

MTOSI Example Using JMS

Abstract

This document illustrates how the rules and recommendations on how to use JMS as MTOSI transport can be exercised, using the support of a concrete example.
The usage of the JMS API will be demonstrated using code snippets.

Table of Contents

MTOSI Example Using JMS	1
Abstract	1
Table of Contents	1
1 Introduction	2
2 Illustrating Request / Response	3
2.1 JMS Queues Management	3
2.2 Scenario 1: <code>SimpleResponse</code> Communication Pattern	7
2.3 Scenario 2: <code>MultipleBatchResponse</code> Communication Pattern	13
3 Illustrating MTOSI Notifications	19
3.1 JMS Topics Management	19
3.2 Scenario 3: Notification Communication Pattern	22
4 References	25
5 Appendix: JMS Platform Used and Configuration	26
6 Administrative Appendix	33
6.1 Document History	33
6.2 Acknowledgments	33
6.3 How to comment on this document	33

1 Introduction

MTOSI requirements call for a transport independent API.

Details of the communication concepts requested at the application level and the different styles supported by MTOSI are presented in [3]. Different Application OSs communicate between each other through the CCV abstraction (Common Communication Vehicle). The operations involve exchanges of XML messages.

Details about how the JMS API can be used to implement the CCV as transport mechanism supporting the MTOSI application level requirements in terms of communication and operation exchange are presented in [4].

This document illustrates those concepts using the support of a concrete example, demonstrating the usage of the JMS API using code snippets. The physical and logical configurations used in the example are shown in [5].

Before going through this document, the reader must read first [3] and [4].

Three OS applications will communicate through the CCV.

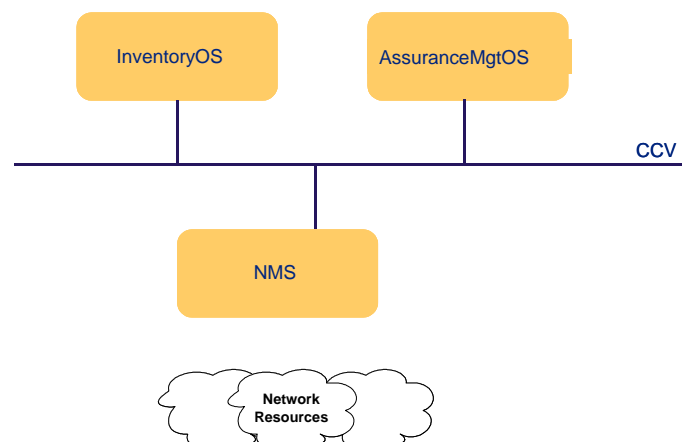


Figure 1. Architecture of the supporting example

The example will demonstrate the usage of JMS in two specific cases only:

- bulk inventory retrieval

The InventoryOS, in a RequestorOS role, will request inventory information from the NMS seen as a SupplierOS. In this demonstration example, the information exchanged between the two JMS applications will use the MTOSI requests and responses dialogue mechanism, and to limit the complexity of this example, only bulk retrieval will be shown.

- notifications

To illustrate notifications, we will consider that the NMS plays the role of a notification publisher and the InventoryOS and the AssuranceMgtOS will both act as notification subscribers.

2 Illustrating Request / Response

In this section, we demonstrate the usage of JMS in the case of bulk inventory retrieval, using point to point communication. The InventoryOS will play the role of the RequestorOS and the NMS the role of the SupplierOS.

Two scenarios will be presented for Request/Response exchanges:

- Scenario 1: `SimpleResponse` communication pattern
The InventoryOS sends a `getInventory` request. The NMS sends a single response back. The InventoryOS collects this response at will.
- Scenario 2: `MultipleBatchResponse` communication pattern
The InventoryOS sends a `getInventory` request. The NMS sends multiple responses back. The InventoryOS collects these responses at will.

To keep things simple, for the scenarios 1 and 2, only one kind of filtering will be considered:

<code>baseInstance</code>	is the list of all the 5 MEs of the monitored network
<code>objectType</code>	is ME, EH, EQ and CC
<code>granularity</code>	is FULL

The decision by the NMS to send a single response or multiple responses is driven by the communication pattern specified by the InventoryOS in each request. From one request to another, the InventoryOS may change the communication pattern.

When the `MultipleBatchResponse` communication pattern is requested, the `requestedBatchSize` parameter must be set (in the payload of the message and also mapped as a JMS specific property field); it will be used by the NMS as follows:

- if `requestedBatchSize` equals 5 or higher, then a single response is sent with information related to the five ME all together (no difference compared to the `SimpleResponse` communication pattern in this case).
- if `requestedBatchSize` is less than 5, then multiple responses will be used, each with a number of MEs not exceeding the value of `requestedBatchSize`.

2.1 JMS Queues Management

In our example, for each of the two point-to-point scenarios 1 and 2, we will use the same principles for the management of the requested JMS Destinations. Two queues only will be used which are created administratively independently from the JMS API itself, since JMS does not define facilities for creating, administering or deleting queues which are not temporary (see the appendix for details about the administrative creation of JMS resources, in the case of the Sun Java™ System Application Server Platform).

We then assume that the following resources have been created.

JMS Resource	Characteristics
Connection Factory	JNDI Name: jms/ConnectionFactory Type: javax.jms.QueueConnectionFactory
Destination Resources	JNDI Name: jms/RtoNMSQA Type: javax.jms.Queue
	JNDI Name: jms/RtoInventoryOSQA Type: javax.jms.Queue

- NMS:

When the NMS starts, it first creates a context to the Application Server and looks up for the connection factory using JNDI. Then it creates a connection that is a session factory, and then it creates a session that will be used as a factory to create queues or topics.

```
// Declare a context, a connectionFactory, a connection and a session
javax.naming.Context      context;
javax.jms.ConnectionFactory connectionFactory;
javax.jms.Connection      connection ;
javax.jms.Session         session ;

// Create a JNDI Context to the application server.
context = new InitialContext()

// Look up the connectionFactory using JNDI.
connectionFactory =
    (ConnectionFactory) context.lookup
        ("jms/ConnectionFactory");

// Ask the factory for a connection.
connection = connectionFactory.createConnection ();

// Create a non transacted session in AUTO_ACKNOWLEDGE mode.
// The messages are acknowledged automatically
session = connection.createSession (false, Session.AUTO_ACKNOWLEDGE);
```

Snippet 1. The NMS creates a session

Then a single message consumer is created and will not be closed until the NMS exits. It corresponds to the queue where incoming requests will be awaited ("jms/RtoNMSQA").

```
// Declare a message consumer
MessageConsumer qConsumer;

// Look up for the destination queue where to receive incoming request
Destination d =(Destination) context.lookup("jms/RtoNMSQA")

// Create a message consumer for this queue
qConsumer = session.createConsumer (d);
```

Snippet 2. The NMS creates a message consumer

At last, the connection is started, activating the message consumer that is now ready to receive incoming messages. Then, the NMS waits indefinitely for incoming requests.

```
// Start the connection, ready for the delivery of incoming requests
connection.start();

// Wait for incoming requests
Message requestMessage = qConsumer.receive();
```

