11 MQ update

May 2000

In this issue

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: +44 1635 550955
e-mail: harryl@xephon.com

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: +1 303 410 9344
Fax: +1 303 438 0290

Contributions

Articles published in *MQ Update* are paid for at the rate of £170 ($250) per 1000 words and £90 ($140) per 100 lines of code. For more information about contributing an article, please check Xephon's Web site, where you can download *Notes for Contributors*.

MQ Update on-line

Code from *MQ Update* is available from Xephon's Web site at www.xephon.com/mqupdate.html (you'll need the user-id shown on your address label to access it). If you've a problem with your user-id or password call Xephon's subscription department on +44 1635 33886.

Editor

Harry Lewis

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.50) each including postage.

A mail system based on JMS and MQSeries

INTRODUCTION

This article describes the Java Message Service API (JMS), its interaction with JNDI (Java Naming and Directory Interface), and its use in conjunction with MQSeries Support Pack MA88. An example application is provided that demonstrates many of the point-to-point messaging features of the JMS.

WHAT IS JMS?

The Java platform, including its standard extensions, provides a wealth of building blocks, from basic networking to dynamic service location and distributed transaction services. Using these components, it's possible to construct sophisticated enterprise applications. The Java Message Service's purpose is to act as a model for an enterprise-quality messaging system used by Java-based clients. JMS thus specifies many of the features found in established message-oriented middleware products, even though its stated aim is not to be a substitute for the collection of features found in these products, but to provide an appropriate set of capabilities with which to implement sophisticated enterprise applications.

JMS encompasses both traditional point-to-point messaging and the new publish/subscribe model. It defines a set of Java interfaces and, as such, is not a product in its own right – it requires implementation in a program. Although it's possible to create a pure Java application that uses nothing but JMS for messaging, existing messaging middleware can easily be used as the basis for a JMS infrastructure, where the JMS layer interacts with an underlying messaging system either through JNI (Java Native Interface) methods or across the network. This article concentrates on using JMS with MQSeries for point-to-point messaging.

JMS primitives

JMS defines a number of object types that model the primitive elements of a messaging system. The most fundamental of them should be familiar to MQSeries users:

- A *Message* is the fundamental unit of data exchange between applications. Sub-types of *Message* include messages with particular types of content, such as *TextMessage, BytesMessage*, and *ObjectMessage*.
- A *Destination* is a store to which messages are addressed and from which they can be retrieved. A *Queue* is a sub-type of *Destination*.
- A *Connection* is an application's handle to the JMS provider, and a *Session* is a more limited context derived from a *Connection*.
- A *Provider* is the implementation that provides JMS services to clients.

When applications send messages, they interact with queues through *MessageProducers*, and when they receive messages, they interact through *MessageConsumers*. A *Connection* to a JMS provider is obtained from a *ConnectionFactory*. An interesting feature of JMS is the way that a factory is located by a client application.

JMS does not define an addressing scheme. The address of a *Destination* is effectively determined either by the naming scheme under which it is stored in the JNDI or by the provider-specific mechanism through which the *ConnectionFactory* and *Destination* objects are directly instantiated.

JMS-administered objects and JNDI

Although JMS is wholly dependent on the underlying messaging system, it is intended that JMS clients should require no direct knowledge of the messaging system's implementation-specific identity or properties. A JMS client's connection to a provider of JMS services

and its access to *Destination* objects is obtained through JMS-administered objects, which are, in turn, generally obtained from a look-up service via the JNDI.

The JNDI, like JMS itself, is an abstract definition of a look-up service for Java clients (not necessarily JMS clients – the JNDI is independent of JMS). Its function is to return a reference to a Java object from a name-keyed repository to a client. In common with JMS, JNDI uses service providers to provide a common interface to various underlying look-up service implementations. The two JNDI providers that are most likely to be used in conjunction with JMS use an LDAP directory and a filesystem store.

Instances of JMS-administered objects – *ConnectionFactory* and *Destination* – are created from what are effectively templates stored in the look-up service repository. The repository's records contain provider-specific attributes, including the Java class name from which to instantiate actual objects. JMS providers are expected to provide administration tools to create and maintain the administered object definitions (the *JMSAdmin* command-line tool is provided in the MQSeries JMS package).

Messages and message properties

Unlike MQSeries messages, JMS messages are inherently typed. JMS provides the structured message types *MapMessage*, *ObjectMessage*, and *StreamMessage*, which are sub-types of *Message*, in addition to the *TextMessage* and *BytesMessage* types. A *MapMessage* contains a set of randomly-accessible named fields, an *ObjectMessage* can be used to hold Java objects (a single container object can contain other objects in the message), and a *StreamMessage* contains a stream of Java data elements of primitive type (that is, simple numeric and string types). The *TextMessage* type is expected to be widely used for XML message data. A JMS provider handling messages originating from non-JMS sources is expected to handle these messages using the most appropriate JMS message type.

