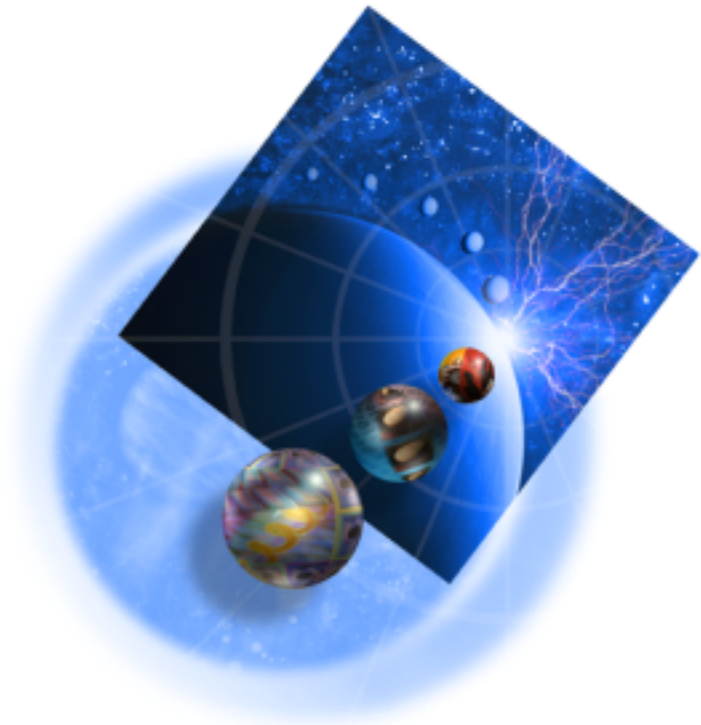


IBM WebSphere Application Server Advanced Edition 3.5

WebSphere JMS/JTA support for MQSeries Overview



Joanna Hodgson
Nathan Kirkham
Dan Murphy
Annette Niederheiser

WebSphere JMS/JTA Support for MQSeries Overview (May 2001)

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Director of Licensing
IBM Corporation
North Castle Drive
Amonk, NY 10504-1785
USA

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

(C) Copyright International Business Machines Corporation 2001. All rights reserved

Note to U.S. Government users -- Documentation related to restricted rights -- Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule contract with IBM Corp.

Summary

WebSphere Application Server is a leading Java transaction server which uses the EJB programming model to run transactional business logic. MQSeries is the leading messaging integration infrastructure for enterprises. Recently a number of companies have been developing application architectures that use the best features of both systems to provide robust, reliable web-based applications.

WebSphere provides the ability to interface transactional database resources to web-based applications to build highly available robust websites. The basic WebSphere model is based on synchronous access to relational databases and transaction systems, which must be highly available. However, the latest e-business applications need to reach out to many kinds of existing systems - for example ERP applications such as SAP/R3 and Bahn and bespoke systems running on many different platforms. Some of these backend systems may have long response times, or may be down for certain periods of time such as during batch processing. The *asynchronous* model offered by MQSeries, combined with the large range of platforms supported make it an excellent infrastructure to deal with such issues.

MQSeries offers a guarantee of delivery using "persistent" messaging. This capability ensures that the message will be delivered, even if key systems crash during processing. To take advantage of this feature, the messaging operations must be transactionally co-ordinated with the database resources controlled by WebSphere. For example, placing an order may involve updating the customer's web based account record in DB2, and sending an XML message to SAP/R3 over MQSeries. It is essential that these two activities are treated as a single unit of work.

WebSphere Application Server 3.5.3 and MQSeries 5.2 now offer this capability using the JMS/JTA interface. Although other application servers have offered such a capability before (including WebSphere Enterprise Edition), we believe this is the first time this has been offered to connect a Java application server to a leading, cross-platform messaging infrastructure using the J2EE APIs. This paper provides the detailed hands-on information about how to use this feature.

1. Introduction	Page 4
1.1 Support provided	Page 4
1.2 What is not covered by this paper	Page 5
2. Business scenarios	Page 5
2.1 Best effort Request / Reply	Page 5
2.1.1 <i>Application design considerations</i>	Page 6
2.2 Datagram : a resilient one way message	Page 8
2.2.1 <i>Application design considerations</i>	Page 8
2.3 Combination : resilient request / best effort reply	Page 9
2.3.1 <i>Application design considerations</i>	Page 9
2.4 Other permutations	Page 10
2.4.1 <i>Message consumers and listeners</i>	Page 10
2.4.2 <i>Message Driven Beans</i>	Page 10
2.4.3 <i>One approach for an interim solution</i>	Page 11
3. Configuring WebSphere to support transactional messages	Page 12
3.1 Install the required software	Page 12
3.2 Setup your system environment	Page 12
3.2.1 <i>Verify your setup so far</i>	Page 13
3.3 Setup JMSAdmin to use the WebSphere JNDI name service	Page 14
3.3.1 <i>Verify your JNDI setup</i>	Page 14
3.4 First Steps with JMS/JTA	Page 15
3.5 Connection factories	Page 16
3.6 Explanation of some JMSAdmin commands	Page 17
3.7 Setup the WebSphere environment	Page 18
3.8 Deploy and test Sample 1	Page 19
4. Datagram example	Page 20
4.1 JNDI configuration	Page 21
4.2 DB2 configuration	Page 21
4.2.1 <i>Setting up JDBC2</i>	Page 21
4.2.2 <i>Setting up a datasource</i>	Page 22
4.3 Transactional properties of EJBs in WebSphere	Page 24
4.4 Using the Datagram example	Page 24
5. Quality of service	Page 26
5.1 Connection Pooling	Page 26
5.2 Persistent vs Non-persistent messages	Page 27
5.3 Development environment	Page 27
6. Performance and Scaling	Page 28
6.1 One phase commit optimisation	Page 28
6.2 Impact of transactions	Page 29
6.3 Improving performance by caching MQ JMS objects	Page 29
6.4 Using non-persistent messages	Page 30
6.5 MQSeries Clustering	Page 31
7. Thanks	Page 31
Appendix A - Debugging in WebSphere	Page 32
A.1 WebSphere environment problems	Page 32
A.2 Tracing problems in WebSphere	Page 32
Appendix B - Datagram example : source code	Page 35

1. Introduction

This paper describes the support for JMS and MQSeries® from IBM® WebSphere® Application Server™ 3.5. The main focus is the transactional messaging support provided by WebSphere 3.5.3 Advanced Edition (fixpak 3 for WebSphere 3.5) and MQSeries 5.2. It shows, by example, how to configure a system to support transactional JMS applications talking to MQSeries and also discusses how to develop such applications.

Basic practical knowledge of DB2® and MQSeries, and a thorough knowledge of WebSphere administration is assumed for this document. A conceptual understanding of EJB transactions and knowledge of the JMS API would be a benefit.

This document was prepared using Windows® NT 4.0. However, most of the information is relevant to all supported platforms.

1.1 Support provided

It is possible to run JMS applications within WebSphere Application Server 3.5 that communicate with a messaging server, such as MQSeries. To do this you need to use the JMS API implementation specific to the vendor's messaging software you are using. For MQSeries, the JMS¹ implementation comes as a SupportPac, MA88 (MQSeries classes for Java™ and MQSeries Classes for JMS). This has been available since December 1999.

With the latest version of MA88 and WebSphere, transactional support for messaging has been added. To get this new functionality, you need :

- WebSphere Application Server 3.5 Advanced Edition **plus** fixpak 3
- MQSeries 5.2
- MQSeries SupportPac MA88 : MQSeries classes for Java and MQSeries Classes for JMS 5.2

Ensure you have the latest version² available from:

<http://www-4.ibm.com/software/ts/mqseries/txppacs/ma88.html>

For WebSphere applications, MQSeries can now be a resource manager in a transaction, in the same way that databases³ can. This means that WebSphere can be the transaction coordinator for two phase commit (2PC) transactions between two or more MQSeries and/or database resources.

This paper is provided in addition to the information in the WebSphere InfoCenter and the *Using Java* (SC34-5456-06) documentation available as a separate download from the MA88 web page : <http://www-4.ibm.com/software/ts/mqseries/txppacs/ma88.html>.

¹ JMS for MQSeries is available for AIX, HP_UX, NT, Solaris and Linux

² The first release of MA88 for MQSeries 5.2 had a dependency on Sun's J2EE connector implementation which meant you had to download a file, connector.jar, from their site. Subsequent versions of MA88 do not have this dependency.

³ For a list of databases supported by WebSphere see :

http://www-4.ibm.com/software/webservers/appserv/doc/latest/idx_aas.htm

1.2 What is not covered by this paper

This paper does not discuss JMS in relation to :

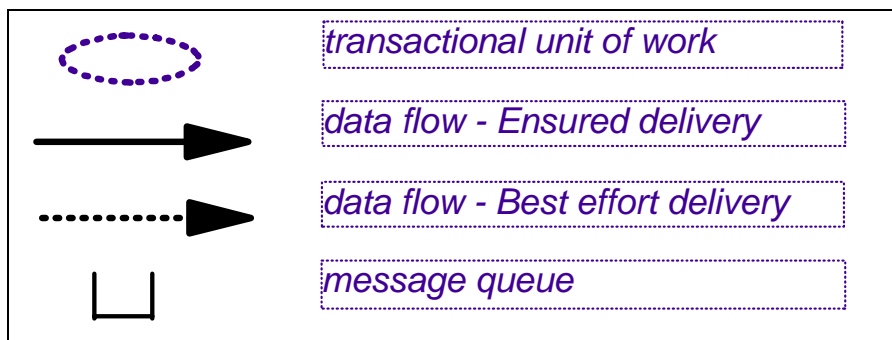
- Publish / Subscribe
- WebSphere scaling and failover
- Security

2. Business scenarios

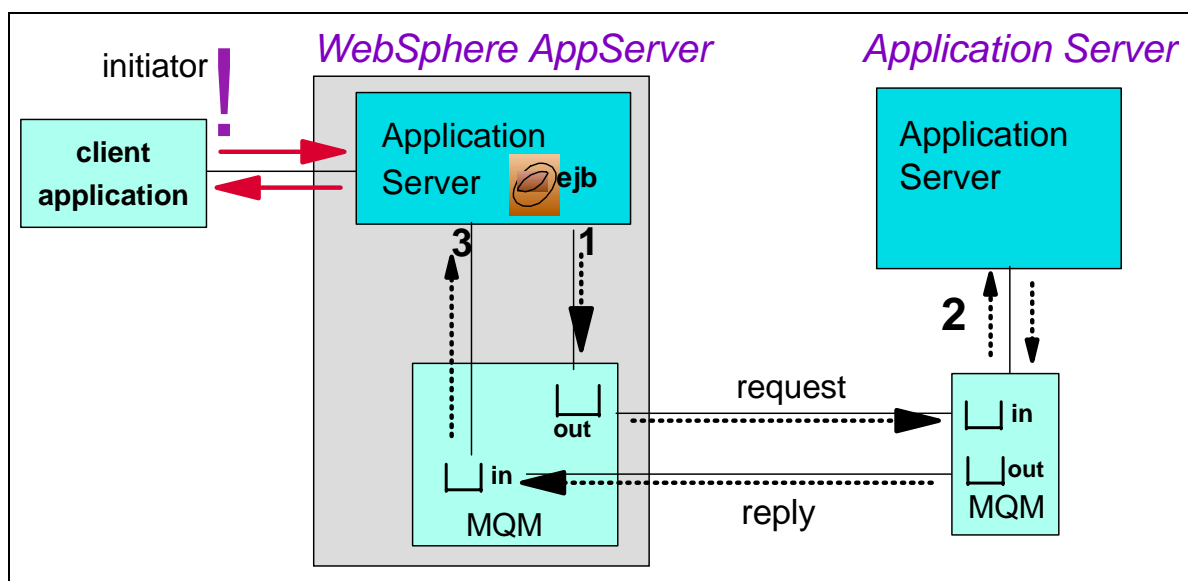
Here we describe three generic business scenarios. These assume that a client application (e.g. a web browser) is the initiator, that WebSphere handles the request from the client then sends a message over MQSeries to another application server (this may or may not be WebSphere).

In section 2.4 (*Other permutations*) we discuss other scenarios, including where the initiator is something other than a web browser.

The key to symbols used in the graphics in this section is :



2.1 Best effort Request / Reply



In this scenario the client application initiates a request for information, for example, a balance enquiry or an order status enquiry. The WebSphere application puts the message on its *out* queue and waits for a reply back to its *in* queue. When the remote application server receives the message it handles it and returns the information requested.

This is a 'best effort' scenario, so there is no guarantee that the message was sent, or that the client will receive a reply within a reasonable time. If the end user does not get a reply, they would be expected to re-submit their request.

In this simple request/reply scenarios there is no need for any work to be included in a transaction or to use persistent messages. If a transaction is not required then the resource managers can avoid a significant amount of overhead (for example, MQSeries logs persistent messages to a file system as part of its assured delivery service). See chapter 5 (*Quality of Service*) for a further discussion of persistent and non-persistent messages.

2.1.1 Application design considerations

MQSeries is an asynchronous communications mechanism; Java method calls including invocation of such as in EJB methods are synchronous in nature.