Snippet 3. The NMS starts the connection and waits for incoming requests

- **InventoryOS:**

The InventoryOS proceeds the same way for the look up of the connection factory, the creation of a connection and the creation of a session.

```
// Declare a context, a connectionFactory, a connection and a session
javax.naming.Context          context;
javax.jms.ConnectionFactory    connectionFactory;
javax.jms.Connection           connection ;
javax.jms.Session              session ;

// Create a JNDI Context to the application server.
context = new InitialContext()

// Look up the connectionFactory using JNDI.
connectionFactory = (ConnectionFactory)context.lookup ("jms/ConnectionFactory");

// Ask the factory for a connection.
connection = connectionFactory.createConnection ();

// Create a non transacted session in AUTO_ACKNOWLEDGE mode.
// The messages are acknowledged automatically
session = connection.createSession (false, Session.AUTO_ACKNOWLEDGE);
```

Snippet 4. The InventoryOS creates a session

Then, the InventoryOS creates a message producer (queue "jms/RtoNMSQA") and a message consumer (queue "jms/RtoInventoryOSQA") in any order:

```
// Declare a message consumer and a message producer
MessageConsumer qConsumer;
MessageProducer qProducer;

// Create a message producer
qProducer = session.createProducer (context.lookup("jms/RtoNMSQA"));

// Create a message consumer
qConsumer = session.createConsumer (context.lookup("jms/RtoInventoryOSQA"));

// Start the connection
connection.start();
```

Snippet 5. The InventoryOS creates a message producer and a message consumer

At the end of those steps, the configuration is as shown on the Figure 2 below.

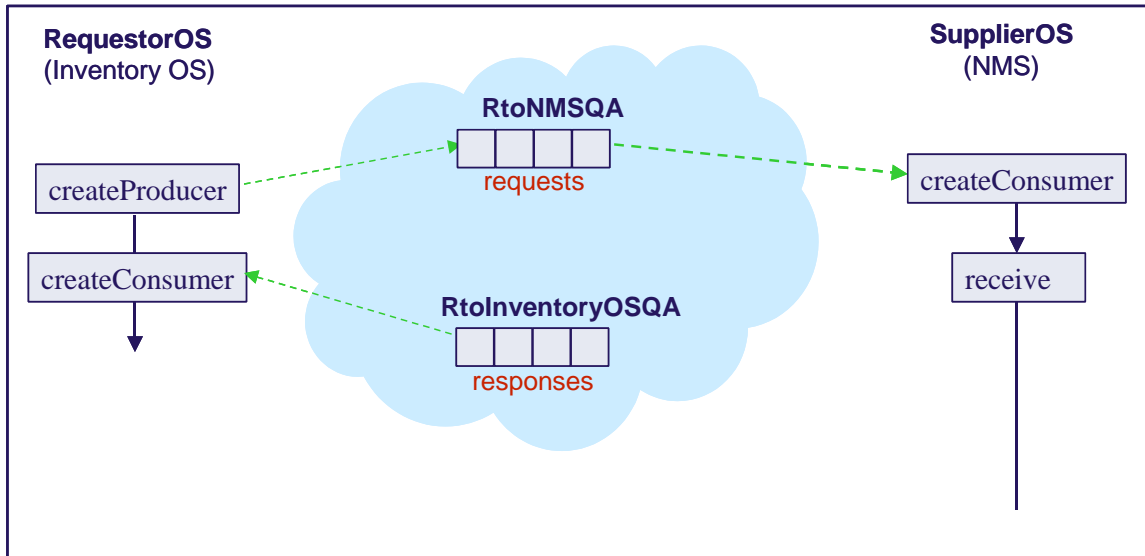


Figure 2. The InventoryOS has created a producer and a consumer; the NMS has created a consumer

2.2 Scenario 1: SimpleResponse Communication Pattern

The InventoryOS sends a request. The NMS sends a single response back. The InventoryOS collects this response at will.

The actual delivery between the JMS Provider and the InventoryOS can be either synchronous (using the `receive` method of the `MessageConsumer`) or asynchronous (creating a `MessageListener` which the JMS Provider will invoke as soon as the response arrives). In our example, we will use the synchronous delivery.



The communication abstraction at the application level is shown on the figure above. The response is sent by the NMS to the InventoryOS without any synchronization dependencies. The InventoryOS can collect this response any time after it has been sent by the NMS.

The application layer passes the `getInventory` request in XML format and nested into a SOAP envelope, as shown below:

```

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns="tmf854.v1">
  <soap:Header>

    <header tmf854Version="1.0">
      <activityName>getInventory</activityName>
      <msgName>getInventoryRequest</msgName>
      <msgType>REQUEST</msgType>
      <correlationId>0001</correlationId>
      <senderURI>jms/RtoInventoryOSQA</senderURI>
    
```

```

    <destinationURI>jms/RtoNMSQA</destinationURI>
    <failureReplytoURI>jms/RtoInventoryOSQA</failureReplytoURI>
    <communicationPattern>SimpleResponse</communicationPattern>
    <communicationStyle>MSG</communicationStyle>
    <timestamp>20051004140305</timestamp>
  </header>

</soap:Header>

```

```

<soap:Body>

  <invr:getInventoryRequest>
    <invr:filter>
      <invr:baseInstance>
        <nam:rdn>
          <nam:type>MD</nam:type>
          <name:value>N1/XdrEMS/Server1</name:value>
        </nam:rdn>
      </invr:baseInstance>
      <invr:includedObjectType>
        <invr:objectType>ME</invr:objectType>
        <invr:granularity>FULL</invr:granularity>
      </invr:includedObjectType>
      <invr:includedObjectType>
        <invr:objectType>EH</invr:objectType>
        <invr:granularity>FULL</invr:granularity>
      </invr:includedObjectType>
      <invr:includedObjectType>
        <invr:objectType>EQ</invr:objectType>
        <invr:granularity>FULL</invr:granularity>
      </invr:includedObjectType>
      <invr:includedObjectType>
        <invr:objectType>CC</invr:objectType>
        <invr:granularity>FULL</invr:granularity>
      </invr:includedObjectType>
    </invr:filter>
  </invr:getInventoryRequest>

</soap:Body>

</soap:Envelope>

```


MTOSI Example Using JMS

Because the SOAP/XML message constructed at the application level is opaque to the JMS layer, some of the XML header parameters must be passed explicitly to the JMS layer which will map them into JMS header or properties fields.

As we saw in [4], after mapping MTOSI headers to JMS headers/properties the resulting mapped values are as shown below:

MTOSI_destinationURI="jms/RtoNMSQA"
MTOSI_senderURI="jms/RtoInventoryOSQA"
MTOSI_activityName="getInventory"
MTOSI_msgType="REQUEST"
MTOSI_msgName="getInventoryRequest"
JMSReplyTo= The JMS destination (of type Destination) whose JNDI name is "jms/RtoInventoryOSQA"
MTOSI_failureReplytoURI="jms/RtoInventoryOSQA"
JMSPriority=4
MTOSI_communicationPattern="SimpleResponse"
MTOSI_communicationStyle="MSG"

MTOSI Example Using JMS

Only two JMS property fields are used (JMSReplyTo, JMSPriority). The other mapped fields are JMS application specific properties. Note that the correlationID is not mapped.

