Using MQSeries as a Transactional Resource Manager with WebLogic Server

Version 1.0
October 25, 2001

Copyright
Copyright © 2001 BEA Systems, Inc. All Rights Reserved.
October 25, 2001

Restricted Rights Legend
This document may not, in whole or in part, be photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc. Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems, Inc.

Limitation of Liability
BEA Systems, Inc. assumes no liability for use or misuse of the information contained in this document. No guarantee is made as to the accuracy or correctness of the information contained herein. This document falls under the limitation of liability in the WebLogic Server license document.

Trademarks
BEA, WebLogic and Tuxedo are registered trademarks of BEA Systems, Inc. BEA WebLogic Server, BEA WebLogic E-Business Platform, BEA WebLogic Integration, BEA WebLogic Portal, BEA WebLogic Commerce Server, BEA WebLogic Personalization Server, and BEA Campaign Manager for WebLogic, are trademarks of BEA Systems.
All other trademarks are the properties of their respective owners.

Contents

1 Introduction

The MQSeries Java Messaging Service (JMS) driver may be used to incorporate MQSeries as a resource manager in WebLogic Server (WLS) distributed transactions. This document describes a set of support classes, utilities and techniques for utilizing MQSeries JMS with WLS.

Because the MQSeries JMS driver transaction support is generic with respect to external transaction managers, it is necessary to provide a layer of abstraction between the MQSeries JMS implementation and WLS to perform such application server specific functions as resource registration and dynamic transaction enlistment. This integration layer manages the interaction of the MQSeries resource manager with the WLS Transaction Manager (TM). Since the integration layer conforms to standard JMS interfaces, applications maintain portability.

In addition to the integration layer, a helper-class is provided to create and configure WLS-specific MQSeries connection factories based on previously configured MQSeries connection factories. The utility retrieves MQSeries JMS objects from the specified Java Naming and Directory Interface ™ (JNDI) context, creates the appropriate WLS MQSeries JMS objects, and places them in the JNDI context of an active WLS for use by application components.

The following sections describe how to configure MQSeries for use with WLS, how to administer MQSeries JMS objects and how to manipulate these objects at runtime within the scope of a global WLS transaction. Also, guidelines for verifying MQSeries transaction participation are presented. Throughout the following sections a simple example is presented that illustrates various configuration and programming issues. The example is comprised of a WLS startup class that services RMI requests to perform MQSeries work transactionally. Note that the example involves JMS queues, however the concepts and procedures are also applicable to JMS topics.

2 Configuring MQSeries for Use with WebLogic Server

This section describes at a high level how to configure MQSeries for use with WLS. For additional information, refer to [2].

1. Install MQSeries 5.2. A 60-day evaluation copy for Windows NT and Windows 2000 is available at http://www.ibm.com/software/ts/mqseries/downloads. Follow the appropriate installation procedures for the target operating system.

2. Download and install the MQSeries SupportPac MA88: MQSeries classes for Java and MQSeries classes for Java Message Service from http://www.ibm.com/software/ts/mqseries/txppacs/ma88.html.

3. For utilizing Topics, download and install the MQSeries SupportPac MA0C: MQSeries - Publish/Subscribe from http://www.ibm.com/software/ts/mqseries/txppacs/ma0c.html

4. Add the following settings to the WLS environment scripts (startWebLogic.cmd and setEnv.cmd).

```
set MQ_INSTALL_PATH=[path of MQSeries installation]
      e.g. c:\Program Files\MQSeries
set MQ_JAVA_INSTALL_PATH=[path of Java installation]
      e.g. c:\Program Files\MQSeries
set WLS_MQ_JAVA_INSTALL_PATH=[path of WLS MQSeries installation]
      e.g. c:\WLSMQSeries
set
CLASSPATH=%CLASSPATH%;%MQ_JAVA_INSTALL_PATH%\Java\samples\base;%MQ_J
AVA_INSTALL_PATH%\java\lib\com.ibm.mq.jar;%MQ_JAVA_INSTALL_PATH%\jav
a\lib\com.ibm.mqjms.jar;%MQ_JAVA_INSTALL_PATH%\java\lib\jms.jar;%MQ_
JAVA_INSTALL_PATH%\java\lib\jndi.jar;%MQ_JAVA_INSTALL_PATH%\java\lib
\fscontext.jar;%MQ_JAVA_INSTALL_PATH%\java\lib\providerutil.jar;%MQ_
JAVA_INSTALL_PATH%\java\lib\connector.jar;%WLS_MQ_JAVA_INSTALL_PATH%
\lib\wlsmqseries.jar
set
PATH=%PATH%;%MQ_JAVA_INSTALL_PATH%\bin;%MQ_JAVA_INSTALL_PATH%\java\b
in;%MQ_JAVA_INSTALL_PATH%\java\lib
```

   Invoke `setEnv.cmd`

5. Start the Queue Manager

```
strmqm QM_hostname
```

6. Run the **runmqsc** utility to define a sample channel called JAVA.CHANNEL

```
DEF CHL('JAVA.CHANNEL') CHLTYPE(SVRCONN)TRPTYPE(TCP) MCAUSER(' ')
DESCR('Sample')
END
```