In addition to their content, JMS messages also have a header that contains a set of control fields. A small set of JMS-defined fields (or

properties) are present in every message, and provider- and application-defined properties can be added to them. Properties can be simple numeric types and strings. A powerful feature of JMS is its ability to filter messages using *selectors*, where messages are retrieved selectively based on field values using SQL expressions.

JMS client operation

JMS specifies the following steps for setting up a typical JMS client:

- Use JNDI to locate a *ConnectionFactory* object
- Use JNDI to find one or more *Destination* objects
- Use the *ConnectionFactory* to create a JMS *Connection*
- Use the *Connection* to create one or more *Sessions*
- Use a *Session* and *Destinations* to create *MessageProducers* and *MessageConsumers*, as required.

Once the message producers and consumers that are required are created, the client is then in a position to begin sending and receiving messages.

Security

JMS does not define a security model – JMS providers are expected to implement their own security schemes. Authentication may be performed when a *Connection* is obtained from a *ConnectionFactory*.

Transactional support

JMS *Sessions* can support transactions, though how this is done depends on the abilities of the provider. The choices are either to use the provider's own transactional support by calling methods in the *XASession* class or to use JTA's (Java Transaction API) transactional support by calling methods in the *Session* class.

RELATIONSHIP AND MAPPING OF JMS TO MQSERIES

For those familiar with MQSeries, there is a conveniently

straightforward mapping between most JMS features and MQSeries API features. Any awkwardness is more to do with the fact that MQI is procedural and JMS object-oriented than with differences between the underlying design of the two APIs. (A comparison with the MQSeries Java classes is easier to draw but perhaps not so useful.) Furthermore, the MQSeries JMS provider uses the MQSeries Java classes in its implementation. A comparison of MQSeries and JMS is shown in the following tables.

Objects and types:

MQSeries object or type	JMS object or property
Queue manager	ConnectionFactory
Queue	Destination or Queue
MQOD	Administered object (ConnectionFactory or Destination)
MQHCONN	Connection or Session
MQHOBJ	MessageProducer, MessageConsumer, and their subclasses
MQGMO	Attributes of Session or MessageConsumer, arguments of MessageConsumer's methods
MQPMO	Attributes of Session or MessageProducer, arguments of MessageProducer's methods
MQMD and other header types	Attributes of Message and its subclasses

Functions and methods

MQSeries function	JMS equivalent
MQCONN	The acquisition or instantiation of a ConnectionFactory, Connection, and Session
MQOPEN	Session methods that create instances of MessageProducer and MessageConsumer
MQPUT/MQPUT1	MessageProducer send methods
MQGET	MessageConsumer receive methods
MQCMIT	Session commit method
MQBACK	Session roll-back method

MQCLOSE	MessageProducer's or MessageConsumer's close methods
MQDISC	Connection's close method
MQINQ/MQSET	Provider-specific methods of administered objects

Message properties

MQMD field	JMS message property	Set in JMS by:
MsgType	JMS_IBM_MsgType	IBM JMS provider
Expiry	JMSExpiration	Send method
Format	JMS_IBM_Format	IBM JMS provider
Priority	JMSPriority	Send method
Persistence	JMSDeliveryMode	Send method
MsgId	JMSMessageID	Send method
CorrelId	JMSCorrelationID	Send method
BackoutCount	JMSRedelivered/JMSXDeliveryCount	Provider
UserIdentifier	JMSXUserID	Provider
PutApplType	JMS_IBM_PutApplType	IBM JMS provider
PutApplName	JMSXAppID	Provider
PutDate/PutTime	JMSTimestamp	Provider
GroupId	JMSXGroupID	Client
MsgSeqNumber	JMSXGroupSeq	Client

JMS messages also contain a record of their destination, storing the destination in the *JMSDestination* property.

USING JMS WITH MQSERIES: AN EXAMPLE APPLICATION

About the example

Here we provide an example JMS application, JMSMail, which is a simple mail system with a user interface loosely based on the simple mail client found on many Unix systems. The application demonstrates the fundamental features of JMS's point-to-point messaging, including:

- The use of JNDI to obtain references to administered objects
- Connection and session creation
- The use of message producers and consumers
- Message properties and selectors.