If your application design puts a message to a queue and then expects a reply care must be taken. You do not want to wait indefinitely on the queue for the reply message to arrive because you do not know how long that will be. Time-outs for session beans, HTTP requests, transactions and other WebSphere objects may occur before the reply is received. An application must be designed to handle a delayed reply or no reply at all.

Regardless of whether the application was initiated by a user request (e.g. an HTTP request) or another application, the application design should consider what to do when a reply message is not received in a reasonable time.

While waiting for the reply message the thread in the WebSphere application is blocked. To ensure that it does not wait forever, you should set a reasonable time-out value, e.g.

```
Message inMessage = queueReceiver.receive(5000);
```

In this example, the WebSphere application would wait for a maximum of 5000 milliseconds. If a reply has not been received in that time, then the application should deal with this situation, by sending an appropriate response to the client, for example, asking the end user to retry later.

Care must be taken at the remote application where the reply message is created and put to the reply queue. It is possible that the WebSphere application would time-out and the message sent by the remote application would never be taken off the queue, so you would want to set an appropriate expiry value for the reply message, e.g. set the JMS message expiry time to 10000ms.

Even with an expiry on the message, the requesting application should defend against getting a delayed reply message from a previous request. As an example, this could be done by selecting messages based on its correlation id.

Note: Since both the request and reply messages are “best-effort”, there should be no need to use MQSeries persistent messages in this scenario. Using **non-persistent** messages leads to significant performance improvements in the underlying message transport.

To send non-persistent messages you have three options. You can set the persistence property:

- directly on the queue within the queue manager
- on the queue object within JMSAdmin
- on a per message basis within your JMS application.

When defining a queue in the JNDI namespace, the JMSAdmin tool can be used to set the persistence properties for the queue :

```
InitCtx> DEFINE Q(TESTQ) PERSISTENCE(xxx)
```

Where xxx can be

APP	(Default) Persistence is defined by application
QDEF	Persistence is defined by the queue default (Set in the queue manager)
PERS	Messages are persistent
NON	Messages are not persistent

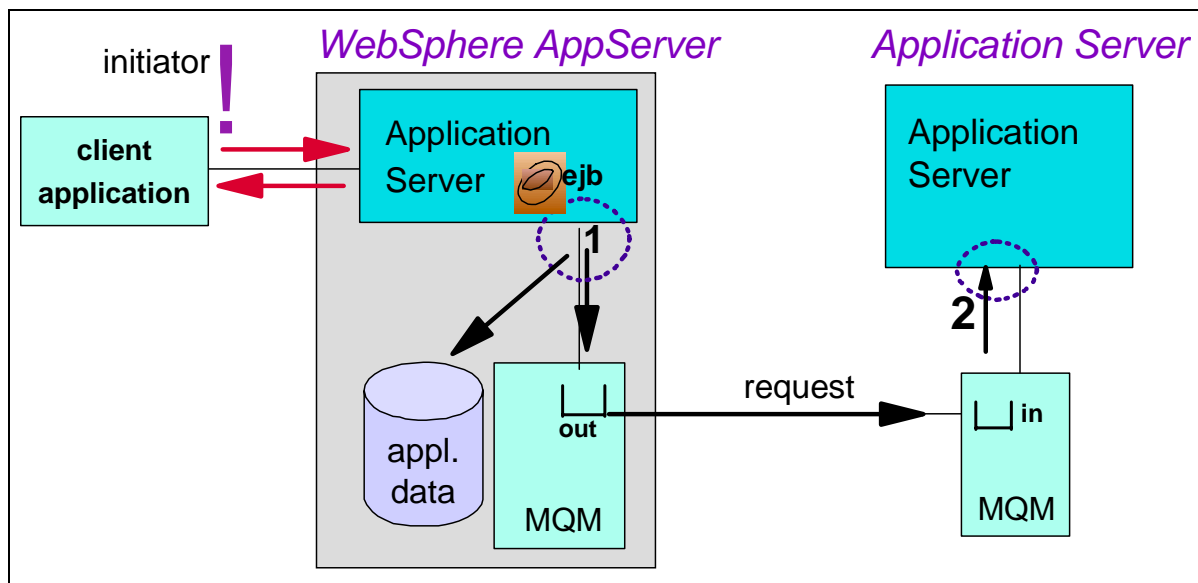
Note that the default case is that messages will be persistent. When using JMSAdmin, message persistence is left to the application by default, and the default delivery mode for JMS messages is also persistent.

Message persistence can be modified on the QueueSender, or specified on the call to the send method, but not directly on the message.

A JMS code fragment to set non-persistence and expiry time on messages sent:

```
QueueSender qS1 = queueSession.createSender(ioQueue);  
qS1.setDeliveryMode(DeliveryMode.NON_PERSISTENT);  
qS1.setTimeToLive(10000); /* 10000 ms */
```

2.2 Datagram : a resilient one way message



In this scenario the client application initiates some action, e.g. submitting a payment. In this kind of scenario, it may be desirable to update some local state which indicates that the message was actually sent. A WebSphere application can achieve this by putting the message on its *out* queue and within the same WebSphere transaction updating a database to log that the message has been sent.

The WebSphere application does not expect any reply message from the remote application. Depending on the type of client application, a response may be necessary: a web browser, for example, would require some sort of acknowledgement that a request had been made. If the user fails to receive that acknowledgement for any reason, they can now inquire the local state (using another function or screen in the web application) to check whether their request was indeed submitted.

2.2.1 Application design considerations

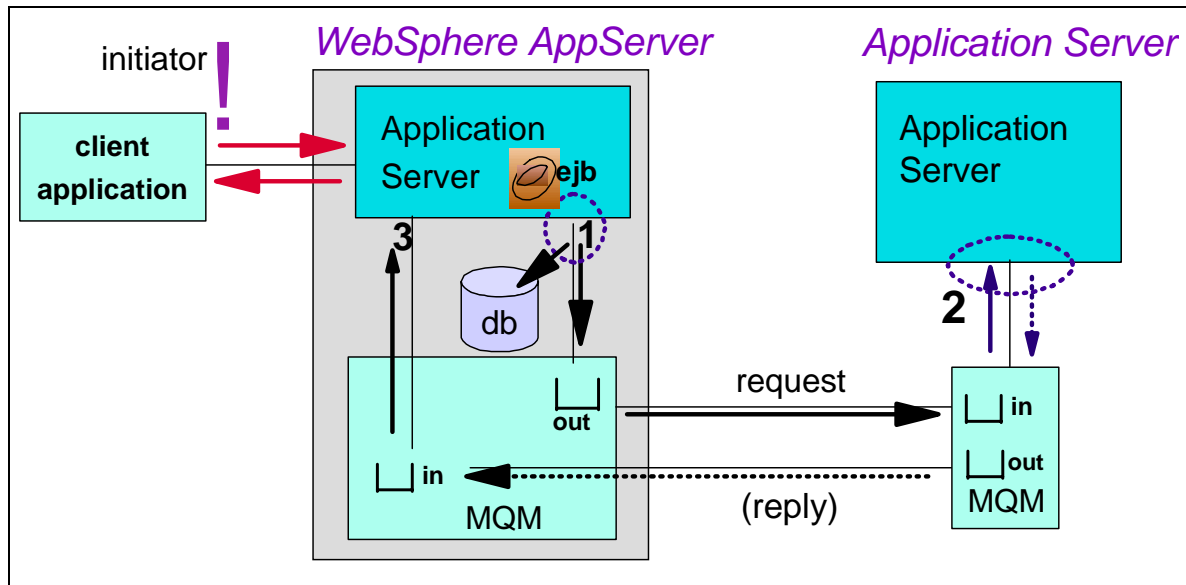
Transactions to MQSeries are boundary transactions, not end-to-end transactions. That is, only the *put* to a queue, or the *get* from a queue is part of a transaction. Once the message is on a queue that is the end of the scope of the transaction as far as that message is concerned. The flow to the remote application is not part of that same transaction. In order to guarantee that the message is received by the remote application you can now rely on the assured delivery capabilities that MQSeries brings for persistent messages..

By including a message *put* in the scope of a transaction you will generally want to ensure that the message is delivered using a persistent delivery mode. See section 2.1.1 for an explanation of how to set persistent message delivery.

As the client application is not expecting a reply from the remote application, the MQSeries asynchronous model will be reflected by the functionality of the application which is exposed to the client. If there is an end user driving the application they will want a way to verify that their request has been submitted, and perhaps a way to check on the status of it.

Taking the submitting a payment example, the WebSphere application could present the user with a page with two links, one to return whether the request was successfully submitted (query the database which was written to during the transaction), the other to fire off a best effort request/reply, similar to scenario 1 above, to determine whether the payment has been fully processed or not.

2.3 Combination : resilient request / best effort reply



This scenario is a combination of the datagram and the best effort request/reply scenarios. An example of where this could be used is in an intra-bank balance transfer.

The client initiates a request for some action to be taken. WebSphere starts a transaction. The application puts the message on its *out* queue and updates a database to log that the message has been sent. Then WebSphere commits the transaction. The application then waits for a reply back to its *in* queue. When the remote application server receives the message it handles it and returns information on the state of the request.

In this scenario, it is possible that the reply message is not received in an appropriate amount of time. If this happens an end user would need an alternative way of checking the status of the request.

2.3.1 Application design considerations

The same issues of time-outs, persistence and end user experience apply as in the best effort request/reply and datagram scenarios (see sections 2.1.1 and 2.2.1).

It is not possible to put a message to a queue and expect a reply within the same WebSphere transaction. The put will only be committed (or sent) when the whole WebSphere transaction is committed, but the reply is not available until later when the request has been processed by the remote application.

If you are doing a *put* then a *get* of a related request/reply pair from the same EJB method, ensure that the method is not in a transaction. You could enforce this by setting the transaction descriptor to TX_NOT_SUPPORTED for your calling bean's method, e.g.

```
Public void CallingBean-put-method // This method cannot be in a
transaction
{
    String id = MsgBean.put(myMessage); // use transaction on MsgBean put
    String response = MsgBean.get(id); // and/or get methods
}
```

Note: Alternatively, the application could use a second non-transactional connection and session to *put* a messages outside of the global transaction if the method was in a transaction.

2.4 Other permutations

The same scenarios apply if the initiation of a request is not done by WebSphere, but by the remote application server. In this case, WebSphere would host the remote application and you would want an EJB, for example, to be triggered when a message appeared on its *in* queue. To do this you need something to monitor the incoming queue, and when a message comes in to determine what action needs to be taken.

2.4.1 Message consumers and listeners

Generic JMS consumers and listeners are not provided with WebSphere 3.5.3. You can use the MQ JMS classes to create your own message consumers and listeners which will wait for a message to arrive and then process it. Typically, a multi-threaded design is used such that the message is passed to a worker thread which actually actions the request and is also responsible for sending any reply (if required). This way the message consumer is not blocked by the processing of the message before processing the next message.

The EJB 1.0 and EJB 1.1 specifications state that “an EJB should not start new threads or attempt to terminate the running thread”. The reason for this statement is that it prevents the container from being able to manage the resources properly. This means that until EJB 2.0 delivers message driven beans, an alternative strategy is required.

2.4.2 Message Driven Beans

When message driven beans (MDB) are available, one of the interfaces a bean provider must implement is `javax.jms.MessageListener`, resulting in the following method signature:

```
void onMessage(javax.jms.Message aMessage)
```

This method will then be called by the EJB container when a message is received. Therefore, if you want to design beans that can be migrated to MDB easily then they should follow the same principle.

2.4.3 One approach for an interim solution

In order to avoid breaking the EJB specification, you can monitor a message queue outside of the EJB container. You can do this by implementing the interface

`com.ibm.ejs.sm.server.ServiceInitializer`, which defines two methods :

```
public void initialize(Context initialNamingContext) throws Exception;
public void terminate(Context initialNamingContext) throws Exception;
```

then adding the following line to the application server command line arguments :

```
-Dcom.ibm.ejs.sm.server.ServiceInitializer=myService;
```

This causes the application server to call the *initialise* method of the *myService* class as part of the application server start-up. Then when a request is made to stop the application server, it calls the *terminate* method of the class.

The following pseudo code demonstrates how this consumer and listener could work :

```
myService
initalize (...) {
    Create a pool of worker threads
    While (!request_to_stop) {
        wait for a message to arrive
        Retrieve the message ID
        Pass the message ID to the worker thread
    }
}
terminate () {
    request_to_stop = true
    Terminate worker threads
}

workerThread
doService(msgID) {
    Lookup messagebean
    Messagebean.doService(msgID)
}

messageBean
doService(msgID) {
    Retrievemessage
    this.onMessage(message)
}
```

As *myService* will run within an application server JVM, great care must be taken when creating and managing the thread pool because each thread is using system resources. If care is not taken, it is possible that the number of threads used will seriously effect the overall performance of the other services controlled by the application server (such as the servlet engine or the EJB container), or in fact the whole system.

The *messageBean* must have a way to deal with invalid messages. If a message is not able to be processed for some reason after a certain number of tries it should be removed from the input queue and placed on a failure queue. This prevents the bean from entering a loop by continuously processing an invalid message.