7. Start the listener process

```
start runmqlsr -t tcp -p 1414 -m QM_hostname
```

8. Test the installation:

```
java MQIVP
```

   Output follows:

```
MQSeries for Java Installation Verification Program
5639-B43 (C) Copyright IBM Corp. 1997, 1998. All Rights Reserved.
===========================================================

Please enter the type of connection (MQSeries or VisiBroker)  : (MQSeries)
Please enter the IP address of the MQSeries server            : localhost
Please enter the port to connect to                           : (1414)
Please enter the server connection channel name               : JAVA.CHANNEL
Please enter the queue manager name                           :
Success: Connected to queue manager.
Success: Opened SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Put a message to SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Got a message from SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Closed SYSTEM.DEFAULT.LOCAL.QUEUE
Success: Disconnected from queue manager


Tests complete -
SUCCESS: This MQSeries Transport is functioning correctly.
Press Enter to continue...
```

9. To utilize Topics, start the Pub/Sub Broker

   ```
   strmqbrk -m QM_hostname
   ```

   where *QM_hostname* is the name of the queue manager created at installation time. Verify that the broker is active by running:

   ```
   dspmqbrk -m QM_hostname
   ```

   Then set up the necessary system queues once by running the following script in the `%MQ_JAVA_INSTALL_PATH%\java\bin` directory.

   ```
   runmqsc QM_hostname < MQJMS_PSQ.mqsc
   ```

3 Administering MQSeries JMS Objects

To help achieve application portability, JMS connection factories, queues and topic objects may be defined prior to application runtime and stored using a JNDI provider. An application can then retrieve the JMS provider-specific objects from the JNDI context at runtime and manipulate them using standard JMS interfaces. The MQSeries JMS distribution provides a command line utility, `JMSAdmin`, that allows for the creation, configuration and persisting of MQSeries JMS objects with various JNDI providers.

In order to use MQSeries as a transactional resource with WLS, an application must define and manipulate WLS MQSeries XA connection factories. These WLS MQSeries XA connection factories are based on the MQSeries XA connection factories and perform operations specific to the WLS transaction manager. This section describes how to administer MQSeries and WLS MQSeries JMS objects. Section 4 discusses how the WLS MQSeries objects may be used in an application.

3.1 JMSAdmin

The following steps may be taken to configure MQSeries connection factories, queues and topics using a third party JNDI provider. These procedures utilize the `JMSAdmin` command line utility provided with the MQSeries Support Pac *MA88: MQSeries classes for Java and MQSeries classes for Java Message Service*.

1. Modify the `JMSAdmin` configuration file, located at `%MQ_JAVA_INSTALL_PATH%\Java\bin\JMSAdmin.config`, as appropriate for the desired JNDI provider. The following is a sample configuration file for use with Sun's file system implementation (with comments removed for compactness).

   ```
   INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
   PROVIDER_URL=file:/C:/MQSeries/JNDI
   SECURITY_AUTHENTICATION=none
   ```

2. Invoke `JMSAdmin` from the `%MQ_JAVA_INSTALL_PATH%\Java\bin` directory and configure a `XAQueueConnectionFactory` object and a `Queue` object for use by the sample application. Queue definitions should be defined as persistent to avoid losing messages in the event of failure.

   ```
   JMSAdmin
   5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
   Starting MQSeries classes for Java(tm) Message Service Administration

   InitCtx> DEFINE XAQCF(mqXAQCF)

   InitCtx> DEFINE Q(mqQ) QUEUE(default) PERSISTENCE(PERS)

   InitCtx> END

   Stopping MQSeries classes for Java(tm) Message Service Administration
   ```

In order to use MQSeries as a transactional resource manager with the WLS TM, connection factories must be defined as XA connection factories. If a non-XA connection factory is employed by an application, updates to destinations using a `Session` obtained from the non-XA connection factory will not participate in distributed transactions. In the event of failure, data corruption could occur.

3.2 JNDIMapper

After the MQSeries JMS objects have been configured and stored with the JNDI provider, the WLS MQSeries connection factories must be defined. These objects encompass MQSeries XA connection factories and perform the necessary WLS TM registration and dynamic enlistment operations. The helper class `weblogic.jms.foreign.mqseries.JNDIMapper` retrieves MQSeries XA connection factories from a third party JNDI context, creates WLS connection factories based on these objects and stores them in the specified WLS JNDI context. The class has two methods that copy objects from one provider context to the other.

```
public void map(String aMQName, String aWLName);
public void map(String aMQName, String aWLName, String aResourceName);
```

These `map()` methods store the object found at the specified foreign context in the local JNDI context. If the object being stored is an MQSeries `XAConnectionFactory` then the corresponding WLS factory is created and stored in the local context. Otherwise, a copy of the object from the foreign context is stored. If the method with the resource name parameter is invoked and the mapped object is a WLS connection factory, then the object's resource name attribute will be set accordingly.