Pre-requisites and environment

The MQSeries JMS provider classes are supplied in the MQSeries MA88 support pack, which is available for free download from the IBM MQSeries for Windows NT, AIX, HP-UX, and Solaris Web sites. MQSeries 5.1 and JDK 1.1.6 or later are required (the requirement is JDK 1.1.7 or later on HP-UX). The MQSeries 5.1.1 Java classes, which are also required, are supplied in the support pack. The description of the example application that follows assumes that these components have been installed.

The example code has been tested and used successfully on two platforms:

- AIX 4.1.5 with MQSeries 5.0 (this worked despite the stated requirement for MQSeries 5.1) and JDK 1.1.6
- Microsoft Windows NT 4.0 with MQSeries 5.1 and JDK 1.2.2.

Configuring the MQSeries JMS environment for use

There are three basic steps required to set up the environment to run MQSeries JMS applications:

- The *classpath* parameter must include classes in the MQSeries, JMS, and JNDI archives.
- To enable JNDI to look up JMS-administered objects, a JNDI-accessible repository must be configured and populated with one or more *ConnectionFactory* and *Destination* entries.
- Any MQSeries objects that are referred to by JMS-administered object definitions must have definitions in MQSeries and be available.

Setting the classpath

To run any MQSeries JMS program, it is necessary that the classes in the following MA88 *.jar* archives are accessible via *classpath* (this is in addition to the base JDK classes):

- *com.ibm.mq.jar*
- *com.ibm.mqbind.jar*
- *com.ibm.mqjms.jar*
- *jms.jar*
- *jndi.jar*
- *fscontext.jar*
- *ldap.jar*
- *providerutil.jar*.

These archives are located in subdirectory *java/lib* (or *java\lib*) of the MQSeries MA88 installation.

Defining JMS-administered objects

JMS-administered object definitions are created using the JMSAdmin command-line administration tool, which must first be configured by editing the *JMSAdmin.config* configuration file. Two entries in this file should be edited: *INITIAL_CONTEXT*, which specifies the JNDI service provider's class name and determines the mechanism used to access the repository, and *PROVIDER_URL*, which is the URL of the repository.

The easiest type of repository to get up and running is a filesystem-based repository; the provider class name of this type of repository is *com.sun.jndi.fscontext.RefFSContextFactory* and the provider URL is a file-based URL. As an example, the following entries can be used:

```
INITIAL_CONTEXT=com.sun.jndi.fscontext.RefFSContextFactory
PROVIDER_URL=file:/usr/mqm/java/directory
```

The JMSAdmin tool and the properties file are found in the *java/bin* (or *java\bin*) sub-directory of the MQSeries MA88 installation. To

run JMSMail, we need definitions for:

- A queue connection factory
- One or more destinations corresponding to mail system users' mailboxes.

To create a queue connection factory called *jmsmail*, which corresponds to the local queue manager *LYCHEE.QM*, and destinations *chris* and *bob*, which correspond to MQSeries queues named *JMSMAIL.CHRIS* and *JMSMAIL.BOB* respectively, you would issue the following JMSAdmin commands ('*InitCtx>*' is the prompt):

```
InitCtx> define qcf(JMSMAIL) qmanager(LYCHEE.QM)
InitCtx> define q(chris) queue(JMSMAIL.CHRIS)
InitCtx> define q(bob) queue(JMSMAIL.BOB)
```

Here, the queue's connection factory definition refers to a local queue manager by name, and this results in the use of MQSeries Java bindings. Alternatively, a queue connection factory can be defined as an MQSeries client connection as follows:

```
InitCtx> define qcf(JMSMAIL) hostname(lychee) port(1414)
channel(CLIENT) + transport(client)
```

The transport type *client* is required in this case to indicate that this should be a client connection. For client connections, an MQSeries channel of type *SVRCONN* must be defined on the target queue manager and either an *inetd* entry (Unix) or active listener must exist on the specified port of the target host.

Creating MQSeries definitions

Finally, the real MQSeries local queues *JMSMAIL.CHRIS* and *JMSMAIL.BOB* must be defined on the queue manager *LYCHEE.QM*.

Invocation scripts

Because of the environmental pre-requisites for running a JNDI/JMS application, it's often convenient to create a shell script or batch file to invoke the application with the correct configuration parameters. A simple Korn shell script *jmsmail* (for Unix environments) to invoke JMSMail would look like this:

JMSMAIL

```
#!/bin/ksh

JMS_HOME=/usr/mqm/java
JMS_LIB=$JMS_HOME/lib
JMS_CLASSPATH=/usr/jdk_base/lib/classes.zip:\
$JMS_LIB:\
$JMS_LIB/com.ibm.mqjms.jar:\
$JMS_LIB/jms.jar:\
$JMS_LIB/jndi.jar:\
$JMS_LIB/fscontext.jar:\
$JMS_LIB/ldap.jar:\
$JMS_LIB/providerutil.jar:\
$JMS_LIB/com.ibm.mq.jar:\
$JMS_LIB/com.ibm.mqbind.jar:.
FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
REPOSITORY=file:$JMS_HOME/directory

echo Starting JMSMail...

java -classpath $JMS_CLASSPATH \
-Djava.naming.factory.initial=$FACTORY \
-Djava.naming.provider.url=$REPOSITORY \
JMSMail $@
```

The equivalent Windows batch file, *jmsmail.bat*, is shown below. Note the use of the continuation character, '➤', to indicate a formatting line break that's not present in the original code.