3. Configuring WebSphere to support transactional messages

3.1 Install the required software

- Websphere Application Server Advanced Edition Version 3.5 with Fixpack 3
Note: This also installs JDK 1.2.2
- MQSeries Version 5.2
- Latest SupportPac MA88: MQSeries classes for Java and MQSeries Classes for JMS 5.2
Follow the instructions in chapter 2 (*Installation procedures*) of the *Using Java* manual to help you.

Note1: When installing MA88 the setup program, by default, installs into the target directory C:\Program Files\Ibm\MQSeries\Java . Use Custom Install if you want to change the target directory (e.g. to x:\MQSeries\java).

Note2: If you do not have the 'Microsoft Windows Installer Redistributable' installed on your Windows NT system, you are not able to run the setup.exe . In this case either download the Installer from the Microsoft site, or copy the Java subdirectory manually from <ma88_root_dir>\program files\ibm\MQSeries\Java to your MQSeries directory x:\MQSeries\java. You find a link to the Microsoft download page on the IBM MA88 SupportPac URL :

<http://www-4.ibm.com/software/ts/mqseries/txppacs/ma88.html>

3.2 Setup your system environment

Here we describe the setup required to get JMS working on your system. Check the environment variables, MQ_JAVA_INSTALL_PATH, PATH and CLASSPATH, are set and contain entries similar to these at the beginning of them:

MQ_JAVA_INSTALL_PATH

- x:\MQSeries\Java

Note: MQ_JAVA_INSTALL_PATH should point to the directory where JMS is installed. Many of the samples supplied with MA88 assume that this environment variable is set.

PATH

- x:\MQSeries\bin
- x:\MQSeries\java\lib
- x:\MQSeries\java\bin
- x:\WebSphere\AppServer\jdk\jre\bin

Note1: This is not a complete PATH, but just the portion relevant to MQSeries and JMS.

Note2: x:\WebSphere\AppServer\jdk\bin contains the WebSphere JDK 1.2.2 . Check that there are no older JDK libraries included in the PATH variable.

CLASSPATH

- x:\MQSeries\java\lib\com.ibm.mq.jar
- x:\MQSeries\java\lib\com.ibm.mqjms.jar
- x:\WebSphere\AppServer\lib\ujc.jar
- x:\WebSphere\AppServer\lib\ejs.jar

Note: This differs from the recommended CLASSPATH settings in the MA88 readme.txt file and in the MA88 *Using Java* manual. You do not need to include jms.jar, jndi.jar and jta.jar because they are now included in ujc.jar. You only need to include providerutil.jar, fscontext.jar and ldap.jar if you do not use the WebSphere JNDI name service.

3.2.1 Verify your setup so far

To verify that MA88 and MQSeries are setup correctly you can run the Installation Verification Test (IVT) supplied with MA88. Check that the MQSeries service is started and that your queue manager is running. **Note:** The IVT tests assume that you have a default queue manager configured.

From a command prompt change to the x:\MQSeries\java\bin directory. Type:

```
>ivtRun -nojndi
```

This tests that your setup is correct for putting and getting a message on the default local queue. The output of a successful run will look like :

```
Creating a QueueConnectionFactory
Creating a Connection
Creating a Session
Creating a Queue
Creating a QueueSender
Creating a QueueReceiver
Creating a TextMessage
Sending the message to SYSTEM.DEFAULT.LOCAL.QUEUE
Reading the message back again

Got message:
JMS Message class: jms_text
JMSType: null
JMSDeliveryMode: 2
JMSExpiration: 0
JMSPriority: 4
JMSMessageID: ID:414d51204d554e43484b494e2e495343b9a8c93a12600100
JMSTimestamp: 986370848160
JMSCorrelationID:null
JMSDestination: queue:///SYSTEM.DEFAULT.LOCAL.QUEUE
JMSReplyTo: null
JMSRedelivered: false
JMS_IBM_MsgType:8
JMSXAppID:pserver\jdk\jre\bin\java.exe
JMSXUserID:Administrato
JMSXDeliveryCount:1
JMS_IBM_PutApplType:11
JMS_IBM_Format:MQSTR
A simple text message from the MQJMSIVT program
Reply string equals original string
Closing QueueReceiver
Closing QueueSender
Closing Session
Closing Connection
IVT completed OK
IVT finished
```

See the section in chapter 4 of the *Using Java* manual called *Running the point-to-point IVT* for more information on the verification tests.

3.3 Setup JMSAdmin to use the WebSphere JNDI name service

The JMSAdmin tool comes with MA88 and allows you to define JNDI objects in a namespace. JMSAdmin supports most JNDI name service providers, including WebSphere. Here, we will only give instructions for WebSphere, but for more information see the MA88 readme.txt and the *Using Java* manual.

First, set JMSAdmin to use WebSphere as the name service. Edit **JMSAdmin.config** in `x:\MQSeries\java\bin` so that it includes these lines:

```
INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory
PROVIDER_URL=iiop://hostname/
```

Where *hostname* can be **localhost** if the WebSphere service is running on the same machine as JMSAdmin.

Note: In *jmsadmin.bat*, change the occurrence of `-DMQJMS_INSTALL_PATH` to `-DMQ_JAVA_INSTALL_PATH`. This allows you to run JMSAdmin from anywhere, rather than just the `x:\MQSeries\java\bin` directory.

3.3.1 Verify your JNDI setup

To verify that the JNDI service is setup correctly we will use the IVT again. First we need to create the required objects for the test in the JNDI namespace. Check that the WebSphere and MQSeries services are started, and that your queue manager is running.

From a command prompt change to the `x:\MQSeries\java\bin` directory. Type:

```
>ivtSetup
```

The output of a successful run will look like :

```

G:\MQSeries\java\bin>rem
-----
G:\MQSeries\java\bin>rem  MQSeries Support for Java Message Service
G:\MQSeries\java\bin>rem  5648-C60 (c) Copyright IBM Corp. 1999. All
Rights Reserved
G:\MQSeries\java\bin>rem
G:\MQSeries\java\bin>rem  Installation Verification Tests Setup Script
G:\MQSeries\java\bin>rem
-----
+ Creating script for object creation within JMSAdmin
+ Calling JMSAdmin in batch mode to create objects

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

InitCtx>
InitCtx>
InitCtx>
InitCtx>
InitCtx>
Stopping MQSeries classes for Java(tm) Message Service Administration

+ Administration done; tidying up files
+ Done!

```

This creates the queue and queue connection factory objects needed for the following test.

Note: Once the objects have been created, if you run the `ivtSetup` again, you will see 'Unable to bind object' messages, because those objects already exist.

Now run the test using the WebSphere JNDI name service. Type:

```
>ivtRun -url iiop://localhost/ -icf com.ibm.ejs.ns.jndi.CNInitialContextFactory
```

The output of a successful run should look similar to the previous test in section 3.2.1 (*Verify your setup so far*). The difference is that rather than creating `QueueConnectionFactory` and `Queue` objects, it retrieves them from JNDI.

3.4 First Steps with JMS/JTA

MA88 comes with three WebSphere JMS/JTA sample applications. Sample 1 has an EJB that does a simple *put* and *get* of a message in a queue using container-managed transactions. We will use this sample to demonstrate setting up the transactional environment correctly.

You must have a queue manager defined for the sample to work. This can either be the default or a named queue manager. Here, we assume that you are using the default queue manager. If you do not have a default queue manager you will have to alter the queue connection factory later. Instructions on how to do this are later in this section.

For Sample 1, you need to add two objects into the JNDI namespace, a queue connection factory and a queue.

From the Sample1 subdirectory (x:\MQSeries\java\samples\jms\ws\sample1), at a command prompt enter:

```
JMSAdmin <admin.scp
```

Admin1.scp is a configuration file that defines the context then stores a WebSphere specific queue connection factory and queue into the JNDI namespace.

Admin1.scp contains the following statements :

```
# create the jms subcontext.
def ctx(jms)
chg ctx(jms)

# create a samples subcontext
def ctx(Samples)
chg ctx(Samples)

# create a WebSphere specific queue connection factory
def wsqcf(QCF1)

# create a Queue. The default queue can be replaced with an alternative
# local queue of your choice.
def q(Q1) queue(SYSTEM.DEFAULT.LOCAL.QUEUE)

# display the current context to provide feedback
dis ctx
# exit
End
```

You should see the output from the *display context* command (dis ctx) on the screen. It should look like this :

```
Contents of InitCtx/jms/Samples
```

```
  a  Q1          com.ibm.mq.jms.MQQueue
  a  QCF1       com.ibm.ejs.jms.mq.JMSWrapXAQueueConnectionFactory

  2 Object(s)
    0 Context(s)
    2 Binding(s), 2 Administered
```

If you do not have a default queue manager you need to alter the queue connection factory definition in JNDI. Start JMSAdmin and enter the following commands :

```
InitCtx>chg ctx(jms/Samples)
InitCtx/jms/Samples>alter wsqcf(QCF1) qmanager(my.queuemanager.name)
InitCtx/jms/Samples>end
```

3.5 Connection factories

If you used the Installation Verification Test in section 3.3.1 (*Verify your JNDI setup*) and have completed the setup for Sample 1, as described above, you may have noticed that you have defined instances of two different types of queue connection factories in your namespace:

- MQConnectionFactory
- JMSWrapXAQueueConnectionFactory

The main difference of these two connection types is their transactional capability.

The basic MQConnectionFactory offers no transactional functionality, messages put to a queue inside a WebSphere transaction will not be rolled back should the transaction be rolled back in WebSphere.

JMSWrapXAQueueConnectionFactory shares the transactional environment of WebSphere. Puts and gets are committed with the transaction and can be rolled back. This allows an MQSeries resource to be involved in transactions with WebSphere and other resources such as DB2. It can be used as part of a 2 phase commit transaction, the transaction being coordinated across multiple resources.

Note: In the JNDI namespace objects defined to one of these connection factory types are discrete from the other type. That is, if you have defined an object called QCF1 as a JMSWrapXAQueueConnectionFactory (wsqcf), then using 'dis qcf(QCF1)' in JMSAdmin to display the object properties will fail. You need to use 'dis wsqcf(QCF1).

This difference is hidden from the JMS programmer. Within your java code you call out to the JNDI namespace to get a QueueConnectionFactory based on an object name. You are then returned an MQ or a WebSphere specific connection factory depending on how the object was defined.

3.6 Explanation of some JMSAdmin commands

You can use the JMSAdmin tool to view and change the properties of the objects stored in the JNDI namespace. Here are some useful commands :

[Display all objects relative to the current context]

```
InitCtx> dis ctx
```

```
Contents of InitCtx
```

```
[D] jta com.ibm.ejs.ns.jndi.CNContextImpl
  a ivtT com.ibm.mq.jms.MQTopic
[D] jdbc com.ibm.ejs.ns.jndi.CNContextImpl
  a ivtQ com.ibm.mq.jms.MQQueue
  a ivtQCF com.ibm.mq.jms.MQQueueConnectionFactory
[D] jms com.ibm.ejs.ns.jndi.CNContextImpl
  a ivtTCF com.ibm.mq.jms.MQTopicConnectionFactory
[D] ejadmin com.ibm.ejs.ns.jndi.CNContextImpl
[D] NATHAN com.ibm.ejs.ns.jndi.CNContextImpl
```

```
21 Object(s)
  7 Context(s)
  14 Binding(s), 4 Administered
```

[Display the MQQueueConnectionFactory object called ivtQCF]

```
InitCtx> dis qcf(ivtQCF)
```

```
TRANSPORT(BIND)
TEMPMODEL(SYSTEM.DEFAULT.MODEL.QUEUE)
MSGRETENTION(YES)
QMANAGER()
VERSION(2)
```

[Change the context to jms/Samples]

```
InitCtx> chg ctx(jms/Samples)
```

[Display all objects relative to the current context - jms/Samples]

```
InitCtx/jms/Samples> dis ctx
```

Contents of InitCtx/jms/Samples

```
a Q1 com.ibm.mq.jms.MQQueue
a QCF1 com.ibm.ejs.jms.mq.JMSWrapXAQueueConnectionFactory
```

```
2 Object(s)
0 Context(s)
2 Binding(s), 2 Administered
```

[Display the MQQueueConnectionFactory object called QCF1 - using the wrong qcf type]

```
InitCtx/jms/Samples> dis qcf(QCF1)
```

Expected and actual object types do not match

[Display the WebSphere specific JMSWrapXAQueueConnectionFactory called QCF1]

```
InitCtx/jms/Samples> dis wsqcf(QCF1)
```

```
TRANSPORT(BIND)
TEMPMODEL(SYSTEM.DEFAULT.MODEL.QUEUE)
MSGRETENTION(YES)
QMANAGER()
VERSION(2)
```

Some things to note here are :

- When QCF1 was defined a queue manager was not explicitly specified so the object will use the default queue manager defined in MQSeries. If there is no default queue manager defined then connections to the queue would fail, i.e. your application would fail with a JMS exception with a reason code of 2085.
- The transport mode on QCF1 is set to Bind, the default. This uses local native code bindings to connect to MQSeries (i.e. you are not connecting remotely to MQSeries). The Bind transport is the only mode that transactions will work with.