The following `JNDIMapper` example shows how the helper class can be used in a WLS startup class to create a WLS connection factory in the server's local JNDI context. An MQSeries `Queue` definition is also stored in the local context.

```
// from MQSeriesHelperImpl.java
public static void main(String[] argv) throws Exception
{
  // advertise in WLS JNDI
  MQSeriesHelperImpl impl = new MQSeriesHelperImpl();
  Context ctx = new InitialContext();
  ctx.bind("MQSeriesHelper", impl);
  ctx.close();
  System.out.println("*** MQSeriesHelper bound ***");

  // map MQSeries objects
  JNDIMapper mapper = new JNDIMapper(
    "com.sun.jndi.fscontext.RefFSContextFactory",
    "file://localhost/c:/mqseries/JNDI");
  mapper.map("mqXAQCF", "wlsmqXAQCF", "MQSeries");
  mapper.map("mqQ", "mqQ");
}
```

In the main method of the WLS startup class `MQSeriesHelperImpl`, the first section advertises an instance of itself in the WLS JNDI context. The second section of the method creates a `JNDIMapper` instance with the initial context factory and provider URL of the JNDI provider that was used by `JMSAdmin` to store the MQSeries JMS objects. The first `map()` method call creates a WLS connection factory, based on the MQSeries XA connection factory found in the foreign JNDI context, with the resource name attribute of "MQSeries". The second call to `map()` simply copies the MQSeries queue definition from the foreign provider context to the local JNDI context.

3.2.1 Resource Naming

The `JNDIMapper.map(mqName, wlsName, resourceName)` method associates a resource name with the WLS connection factory that is created and stored in the local JNDI context. When the connection factory is used in an application, MQSeries `XAResource` objects, which are obtained from `Session` objects, are registered with the WLS TM under the specified name. In the above example, the MQSeries connection factory stored under the context "mqXAQCF" is used to create the WLS connection factory "wlsmqXAQCF" with a resource name attribute of “MQSeries”. All `QueueSession` objects created from the connection factory will be registered with the WLS TM under the “MQSeries” resource name. If a WLS connection factory is created without an explicit resource name, then a default name is generated and used for registration. This name is comprised of the MQSeries Queue Manager name combined with the WLS domain name and server name. For example, *QM_hostname@domain+server*.

Note that the WLS TM uses resource registration names to determine transaction branches. If two resources are registered with the TM under the same name, then they will be treated as the same logical branch of a transaction. In such a scenario, one of the resources will not participate in the commit protocol. Care should be taken when assigning resource names to avoid resource name collisions.

4 WLS MQSeries Classes

The WLS MQSeries classes implement the standard JMS interfaces. Instances of these classes intercept application method invocations that update JMS destinations; such as send, publish and receive operations; and perform dynamic enlistment with the WLS TM prior to executing the actual MQSeries JMS operation.

The typical usage pattern is to define a WLS MQSeries `XAConnectionFactory` object that wraps an MQSeries `XAConnectionFactory` object. This WLS-specific connection factory might then be stored in a WLS JNDI context for use at application runtime. An application would retrieve the connection factory and use it to create the appropriate connection, session, producer and consumer objects. Each of these objects derived from the connection factory will be a WLS MQSeries object that delegates method invocations to the corresponding MQSeries implementation object.

4.1 Programming

This section describes how the MQSeries classes may be used by an application to perform transactional updates to MQSeries destinations using standard JMS and JTA interfaces. The following example makes use of an MQSeries queue destination. Note that exception handling is omitted for readability.

4.1.1 Retrieving the MQSeries Connection Factory

As discussed in the *Administering MQSeries JMS* Objects section, a WLS MQSeries connection factory may be stored prior to application runtime in a local WLS JNDI context. An application then retrieves the connection factory at runtime, as shown below.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, WLS_ICF);
env.put(Context.PROVIDER_URL, WLS_PROVIDER_URL);
ctx = new InitialContext(env);
QueueConnectionFactory mqQCF =
  QueueConnectionFactory)ctx.lookup("wlsmqXAQCF");
```

In the above example, the `WLS_ICF` and `WLS_PROVIDER_URL` variables contain the WLS JNDI initial context factory class name and provider URL, respectively. Alternatively, the `weblogic.jndi.Environment` class may be used to obtain the local context, as follows.

```
Environment wlenv = new Environment();
ctx = wlenv.getInitialContext();
QueueConnectionFactory mqQCF =
  QueueConnectionFactory)ctx.lookup("wlsmqXAQCF");
```

Note that the JMS specification prescribes the use of "XA" classes methods to integrate with a JTS/JTA provider. The WLS MQSeries classes do not require the use of these XA classes and methods in order for MQSeries JMS to participate in distributed transactions with the WLS TM. If a WLS connection factory is employed, then all MQSeries JMS operations will participate in global transactions. In fact, using a WLS XA connection factory requires the use of global transactions. If an MQSeries destination is accessed outside the scope of a transaction then the MQSeries JMS exception ***JMSException: MQJMS2007*** will be thrown. Refer to the *Troubleshooting* section for more information.

4.1.2 Retrieving the MQSeries Queue

The following code sample shows retrieving the predefined MQSeries Queue definition from the local JNDI context of the server.