JMSMAIL.BAT

```
@echo off

SET JMS_HOME=\MQSeries\java
SET JMS_LIB=%JMS_HOME%\lib
SET JMS_CLASSPATH=%JMS_LIB%;%JMS_LIB%\com.ibm.mqjms.jar;
➤  %JMS_LIB%\jms.jar;%JM S_LIB%\jndi.jar;%JMS_LIB%\fscontext.jar;
➤  %JMS_LIB%\ldap.jar;%JMS_LIB%\prov iderutil.jar;
➤  %JMS_LIB%\com.ibm.mq.jar;%JMS_LIB%\com.ibm.mqbind.jar;.
SET FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
SET REPOSITORY=file:/MQSeries/mqm/java/directory

echo Starting JMSMail...

java -classpath %JMS_CLASSPATH% -Djava.naming.factory.initial=
➤  %FACTORY% Djava.naming.provider.url=%REPOSITORY% JMSMail
➤  %1 %2 %3 %4 %5
```

The JMSMail code

This section discusses the code of the sample application. For clarity, we refer to the code using line numbers (the entire source code is listed at the end of the article, including line numbers). However, remember to strip out line numbers before compiling the code.

The JMSMail application is designed as a single class, *JMSMail* (the class definition begins at line 11), which implements two JMS listener interfaces. An instance of *JMSMail* maintains one JMS provider connection and one session, which is used both for sending and receiving messages. The mailbox queue is where received messages are retrieved. A hash table (line 22) is used to build a map of message indices to message identifiers, which is then used for selecting specific messages, while the *BufferedReader* wrapper (line 23), which wraps the standard input stream, allows us to use the *readLine* method to fetch user input a line at a time. The *javax.naming* and *javax.jms* import statements (lines 3 and 4) allow us to refer to the JNDI and JMS classes respectively.

The *main* method begins at line 35 and ends at line 97. It first initializes the JNDI context using the Java system properties (line 41), and then instantiates an object of the *JMSMail* class (line 48). The *JMSMail* instance has a user name associated with it that is taken either from the *jmsmail.user.name* property (if defined) or the user's login name.

What happens next depends on whether any addressee names were included in the command line – if any were, the application runs in 'send mode', calling the *composeMessage* method (line 55) to build a JMS message from the user's input and attempting to send it to each addressee in turn. If no command line arguments were supplied, the application runs in 'receive mode', responding to a small set of commands that are used to view and manipulate received messages.

When displaying *JMSExceptions*, it's useful to see the underlying exception (if present), as this may give a direct insight into the underlying provider problem. This is accessed with the *getLinkedException* method in *JMSException* (lines 84 and 85).

The *getConnection* (lines 103 to 110) and *getDestination* (lines 116 to 120) methods are 'convenience' methods that use the JNDI's context

look-up and return references to administered objects.

The *JMSMail* constructor (lines 126 to 140) establishes a connection to the JMS provider. It uses the connection to open both a session and the mailbox queue corresponding to the user. Although the application only accesses the mailbox queue in receive mode, the *composeMessage* method also needs a reference to the queue to set the *JMSReplyTo* property in messages sent.

The compose method (lines 148 to 191) creates a JMS *TextMessage* instance from text entered on the command line and sets the *JMSReplyTo* message property to refer to the sender's mailbox. Two further application-defined properties – *from* and *subject* – are set. This method is also used by the *r* (reply) command in receive mode.

The *sendMessage* method (lines 197 to 217) obtains a *Queue* reference using the addressee's name and then uses a *QueueSender* to send the message. The *QueueSender* is explicitly closed (line 216) after it's used so as to free any resources allocated to it by the provider.

In receive mode, the application keeps a record of message identifiers of listed messages (see lines 224 to 238, which shows the method used to find the *JMSMessageID* of a message in the mailbox). These are stored in the hash table *table* and are accessed by the receive commands that are used to process individual messages. The hash table is populated by the *listMessages* method, which returns a 'newline'-delimited string describing the messages in the mailbox.