3.7 Setup the WebSphere environment

Check your WebSphere Application Server environment. You need the following jar files in order to run Sample1 : ejs.jar, ujc.jar, com.ibm.mq.jar and com.ibm.mqjms.jar. As noted before, jms.jar is included in ujc.jar and therefore is not needed.

Check in \WebSphere\Appserver\bin\admin.config that the following files are included in com.ibm.ejs.sm.adminserver.classpath:

```
x\:/WebSphere/AppServer/lib/ejs.jar;
x\:/WebSphere/AppServer/lib/ujc.jar;
```

Instead of putting the com.ibm.mq.jar and com.ibm.mqjms.jar in the adminserver classpath (as described in the MA88 readme.txt) you can place them on the classpath for the application server that you are running your JMS applications in. This is because the adminserver classpath is inherited by all application servers and you only need these files on those servers running JMS applications. By doing this, the application server will need to be running before being able to deploy an EJB to it.

1. Change the application server classpath. From the Websphere Admin Console, select the relevant application server (e.g. Default Server) and enter into the command line arguments:

```
-classpath
x:\MQSeries\java\lib\com.ibm.mqjms.jar;x:\MQSeries\java\lib\com.ibm.mq.jar;
```

2. Add an environment variable to your application server environment. Click the **Environment** button of your server and add the variable `PATH` with value `x:\MQSeries\java\bin;x:\MQSeries\java\lib`.
3. Click **Apply** and restart the application server. (If you do not click Apply you will lose these updates.)

See Appendix A (*Debugging in WebSphere*) for problems relating to your CLASSPATH.

3.8 Deploy and test Sample 1

Now you can deploy the Sample 1 code and test that WebSphere is setup correctly for JMS/JTA application. All code for this sample is in

`x:\MQSeries\java\samples\jms\ws\sample1`

1. Make sure that your application server is started.
2. Deploy the Sample1 EJB in `x:\MQSeries\java\samples\jms\ws\sample1\jmsSample1.jar` into your EJB Container.

<p>Note: After you have deployed the bean, if you select it from the Admin Console, view the 'Deployment descriptor in use' and select the Transactions tab, you will see that the methods <code>getMessage</code> and <code>putMessage</code> have transaction attribute <code>TX_REQUIRED</code>.</p>
--

3. Start the bean you have just deployed.

Now use the client program provided to test that your WebSphere JMS/JTA environment is configured correctly.

1. Extract **Sample1Client.class** from `jmsSample1.jar` into

`x:\MQSeries\java\samples\jms\ws\sample1`

<p>Note: that this class file is packaged in the jar in the <code>sample1</code> directory, so check the location of this file after you have extracted it.</p>
--

2. Edit **runClient.bat** and change the location of WebSphere to where you have it installed:
`@set WAS=c:\Websphere\Appserver`
3. From a command prompt, start the client, passing it a string to send as a message, e.g.
`>runClient Hello`

The output from a successful run will look like:

```
calling Sample1 EJB with string 'Hello'
calling putMessage
calling getMessage
got message 'Hello'
done
```

The EJB write progress messaged to standard out (stdout). To see these look in the stdout.txt file for the application server that the EJB is deployed in.

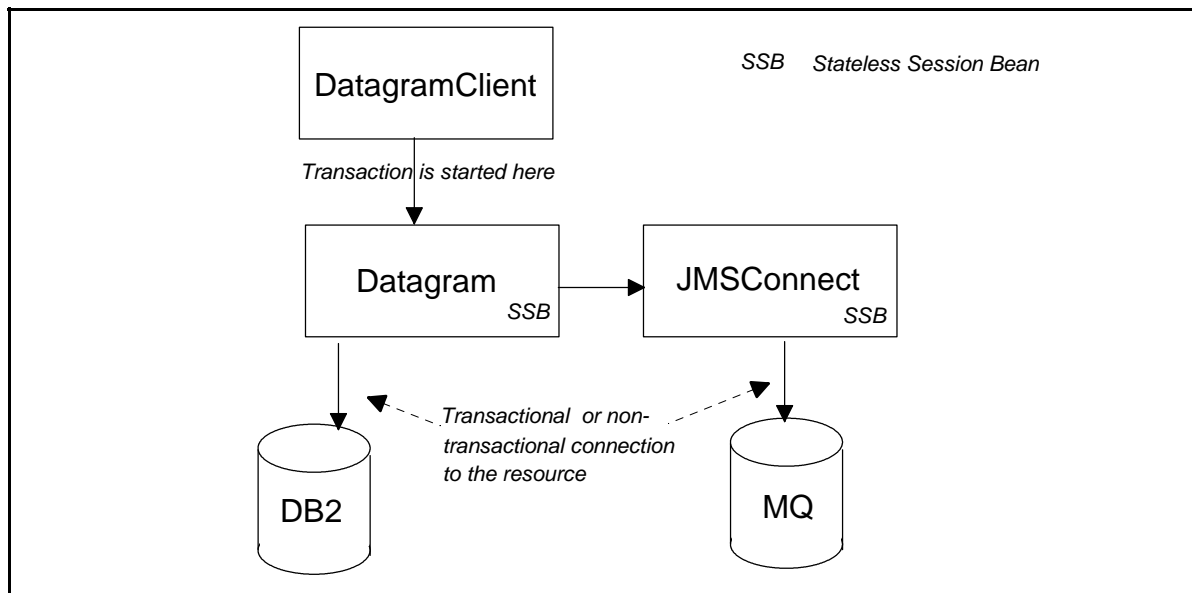
Note: When you create a new application server the stdout.txt and stderr.txt files are created in the system32 folder. You can specify where they are set on the General tab of the applicatin server.

See the readme.txt in the sample1 directory and *Appendix E (JMS JTA/XA interface with WebSphere) of the Using Java manual* for further information on this sample.

4. Datagram example

The Datagram example in section 2.2 (*Datagram : a resilient one way message*) discusses transactional JMS applications for WebSphere. Here we have written some sample code that implements this basic application design. The Datagram code has a client and two EJBs. The source code for these is in Appendix B.

The client calls a stateless session bean that writes to a database and calls another stateless session bean, JMSSConnect. This second bean wraps the calls to JMS to give some degree of abstraction.



This example allows you to use transactional and non-transactional connections to DB2 and MQ, to see the effects that a transaction has.

The transactional capability when connecting to MQ depends on the type of queue connection factory used. The default `MQQueueConnectionFactory` offers no capability, but with the latest JMS support an additional type has been added, `JMSWrapXAQueueConnectionFactory`. See section 3.5 (*Connection factories*) for more information on these two types.

This new connection type allows 2 phase commit between an MQ queue and another resource, in this example DB2.

Because the different connections to MQ or DB2 are stored in the JNDI repository, they can be changed at runtime. The client can be run using transactional and non-transaction connections and the different behaviours compared. In this chapter we explain how to set up and run the Datagram example.

4.1 JNDI configuration

For this example you need a definition of each of the two types of connection factory, one for the transactional tests and one for the non-transactional tests. If you used the Installation Verification Test in section 3.3.1 (*Verify your JNDI setup*) and have completed the setup for Sample 1, as described in chapter 3 (*Configuring WebSphere to support transactional messages*) then you will have the required objects defined in your JNDI namespace :

```
a   ivtQ           com.ibm.mq.jms.MQQueue
a   ivtQCF        com.ibm.mq.jms.MQQueueConnectionFactory
a   Q1            com.ibm.mq.jms.MQQueue
a   QCF1         com.ibm.ejs.jms.mq.JMSWrapXAQueueConnectionFactory
```

If not, you can define them using `JMSAdmin` with the following commands :

```
InitCtx>def qcf(ivtQCF)
InitCtx>def q(ivtQ) queue(SYSTEM.DEFAULT.LOCAL.QUEUE)
InitCtx>def ctx(jms)
InitCtx>chg ctx(jms)
InitCtx/jms>def ctx(Samples)
InitCtx/jms>chg ctx(Samples)
InitCtx/jms/Samples>def wsqcf(QCF1)
InitCtx/jms/Samples>def q(Q1) queue(SYSTEM.DEFAULT.LOCAL.QUEUE)
```

- `ivtQCF` is the non-transactional connection factory
- `QCF1` is the transactional connection factory

4.2 DB2 configuration

When connecting to databases the JDBC driver can be transactional and non-transactional. This is set when the JDBC driver is installed into WebSphere. This section describes how to configure the datasources for this example.

4.2.1 Setting up JDBC2

By default DB2 does not use the `jdbc2` drivers (required for 2PC), these must be installed. If you have used WebSphere for database transactions before you will already have done this step, however, doing it again will not harm your installation.

Lots of applications use the jdbc driver library file and if any are accessing it then it cannot be changed. To ensure that all connections are removed, it may be necessary to reboot your machine. This removes any unknown connections to the files that have started from running applications. Once you have done this, follow these steps to install the jdbc2 driver :

1. Stop all the DB2, IBM Websphere and IBM HTTP services. Stop anything else you know to be accessing the database files such as VisualAge® Java. On NT you can do this from Services.
2. In a command window go to x:\SQLLIB\java12. Run usejdbc2.bat, this should not show any errors about not being able to replace files.
3. Reboot the machine again to start all the services

If you do not use the jdbc2 drivers then WebSphere will be unable to create transactions that include both the database and JMS resources. This causes an exception inside WebSphere which will not be reported back to the user in the client stack trace, see Appendix A, (*Debugging in WebSphere*), on how to trace problems inside the application server.

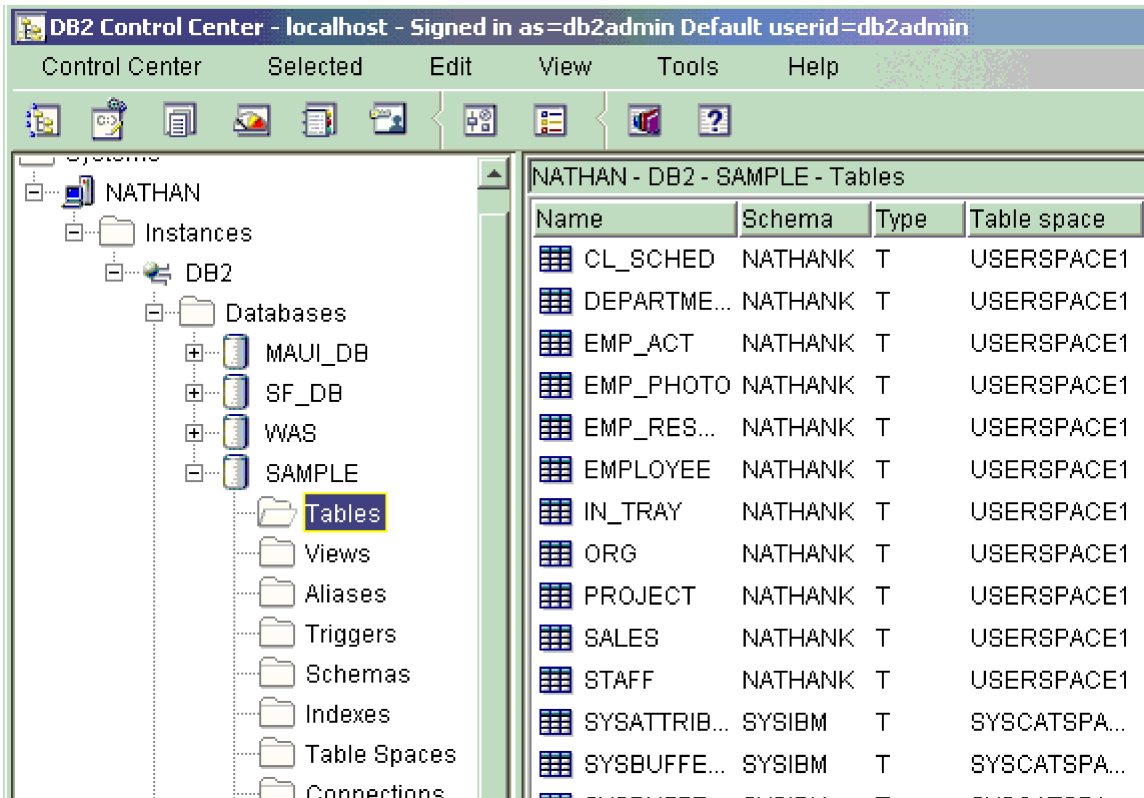
4.2.2 Setting up a datasource

In the same way that we have two connection factories to demonstrate transactional and non-transactional capabilities, we also need two datasources, one that is enabled for transactions and one that is not.

For this example we use the Sample database provided with DB2. To install it select the First Steps from the DB2 program group and create the Sample database.

In DB2, when a table is created it belongs to a schema. If a schema name is not explicitly given when a table is created, it will default to the name of the user creating the table. In addition, when a table is created only the user who created it is authorised to access it. Therefore, when connecting to the datasource you should use the user profile that created the table. If you do not do this then you must fully qualify the table name with the schema name and ensure that the user has authority to access the table.