```
Queue mqQueue = (Queue)ctx.lookup("mqQ");
```

4.1.3 Create Producer

The connection factory that was obtained from JNDI may be used to create the connection, session and producer objects required to send a message to the MQSeries queue.

```
QueueConnection mqConn = mqQCF.createQueueConnection();
mqConn.start();
QueueSession mqSession = mqConn.createQueueSession(true,
  Session.AUTO_ACKNOWLEDGE);
QueueSender mqSender = mqSession.createSender(mqQueue);
```

The `createQueueSession()` method is shown here being called with the values of `true` for the `transacted` parameter and `Session.AUTO_ACKNOWLEDGE` for the `acknowledgeMode` parameter. These parameters are essentially ignored as internally a `XASession` object is created from which the `Session` object is created that is then returned to the caller. Alternately, the JMS XA methods may be used to create the connection, session and producer objects as shown below. Note that if the `getXAQueueSession()` method is called on the `XAQueueConnection` object, then an additional call to `getQueueSession()` on the `XAQueueSession` object is required. The two approaches are essentially equivalent with regard to the objects that are created internally and from the point of view of the WLS TM.

```
XAQueueConnection mqXAConn = mqQCF.createXAQueueConnection();
mqXAConn.start();
XAQueueSession mqXASession = mqConn.createXAQueueSession();
QueueSession mqSession = mqXASession.getQueueSession();
QueueSender mqSender = mqSession.createSender(mqQueue);
```

4.1.4 Sending a Message in a Transaction

Once the session and producer objects have been created, messages may be sent to the MQSeries queue as part of a global transaction. The following example shows how to retrieve a UserTransaction object, start a WLS transaction, send a message to an MQSeries queue, and commit the transaction.

```
UserTransaction ut = (UserTransaction) ctx.lookup(
  "javax/transaction/UserTransaction");
ut.begin();
mqSender.send(msg);
ut.commit();
```

This example shows a single resource being accessed within the scope of a transaction. In this situation the WLS TM will perform a one-phase commit optimization with MQSeries.

4.1.5 Threading Issues

According to the JMS specification, Session objects are not intended for concurrent access across multiple threads. The WLS MQSeries classes have the restriction of single-threaded session access. If two threads infected by two separate transactions access the same session object, a `javax.transaction.xa.XAException` may result. Each server execute thread should obtain its own session object before performing any MQSeries JMS operations in a transaction.

4.1.6 Asynchronous Consumers

Asynchronous message delivery cannot participate in WLS global transactions. To use asynchronous message delivery with WLS, a non-XA connection factory must be used to create the session from which the consumer is created. Also, the MQSeries JMS implementation requires that the `MessageListener` be assigned to the `MessageConsumer` prior to starting the connection.

4.2 Example Application

This section describes how to configure and run the provided example application, listed at the end of this document. The example consists of a WLS startup class, `MQSeriesHelper`, and a client application that makes RMI calls to a startup class instance running in WLS. The RMI methods perform transactional operations with MQSeries.

4.2.1 Building

The example may be built by compiling the three Java files; MQSeriesHelper.java, MQSeriesHelperImpl.java, and MQClient.java; and by running `weblogic.rmic` on the MQSeriesHelperImpl class.

```
javac MQSeriesHelper.java MQSeriesHelperImpl.java MQClient.java
java weblogic.rmic -nomanglednames MQSeriesHelperImpl
```

Copy the resulting ".class" files to a directory that is in the CLASSPATH environment variable.

4.2.2 Configuring and Starting WebLogic

A startup class definition must be added to the WLS configuration file. Add the following entry to the config.xml file for the domain. Modify the *Targets* value to be the actual name of the server being booted.

```
   <StartupClass
     Name="MQSeriesHelper"
     Targets="serverName"
     ClassName="MQSeriesHelperImpl"/>
```

Add a WLS JMS Server definition to the configuration. Modify the Targets value to be the actual name of the server being booted. Create a directory under the server directory named *myfilestore*.

```
<JMSServer Name="TestJMSServer" Targets="serverName"
  Store="FileStore">
  <JMSQueue Name="WLSQueue" JNDIName="WLSQueue"/>
</JMSServer>

<JMSConnectionFactory  Name="WLSCF" JNDIName="WLSCF"
  Targets="server1"  UserTransactionsEnabled="true"
  XAConnectionFactoryEnabled="true"/>