Note that, in this case, there is no requestedBatchSize transmitted in the XML payload, since the communication pattern is SimpleResponse.

```
// The InventoryOS creates the message object to send.
TextMessage message = session.createTextMessage();

// Copy the (opaque) SOAP envelope of the request into the message body
message.setText(...);

// Set the relevant JMS Header fields used
message.setJMSReplyTo(context.lookup("jms/RtoInventoryOSQA"));
message.setJMSPriority (4);

// Set the relevant Message Properties fields used
message.setStringProperty ("MTOSI_destinationURI", "jms/RtoNMSQA");
message.setStringProperty ("MTOSI_senderURI", "jms/RtoInventoryOSQA");
message.setStringProperty ("MTOSI_activityName", "getInventory");
message.setStringProperty ("MTOSI_msgType", "REQUEST");
message.setStringProperty ("MTOSI_msgName", "getInventoryRequest");
message.setStringProperty ("MTOSI_failureReplytoURI", "jms/RtoInventoryOSQA");
message.setStringProperty ("MTOSI_communicationPattern", "SimpleResponse");
message.setStringProperty ("MTOSI_communicationStyle", "MSG");
...
// Sends the request to the "jms/RtoNMSQA" queue through the message producer
qProducer.send(message);

// Keep the messageID of the sent message to correlate the future response
String rqCorrelationID = message.getJMSMessageID ();
```

Snippet 6. The InventoryOS creates a getInventory request message and sends it
(SimpleResponse pattern)

The NMS receives the request transmitted by the JMS provider, through the message consumer associated with the jms/RtoNMSQA queue.

After processing the request, a single response message is prepared to be sent back to the InventoryOS.

At this stage, the InventoryOS uses the senderURI as read from the incoming request to create a temporary producer to the corresponding queue named jms/RfromNMSQA (only the message producer has a temporary existence, the queue itself being permanent, having been created administratively independently from the JMS API).

Then, the message is sent through this message producer. At the end of this operation, the message producer is closed.

Indeed, the communication is not direct between the NMS and the InventoryOS, the response message being sent to the JMS Provider handling the jms/RtoInventoryOSQA queue. At this point in time, the NMS may stop executing, the InventoryOS will be capable to collect the response since it is registered in the JMS Provider.

```
// Wait for incoming requests
TextMessage requestMessage = qConsumer.receive();

// Read the relevant JMS Header fields used
String requestId = requestMessage.getJMSMessageID ();
```

MTOSI Example Using JMS

```
int priority          = requestMessage.getJMSPriority ();
Destination replyTo   = requestMessage.getJMSReplyTo ();

// Read the relevant Message Properties fields used
String msgType = requestMessage.getStringProperty ("MTOSI_msgType");
String msgName = requestMessage.getStringProperty ("MTOSI_msgName");
String pattern = requestMessage.getStringProperty ("MTOSI_communicationPattern");
String style    = requestMessage.getStringProperty ("MTOSI_communicationStyle");

// Create a producer dynamically to the replyTo destination
// specified by the requestor
MessageProducer producer;

producer = session.createProducer (replyTo);

TextMessage message; // Declare a text message for the response

// Prepare response payload (i.e. opaque SOAP)
message.setText (...);

// Set the relevant JMS Header fields used in the response message
// Note that the messageID of the request is used as JMSCorrelationID
message.setJMSCorrelationID (requestId);
message.setJMSPriority (4);

// Set the relevant Message Properties fields used in the response message
message.setStringProperty ("MTOSI_senderURI", "jms/RtoNMSOSQA");
message.setStringProperty ("MTOSI_activityName", "getInventory");
message.setStringProperty ("MTOSI_msgType", "RESPONSE");
message.setStringProperty ("MTOSI_msgName", "getInventoryResponse");
message.setStringProperty ("MTOSI_activityStatus", "SUCCESS");
message.setStringProperty ("MTOSI_communicationPattern", "SimpleResponse");
message.setStringProperty ("MTOSI_communicationStyle", "MSG");
...
// then, send the message and close the producer
producer.send(message) ;
producer.close();
```

Snippet 7. The NMS receives the `getInventory` request, processes it and sends the response back

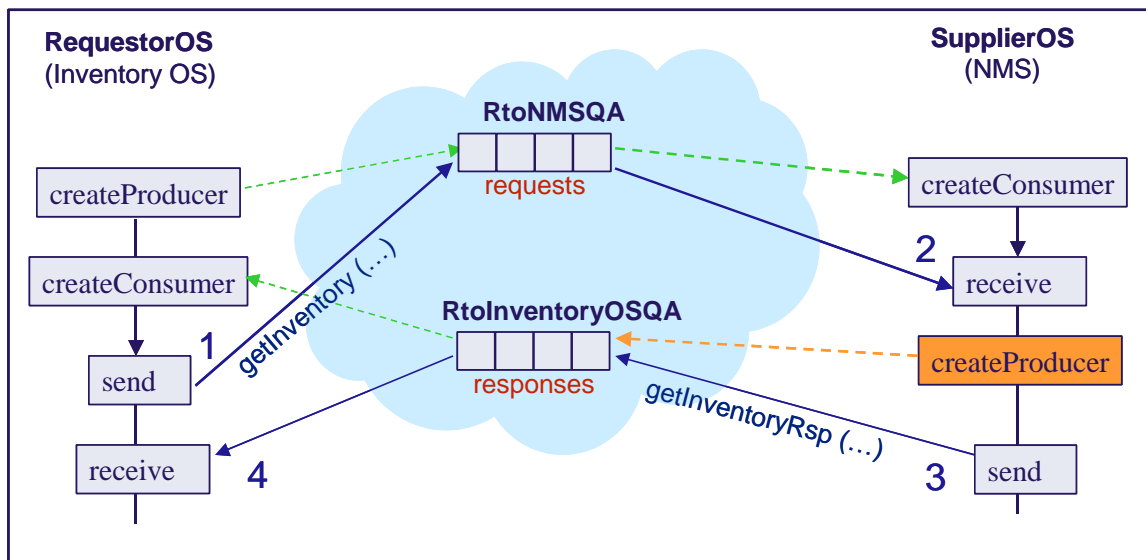


Figure 3. MSG Style, SimpleResponse Pattern

Figure 3 summarizes the different interactions between the two JMS clients and the JMS Provider (blue cloud). The green dotted arrows show the creation of producers and consumers at start time. Then, the request message is sent by the InventoryOS and received by the NMS (solid arrows 1 and 2). The NMS creates a producer dynamically (orange) and then sends its response message to the JMS provider (solid arrow 3).

The InventoryOS can then collect, at any time, the response message available in the JMS provider. (recall that, in this example, we have used the `receive` call to the message consumer; a listener mechanism is also possible, listening for any incoming delivery from the JMS provider, as soon as a new response message is available for this message consumer).

```
Message responseMessage = qConsumer.receive();// a listener is also possible

// Read the relevant JMS Header fields used
String rspCorrelationID = responseMessage.getJMSCorrelationID ();
                        // used to compare with "rqCorrelationID"

// Read the relevant Message Properties fields used
String msgType          = responseMessage.getStringProperty ("MTOSI_msgType");
                        // Should be "RESPONSE"
String activityStatus = responseMessage.getStringProperty ("MTOSI_activityStatus");
...
```

Snippet 8. The InventoryOS receives the response message

2.3 Scenario 2: MultipleBatchResponse Communication Pattern

The InventoryOS sends a single request with an additional parameter compared to the *SimpleResponse* scenario: the `requestedBatchSize` allowing to specify the maximum number of units that may be present in each partial response. In the case this parameter exceeds the actual number of units (in our case a unit is a ME) matching the selection criteria, then the NMS must split the complete response into multiple partial responses, and send as many partial responses as necessary.

The NMS sends multiple responses back without any synchronization dependencies, and without waiting.

The InventoryOS collects these responses at will.