A *QueueBrowser* is used in the *listMessages* method (lines 246 to 280) to provide a snapshot of the messages in a queue. Each line in the string returned contains an index number, starting with index number 1. The sender is identified by the application-defined *from* property, while the timestamp is taken from the message's *JMSTimestamp* property and the subject is read from the application-defined *subject* property. As with message producers and consumers, the browser is explicitly closed (line 277) after use in order to free any resources allocated to it by the provider. For convenience, the message count is appended as the last line of the returned string.

To obtain a browse copy of an individual message (lines 287 to 307), we again use a *QueueBrowser*. We could use a selector to limit the

messages available from the browser to just ones with the correct message ID (as in the *receiveMessage* method in lines 314 to 325), though we would still have to use the browser's *getEnumeration* method to access the final message. In this case, it requires no more work to look for the message explicitly without a selector.

To delete a message, we need to use a *QueueReceiver* – a subclass of *MessageConsumer*. The selector used is a filter that ensures that only messages matching the specified message identifier are returned. When we call the method to receive the message in line 320, we know that the message is already on the queue, so we can use the queue receiver's *receiveNoWait* method. The *MessageConsumer* class also defines receive methods that block until a suitable message arrives.

The JMSMail application implements the JMS *MessageListener* interface, which defines one method, *onMessage* (lines 333 to 350). A *MessageListener* is associated with a *MessageConsumer*, and the application registers itself as a listener when it enters receive mode in the *run* method (lines 371 to 461). The invocation of the *onMessage* method serves as notification that a message has arrived, though the message itself is passed on directly, and so is removed from the queue by the *MessageConsumer* before the method is called. This is unfortunate from the point of view of the JMSMail application, as the new mail message is effectively retrieved as soon as it arrives and is, therefore, not available in the mailbox queue. However, this method serves to illustrate the way a MessageListener works.

The JMSMail application also implements the JMS *ExceptionListener* interface and registers itself as an exception listener in its constructor. The *onException* method (lines 358 to 363) provides a way for JMS to notify the application if a problem occurs with the JMS environment independently of the application's actions. JMS exceptions raised directly as a result of the application's invocation of JMS methods are caught in the normal way.

The run method (lines 371 to 461) executes a set of simple actions on the user's JMSMail mailbox. The commands available to the user are:

n Show message number *n* (sets index to *n*)

l List messages (resets index)

d Delete message at current index

r Reply to message at current index

s Send new message

q Quit.

The message list produced by the *l* command shows index numbers in the left-hand column. The message index maintained is the number of the last message displayed; the *d* and *r* commands act on the message identified by this index. Hitting the *Enter* key at a blank line displays the message at the current index and increments the index by one.

The first thing the JMSMail application does in the *run* method is set up a *QueueReceiver* and associate itself with it as a *MessageListener*. This requires a new *Session*, as a single session cannot support synchronous receive operations (which are performed in the *receiveMessage* method) and asynchronous message delivery using a message listener at the same time. The *QueueReceiver* is established with a selector based on the current date and time. This ensures that only newly-arrived messages are passed to the listener. It is necessary to cast the time value to type *int* (line 374) to avoid a *NumberFormatException* in the SQL parser, which does not accept values of type *long*.

Before a *Connection* can deliver messages to consumers, its *start* method must be called (line 385). The command loop that follows (lines 387 to 457) then processes one command per iteration until either the user quits by issuing the *q* command or a *JMSException* is thrown.

The *run* method uses the *help* method (lines 467 to 47), which returns a list of available commands when it's invoked.

The *disconnect* method (lines 482 to 486) releases the connection to the JMS provider.

Finally, the last line of the code, line 487, is a closing brace that indicates the end of the class enclosure.

Using JMSMail to send a message

To send a message, invoke the *jmsmail* script or the *jmsmail.bat* batch file (both were described above) with the name of the destination user as a parameter. Pre-requisites for this to work are that mailboxes belonging to both the sender and the recipient must exist as JNDI entries and MQSeries queues, that a JNDI entry for the queue manager must already be defined, and that the queue manager must be available. A command sequence for sending a message is shown below.

```
$ jmsmail chris
Starting JMSMail...
Subject: This is JMS
Chris, this is a JMS message!
.
$
```

Using JMSMail in receive mode

To handle JMSMail messages received, invoke the *jmsmail* script or batch file with no arguments. For instance, as user *chris*, my response to the above message being sent would be:

```
$ jmsmail
Starting JMSMail...
1  chris  Mon Mar 06 18:02:12 GMT 2000 This is JMS
1 message.
?
```