 <JMSFileStore Name="FileStore" Directory="myfilestore"
   JMSServer="TestJMSServer"/>
```

The following environment properties may be defined at server startup to control how the MQSeriesHelper object behaves with respect to JNDI providers, context names, etc. The default values correspond to the configuration described in this document.

`wlsmqs.mqs.icf`	The initial context factory for the MQSeries JNDI provider. Default: `com.sun.jndi.fscontext.RefFSContextFactory`
`wlsmqs.mqs.providerurl`	The provider URL for the MQSeries JNDI provider. Default: `file://localhost/c:/mqseries/JNDI`
`wlsmqs.wls.icf`	The initial context factory for the WLS JNDI provider. Default: `weblogic.jndi.WLInitialContextFactory`
`wlsmqs.wls.providerurl`	The provider URL for the WLS JNDI provider. Default: `t3://localhost:7001`
`wlsmqs.mqs.mqxaqcf`	The context name of the MQSeries XAQueueConnectionFactory stored in the MQSeries JNDI provider. Default: `mqXAQCF`
`wlsmqs.mqs.mqqueue`	The context name of the MQSeries Queue stored in the MQSeries JNDI provider. Default: `mqQ`
`wlsmqs.wls.mqxaqcf`	The context name of the WLS XAQueueConnectionFactory stored in the WLS JNDI

	context. Default: `wlsmqXAQCF`
`wlsmqs.wls.resourcename`	The resource name to be assigned to the WLS XAQueueConnectionFactory. Default: `MQSeries`
`wlsmqs.wls.mqqueue`	The context name of the MQSeries Queue that is stored in the local WLS JNDI context. Default: `mqQ`
`wlsmqs.wls.qcf`	The context name of the WLS JMS QueueConnectionFactory stored in the local context. Default: `WLSCF`
`wlsmqs.wls.queue`	The context name of the WLS JMS queue stored in the local JNDI context. Default: `WLSQueue`

In order to modify the default configuration settings for the MQSeriesHelper object, define the appropriate properties in the script used to boot WLS. For example, to change the WLS provider URL to t3://localhost:8801, add the following `java` command-line.

```
java –Dwlsmqs.wls.providerurl=t3://localhost:8801 ... weblogic.Server
```

After changing the server configuration and updating the WLS startup script, start the server.

4.2.3 Invoking the Client

After the server has booted, invoke the `MQClient` client. The client makes RMI calls to the `MQSeriesHelper` startup class to perform transactional MQSeries operations. The application takes two arguments, the WLS server URL and a string message.

```
java MQClient t3://localhost:7001 "MQSeries test message"
```

The client will invoke the MQSeriesHelper object to enqueue a WLS JMS message within a transaction. Another remote method is called to retrieve the message from the WLS JMS queue and send it to the MQSeries queue within a transaction. Finally, the client invokes another method to transactionally receive the message from the MQSeries queue.

Expected client output:

```
sending:  "MQSeries test message"
received: "MQSeries test message"
```

Expected server output:

```
Bridged: "MQSeries test message"
```

4.2.4 Verify Transaction Participation

In order to determine if the MQSeries queue operations performed by running the example actually participated in global transactions, check the statistics of the registered resources. This may be performed by using the WLS Console and viewing the *[domain] > Servers > [server] > Monitoring > JTA > Transaction by Resource* pane, or by running the `weblogic.Admin` command-line utility. The utility may be invoked as follows, substituting configuration-specific values for the *url, username* and *password* arguments:

```
   java weblogic.Admin –url t3://localhost:7001 \
   –username system –password gumby1234 \
   GET –pretty –type TransactionResourceRuntime
```

The output of this command displays the `TransactionResourceRuntime` MBeans that represent registered resources with the WLS TM. The attributes of these MBeans are primarily statistics that indicate the number of transactions processed, and how many were committed, rolled back, etc. After running the example application, the output should look something like the following.