The communication abstraction at the application level is shown on the figure above.

The InventoryOS can collect these partial responses one by one at any time after each of them has been sent by the NMS and collected by the JMS provider.

In our example, as shown in the XML representation of the `getInventory` request below, the communication pattern selected is `MultipleBatchResponse` and value 2 is entered for the `requestedBatchSize`.

The InventoryOS then sends the request to the `jms/RtoNMSQA` (through the corresponding producer). It is then received by the NMS waiting on the same queue as a consumer. So far, to the

MTOSI Example Using JMS

exception of the two parameters mentioned above, there has not been any difference compared to the scenario 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
xmlns="tmf854.v1">
  <soap:Header>

    <header tmf854Version="1.0">
      <activityName>getInventory</activityName>
      <msgName>getInventoryRequest</msgName>
      <msgType>REQUEST</msgType>
      <destinationURI>jms/RtoNMSQA</destinationURI>
      <senderURI>jms/RtoInventoryOSQA</senderURI>
      <failureReplytoURI>jms/RtoInventoryOSQA</failureReplytoURI>
      <correlationId>0002</correlationId>
      <priority>4</priority>
      <communicationPattern>MultipleBatchResponse</communicationPattern>
      <communicationStyle>MSG</communicationStyle>
      <requestedBatchSize>2</requestedBatchSize>
      <timestamp>20051004140406</timestamp>

    </header>

  </soap:Header>
```

The mapping from the MTOSI header parameters into the JMS headers/properties will result in the values shown in the table below:

MTOSI_destinationURI="jms/RtoNMSQA"
MTOSI_senderURI="jms/RtoInventoryOSQA"
MTOSI_activityName="getInventory"
MTOSI_msgType="REQUEST"
MTOSI_msgName="getInventoryRequest"
JMSReplyTo= The JMS destination (of type Destination) whose JNDI name is "jms/RtoInventoryOSQA"
MTOSI_failureReplytoURI="jms/RtoInventoryOSQA"
JMSPriority=4
MTOSI_communicationPattern="MultipleBatchResponse"
MTOSI_communicationStyle="MSG"
MTOSI_requestedBatchSize=2

```
// The InventoryOS creates the message object to send.
TextMessage message = session.createTextMessage();

// Copy the (opaque) SOAP envelope of the request into the message body
```

MTOSI Example Using JMS

```
message.setText (...);

// Set the relevant JMS Header fields used
message.setJMSReplyTo(context.lookup("jms/RtoInventoryOSQA"));
message.setJMSPriority (4);

// Set the relevant Message Properties fields used
message.setStringProperty ("MTOSI_destinationURI", "jms/RtoNMSQA");
message.setStringProperty ("MTOSI_senderURI", "jms/RtoInventoryOSQA");
message.setStringProperty ("MTOSI_activityName", "getInventory");
message.setStringProperty ("MTOSI_msgType", "REQUEST");
message.setStringProperty ("MTOSI_msgName", "getInventoryRequest");
message.setStringProperty ("MTOSI_failureReplytoURI", "jms/RtoInventoryOSQA");
message.setStringProperty ("MTOSI_communicationPattern", "MultipleBatchResponse");
message.setStringProperty ("MTOSI_communicationStyle", "MSG");
message.setIntProperty ("MTOSI_requestedBatchSize", 2);
...
// Sends the request to the "jms/RtoNMSQA" queue through the message producer
qProducer.send (message);

// Keep the messageID of the sent message to correlate the future response
String rqCorrelationID = message.getJMSMessageID ();
```

Snippet 9. The InventoryOS creates a `getInventory` request message and sends it (MultipleBatchResponse pattern)

Upon reception, the NMS processes the request by considering the `requestedBatchSize` parameter, in association with the filter. The `requestedBatchSize` parameter specifies that the maximum number of MEs that may be returned per response message is 2. Since the NMS handles 5 MEs, then 3 response messages are prepared.

- response 1 with ME 1 and 2 (OM3500CO32 54016 and OM3500CO31 768)
- response 2 with ME 3 and 4 (OM3500CO33 19968 and 20005)
- response 3 with ME 5 (19204)

Each response is sent back as soon as it is available. Indeed, each response message can be collected at will by the InventoryOS, once received by the JMS Provider.

After the 3 responses have been sent, the message producer is closed by the NMS.

Indeed, the communication is not direct between the NMS and the InventoryOS, the response messages being sent to the JMS provider handling the `jms/RtoInventoryOSQA` queue. At this point in time, the NMS may stop executing, the InventoryOS will be capable to collect all the partial responses since they are registered in the JMS Provider.

MTOSI Example Using JMS

```
// Wait for incoming requests
TextMessage requestMessage = qConsumer.receive();

// Read the relevant JMS Header fields used
String requestId      = requestMessage.getJMSMessageID ();
int priority          = requestMessage.getJMSPriority ();
Destination replyTo   = requestMessage.getJMSReplyTo ();

// Read the relevant Message Properties fields used
String msgType = requestMessage.getStringProperty ("MTOSI_msgType");
String msgName = requestMessage.getStringProperty ("MTOSI_msgName");
String pattern = requestMessage.getStringProperty ("MTOSI_communicationPattern");
String style   = requestMessage.getStringProperty ("MTOSI_communicationStyle");
int requestedBatchSize;

if (pattern.equals ("MultipleBatchResponse")) {
    requestedBatchSize = requestMessage.getIntProperty ("MTOSI_requestedBatchSize");
}

// Create a producer dynamically to the replyTo destination
// specified by the requestor
MessageProducer producer;

producer = session.createProducer (replyTo);

TextMessage message; // Declare a text message for the responses

int sequenceNb = 0;
boolean isLastResponse = false;
while (true) {
    message.setText(...); // Prepare partial response payload
    sequenceNb++;
    isLastResponse = ...; // Set to true for the last partial response only

    // Set the relevant JMS Header fields used in the response messages
    // Note that the same JMSCorrelationID is used for all the responses
    message.setJMSCorrelationID (requestId);

    // Set the relevant JMS Properties fields used in the response message
    message.setStringProperty ("MTOSI_senderURI", "jms/RtoNMSOSQA");
    message.setStringProperty ("MTOSI_activityName", "getInventory");
    message.setStringProperty ("MTOSI_msgType", "RESPONSE");
    message.setStringProperty ("MTOSI_msgName", "getInventoryResponse");
    message.setStringProperty ("MTOSI_activityStatus", "SUCCESS");
    message.setStringProperty ("MTOSI_communicationPattern", "MultipleBatchResponse");
    message.setStringProperty ("MTOSI_communicationStyle", "MSG");