To see the message, press *Enter* or type *1* (the message index). The message is shown, displaying its properties and the message text:

```
JMS Message class: jms_text
  JMSType:         null
  JMSDeliveryMode: 2
  JMSExpiration:   0
  JMSPriority:     4
  JMSMessageID:    ID:414d51204c59434845452e514d20202038c3de6e00
                   ➤  003013
  JMSTimestamp:    952365732600
  JMSCorrelationID:null
  JMSDestination:  queue:///JMSMAIL.CHRIS
  JMSReplyTo:      queue:///JMSMAIL.CHRIS
  JMSRedelivered:  false
  JMS_IBM_Format:MQSTR
```

```
  JMS_IBM_PutApplType:6
  JMSXDeliveryCount:1
  JMS_IBM_MsgType:1
  subject:This is JMS
  from:chris
  JMSXUserID:chris
  JMSXAppID:
Chris, this is a JMS message!

?
```

For a list of other valid commands, type *help* (or anything that's not a valid command).

ACCESSING MQSERIES ADMINISTERED OBJECTS WITHOUT JNDI

Our sample application happens to be provider-independent, as the JNDI provides access to the JMS provider without any reference to its MQSeries-specific properties. However, in cases where a JNDI repository is not available, it's possible to directly instantiate both *ConnectionFactory* and *Destination* objects that are provider-specific.

To make our sample application use MQSeries-specific administered objects, we'd need to make the following three changes to the code:

1 Add an *import* statement that allows us to refer to classes in the MQSeries JMS package:

```
import com.ibm.mq.jms.*;
```

2 Replace our *getConnection* method with the following:

```
static QueueConnection getConnection (String name)
           throws JMSException
{
   MQQueueConnectionFactory factory =
      new MQQueueConnectionFactory ();

   factory.setQueueManager (name);

   return factory.createQueueConnection ();
}
```

3 Replace the *getDestination* method with the following:

```
static Queue getDestination (String name) throws JMSException
{
```

```
        return session.createQueue (name);
    }
```

Note that these replacement methods do not throw an exception of type *NamingException*, as no JNDI methods are invoked.

The JNDI's value in de-coupling JMS object names from real object names becomes apparent: by making the above changes, we now need to match the factory and destination names used by JMSMail with real MQSeries objects. That is, we need to:

- Either define a queue manager called *jmsmail* or use the *jmsmail.server.name* property to refer to an existing queue manager
- Define MQSeries queues or queue aliases matching user names used by JMSMail.

For completeness, as all JNDI-related code is now removed from the program, the *InitialContext* need not be initialized.

RESOURCES

- JMS information:

  http://java.sun.com/products/jms/index.html

- JNDI information:

  http://java.sun.com/products/jndi/index.html

- MQSeries JMS implementation (support pack MA88):

  http://www.ibm.com/software/ts/mqseries/txppacs/ma88.html

- MQSeries home page:

  http://www.ibm.com/software/ts/mqseries/

JMSMAIL.JAVA

```
1 import java.io.*;
2 import java.util.*;
3 import javax.naming.*;
4 import javax.jms.*;
5
6 /**
```

```
 7 * JMS sample application modelling a simple mail system. The
 8 * user interface is loosely based on Unix's common mail client.
 9 */
10
11 public class JMSMail implements MessageListener, ExceptionListener
12 {
13    final static String NEWLINE =
14       File.separatorChar == '\\' ? "\r\n" : "\n";
15
16    static InitialContext context;
17
18    String username;
19    QueueConnection connection;
20    QueueSession session;
21    Queue mailbox;
22    Hashtable table = new Hashtable ();
23    BufferedReader input = new BufferedReader
24       (new InputStreamReader (System.in));
25
26    /**
27     * Main application's entry point. If any command-line arguments
28     * are supplied, the application runs in send mode, so that a
29     * message is composed and sent to each of the specified
30     * addressees in turn. With no command-line arguments, the
31     * application runs in receive mode and manipulates received
32     * messages in the user's mailbox using simple commands.
33     */
34
35    public static void main (String [] args)
36    {
37       try
38       {
39          // Set up the JNDI context.
40
41          context = new InitialContext (System.getProperties ());
42
43          // The username identifies the 'from' address to use when
44          // sending and the mailbox (queue) to use when receiving.
45
46          String username = System.getProperty ("jmsmail.user.name",
47             System.getProperty ("user.name"));
48          JMSMail app = new JMSMail (username);
49
50          if (args.length > 0)
51          {
52             // Compose a message and send it to each addressee
53             // specified on the command line.
54
55             Message message = app.composeMessage (null);
```

```
56
57             for (int i = 0; i < args.length; i++)
58             {
59                try
60                {
61                   app.sendMessage (message, args [i]);
62                }
63
64                catch (NamingException ne)
65                {
66                   System.err.println (args [i] +
67                      ": JNDI lookup error: " + ne);
68                }
69             }
70          }
71          else
72          {
73             // Run in receive mode operating on messages in the
74             // user's mailbox.
75
76             app.run ();
77          }
78
79          app.disconnect ();
80       }
81
82       catch (JMSException jmse)
83       {
84          System.err.println ("JMS error: " + jmse +
85             " (" + jmse.getLinkedException () + ')');
86       }
87
88       catch (NamingException ne)
89       {
90          System.err.println ("Invalid destination: " + ne);
91       }
92
93       catch (IOException ioe)
94       {
95          System.err.println ("Error reading input: " + ioe);
96       }
97    }
98
99    /**
100    * Returns a connection using a named ConnectionFactory.
101    */
102
103   static QueueConnection getConnection (String name) throws
104      JMSException, NamingException
```

```
105    {
106       QueueConnectionFactory factory =
107          (QueueConnectionFactory) context.lookup (name);
108
109       return factory.createQueueConnection ();
110    }
111
112    /**
113     * Returns a reference to a named queue.
114     */
115
116    static Queue getDestination (String name) throws
117       JMSException, NamingException
118    {
119       return (Queue) context.lookup (name);
120    }
121
122    /**
123     * Constructor for a JMSMail client (sender or receiver).
124     */
125
126    public JMSMail (String username) throws
127       JMSException, NamingException
128    {
129       this.username = username;
130
131       // We normally use ConnectionFactory 'jmsmail', though this
132       // can be overridden using the jmsmail.server.name property.
133
134       connection = getConnection (System.getProperty
135          ("jmsmail.server.name", "jmsmail"));
136       connection.setExceptionListener (this);
137       session = connection.createQueueSession
138          (false, Session.AUTO_ACKNOWLEDGE);
139       mailbox = getDestination (username);
140    }
141
142    /**
143     * Compose a JMS message from the command line. Text entry is
144     * terminated either by an EOF (CTRL-D) or a line containing
145     * only a period ('.').
146     */
147
148    public Message composeMessage (String subject) throws
149       IOException, JMSException
150    {
151       TextMessage message = session.createTextMessage ();
152       StringBuffer buffer = new StringBuffer ();
153       String line;
```

```
154
155         try
156         {
157            System.out.print ("Subject: ");
158
159            if (subject == null)
160            {
161               subject = input.readLine ();
162            }
163            else
164            {
165               System.out.println (subject);
166            }
167
168            while ((line = input.readLine ()) != null &&
169               !line.equals ("."))
170            {
171               buffer.append (line);
172               buffer.append (NEWLINE);
173            }
174         }
175
176         catch (EOFException eofe)
177         {
178            ;
179         }
180
181         message.setText (new String (buffer));
182
183         // Set message properties; 'from' and 'subject' are
184         // application-defined.
185
186         message.setJMSReplyTo (mailbox);
187         message.setStringProperty ("from", username);
188         message.setStringProperty ("subject", subject);
189
190         return message;
191      }
192
193      /**
194       * Send a JMS message to a named destination.
195       */
196
197      public void sendMessage (Message message, String destination)
198         throws JMSException, NamingException
199      {
200         Queue queue;
201         QueueSender sender;
202
```

```
203         try
204         {
205            queue = getDestination (destination);
206         }
207
208         catch (NullPointerException npe)
209         {
210            throw new NamingException ("Invalid destination: " +
211               destination);
212         }
213
214         sender = session.createSender (queue);
215         sender.send (message);
216         sender.close ();
217      }
218
219      /**
220       * Finds the JMSMessageID property of the message in the
221       * mailbox with the specified index (in receive mode).
222       */
223
224      String getMessageID (int index) throws