```
---------------------------
MBeanName:
"qa:Location=server1,Name=JTAResourceRuntime_MQSeries,ServerRuntime=server
1,Type=TransactionResourceRuntime"
        CachingDisabled: true
        Name: JTAResourceRuntime_MQSeries
        ObjectName: JTAResourceRuntime_MQSeries
        Registered: false
        ResourceName: MQSeries
        TransactionCommittedTotalCount: 2
        TransactionHeuristicCommitTotalCount: 0
        TransactionHeuristicHazardTotalCount: 0
        TransactionHeuristicMixedTotalCount: 0
        TransactionHeuristicRollbackTotalCount: 0
        TransactionHeuristicsTotalCount: 0
        TransactionRolledBackTotalCount: 0
        TransactionTotalCount: 2
        Type: TransactionResourceRuntime
---------------------------
MBeanName:
"qa:Location=server1,Name=JTAResourceRuntime_JMS_FileStore,ServerRuntime=s
erver1,Type=TransactionResourceRuntime"
        CachingDisabled: true
        Name: JTAResourceRuntime_JMS_FileStore
        ObjectName: JTAResourceRuntime_JMS_FileStore
        Registered: false
        ResourceName: JMS_FileStore
        TransactionCommittedTotalCount: 2
        TransactionHeuristicCommitTotalCount: 0
        TransactionHeuristicHazardTotalCount: 0
        TransactionHeuristicMixedTotalCount: 0
        TransactionHeuristicRollbackTotalCount: 0
        TransactionHeuristicsTotalCount: 0
        TransactionRolledBackTotalCount: 0
        TransactionTotalCount: 2
        Type: TransactionResourceRuntime
```

There are two MBeans displayed in the above output; one for the MQSeries resource and one for the WLS JMS file store. Each MBean should have the value of two for both the `TransactionTotalCount` and `TransactionCommittedTotalCount` attributes. These values will increase by two each time the example is executed. The WLS JMS resource participates in two transactions, a one-phase transaction to enqueue the initial message and a two-phase transaction to dequeue as part of the transfer to the MQSeries queue. The MQSeries JMS resource also participates in two transactions, a two-phase transaction to transfer a message from the WLS JMS queue to the MQSeries queue and a one-phase transaction to dequeue from the MQSeries queue.

5 Restrictions and Limitations

The following limitations and restrictions apply when utilizing MQSeries JMS with WLS.

- The MQSeries JMS integration with WLS is only supported with the IBM MQSeries JMS driver. The MQSeries Base Java driver is not supported.

- The MQSeries JMS integration with WLS was tested using the "bindings" connection mode only. The MQSeries JMS driver does not provide support for distributed transactions in the "client" connection mode. For a discussion of MQSeries driver connection modes, refer to [2], *Part 1, Chapter 1, Connection options.*

- MQSeries destinations must be accessed from applications components running in WLS in order for updates to participate in distributed transactions. Server components include such constructs as startup classes, RMI objects, EJBs, servlets, etc. This restriction is due to the WLS requirement that resource managers only be registered on a server instance. Client applications may update MQSeries destinations outside of a global transaction by using the standard MQSeries connection factories.

- Message Driven Beans cannot be invoked from an MQSeries destination as part of a WLS distributed transaction. Refer to [6], "Can you use a foreign JMS provider to drive an MDB transactionally?" for more information.

- Multiple sessions derived from the same MQSeries XA connection factory should not be used in the same transaction. If multiple sessions to the same MQSeries Queue Manager must be used, define separate XA connection factories and assign unique resource name attributes.

- Consideration should be given to the following MQSeries restrictions from "MQSeries System Administration, Part 1, Chapter 14, External syncpoint coordination":
  - Only one queue manager at a time may participate in a transaction coordinated by an instance of an external syncpoint coordinator: the syncpoint coordinator is effectively connected to the queue manager, and is therefore subject to the rule that only one connection at a time is supported.

  - A queue manager whose resource updates are coordinated by an external syncpoint coordinator must be started before the external syncpoint coordinator starts. Similarly, the syncpoint coordinator must be ended before the queue manager is ended.

- If you are using an external syncpoint coordinator that terminates abnormally, you should stop and restart your queue manager ***before*** restarting the syncpoint coordinator to ensure that any messaging operations uncommitted at time of the failure are properly resolved.

6 Troubleshooting

This section describes common problems when using MQSeries JMS with WLS.

1. Producer/Consumer operation results in JMSException: *javax.jms.JMSException: MQJMS2007: failed to send message to MQ queue, Linked exception: com.ibm.mq.MQException: Completion Code 2, Reason 2072 (MQRC_SYNCPOINT_NOT_AVAILABLE)*
   - This indicates that a MQSeries JMS operation was performed outside of a global transaction while using an XASession. All XASession producer/consumer operations must be invoked within a global transaction.

7 Glossary

Coordinator	The entity that drives the two-phase commit protocol.
Distributed Transaction	A global transaction involving two or more resources across one or more server instances.
Dynamic Enlistment	Transaction enlistment of a resource that happens only when the resource is accessed.
Global Transaction	A transaction that is managed by a transaction manager and is associated with a thread of control. It may encompass one or more participating resources that are infected with the transaction when they are accessed by an application component. The transaction manager coordinates the commit processing using the two-phase commit protocol.
One-phase Commit	An optimization of the two-phase commit protocol whereby the prepare phase is skipped when there is only one participating resource.
Resource Manager	An entity that manages persistent data, such as a queuing system or database, which can participate in two-phase commit transactions.
Resource Registration	The mechanism whereby a resource manager is registered with a transaction manager.
Static Enlistment	Transaction enlistment of a resource that occurs whenever a transaction is started or becomes active in a thread. This type of enlistment is typically only used with resources that do not support dynamic enlistment.
Transaction Enlistment	The mechanism whereby a resource manager is associated with the transaction context of the caller.
Transaction Manager	Process that manages the transaction lifecycle and subordinate resources, and coordinates the two-phase

	commit protocol. Also referred to in the MQSeries documentation as an external Syncpoint Coordinator.
Two-phase Commit	A protocol for ensuring that updates to multiple resource managers, within the scope of a transaction, are permanently stored or abandoned as a unit. A coordinator instructs each transaction participant to prepare. If any participant responds negatively, the transaction will be aborted (rolled back) and each participant will be notified to revert its changes. If all participants agree, the coordinator will write a commit record to permanent store. At this point the transaction is committed. The coordinator is then responsible for informing each participating resource of the outcome so that its changes will be made durable.

8 References and Related Documents

[1] Using foreign JMS providers with WLS, BEA Developer Center, http://developer.bea.com/docs/jmsproviders.jsp

[2] Using Java, MQSeries Manuals, IBM Corporation, Document Number SC34-5456-05

[3] MQSeries System Administration, Second edition (March 1999), IBM Corporation

[4] Java Transaction API (JTA) Specification (*http://java.sun.com/products/jta*)

[5] X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3

[6] WebLogic Server 6.1 JMS FAQ, http://e-docs.bea.com/wls/docs61/faq/jms.html

9 Example Source Code

MQSeriesHelper.java:

```
import weblogic.rmi.Remote;
import weblogic.rmi.RemoteException;