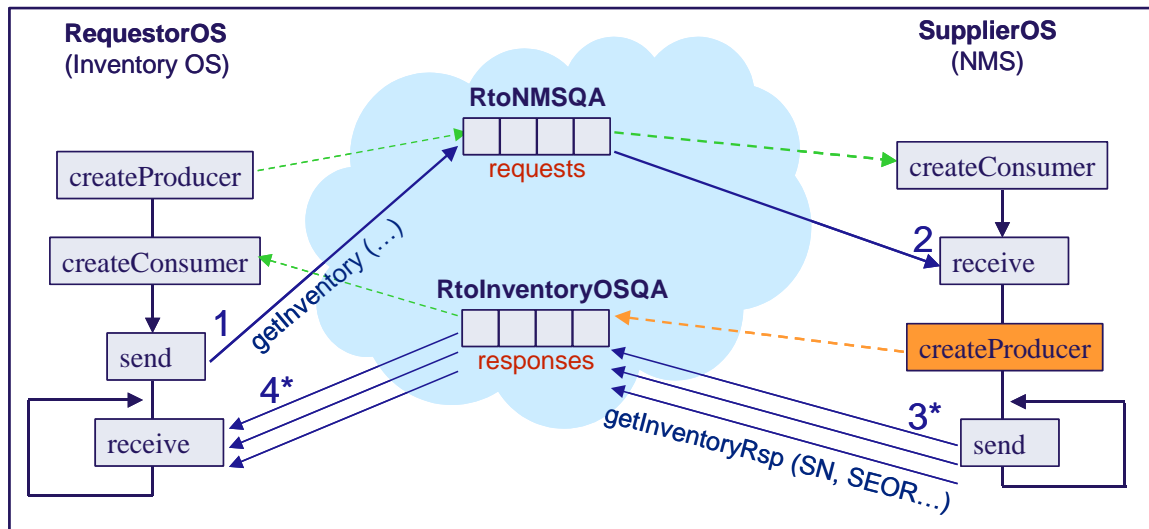
    // Including the batchSize and batchSizeEndOfReply fields
    // since the communication pattern is MultipleBatchResponse
    message.setBooleanProperty ("MTOSI_batchSequenceEndOfReply", isLastResponse)
    message.setIntProperty      ("MTOSI_batchSequenceNumber", sequenceNb);

    // Then send the message
    producer.send (message);

    if (isLastResponse) break;
}

producer.close();
```

Snippet 10. The NMS receives the `getInventory` request, processes it and sends multiple responses back



The Figure 4 summarizes the different interactions between the two JMS clients and the JMS provider (blue cloud).

The process is the same as for the scenario 1 up to the reception of the request and the dynamic creation of producer by the NMS.

Each of the three responses is sent individually to the JMS Provider (solid arrow 3).

The InventoryOS can then collect them, at any time, by querying the JMS provider.

A degraded case of this scenario is when the `requestedBatchSize` is big enough to allow the InventoryOS to send a single response. In this case, indeed, a single response is supplied with the `batchSequenceEndOfReply` parameter set to `true`.

3 Illustrating MTOSI Notifications

In this section, we demonstrate the usage of JMS in the case when the NMS publishes notifications, using publish/subscribe communication.

The scenario that will be presented is as follows:

- **Scenario 3: Notification communication pattern**
The NMS publishes MTOSI notifications while the InventoryOS and the FaultManagementOS act as subscribers.

3.1 JMS Topics Management

We will consider notifications from two different MTOSI topics: MTOSI Inventory Topic notifications and MTOSI Fault Topic notifications. As recommended in [4], the NMS will use two JMS topics, one for each of the two MTOSI notification topics considered.

(The term “topic” is overloaded here: we will always use either MTOSI or JMS as prefix to avoid confusion).

We, then, create administratively the two following additional JMS resources.

JMS Resource	Characteristics	
Destination Resources	JNDI Name:	jms/RinventoryTopic
	Type:	javax.jms.Topic
	JNDI Name:	jms/RfaultTopic
	Type:	javax.jms.Topic

- **NMS**

In addition to the single message consumer already created for the Request/Response scenario as shown in the Snippet 1, the NMS creates two message producers, as shown below, to publish respectively MTOSI Inventory Topic notifications and MTOSI Fault Topic notifications.

```
// Declare message producers
MessageConsumer tProducerInventory,
                tProducerFault;

// Create a message producer for MTOSI Inventory notifications
tProducerInventory = session.createProducer
                    (context.lookup("jms/RinventoryTopic"));
...
// Create a second producer for MTOSI Fault notifications
tProducerFault = session.createProducer
```

Snippet 11. The NMS creates two message producers

- **InventoryOS**

Our rationale, in this example, is the following: after the InventoryOS has executed the bulk inventory retrieval, as shown in the previous section, and has aligned its database, we assume that it is interested in receiving from the NMS any notification relative to the creation of new objects or the deletion of existing objects that may occur in the network under control, and also notifications relative to attribute change.

To start this process, the InventoryOS creates an additional message consumer, to consume the notifications published on the `jms/RinventoryTopic` JMS topic. Because it is interested only in the `ObjectCreation`, `ObjectDeletion` and `AttributeValueChange` notification types, it uses an appropriate message selector while creating the message consumer, as shown in the code snippet below:

```
// Declare a message consumer
MessageConsumer tConsumer;

// Look up for the destination topic where to receive notifications
Destination d =(Destination) context.lookup("jms/RinventoryTopic")

// Create a message consumer for this topic, with the appropriate message selector
tConsumer =
    session.createConsumer
    (d,
    "MTOSI_EventType = 'ObjectCreation' OR
    MTOSI_EventType = 'ObjectDeletion' OR
    MTOSI_EventType = 'AttributeValueChange'");
```

Snippet 12. The InventoryOS creates a message consumer with the appropriate selector

- AssuranceMgtOS

We then assume that the AssuranceMgtOS is interested to be notified of the creation and deletion of objects in the NMS and of the faults raised by the NMS. On the contrary to the InventoryOS, we also assume that it is NOT interested into the `AttributeValueChange` notification.

As a consequence, the AssuranceMgtOS needs to subscribe to the two JMS topics to which the NMS is publishing, to receive both MTOSI Inventory Topic notifications and MTOSI Fault Topic notifications.

Because it is interested only in the `ObjectCreation` and `ObjectDeletion` inventory notification types (and not by the `AttributeValueChange` notification), it will use the following message selector when creating the message consumer associated with the `jms/RinventoryTopic`:

```
MTOSI_EventType = 'ObjectCreation' OR
MTOSI_EventType = 'ObjectDeletion'
```

To complicate a bit our story, we also assume that the AssuranceMgtOS wants to receive alarms only if the `perceivedSeverity` level is either `PS_INDETERMINATE` (0) or `PS_CRITICAL` (1) or `PS_MAJOR` (2). Then the following message selector will be used:

```
MTOSI_EventType = 'AlarmInformation' AND
(MTOSI_perceivedSeverity = 0 OR
MTOSI_perceivedSeverity = 1 OR
MTOSI_perceivedSeverity = 2)
```

MTOSI Example Using JMS

Here below is the corresponding code snippet (indeed, we make the assumption that the same steps as shown in Snippet 1 have been executed first for the AssuranceMgtOS):

```
// Declare message consumers
MessageConsumer tConsumerInventory,
                tConsumerFault;

// Create a message consumer for the jms/RinventoryTopic topic,
// with the appropriate message selector
tConsumerInventory =
    session.createConsumer
        (context.lookup("jms/RinventoryTopic"),
         "MTOSI_EventType = 'ObjectCreation' OR
          MTOSI_EventType = 'ObjectDeletion'");