If you are not sure who created a table you can use the DB2 Control Center to determine the schema name and user authorities to the table. To do this, from the Control Center open up the Sample database and select the **table** folder. The schema name is in the schema column :

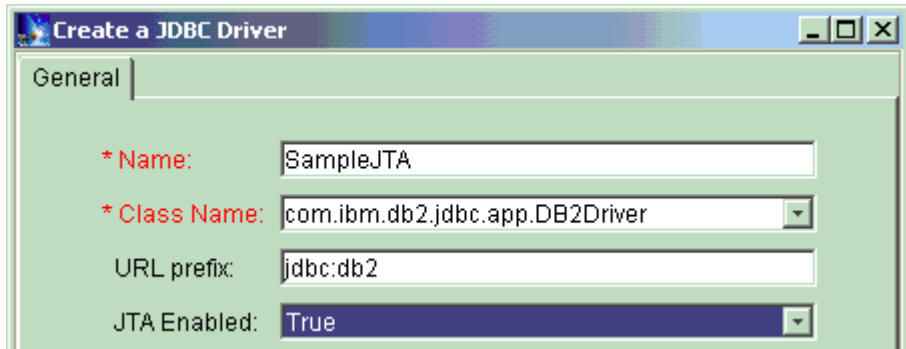


You will need this userid and the correct password when setting up the datasource inside WebSphere.

Now setup WebSphere to access the database. For this exercise we want to set up two JDBC connections, one with transactional support enabled and one with it disabled.

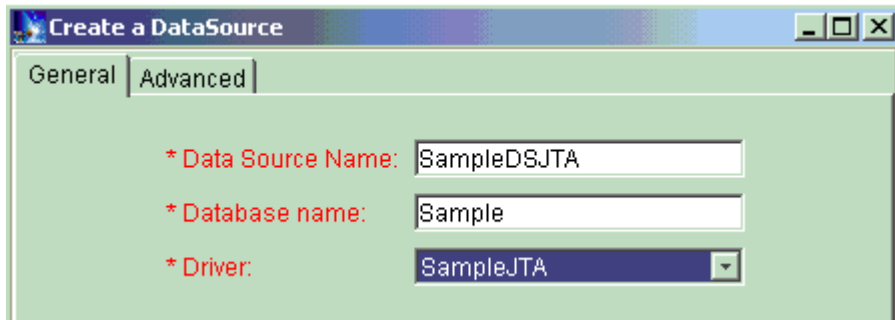
From the WebSphere Admin Console, go to the *Type* view (**View->Type**)

Create a JDBC Driver : Right click on the JDBC drivers folder and select **create**. You will be presented with a dialog box. Fill it in as follows:



Create another one, with the Name **SampleNOJTA** with JTA enabled set to **false**.

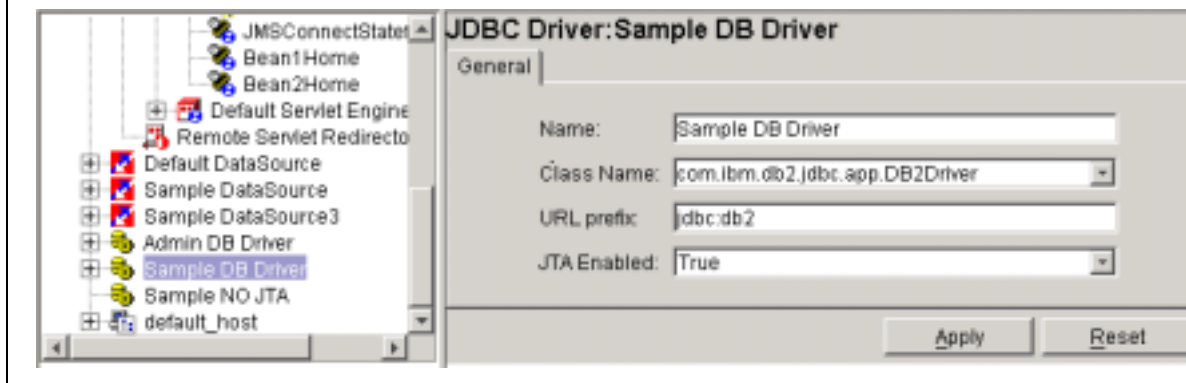
Create two datasources : Right click on the DataSource folder and select **create**. You will be presented with a dialog box. Fill it in as follows:



Create a second one, SampleDSNOJTA, which uses the non transactional SampleNOJTA JDBC driver.

Now switch back to the *Topology* view (**view->Topology**). You will see the two jdbc drivers you have just created in this view. Select each of them in turn, right click and select **install**.

Note: In order to do transactions to a database, the JDBC driver in the WebSphere Admin Console needs to be JTA enabled. If you are changing an existing JDBC Driver to be JTA enabled you must restart the application server (e.g. Default Server) rather than just the container, otherwise the change will not be picked up.



4.3 Transactional properties of EJBs in WebSphere

Setting the transaction attributes and isolation levels on EJB methods can be done from the Websphere Admin Console or the development tool VisualAge Java. Within a transaction, the isolation level for all methods called must be set to the same value. If possible, it is recommended that you set the isolation levels from the development tool to ensure all values are the same.

4.4 Using the Datagram example

In the Datagram session bean there is a method *put* whose transactional setting is TX_REQUIRED.

```
public void put(String message, String datasourceName, String qcfName,
                String queueName, boolean rollback) throws Exception
{
    // Get handle to my JMS helper bean
    InitialContext ic = new InitialContext();
    Object objref = ic.lookup("JMSSconnect");
```

```

JMSConnectHome home =
    (JMSConnectHome) javax.rmi.PortableRemoteObject.narrow(objref,
                                                            JMSConnectHome.class);
JMSConnect jmsBean = home.create();

// Get handle to my DB2 datasource
DataSource ds = (DataSource)ic.lookup(datasourceName);
java.sql.Connection conn = ds.getConnection();
PreparedStatement stmt = conn.prepareStatement("INSERT INTO IN_TRAY
      (RECEIVED,SOURCE,SUBJECT) VALUES(?, 'DBMQWrit',?)");
stmt.setDate(1,new java.sql.Date(System.currentTimeMillis()));
stmt.setString(2,message);

// Write out to the database
stmt.executeUpdate();

// Write out to the queue
jmsBean.putTextMessage(message,qcfName,queueName);

// Rollback transaction if we want to
if (rollback) mySessionCtx.setRollbackOnly();
stmt.close();
}

```

The DatagramClient takes a string as a parameter. This string is then sent as a message to MQSeries, and written to a database.

Using this client program and the script file to run it, Datagram.bat, we can use queue connection factories and datasources that have different transactional capabilities.

Messages are placed in the SYSTEM.DEFAULT.LOCAL.QUEUE of the default queue manager. Database records are written to the IN_TRAY table of the Sample database.

You can verify that a string has been written to the database by listing all the records in the table and seeing if it is included. To do this, issue the following SQL statements from the DB2 Command Center tool.

```

CONNECT TO Sample
SELECT * from IN_TRAY

```

From a system command prompt, use the following commands to see different behaviour depending on the datasource and the connection factory used:

```
> Datagram "Hello World" jdbc/SampleDSJTA userid password ivtQCF ivtQ false
```

This writes *Hello World* to the queue and the database table

```
> Datagram "Hello World" jdbc/SampleDSJTA userid password ivtQCF ivtQ true
```

This will rollback the transaction but MQ will still have the message on its queue. This is because the queue connection factory used, ivtQCF, is not transactional. Changing it to:

```
> Datagram "Hello World" jdbc/SampleDSJTA userid password jms/Samples/QCF1
jms/Samples/Q1 true
```

Will cause the queue message to rollback too, the queue connection factory 'jms/Samples/QCF1' uses the new transactional factory.

```
> Datagram "Hello World" jdbc/SampleDSNOJTA userid password jms/Samples/QCF1
jms/Samples/Q1 true
```

This will fail with a transaction rollback because it uses a JDBC datasource that does not have transactions enabled.

To find the underlying cause of a transaction rollback exceptions, you can use the WebSphere trace mechanism. See Appendix A (Debugging in WebSphere) to see how to use WebSphere tracing. If you did this you would see that it fails because the transaction needs to perform 2PC between MQ and DB2. Since it cannot do this it does not allow only half the operation to complete, and so the whole transaction is rolled back.

Below is a table of options you could use to test all the possible combinations with this Datagram example.

Case	Queue Connection Factory	Datasource JTA Enabled	Rollback	Outcome
1	WSQCF	Yes	False	The data is on the Queue and in the database
2	WSQCF	Yes	True	The data is not in the Queue and not in the database
3	WSQCF	No	False	Throws exception because it cannot guarantee the transaction with database. Using trace you see the exception showing it cannot use 1PC database driver
4	WSQCF	No	True	Rolls back message on queue, never writes message to database. No exception is thrown
5	QCF	Yes	False	The data is on the Queue and in the database
6	QCF	Yes	True	The data still appears on the queue and is not rolled back. Database is rolled back correctly. This is different from case 3 when the database did not honour transactions, in that case an exception was thrown
7	QCF	No	False	The data is on the Queue and in the database
8	QCF	No	True	Data rolled back from database (using 1PC) Message appears in Queue.

5. Quality of service

5.1 Connection Pooling

MQSeries 5.2 provides significantly better underlying performance than previous versions of MQSeries. In particular, getting a connection to a queue manager is much faster.

The latest MA88 provides a default connection pool through the default connection manager. When a JMS Session is closed, the underlying connection is pooled. The scope of the pooling is global for the JVM. If many JMS Sessions are simultaneously active, the connections will be pooled between all of them. However, when the number of Sessions falls to zero, the pooling is disabled and all pooled connections are closed. Hence, if one Session at a time is sequentially used there will be no pooled connections persisted between them.

If an application needs to change the default behaviour of the pool, for example, to enable pooling between sequentially used JMS Sessions, it can use a base-MQ specific techniques to extend the lifetime of the default pool.

The following code uses the `MQSimpleConnectionManager` class to replace the default connection manager. The connection manager constructed will keep up to 100 unused connections alive for reuse for up to 1000 milliseconds and will always return a connection to the pool.

```
myConMgr = new MQSimpleConnectionManager();
myConMgr.setTimeout(1000);
myConMgr.setHighThreshold(100);
myConMgr.setActive(MQSimpleConnectionManager.MODE_ACTIVE);

MQEnvironment.setDefaultConnectionManager(myConMgr);
```

There is no hard limit on the number of connections in use. It is possible to have more connections in use than you have your connection pool configured to. This is because the pool is only keeping unused connections and not keeping track of any inuse connections. When a connection is closed and returned to the pool, if the pool is already at maximum size, the oldest connection in the pool is removed.

With no hard limit on the number of connections in use, it is possible to exhaust system resources if you try and have too many connections running at one time. Care must be taken over how many connections are in use.

5.2 Persistent vs Non-persistent messages

Within MQSeries a queue can hold persistent and non-persistent messages. Persistent messages are always written directly to disk, and are restored to the queue if the queue goes down for some reason. This type of message is resilient, but there is a performance cost because of the disk access.

If you wish to do transactional writing to the queue then you will want this level of resilience too. JMS does not enforce the type of message it writes in a transaction so you must ensure that the message is persistent yourself.

When using JMSAdmin, persistent messages are the default. If you do not explicitly set the persistence property of the queue, it will use the default, `PERSISTENCE(APP)`, i.e. the application determines persistence for the queue in JMSAdmin. The default delivery mode for JMS messages is also persistent. This property is set on the `QueueSender`, or specified on the call to the `send` method, not directly on the message.

If you want to change this

1. Set the persistence in the definition of the Queue, using JMSAdmin and/or
2. Use `setDeliveryMode` on the `QueueSender` or on the `send` method. Setting it on the Message will have no effect when sending messages (see `Message.setJMSDeliveryMode`).

5.3 Development environment

When developing 2 phase commit (2PC) transactional code accessing MQSeries you cannot test the transactional piece using the current version of VisualAge Java (VAJ), v3.5.3. The queue connection factory in VAJ and Websphere Test Environment (WTE) have not yet been updated.

To get around this, you can develop your code in VAJ using the existing JMS QueueConnectionFactory then test the transactional parts in WebSphere once the rest of the code works in VAJ.

If you want to be able to do all your testing within VAJ then, as an unsupported workaround, you can import the required files into your workbench. The files you need are contained within <WAS_HOME>\lib\ejb.jar. Extract the following files and import them:

- EnlistableResource.class
- JMSWrapQueueConnection.class
- JMSWrapQueueConnectionFactory.class
- JMSWrapQueueReceiver.class
- JMSWrapQueueSender.class
- JMSWrapQueueSession.class
- JMSWrapTopicConnection.class
- JMSWrapTopicConnectionFactory.class
- JMSWrapTopicPublisher.class
- JMSWrapTopicSession.class
- JMSWrapTopicSubscriber.class
- JMSWrapXAQueueConnection.class
- JMSWrapXAQueueConnectionFactory.class
- JMSWrapXAQueueSession.class
- JMSWrapXATopicConnection.class
- JMSWrapXATopicConnectionFactory.class
- JMSWrapXATopicSession.class
- JMSXAResourceFactory.class
- JMSXAResourceInfo.class
- WrapObjectFactory.class

Detailed information on writing JMS application, including the API reference is included in the *Using Java* documentation that accompanies MA88.