/**
 * Remote interface for the MQSeriesHelper startup-class/RMI object.
 */
public interface MQSeriesHelper extends Remote
{
  /**
   * Sends a TextMessage to a WLS queue transactionally.
   */
  void sendWLSMessage(String msg) throws RemoteException;

  /**
   * For the number of messages specified, starts a transaction
   * dequeues from WLS queue, enqueues message to MQSeries
   * queue and commits the transaction.
   */
  void bridgeWLS2MQS(int numMsgs) throws RemoteException;

  /**
   * Receives a message from an MQSeries queue and returns
   * the text.
   */
  String receiveMQMessage() throws RemoteException;
}
```

MQSeriesHelperImpl.java:

```
import java.util.Hashtable;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.QueueReceiver;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.XAQueueConnection;
import javax.jms.XAQueueConnectionFactory;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.transaction.UserTransaction;
import weblogic.jms.foreign.mqseries.JNDIMapper;
import weblogic.jndi.Environment;
import weblogic.rmi.RemoteException;
import weblogic.transaction.TxHelper;

/**
 * Implementation of the MQSeriesHelper startup-class/remote interface.
 * When the startup class is invoked during server boot, an instance
 * will be bound in the local JNDI context and MQSeries objects will
 * be copied from the MQSeries JNDI provider to the local context.
 * As part of the mapping of JMS objects a WLS MQSeries
 * XAQueueConnectionFactory will be created with the specified
 * resource name attribute.
 */
public class MQSeriesHelperImpl implements MQSeriesHelper {

  private static String mqICF;
  private static String mqProviderURL;
  private static String wlICF;
  private static String wlProviderURL;
  private static String mqMQXAQCFName;
  private static String mqMQQueueName;
  private static String wlMQXAQCFName;
  private static String wlMQQueueName;
  private static String wlQCFName;
  private static String wlQueueName;
  private static String wlResourceName;

  /**
   * Invoked during server boot.  Registers instance in local
   * WLS JNDI context to serve RMI requests.  Reads system properties
   * if any to override default settings.  Binds MQSeries object in
   * local JNDI context.
   */
  public static void main(String[] argv) throws Exception
  {
    // advertise in WLS JNDI
    MQSeriesHelperImpl impl = new MQSeriesHelperImpl();
    Context ctx = new InitialContext();
    ctx.bind("MQSeriesHelper", impl);
    ctx.close();
    System.out.println("*** MQSeriesHelper bound ***");

    getSystemProperties();

    // map MQSeries objects
    JNDIMapper mapper = new JNDIMapper(mqICF, mqProviderURL);
    mapper.map(mqMQXAQCFName, wlMQXAQCFName, wlResourceName);
    mapper.map(mqMQQueueName, wlMQQueueName);
  }

  /**
   * @see MQSeriesHelper#sendWLSMessage
   */
  public void sendWLSMessage(String msgText) throws RemoteException
  {
    JMSObject wljms = null;
    QueueSession session = null;
    QueueSender sender = null;

    try {
      wljms = new JMSObject(wlICF, wlProviderURL, wlQCFName,
                            wlQueueName);

      session = wljms.getSession();
      sender = wljms.getSender();

      TextMessage msg = session.createTextMessage();
      msg.setText(msgText);

      UserTransaction ut = getUserTransaction();

      ut.begin();
      sender.send(msg);
      ut.commit(); // 1PC

    } catch (Exception e) {
      System.err.println(e.toString());
      e.printStackTrace();
      throw new RemoteException(e.toString(), e);
    } finally {
      wljms.cleanup();
    }
  }

  /**
   * @see MQSeriesHelper#bridgeWLS2MQS
   */
  public void bridgeWLS2MQS(int numMsgs) throws RemoteException
  {
    JMSObject wljms = null;
    QueueReceiver wlReceiver = null;
    JMSObject mqjms = null;
    QueueSender mqSender = null;
    UserTransaction ut = getUserTransaction();

    try {
      wljms = new JMSObject(wlICF, wlProviderURL, wlQCFName,
                            wlQueueName);
      wlReceiver = wljms.getReceiver();

      mqjms = new JMSObject(wlICF, wlProviderURL, wlMQXAQCFName,
                            wlMQQueueName);
      mqSender = mqjms.getSender();

      for (int i=0; i<numMsgs; i++) {
        ut.begin();
        Message msg = wlReceiver.receive();
        mqSender.send(msg);
        ut.commit(); // 2PC
        if (msg instanceof TextMessage) {
          System.out.println("bridged: \"" + ((TextMessage)msg).getText() +
                             "\"");
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
      throw new RemoteException(e.toString(), e);
    } finally {
      if (wljms != null) wljms.cleanup();
      if (mqjms != null) mqjms.cleanup();
    }
  }

  /**
   * @see MQSeriesHelper#receiveMQMessage
   */
  public String receiveMQMessage() throws RemoteException
  {
    JMSObject mqjms = null;
    QueueSession session = null;
    QueueReceiver receiver = null;

    try {
      mqjms = new JMSObject(wlICF, wlProviderURL, wlMQXAQCFName,
                            wlMQQueueName);
      session = mqjms.getSession();
      receiver = mqjms.getReceiver();