// Create a message consumer for the jms/RfaultTopic topic,
// with the appropriate message selector
tConsumerFault =
    session.createConsumer
        (context.lookup("jms/RfaultTopic"),
         "MTOSI_EventType = 'AlarmInformation' AND
          (MTOSI_perceivedSeverity = 0 OR
           MTOSI_perceivedSeverity = 1 OR
           MTOSI_perceivedSeverity = 2)");
```

Snippet 13. The AssuranceMgtOS creates two message consumers with the appropriate selectors

At the end of those steps, the configuration is as shown on the Figure 5 below.

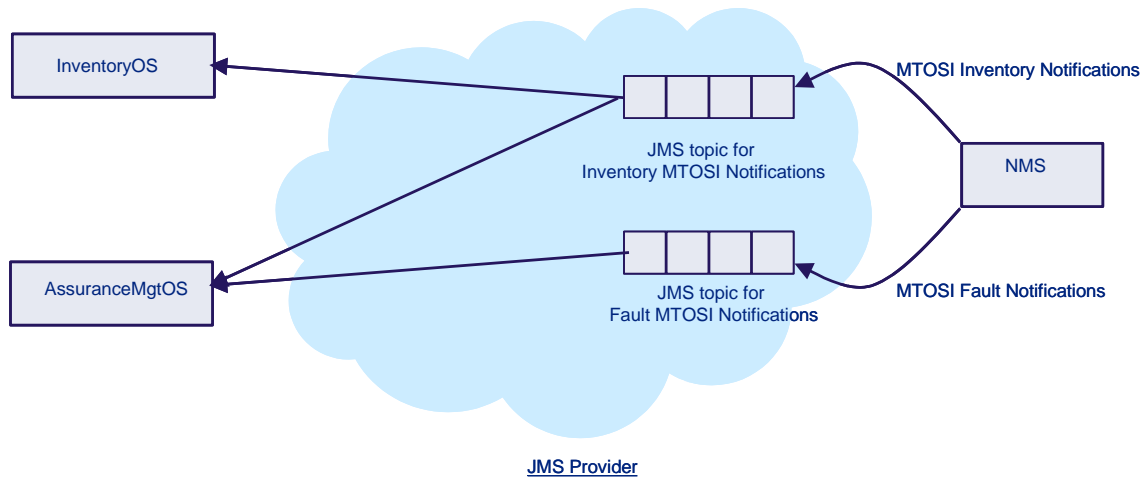


Figure 5. Two JMS topics and the associated flow of information

3.2 Scenario 3: Notification Communication Pattern

The NMS will send four MTOSI notifications as follows:

AlarmInformation notification with perceivedSeverity equals PS_MINOR (3)
 AlarmInformation notification with perceivedSeverity equals PS_CRITICAL (1)
 ObjectCreation notification
 AttributeValueChange notification

The code snippet below is common to all those 4 notifications publishing:

```
// The NMS creates the message object to send
TextMessage message = session.createTextMessage();

// Create the message body containing the actual notification in XML format
message.setText(...);

// Set the relevant JMS Header fields used
// No JMSReplyTo, No JMSCorrelationID
message.setJMSPriority (4);

// Set the Message Properties fields relevant for any notification
// No MTOSI_senderURI, No MTOSI_destinationURI
message.setStringProperty ("MTOSI_msgType", "NOTIFICATION");
message.setStringProperty ("MTOSI_communicationPattern", "Notification");
```

MTOSI Example Using JMS

Snippet 14. The NMS prepares general message properties fields for notifications

For each of the four notifications, a specific mapping of the relevant parameters to message properties fields is done:

```
// Set the Message Properties fields relevant for the specific notification
message.setStringProperty ("MTOSI_EventType", "AlarmInformation");
message.setIntProperty ("MTOSI_perceivedSeverity", 3); // PS_MINOR
...

// Sends the notification to the appropriate JMS topic
tProducerFault.send(message);
```

Snippet 15. The NMS sends a `AlarmInformation` notification
with `perceivedSeverity equals PS_MINOR (3)`

The `AlarmInformation` notification with `perceivedSeverity equals PS_MINOR (3)` is filtered out by the message selector registered on the `jms/RfaultTopic` JMS topic on behalf of the AssuranceMgtOS.

The AssuranceMgtOS will not receive it.

```
// Set the Message Properties fields relevant for the specific notification
message.setStringProperty ("MTOSI_EventType", "AlarmInformation");
message.setIntProperty ("MTOSI_perceivedSeverity", 1); // PS_CRITICAL
...

// Sends the notification to the appropriate JMS topic
tProducerFault.send(message);
```

Snippet 16. The NMS sends a `AlarmInformation` notification
with `perceivedSeverity equals PS_CRITICAL (1)`

The `AlarmInformation` notification with `perceivedSeverity equals PS_CRITICAL (1)` is filtered in by the message selector registered on the `jms/RfaultTopic` JMS topic on behalf of the AssuranceMgtOS.

The AssuranceMgtOS will receive it.

```
// Set the Message Properties fields relevant for the specific notification
message.setStringProperty ("MTOSI_EventType", "ObjectCreation");
...

// Sends the notification to the appropriate JMS topic
tProducerInventory.send(message);
```

Snippet 17. The NMS sends a `ObjectCreation` notification

The `ObjectCreation` notification is filtered in by the two message selectors registered on the `jms/RfaultTopic` JMS topic on behalf of the AssuranceMgtOS and the InventoryOS.

The AssuranceMgtOS and the InventoryOS will receive it.

```
// Set the Message Properties fields relevant for the specific notification
message.setStringProperty ("MTOSI_EventType", "AttributeValueChange");
...