6. Performance and Scaling

6.1 One phase commit optimisation

Using a two phase commit transaction incurs a performance overhead. However, if applicable, WebSphere will optimise this by using one phase commitment where possible.

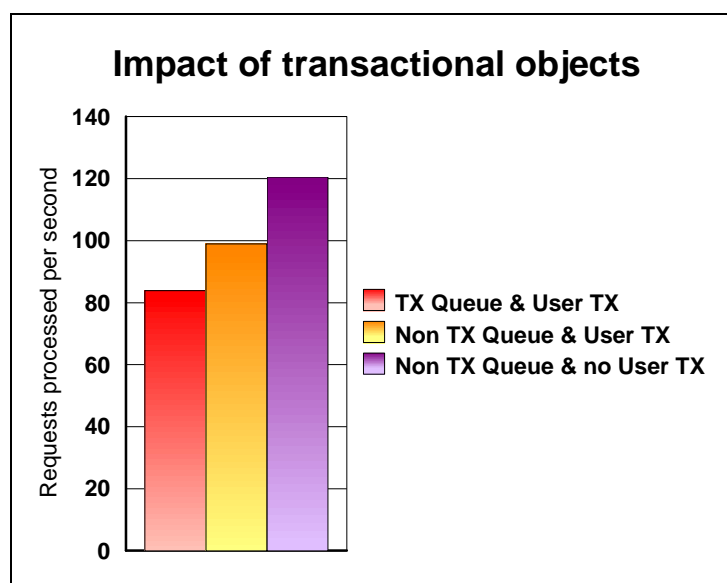
When a transaction is due to be committed, the WebSphere transaction coordinator will check to see if, in fact, a transaction is being performed across two resources. If only one resource is being used, then it will automatically perform a one phase commit (1PC) and so optimise transaction performance.

6.2 Impact of transactions

There is a performance cost for using a transactional queue connection factory (wsqcf). If an application does not require transactions then should you care about which connection factory objects you define?

To determine if the impact of transactional versus non-transactional connection factories in a IPC environment was significant, we used a simple test servlet.

When the servlet was invoked it was passed a 250 byte message. An initialisation parameter was used on the servlet to determine whether it would create a user transaction or not. If so, it would start it, write to the queue and then commit the transaction. This test was repeated using a combination of transactional and non-transactional queue connection factories, with and without user transactions. The results are illustrated in the following graph :



As you can see, the cost of using a transactional queue, even when IPC optimisation takes place may be significant if a high throughput is required. Therefore, if you **do not** require transaction support and want to ensure maximum throughput, you should:

1. ensure transactions are not created
2. not use transactional queue connection factories

6.3 Improving performance by caching MQ JMS objects

It is likely that your application will do more than just write one message to a message queue. Being able to cache some JMS objects in your beans would provide some performance enhancements. If you are intending to do this then you need to be aware that some of the objects are not thread safe. The following table lists the thread safety of JMS objects, according to chapter 10 of the MA88 *Using Java* manual.

JMS Object	Thread Safe
Connection Factory	Yes
Connection	Yes
Queue / Topic	Yes

Session	No
Message Producer	No
Message Consumer	No
Message	No

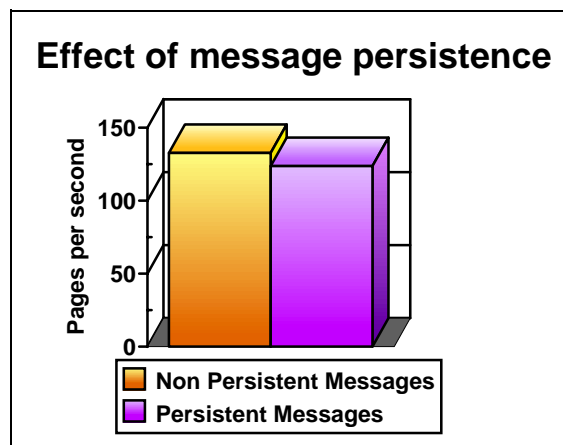
Thread safe objects can be cached as static fields in a session bean. The scope of a static field is class-wide, allowing all instances of the bean to use the same cached object. Since only the first instance created will lookup the thread safe objects, all instances of this session bean will perform better. Caching these objects does have a limitation; you should not change the queue connection factory or the queue after the first invocation or you could get unpredictable results. If this is an issue, your stateless session beans should not make use of static fields.

If a client makes repeated *put* or *get* calls to a stateful session bean, the bean should cache JMS objects. For example, an audit bean may be called repeatedly to send messages to a auditing system. In this case, the thread safe objects can be static fields (i.e. cached class-wide) since the queue connection factory and queue objects would not need to change, and non-thread safe objects can be instance fields (i.e. cached at the instance level).

6.4 Using non-persistent messages

It is possible to use non-persistent messages with JMS. This means that the message is passed to the underlying MQ transport, but not written to disk before the calling application is allowed to continue working. This could improve the performance of an application.

To prove this, we tested using a simple JMS servlet. When the servlet is invoked it is passed a 250 byte string, this is then sent as a JMS message to an MQ queue. This was repeated using both persistent messages and non persistent messages a number of times. The results are illustrated in the graph below :



As you can see, throughput was higher for non-persistent messages than persistent messages. In our tests there was about seven percent (7%) better performance when using non-persistent messages. The performance difference likely to be found on a production machine is much more significant. For details of a production-like scenario see the SupportPac, *MP6D* : *MQSeries for AIX® version 5.2 -- Performance highlights* at <http://www-4.ibm.com/software/ts/mqseries/txppacs/supportpacs/mp6d.pdf>

See chapter 2 (Business scenarios) for a discussion of when to and when not to use persistent messages.

6.5 MQSeries Clustering

JMS applications can use MQSeries queue manager clusters. Clusters allow you to scale your MQ applications and give you workload management, a highly available environment and easier system administration than administering many individual queues.

It is possible to *put* and *get* messages to a clustered queue, however, you can only *get* from a local queue (local to the queue manager to which the receiving application is connected). MQSeries clusters are especially useful in Datagram scenarios; where you put messages to queues and do not request a reply.

If you define a clustered queue that is hosted on several queue managers in the cluster (i.e. several queue managers have a local shared instance of the same queue) then MQSeries uses a workload management algorithm (by default this is a round-robin approach based on channel status and queue availability) to determine the queue manager to route a message to. If there is a local instance of the clustered queue, then this one is always tried first.

To set up a queue manager cluster for use with JMS you have to consider several things:

- The JMS Queue object should not contain a QMANAGER . If there is a QMANAGER defined to the queue, JMS puts it in the message header and the message will always be delivered to this queue manager.
- JMS needs to know a MQSeries queue manager to create a connection to . Either you define a default queue manager in your MQSeries Cluster configuration or you define a QMANAGER in your queue connection factory object in JNDI namespace. The second way gives you more flexibility. For example, if you have a cloned WebSphere application, then you can define different queue connection factories for each clone running on a different machine.
- The connected queue manager must not contain a local private queue with the same name as the target clustered queue. Otherwise the messages sent to the clustered queue will always be put into the local private queue. If this queue is not available, the message is not forwarded to the clustered queue.
- If you change the default queue manager or the JNDI queue connection factory definition, you have to restart your application server (e.g. Default Server) to pick up the change. This is because it caches the default queue manager.

7. Thanks

Thanks to everyone who gave their support and feedback to this paper. In particular, David Currie, Paul Fremantle, David Grainger, Vernon Green, Colleen Haffey, James Kingdon, Fintan McElroy, Jamie Roots, Greg Wadley

Appendix A - Debugging in WebSphere

A.1 WebSphere environment problems

The following are examples of exceptions caused because of a problem with the CLASSPATH. This is not the only cause of them, but it is a common one:

- `Com.ibm.ejs.container.UncheckedException NoClassDefFound:
com/ibm/mq/MQXAResource`
- `Com.ibm.ejs.container.UncheckedException ClassCastException:
javax.naming.Reference`

If you see this, carefully check your CLASSPATH for typos. Simple things like using forward slashes (/) instead of back slashes (\), or spaces between file names can cause this type of problem. For indepth information about WebSphere classpaths see the InfoCenter.

If you get a transaction rollback exception, and the underlying exception in a WebSphere trace is something like:

- `com.ibm.ejs.container.UncheckedException :
java.lang.UnsatisfiedLinkError: no library mqjbd02 (mqjbd02.dll) in
java.library.path`

this could mean that your PATH is set incorrectly. Check your system PATH for `x:\MQSeries\java\bin;x:\MQSeries\java\lib`. Or set an environment variable for your application server in the Websphere Administrative Console:

```
PATH=x:\MQSeries\java\bin;x:\MQSeries\java\lib
```

A.2 Tracing problems in WebSphere

Most exceptions that are thrown inside a transaction will be reported back to the caller as a Transaction Rollback or a Corba exception, without telling you what exactly happened. Using the trace mechanisms will give you a clearer picture of what actually went wrong so you can fix it

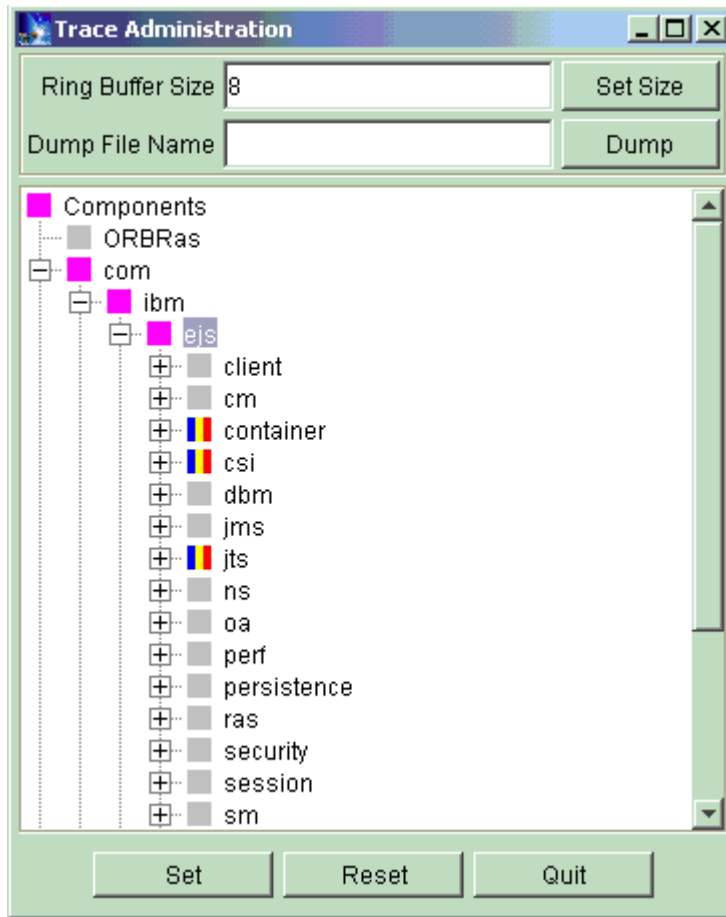
To trace an application server using the WebSphere admin console:

- select it (e.g. Default Server)
- right click and select **Trace**

This brings up the Trace Administration.

You can now select the packages you wish to trace. To trace the area involved with transactions:

- double click on **Components**
- select package **com.ibm.ejs.container.***
To do this, expand the tree, select **container**, right click and select **All**.
- repeat for packages **com.ibm.ejs.csi.*** and **com.ibm.ejs.jts.***
- click the **Set** button to enable tracing.



Run your application that causes the exception. Once you have seen the problem, invoke the Trace Administration again. This time enter a dump file name and click the **Dump** button. If your code is reporting a transaction rollback then this dump will often include the exception that caused the rollback to occur.

If you don't seem to get any useful data out, try tracing from a higher level package, such as `com.ibm.ejs.*`. You will get more output but it might also include the trace information you need.

Finally, when you have diagnosed your problem disable tracing by deselecting your trace settings from the Trace Administration.

A useful class is `BeanMetaData`. This creates an output of a bean's methods, and their transaction and isolation settings. To use this you must switch on tracing in the advanced tab of the application server. To do this, set the following fields :

```
Trace specification:      com.ibm.ejs.container.*=all=enabled
Trace output file:      <a_unique_file.name>
```

[For a full description see the WebSphere InfoCenter.]

This will cause dumps of the beans to the file you specified. This can be useful if you are having problems with transactions and isolation levels.