      UserTransaction ut = getUserTransaction();

      ut.begin();
      Message msg = receiver.receive();
      ut.commit(); // 1PC

      if (msg == null || !(msg instanceof TextMessage)) {
        return "[not a TextMessage]";
      }

      TextMessage textMsg = (TextMessage)msg;
      return textMsg.getText();
    } catch (Exception e) {
      e.printStackTrace();
      throw new RemoteException(e.toString(), e);
    } finally {
      mqjms.cleanup();
    }
  }

  private static void getSystemProperties()
  {
    mqICF = System.getProperty(
      "wlsmqs.mqs.icf", "com.sun.jndi.fscontext.RefFSContextFactory");
    mqProviderURL = System.getProperty(
      "wlsmqs.mqs.providerurl", "file://localhost/c:/mqseries/JNDI");
    wlICF = System.getProperty(
      "wlsmqs.wls.icf", "weblogic.jndi.WLInitialContextFactory");
    wlProviderURL = System.getProperty(
      "wlsmqs.wls.providerurl", "t3://localhost:7001");

    mqMQXAQCFName = System.getProperty("wlsmqs.mqs.mqxaqcf", "mqXAQCF");
    mqMQQueueName = System.getProperty("wlsmqs.mqs.mqqueue", "mqQ");

    wlMQXAQCFName = System.getProperty("wlsmqs.wls.mqxaqcf", "wlsmqXAQCF");
    wlResourceName = System.getProperty("wlsmqs.wls.resourcename",
      "MQSeries");
    wlMQQueueName = System.getProperty("wlsmqs.wls.mqqueue", "mqQ");

    wlQCFName = System.getProperty(
      "wlsmqs.wls.qcf", "WLSCF");
    wlQueueName = System.getProperty(
      "wlsmqs.wls.queue", "WLSQueue");
  }

  private UserTransaction getUserTransaction()
  {
    // could also do JNDI lookup of javax.transaction.UserTransaction
    // in local WLS context
    return TxHelper.getUserTransaction();
  }

  class JMSObject
  {
    private InitialContext ctx;
    private Queue queue;
    private XAQueueConnection connection;
    private XAQueueConnectionFactory factory;
    private QueueReceiver queueReceiver;
    private QueueSender queueSender;
    private QueueSession session;

    JMSObject(String icf, String url, String qcf, String qname)
      throws Exception
    {
      // Get the initial context
      Hashtable env = new Hashtable();
      env.put(Context.INITIAL_CONTEXT_FACTORY, icf);
      env.put(Context.PROVIDER_URL, url);
      env.put(Context.REFERRAL, "throw");
      ctx = new InitialContext(env);
      factory = (XAQueueConnectionFactory)ctx.lookup(qcf);

      connection = factory.createXAQueueConnection();
      session = connection.createXAQueueSession().getQueueSession();
      queue = (Queue)ctx.lookup(qname);
      connection.start();
      queueSender = session.createSender(queue);
      queueReceiver = session.createReceiver(queue);
    }

    QueueSession getSession()
    {
      return session;
    }

    QueueSender getSender()
    {
      return queueSender;
    }

    QueueReceiver getReceiver()
    {
      return queueReceiver;
    }

    void cleanup()
    {
      try {
        queueSender.close();
        queueReceiver.close();
        session.close();
        connection.close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
  }
}
```

MQClient.java:

```java
import weblogic.jndi.Environment;

/**
 * A simple RMI client that invokes the MQSeriesHelper startup-class/
 * RMI object to transactionally send and receive TextMessages to and
 * from WLS and MQSeries queues using standard JMS interfaces.
 */
public class MQClient
{
  /**
   * After obtaining a stub to the MQSeriesHelper RMI object in the
   * specified WLS server, three remote methods are invoked.  The first
   * method enqueues a TextMessage, based on the provided string, to a
   * WLS queue in a one-phase transaction.  The second method, within
   * the scope of a transaction, receives the message from the WLS queue
   * and sends it to an MQSeries queue (two-phase commit).  The third
   * method call receives the message from the MQSeries queue in a
   * one-phase transaction.
   * <p>
   * Usage: java MQClient serverURL messageText
   */
  public static void main(String[] argv) throws Exception
  {
    String serverURL = null;
    String messageText = null;
    if (argv.length != 2) {
      System.out.println(
        "usage: java MQClient <serverURL> <messageText>");
      System.exit(1);
    }
    serverURL = argv[0];
    messageText = argv[1];

    // lookup RMI object
    Environment env = new Environment();
    env.setProviderUrl(serverURL);
    Context ctx = env.getInitialContext();
    MQSeriesHelper helper = (MQSeriesHelper)
      ctx.lookup("MQSeriesHelper");

    System.out.println("sending:  \"" + messageText + "\"");

    // enqueue message string to WLS queue
    helper.sendWLSMessage(messageText);

    // dequeue from WLS queue and enqueue to MQSeries queue
    helper.bridgeWLS2MQS(1);

    // dequeue message from MQSeries queue
    System.out.println("received: \"" + helper.receiveMQMessage() +
                       "\"");
  }
}
```