// Sends the notification to the appropriate JMS topic
tProducerInventory.send(message);
```

Snippet 18. The NMS sends a `AttributeValueChange` notification

The `AttributeValueChange` notification is filtered out by the message selector registered on the `jms/RfaultTopic` JMS topic on behalf of the `AssuranceMgtOS`.

The `AssuranceMgtOS` will not receive it.

The same notification is filtered in by the message selector registered on the `jms/RfaultTopic` JMS topic on behalf of the `InventoryOS`.

The `InventoryOS` will receive it.

4 References

- [1] Specification document: Java Message Service, Version 1.1, April 12, 2002
- [2] J2EE 1.4 and JMS 1.1 Tutorial, <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/J2EETutorial.pdf>
- [3] [SD2-5](#), Communication Styles, 2008.
- [4] [SD2-9](#), Using JMS as MTOSI Transport, 2008.
- [5] [SD2-4](#), Transport Independent Example of MTOSI, 2008.

5 Appendix: JMS Platform Used and Configuration

Note: this appendix is based on early version of the J2EE environment from Sun. More recent versions of the Sun Application Server are available, but the principles remain unchanged.

- Download

The JMS platform used as support for the example is the

Sun Java™ System Application Server Platform Edition 8 2004Q4 Beta

which you can download from <http://java.sun.com/j2ee/1.4/download.html>.

The simplest is to download the full bundle as shown in the **All-In-One Bundle** frame. Be careful to use release 2004Q4 Beta and not simply the plain version 8, since some nasty bugs have been fixed.

Download also the following tutorial:

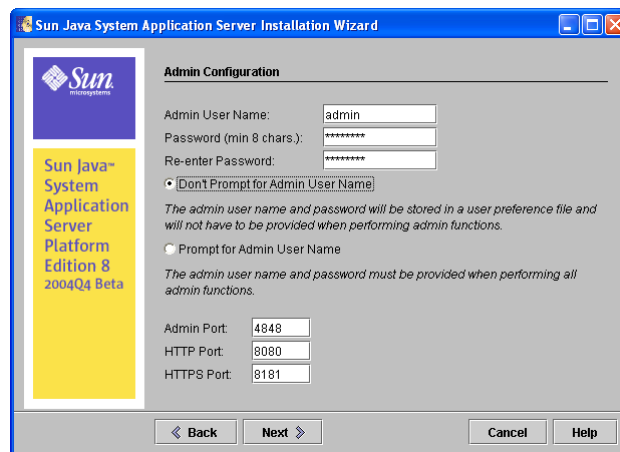
J2EE 1.4 Tutorial Update 3 (for Sun Java System Application Server Platform Edition 2004Q4 Beta) *September 20, 2004*

from <http://java.sun.com/j2ee/1.4/download.html#tutorial>

Indeed, the code of the example could also be used with any other platform supporting the JMS API.

- Install

- install the Sun Application Server
(double click on the “sjsas_pe-8_2004Q4-beta-windows.exe” file).
When prompted for a User Name and a Password, I used “admin” and “password” resp.



We will assume you have installed the Application Server under the default directory “C:\Sun\AppServer”.

- Install the tutorial.

We will assume you have installed the tutorial under “C:\Sun\j2eetutorial14”.

(If you use a different location, remember that you should not install the tutorial to a location with spaces in the path).

MTOSI Example Using JMS

- **Configure**

You will find the instructions to configure your environment in the tutorial document, chapter “About This Tutorial”, section “Building the Examples”.

The configuration steps are:

- Add the following to your CLASSPATH system environment variable:
C:\Sun\AppServer\lib\j2ee.jar
- Add the following to your PATH system environment variable:
C:\Sun\AppServer\bin
- Edit the file “C:\Sun\j2eetutorial14\examples\common\build.properties” and set the three following variables (note the use of “/” instead of “\”):
j2ee.home=C:/Sun/AppServer
j2ee.tutorial.home=C:/Sun/j2eetutorial14
admin.user=admin
- Edit the file “C:\Sun\j2eetutorial14\examples\common\admin-password.txt” and set the unique variable as follows
AS_ADMIN_PASSWORD=password

- **Validate the installation**

While this step is not strictly necessary, it is highly recommended.

To validate the installation we will use the simplest example supplied as part of the tutorial samples.

Under “C:\Sun\j2eetutorial14\examples\jms\simple”, we will use files SimpleProducer.java and SimpleSynchConsumer.java to construct a Producer and a Consumer exchanging messages using a JMS queue.

You can follow the instructions in chapter 33 “The JAVA Message Service API”, section “Writing Simple JMS Client Applications”, subsection “Creating JMS Administered Objects”.

- Creating JMS Administered Objects

Start the Application Server:

Programs → Sun Microsystems → Application Server → Start Default Server

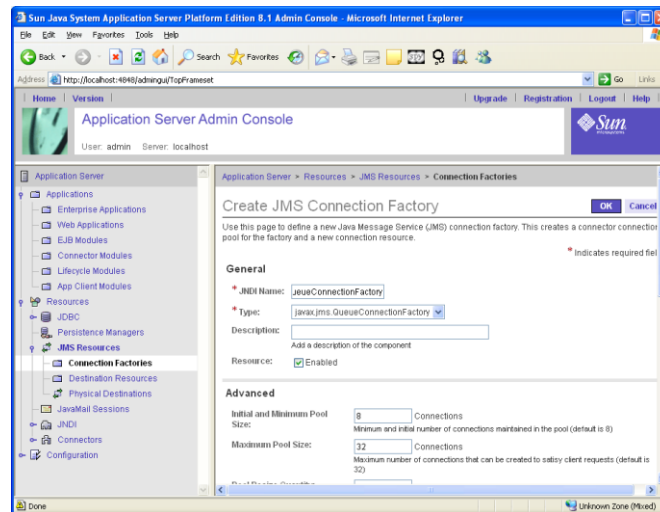
Start the admin console (Note that there is also a cmd line interface which can be used alternatively: “asadmin.bat”):

Programs → Sun Microsystems → Application Server → Admin Console

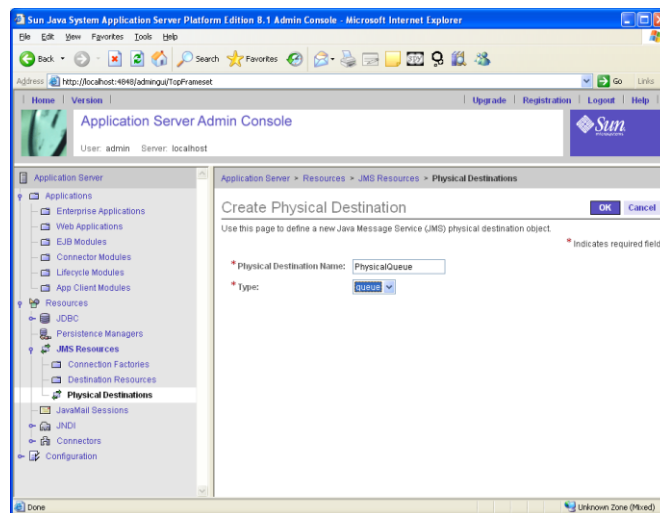
- Creating a connection factory:
 1. In the tree component, expand the Resources node, then expand the JMS Resources node.
 2. Select the Connection Factories node.
 3. On the JMS Connection Factories page, click New. The Create JMS Connection Factory page appears.
 4. In the JNDI Name field, type `jms/QueueConnectionFactory`.
 5. Choose `javax.jms.QueueConnectionFactory` from the Type combo box.

MTOSI Example Using JMS

6. Select the Enabled checkbox. The Admin Console appears as shown below.
7. Click OK to save the connection factory.



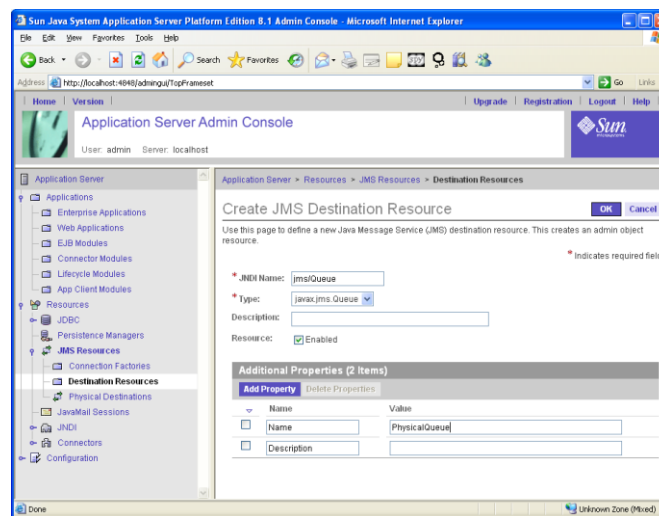
- Creating a physical queue:
 1. In the tree component, expand the Resources node, then expand the JMS Resources node.
 2. Select the Physical Destinations node.
 3. On the Physical Destinations page, click New. The Create Physical Destination page appears.
 4. In the Physical Destination Name field, type `PhysicalQueue`.
 5. Choose `queue` from the Type combo box.
 6. Click OK.



- Creating a resource destination queue:

MTOSI Example Using JMS

1. In the tree component, expand the Resources node, then expand the JMS Resources node.
2. Select the Destination Resources node.
3. On the JMS Destination Resources page, click New. The Create JMS Destination Resource page appears.
4. In the JNDI Name field, type `jms/Queue`.
5. Choose `javax.jms.Queue` from the Type combo box.
6. Select the Enabled checkbox.
7. In the Additional Properties area, type `PhysicalQueue` in the Value field for the Name property.
8. Click OK.



○ Packaging the SimpleProducer and the SimpleSynchConsumer

The simplest way to run these examples using the Application Server is to package each one in an application client JAR file.

First, start deploytool:

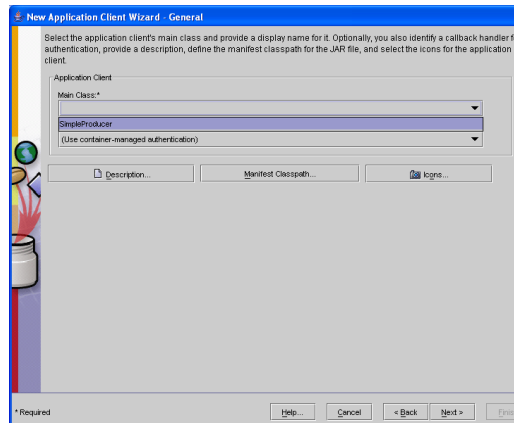
Start → Programs → Sun Microsystems → Application Server → Deploytool

Package the SimpleProducer example as follows:

1. Choose File → New → Application Client to start the Application Client wizard.
2. In the JAR File Contents screen, select the radio button labeled Create New Stand-Alone AppClient Module.
3. Click Browse next to the AppClient Location field and navigate to the `C:\Sun\j2eetutorial14\examples\jms\simple` directory.
4. Type `SimpleProducer` in the File Name field, and click Create Module File.
5. Verify that `SimpleProducer` appears in the AppClient Name field.
6. Click the Edit Contents button next to the Contents text area.
7. In the dialog box, locate the `build` directory. Select `SimpleProducer.class` from the Available Files tree. Click Add and then OK.

MTOSI Example Using JMS

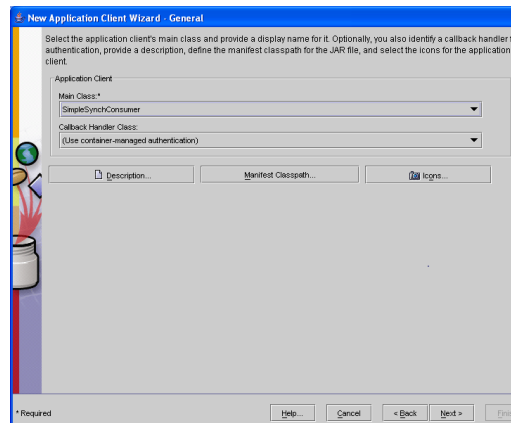
8. In the General screen, select `SimpleProducer` in the Main Class combo box.
9. Click Next.
10. Another panel will open. Go to the Main Class dialog box and select `SimpleProducer`



11. Click Next.
12. Click Finish.

Package the `SimpleSynchConsumer` example as follows:

13. Choose File → New → Application Client to start the Application Client wizard.
14. In the JAR File Contents screen, select the radio button labeled Create New Stand-Alone AppClient Module.
15. Check that `C:\Sun\j2eetutorial14\examples\jms\simple` is displayed in the AppClient Location field.
16. Click Browse next to the AppClient Location field.
17. Type `SimpleSynchConsumer` in the File Name field, and click Create Module File.
18. Verify that `SimpleSynchConsumer` appears in the AppClient Name field.
19. Click the Edit Contents button next to the Contents text area.
20. In the dialog box, locate the build directory. Select `SimpleSynchConsumer.class` from the Available Files tree. Click Add and then OK.
21. In the General screen, select `SimpleSynchConsumer` in the Main Class combo box.
22. Click Next
23. Another panel will open. Go to the Main Class dialog box and select `SimpleSynchConsumer`



24. Click Next.
25. Click Finish.

o Running the SimpleProducer and the SimpleSynchConsumer

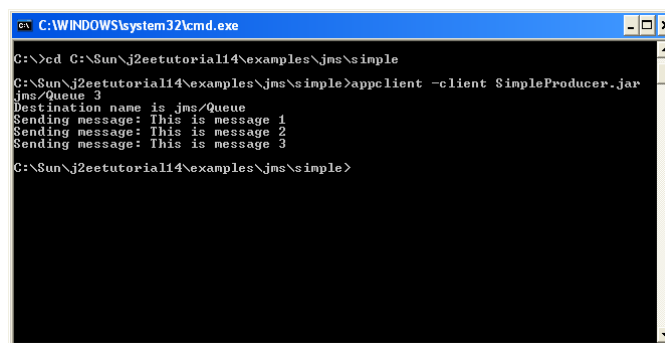
You run the sample programs using the `appclient` command. Each of the programs takes command-line arguments: a destination name, a destination type, and, for `SimpleProducer`, a number of messages.

Open a cmd window and go to the `C:\Sun\j2eetutorial14\examples\jms\simple` directory:

1. Run the `SimpleProducer` program, sending three messages to the queue `jms/Queue`:

```
appclient -client SimpleProducer.jar jms/Queue 3
```

The output of the program looks like this:

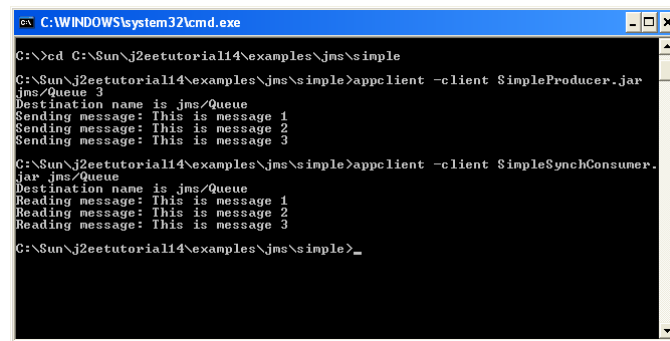


1. In the same window, run the `SimpleSynchConsumer` program, specifying the queue name:

```
appclient -client SimpleSynchConsumer.jar jms/Queue
```

The output of the program looks like this:

MTOSI Example Using JMS



```
C:\WINDOWS\system32\cmd.exe
C:\>cd C:\Sun\j2eetutorial14\examples\jms\simple
C:\Sun\j2eetutorial14\examples\jms\simple>appclient -client SimpleProducer.jar
jms/Queue 3
Destination name is jms/Queue
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
C:\Sun\j2eetutorial14\examples\jms\simple>appclient -client SimpleSynchConsumer.
jms/Queue
Destination name is jms/Queue
Reading message: This is message 1
Reading message: This is message 2
Reading message: This is message 3
C:\Sun\j2eetutorial14\examples\jms\simple>
```


6 Administrative Appendix

6.1 Document History

Version	Date	Description of Change
1.0	May 2005	This is the first version of this document and as such, there are no revisions to report.
1.1	Dec 2005	Added reference [6]. Replaced "originatorURI" by "senderURI" Replaced "operationRespStatus" by "activityStatus" Removed "Domain" Removes "payloadVersion" Updated the snippets (XML and Java)
1.2	May 2008	Aligned with the MTOSI 2.0 artifacts

6.2 Acknowledgments

First Name	Last Name	Company
Paul	McCluskey	Amdocs OSS
Paul	Watkin	Amdocs OSS

6.3 How to comment on this document

Comments and requests for information must be in written form and addressed to the contact identified below:

Michel	Besson	Amdocs OSS
Phone:	+44 7717 692 178	
Fax:		

e-mail:	Michel.Besson@Amdocs.com
---------	--

Please be specific, since your comments will be dealt with by the team evaluating numerous inputs and trying to produce a single text. Thus we appreciate significant specific input. We are looking for more input than wordsmith" items, however editing and structural help are greatly appreciated where better clarity is the result.