You will see something similar to:

```

[01.03.21 14:28:48:289 GMT] 8840f055 BeanMetaData d -- BeanMetaData Dump --
    "com.ibm.ejs.container.BeanMetaData@bc1df05d"
    BeanMetaData(STATELESS_SESSION, casel.JMSConnectBean)
    "HomeInterfaceClass = interface casel.JMSConnectHome"
    "RemoteInterfaceClass = interface casel.JMSConnect"
    "RemoteImplClass = class casel.EJSRemoteJMSConnect"
    "HomeRemoteImplClass = class casel.EJSRemoteJMSConnectHome"
    "pKeyClass = null"
    "Home Name = JMSConnect"
    "Reentrant = false"
    "exclusivePersistentStore = false"
    "findForUpdate = false"
    "No associated beans"
[01.03.21 14:28:48:299 GMT] 8840f055 BeanMetaData d Method attributes for: getTextMessage(0)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:309 GMT] 8840f055 BeanMetaData d Method attributes for: getTextMessage(1)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:359 GMT] 8840f055 BeanMetaData d Method attributes for: putTextMessage(8)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:359 GMT] 8840f055 BeanMetaData d Method attributes for: putTextMessage(9)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:369 GMT] 8840f055 BeanMetaData d Method attributes for: get(10)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:379 GMT] 8840f055 BeanMetaData d Method attributes for: get(11)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:379 GMT] 8840f055 BeanMetaData d Method attributes for: remove(12)
    "TX_SUPPORTS"
    "TRANSACTION_READ_COMMITTED"
    "read-only = false"
[01.03.21 14:28:48:389 GMT] 8840f055 BeanMetaData d Home method attributes for: create(0)
    "TX_REQUIRED"
    "TRANSACTION_READ_COMMITTED"
[01.03.21 14:28:48:389 GMT] 8840f055 BeanMetaData d Home method attributes for:
getEJBMetaData(1)
    "TX_NOT_SUPPORTED"
    "TRANSACTION_READ_COMMITTED"
[01.03.21 14:28:48:389 GMT] 8840f055 BeanMetaData d Home method attributes for: remove(2)
    "TX_SUPPORTS"
    "TRANSACTION_READ_COMMITTED"
[01.03.21 14:28:48:399 GMT] 8840f055 BeanMetaData d Home method attributes for: remove(3)
    "TX_SUPPORTS"
    "TRANSACTION_READ_COMMITTED"

```

Appendix B - Datagram example : source code

JMSConnectBean.java

```
package casel;

import java.rmi.RemoteException;
import java.security.Identity;
import java.util.*;
import java.util.Properties;
import javax.ejb.*;
import javax.naming.*;
import javax.jms.*;
import java.io.Serializable;
/**
 * This is a Session Bean Class
 */
public class JMSConnectBean implements SessionBean {
    private javax.ejb.SessionContext mySessionCtx = null;
    final static long serialVersionUID = 3206093459760846163L;
    private transient QueueConnectionFactory qcf = null;
    private transient Queue putQueue = null;
    private transient Queue getQueue = null;

    /**
     * The standard IDs are specified in the JMS Spec section 3.4
     * This example does not include them all
     */
    static String CORRELATION_ID = "JMSCorrelationID";
    static String MESSAGE_ID = "JMSMessageID";

    static short DEFAULT_TIMEOUT = 5000;

    private final static short TYPE_STRING = 1;
    private final static short TYPE_BYTE = 2;
    //private final static short TYPE_MAP = 3;
    private final static short TYPE_OBJECT = 4;
    private final static short TYPE_STREAM = 5;

    /**
     * ejbActivate method comment
     * @exception java.rmi.RemoteException The exception description.
     */
    public void ejbActivate() throws java.rmi.RemoteException {
    }
    /**
     * ejbCreate method comment
     * @exception javax.ejb.CreateException The exception description.
     * @exception java.rmi.RemoteException The exception description.
     */
    public void ejbCreate()
        throws javax.ejb.CreateException, java.rmi.RemoteException {
    }
    /**
     * ejbPassivate method comment
     * @exception java.rmi.RemoteException The exception description.
     */
    public void ejbPassivate() throws java.rmi.RemoteException {
    }
    /**
     * ejbRemove method comment
     * @exception java.rmi.RemoteException The exception description.
     */
    public void ejbRemove() throws java.rmi.RemoteException {
```

```

}
/**
 * getSessionContext method comment
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
    return mySessionCtx;
}
/**
 * Insert the method's description here.
 * Creation date: (14/03/2001 10:22:26)
 * @param jndiName java.lang.String
 * @exception javax.naming.NamingException The exception description.
 */
public void init(String jndiName) throws javax.naming.NamingException {
}
/**
 * Insert the method's description here.
 * Creation date: (14/03/2001 10:22:26)
 * @param jndiName java.lang.String
 * @exception javax.naming.NamingException The exception description.
 */
public void init(String qcfName, String putQName, String getQName)
    throws javax.naming.NamingException {
    setQCF(qcfName);
    setPutQ(putQName);
    setGetQ(getQName);
}

/**
 * Set the QueueConnectionFactory JNDI name
 * Creation date: (14/03/2001 10:22:26)
 * @param qcfName String thje jndiName of the QueueConnectionFactory
 * @exception javax.naming.NamingException The jndi name could not be
 * found
 */
private void setQCF(String qcfName) throws NamingException {
    InitialContext ic = new InitialContext();
    Object objRef = ic.lookup(qcfName);
    qcf = (QueueConnectionFactory) javax.rmi.PortableRemoteObject.narrow(
        ObjRef, QueueConnectionFactory.class);
}

/**
 * Set the outgoing put Queue JNDI name
 * Creation date: (14/03/2001 10:22:26)
 * @param putQName String thje jndiName of the put Queue
 * @exception javax.naming.NamingException The jndi name could not be
 * found
 */
private void setPutQ(String putQName) throws NamingException {
    InitialContext ic = new InitialContext();
    Object objRef = ic.lookup(putQName);
    putQueue = (Queue) javax.rmi.PortableRemoteObject.narrow(objRef,
        Queue.class);
}

/**
 * Set the incoming get Queue JNDI name
 * Creation date: (14/03/2001 10:22:26)
 * @param putQName String thje jndiName of the get Queue
 * @exception javax.naming.NamingException The jndi name could not be
 * found
 */
private void setGetQ(String getQName) throws NamingException {
    InitialContext ic = new InitialContext();

```

```

        Object objRef = ic.lookup(getQName);
        getQueue = (Queue) javax.rmi.PortableRemoteObject.narrow(objRef,
            Queue.class);
    }

    public String putTextMessage(String putText,String qcfName,
        String putQueueName)
        throws JMSEException, NamingException {
        setQCF(qcfName);
        setPutQ(putQueueName);
        return put(putText, TYPE_STRING, qcf, putQueue);
    }

    public String putTextMessage(String putText, QueueConnectionFactory qcf,
        Queue putQueue)
        throws JMSEException {
        return put(putText, TYPE_STRING,qcf,putQueue);
    }

    public String putObjectMessage(Object obj,String qcfName,
        String putQueueName)
        throws JMSEException, NamingException {
        setQCF(qcfName);
        setPutQ(putQueueName);
        return put(obj, TYPE_OBJECT, qcf, putQueue);
    }

    public String putObjectMessage(Object obj, QueueConnectionFactory qcf,
        Queue putQueue) throws JMSEException {
        return put(obj, TYPE_OBJECT,qcf,putQueue);
    }

    public String putStreamMessage(byte[] bytes,String qcfName,
        String putQueueName)
        throws JMSEException, NamingException {
        setQCF(qcfName);
        setPutQ(putQueueName);
        return put(bytes, TYPE_BYTE, qcf, putQueue);
    }

    public String putStreamMessage(byte[] bytes, QueueConnectionFactory qcf,
        Queue putQueue) throws JMSEException {
        return put(bytes, TYPE_STREAM,qcf,putQueue);
    }

    public Message get(String messageID, String selectorHeader, int timeout,
        String qcfName, String getQueueName)
        throws JMSEException, NamingException {
        setQCF(qcfName);
        setGetQ(getQueueName);
        return get(messageID,selectorHeader,timeout,qcf,getQueue);
    }

    public Message get(String messageID, String selectorHeader, int timeout,
        QueueConnectionFactory qcf, Queue getQueue)
        throws JMSEException {
        if (getQueue == null || qcf == null)
            throw new JMSEException("JMS Bean not initialized");
        QueueSession session = null;
        QueueSender sender = null;
        QueueConnection jmsCon = null;

        Message inMessage = null;

        //System.out.println("request to get message '"+messageID+"'");

        try {

            // Create connection
            jmsCon = qcf.createQueueConnection();
            jmsCon.start();

```

```

        // Create session
        session = jmsCon.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);

        // Create a receiver
        String selector = selectorHeader + " = '" + messageID + "'";
        QueueReceiver queueReceiver = session.createReceiver(getQueue,
            selector);

        System.out.println("getting message");
        inMessage = queueReceiver.receive(timeout);

        if (inMessage == null) {
            System.out.println(selector + " message not found");
        }

        System.out.println("closing receiver");
        queueReceiver.close();

        System.out.println("closing connection");
        jmsCon.close();
        jmsCon = null;

    } catch (JMSEException je) {
        System.out.println("failed with " + je);
        Exception le = je.getLinkedException();
        if (le != null)
            System.out.println("linked exception: " + le);

        // pass exception back to client
        throw je;

    } /* catch (Exception e) {
        System.out.println("failed with "+e);

        // pass exception back to client
        throw e;
    }*/

    return inMessage;
}

public String getTextMessage(
    String messageID,
    String selectorHeader,
    int timeout,
    QueueConnectionFactory qcf,
    Queue getQueue)
    throws JMSEException {
    Message msg = get(messageID, selectorHeader, timeout, qcf, getQueue);
    if (msg != null) {
        if (!(msg instanceof TextMessage))
            throw new MessageFormatException(
                selectorHeader + messageID + " is not a text message");

        return ((TextMessage) msg).getText();
    } else
        return null; // no message matched
}

/**
 * setSessionContext method comment

```

```

    * @param ctx javax.ejb.SessionContext
    * @exception java.rmi.RemoteException The exception description.
    */
    public void setSessionContext(javax.ejb.SessionContext ctx)
        throws java.rmi.RemoteException {
        mySessionCtx = ctx;
    }

private String put(Object putObject, int type, QueueConnectionFactory qcf,
                  Queue putQueue) throws javax.jms.JMSEException {
    if (putQueue == null || qcf == null)
        throw new JMSEException("JMS Bean not initialized");
    String messageID = null;
    QueueSession session = null;
    QueueConnection jmsCon = null;
    try {
        Message putMessage = null;
        jmsCon = qcf.createQueueConnection();
        jmsCon.start();
        // Create a session. Note that the parameters are ignored if
        // the connection factory is a WebSphere Specific one (as per the JMS
        // spec' for an XAQueueConnection).
        session =
            jmsCon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        switch (type) {
            case TYPE_STRING :
                putMessage = session.createTextMessage(putObject.toString());
                break;
            case TYPE_STREAM :
                putMessage = session.createStreamMessage();
                ((StreamMessage)putMessage).writeBytes((byte[])putObject);
                break;
            case TYPE_BYTE :
                putMessage = session.createBytesMessage();
                ((BytesMessage)putMessage).writeBytes((byte[])putObject);
                break;
            case TYPE_OBJECT :
                if (!(putObject instanceof Serializable))
                    throw new JMSEException("Can not create ObjectMessage with
                        non-serializable object");
                putMessage =
                    session.createObjectMessage((Serializable)putObject);
                break;
            default :
                throw new JMSEException("Unknown message type");
        }
    }

    // Create a QueueSender
    System.err.println("creating queue sender");
    QueueSender sender = session.createSender(putQueue);

    // ...and send it
    System.out.println("sending message");
    sender.send(putMessage);

    // store the message id for the return value
    messageID = putMessage.getJMSMessageID();

    // Close the connection (close calls will cascade to other objects)
    jmsCon.close();
    jmsCon = null;

    System.out.println("done");
} catch (JMSEException je) {
    System.out.println("failed with " + je);
}

```

```

        Exception le = je.getLinkedException();
        if (le != null)
            System.out.println("linked exception " + le);

        // Pass the exception back to the caller
        throw je;

    } finally {
        // Ensure that the Session always gets closed
        if (session != null) {
            try {
                session.close();
            } catch (JMSEException je) {
            }
        }
        // Ensure that the Connection always gets closed
        if (jmsCon != null) {
            try {
                jmsCon.close();
            } catch (JMSEException je) {
            }
        }
    }
    return messageID;
}
public String putByteMessage(byte[] bytes, String qcfName,
                             String putQueueName)
    throws JMSEException, NamingException {
    setQCF(qcfName);
    setPutQ(putQueueName);
    return put(bytes, TYPE_BYTE, qcf, putQueue);
}
public String putByteMessage(byte[] bytes, QueueConnectionFactory qcf,
                             Queue putQueue)
    throws JMSEException {
    return put(bytes, TYPE_BYTE, qcf, putQueue);
}
public String getTextMessage(String messageID, String selectorHeader,
                             int timeout, String qcfName, String getQueueName)
    throws JMSEException, NamingException {
    setQCF(qcfName);
    setGetQ(getQueueName);
    return getTextMessage(messageID, selectorHeader, timeout, qcf, getQueue);
}
}

```

JMSConnectHome.java

```

package casel;

/**
 * This is a Home interface for the Session Bean
 */
public interface JMSConnectHome extends javax.ejb.EJBHome {

    /**
     * create method for a session bean
     * @return casel.BestEffort
     * @exception javax.ejb.CreateException The exception description.
     * @exception java.rmi.RemoteException The exception description.
     */
    casel.JMSConnect create()
        throws javax.ejb.CreateException, java.rmi.RemoteException;
}

```

JMSConnect.java

```
package casel;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface JMSConnect extends javax.ejb.EJBObject {

/**
 *
 * @return javax.jms.Message or null if no message matched
 * @param messageID java.lang.String
 * @param selectorHeader java.lang.String
 * @param timeout int
 * @param qcfName java.lang.String
 * @param getQueueName java.lang.String
 * @exception String The exception description.
 * @exception String The exception description.
 * @exception String The exception description.
 */
    javax.jms.Message get(java.lang.String messageID,
        java.lang.String selectorHeader, int timeout,
        java.lang.String qcfName, java.lang.String getQueueName)
        throws javax.naming.NamingException, java.rmi.RemoteException,
        javax.jms.JMSEException;

/**
 *
 * @return javax.jms.Message or null if no message matched
 * @param messageID java.lang.String
 * @param selectorHeader java.lang.String
 * @param timeout int
 * @param qcf javax.jms.QueueConnectionFactory
 * @param getQueue javax.jms.Queue
 * @exception String The exception description.
 * @exception String The exception description.
 */
    javax.jms.Message get(java.lang.String messageID,
        java.lang.String selectorHeader, int timeout,
        javax.jms.QueueConnectionFactory qcf, javax.jms.Queue getQueue)
        throws java.rmi.RemoteException, javax.jms.JMSEException;

/**
 *
 * @return java.lang.String
 * @param messageID java.lang.String
 * @param selectorHeader java.lang.String
 * @param timeout int
 * @param qcfName java.lang.String
 * @param getQueueName java.lang.String
 * @exception String The exception description.
 * @exception String The exception description.
 * @exception String The exception description.
 */
    java.lang.String getTextMessage(java.lang.String messageID,
        java.lang.String selectorHeader, int timeout, java.lang.String qcfName,
        java.lang.String getQueueName)
        throws javax.naming.NamingException, java.rmi.RemoteException,
        javax.jms.JMSEException;

/**
 *
 * @return java.lang.String or null if no message matched
 * @param messageID java.lang.String
 * @param selectorHeader java.lang.String
 * @param timeout int
 */
}
```

```

* @param qcf javax.jms.QueueConnectionFactory
* @param getQueue javax.jms.Queue
* @exception String The exception description.
* @exception String The exception description.
*/
java.lang.String getTextMessage(java.lang.String messageID,
    java.lang.String selectorHeader, int timeout,
    javax.jms.QueueConnectionFactory qcf, javax.jms.Queue getQueue)
    throws java.rmi.RemoteException, javax.jms.JMSEException;
/**
*
* @return java.lang.String
* @param bytes byte []
* @param qcfName java.lang.String
* @param putQueueName java.lang.String
* @exception String The exception description.
* @exception String The exception description.
* @exception String The exception description.
*/
java.lang.String putByteMessage(byte [] bytes, java.lang.String qcfName,
    java.lang.String putQueueName)
    throws javax.naming.NamingException, java.rmi.RemoteException,
    javax.jms.JMSEException;
/**
*
* @return java.lang.String
* @param bytes byte []
* @param qcf javax.jms.QueueConnectionFactory
* @param putQueue javax.jms.Queue
* @exception String The exception description.
* @exception String The exception description.
*/
java.lang.String putByteMessage(byte [] bytes,
    javax.jms.QueueConnectionFactory qcf, javax.jms.Queue putQueue)
    throws java.rmi.RemoteException, javax.jms.JMSEException;
/**
*
* @return java.lang.String
* @param obj java.lang.Object
* @param qcfName java.lang.String
* @param putQueueName java.lang.String
* @exception String The exception description.
* @exception String The exception description.
* @exception String The exception description.
*/
java.lang.String putObjectMessage(java.lang.Object obj,
    java.lang.String qcfName, java.lang.String putQueueName)
    throws javax.naming.NamingException, java.rmi.RemoteException,
    javax.jms.JMSEException;
/**
*
* @return java.lang.String
* @param obj java.lang.Object
* @param qcf javax.jms.QueueConnectionFactory
* @param putQueue javax.jms.Queue
* @exception String The exception description.
* @exception String The exception description.
*/
java.lang.String putObjectMessage(java.lang.Object obj,
    javax.jms.QueueConnectionFactory qcf, javax.jms.Queue putQueue)
    throws java.rmi.RemoteException, javax.jms.JMSEException;
/**
*
* @return java.lang.String
* @param bytes byte []
* @param qcfName java.lang.String

```

```

    * @param putQueueName java.lang.String
    * @exception String The exception description.
    * @exception String The exception description.
    * @exception String The exception description.
    */
    java.lang.String putStreamMessage(byte [] bytes, java.lang.String qcfName,
        java.lang.String putQueueName)
        throws javax.naming.NamingException, java.rmi.RemoteException,
        javax.jms.JMSEException;
    /**
    *
    * @return java.lang.String
    * @param bytes byte []
    * @param qcf javax.jms.QueueConnectionFactory
    * @param putQueue javax.jms.Queue
    * @exception String The exception description.
    * @exception String The exception description.
    */
    java.lang.String putStreamMessage(byte [] bytes,
        javax.jms.QueueConnectionFactory qcf, javax.jms.Queue putQueue)
        throws java.rmi.RemoteException, javax.jms.JMSEException;
    /**
    *
    * @return java.lang.String
    * @param putText java.lang.String
    * @param qcfName java.lang.String
    * @param putQueueName java.lang.String
    * @exception String The exception description.
    * @exception String The exception description.
    * @exception String The exception description.
    */
    java.lang.String putTextMessage(java.lang.String putText,
        java.lang.String qcfName, java.lang.String putQueueName)
        throws javax.naming.NamingException, java.rmi.RemoteException,
        javax.jms.JMSEException;
    /**
    *
    * @return java.lang.String
    * @param putText java.lang.String
    * @param qcf javax.jms.QueueConnectionFactory
    * @param putQueue javax.jms.Queue
    * @exception String The exception description.
    * @exception String The exception description.
    */
    java.lang.String putTextMessage(java.lang.String putText,
        javax.jms.QueueConnectionFactory qcf, javax.jms.Queue putQueue)
        throws java.rmi.RemoteException, javax.jms.JMSEException;
}

```

DatagramBean.java

```

package casel;

import java.rmi.RemoteException;
import java.security.Identity;
import java.util.*;
import java.util.Properties;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import javax.sql.*;
import java.sql.*;

import javax.transaction.*;

```

```

/**
 * This is a Session Bean Class
 */
public class DatagramBean implements SessionBean {
    private javax.ejb.SessionContext mySessionCtx = null;
    final static long serialVersionUID = 3206093459760846163L;

/**
 * ejbActivate method comment
 * @exception java.rmi.RemoteException The exception description.
 */
public void ejbActivate() throws java.rmi.RemoteException {}

/**
 * ejbCreate method comment
 * @exception javax.ejb.CreateException The exception description.
 * @exception java.rmi.RemoteException The exception description.
 */
public void ejbCreate() throws javax.ejb.CreateException,
java.rmi.RemoteException {}

/**
 * ejbPassivate method comment
 * @exception java.rmi.RemoteException The exception description.
 */
public void ejbPassivate() throws java.rmi.RemoteException {}

/**
 * ejbRemove method comment
 * @exception java.rmi.RemoteException The exception description.
 */
public void ejbRemove() throws java.rmi.RemoteException {}

/**
 * getSessionContext method comment
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
    return mySessionCtx;
}

/**
 * Insert the method's description here.
 * Creation date: (14/03/2001 14:47:36)
 * @param message java.lang.String
 */
public void put(String message, String datasourceName, String qcfName,
                String queueName, boolean rollback) throws Exception
{
    InitialContext ic = new InitialContext();
    Object objref = ic.lookup("JMSConnect");
    JMSConnectHome home =
        (JMSConnectHome) javax.rmi.PortableRemoteObject.narrow(objref,
            JMSConnectHome.class);
    JMSConnect jmsBean = home.create();
    DataSource ds = (DataSource)ic.lookup(datasourceName);
    java.sql.Connection conn = ds.getConnection();
    PreparedStatement stmt = conn.prepareStatement("INSERT INTO IN_TRAY
        (RECEIVED,SOURCE,SUBJECT) VALUES(?, 'DBMQWrit',?)");
    stmt.setDate(1,new java.sql.Date(System.currentTimeMillis()));
    stmt.setString(2,message);

    stmt.executeUpdate();
    jmsBean.putTextMessage(message,qcfName,queueName);
    if (rollback) mySessionCtx.setRollbackOnly();
    stmt.close();
}

/**
 * setSessionContext method comment
 * @param ctx javax.ejb.SessionContext
 * @exception java.rmi.RemoteException The exception description.

```

```

*/
public void setSessionContext(javax.ejb.SessionContext ctx) throws
java.rmi.RemoteException {
    mySessionCtx = ctx;
}
/**
 * Insert the method's description here.
 * Creation date: (14/03/2001 14:47:36)
 * @param message java.lang.String
 */
public void put(String message, String datasourceName, String userid,
String password, String qcfName, String queueName,
boolean rollback) throws Exception
{
    InitialContext ic = new InitialContext();
    Object objref = ic.lookup("JMSConnect");
    JMSConnectHome home =
        (JMSConnectHome)javax.rmi.PortableRemoteObject.narrow(objref,
        JMSConnectHome.class);
    JMSConnect jmsBean = home.create();
    UserTransaction ut = mySessionCtx.getUserTransaction();
    DataSource ds = (DataSource)ic.lookup(datasourceName);
    java.sql.Connection conn = ds.getConnection(userid, password);
    ut.begin();
    PreparedStatement stmt = conn.prepareStatement("INSERT INTO IN_TRAY
        (RECEIVED,SOURCE,SUBJECT) VALUES(?, 'DBMQWrit',?)");
    stmt.setDate(1,new java.sql.Date(System.currentTimeMillis()));
    stmt.setString(2,message);

    stmt.executeUpdate();
    jmsBean.putTextMessage(message,qcfName,queueName);
    if (rollback)
        ut.rollback();
    else
        ut.commit();
//    mySessionCtx.setRollbackOnly();
    stmt.close();
}
}

```

DatagramHome.java

```

package casel;

/**
 * This is a Home interface for the Session Bean
 */
public interface DatagramHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
 * @return casel.TranDBMQWrite
 * @exception javax.ejb.CreateException The exception description.
 * @exception java.rmi.RemoteException The exception description.
 */
casel.Datagram create()
    throws javax.ejb.CreateException, java.rmi.RemoteException;
}

```

Datagram.java

```

package casel;

```

```

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface Datagram extends javax.ejb.EJBObject {

/**
 *
 * @return void
 * @param message java.lang.String
 * @param datasourceName java.lang.String
 * @param qcfName java.lang.String
 * @param queueName java.lang.String
 * @param rollback boolean
 * @exception String The exception description.
 * @exception String The exception description.
 */
void put(java.lang.String message, java.lang.String datasourceName,
        java.lang.String qcfName, java.lang.String queueName,
        boolean rollback)
        throws java.rmi.RemoteException, java.lang.Exception;

/**
 *
 * @return void
 * @param message java.lang.String
 * @param datasourceName java.lang.String
 * @param userid java.lang.String
 * @param password java.lang.String
 * @param qcfName java.lang.String
 * @param queueName java.lang.String
 * @param rollback boolean
 * @exception String The exception description.
 * @exception String The exception description.
 */
void put(java.lang.String message, java.lang.String datasourceName,
        java.lang.String userid, java.lang.String password,
        java.lang.String qcfName, java.lang.String queueName,
        boolean rollback)
        throws java.rmi.RemoteException, java.lang.Exception;
}

```

DatagramClient.java

```

import javax.naming.*;
import java.util.*;

import casel.*;
public class DatagramClient {

/**
 * DBMQWrite constructor comment.
 */
public DatagramClient() {
    super();
}

/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
    try
    {
        if (args.length < 7)
        {
            System.out.println("args <Message> <datasource> <ds_userid>
                <ds_passwd> <qcf> <q> <rollback set to true>");

```

```

    return;
}
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.ejs.ns.jndi.CNInitialContextFactory" );
// Replace localhost with the hostname of the server if necessary
env.put( Context.PROVIDER_URL, "iiop://localhost/" );

Context initial = new InitialContext(env);

// Lookup EJB Home object
Object objref = initial.lookup("casel/Datagram");

// Narrow to appropriate type
DatagramHome home =
    (DatagramHome)javax.rmi.PortableRemoteObject.narrow(objref,
        DatagramHome.class);
System.out.println("Putting a message to a queue and database then
    causing rollback." );
System.out.println(" " );

// Create the remote stub for the EJB
Datagram testBean = home.create();
TestBean.put(args[0],args[1],args[2],args[3],args[4],args[5],
    args[6].equalsIgnoreCase("TRUE"));
// Insert code to start the application here.
} catch (Exception ex)
{
    ex.printStackTrace();
}
}
}

```

Trademarks

The following are trademarks of IBM Corporation in the United States or other countries or both:

AIX

DB2

IBM

MQSeries

WebSphere

WebSphere Application Server

Microsoft, Windows, Windows NT and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Other company, product, and service names, may be trademarks or service marks of others.