225         ArrayIndexOutOfBoundsException
226      {
227         String messageID = (String) table.get (new Integer (index));
228
229         if (messageID == null)
230         {
231            throw new ArrayIndexOutOfBoundsException
232               ("Invalid message number " + index + '.');
233         }
234         else
235         {
236            return messageID;
237         }
238      }
239
240      /**
241       * Return a string listing the sender, timestamp, and subject
242       * along with the index of the messages in the mailbox
243       * (receive mode).
244       */
245
246      public String listMessages () throws JMSException
247      {
248         QueueBrowser browser = session.createBrowser (mailbox);
249         Enumeration e = browser.getEnumeration ();
250         StringBuffer sb = new StringBuffer ();
251         int i = 0;
```

```
252
253         synchronized (table)
254         {
255            table.clear ();
256
257            while (e.hasMoreElements ())
258            {
259               Message message = (Message) e.nextElement ();
260               String subject = message.getStringProperty ("subject");
261
262               sb.append (++i);
263               sb.append ("  ");
264               sb.append (message.getStringProperty ("from"));
265               sb.append ("  ");
266               sb.append (new Date (message.getJMSTimestamp ()));
267               sb.append ("  ");
268               sb.append (subject == null ? "" : subject);
269               sb.append (NEWLINE);
270
271               table.put (new Integer (i), message.getJMSMessageID ());
272            }
273         }
274
275         sb.append (i);
276         sb.append (i == 1 ? " message." : " messages.");
277         browser.close ();
278
279         return new String (sb);
280      }
281
282      /**
283       * Returns the message with the specified JMSMessageID from the
284       * mailbox (receive mode).
285       */
286
287      public Message browseMessage (String messageID) throws
288         JMSException
289      {
290         QueueBrowser browser = session.createBrowser (mailbox);
291         Enumeration e = browser.getEnumeration ();
292         Message message = null;
293
294         while (e.hasMoreElements () && message == null)
295         {
296            message = (Message) e.nextElement ();
297
298            if (!message.getJMSMessageID ().equals (messageID))
299            {
300               message = null;
```

```
301          }
302       }
303
304       browser.close ();
305
306       return message;
307    }
308
309    /**
310     * Returns the message with the specified JMSMessageId from
311     * the mailbox, deleting it from the queue.
312     */
313
314    public Message receiveMessage (String messageID) throws
315       JMSException
316    {
317       String selector = "JMSMessageID='" + messageID + '\'';
318       QueueReceiver receiver = session.createReceiver
319          (mailbox, selector);
320       Message message = receiver.receiveNoWait ();
321
322       receiver.close ();
323
324       return message;
325    }
326
327    /**
328     * This MessageListener method is called when a new message is
329     * received. The message is passed directly, and is not retained
330     * in the mailbox queue.
331     */
332
333    public void onMessage (Message message)
334    {
335       System.out.print ((char) 0x07);   // beep
336
337       try
338       {
339          System.out.println ("\nNew mail from " +
340             message.getStringProperty ("from").trim () + ": ");
341       }
342
343       catch (JMSException jmse)
344       {
345          System.out.println ("\nNew mail: ");
346       }
347
348       System.out.println (message);
349       System.out.print ("? ");
```

```
350    }
351
352    /**
353     * This ExceptionListener method is called if a problem occurs
354     * with the connection to the JMS service provider while we're
355     * not looking.
356     */
357
358    public void onException (JMSException jmse)
359    {
360       System.err.println ("JMS connection error: " + jmse +
361          " (" + jmse.getLinkedException () + ')');
362       System.exit (-1);
363    }
364
365    /**
366     * Runs this JMSMail instance as a receiver, reading mailbox
367     * commands from the command line. For a list of commands,
368     * see the help method.
369     */
370
371    public void run () throws JMSException, IOException
372    {
373       String selector = "JMSTimestamp > " +
374          (int) System.currentTimeMillis ();
375       QueueSession listenerSession = connection.createQueueSession
376          (false, Session.AUTO_ACKNOWLEDGE);
377       QueueReceiver listenerReceiver =
378          listenerSession.createReceiver (mailbox, selector);
379       String command;
380       int index = 0;
381       boolean quit = false;
382
383       listenerReceiver.setMessageListener (this);
384       System.out.println (listMessages ());
385       connection.start ();
386
387       do
388       {
389          System.out.print ("? ");
390
391          try
392          {
393             if ((command = input.readLine ().trim ()).equals (""))
394             {
395                System.out.println
396                   (browseMessage (getMessageID (++index)));
397             }
398             else if (command.equals ("l"))
```

```
399             {
400                System.out.println (listMessages ());
401                index = 0;
402             }
403             else if (command.equals ("d"))
404             {
405                receiveMessage (getMessageID (index));
406                System.out.println
407                   ("Message " + index++ + " deleted.");
408             }
409             else if (command.equals ("r"))
410             {
411                Message message = browseMessage (getMessageID
                                                 ➤  (index));
412
413                sendMessage (composeMessage ("Re: " +
414                   message.getStringProperty ("subject")),
415                      message.getStringProperty ("from"));
416                System.out.println ("Reply sent.");
417             }
418             else if (command.equals ("s"))
419             {
420                System.out.print ("To: ");
421
422                String to = input.readLine ();
423
424                sendMessage (composeMessage (null), to);
425                System.out.println ("Message sent.");
426             }
427             else if (command.equals ("q"))
428             {
429                quit = true;
430             }
431             else
432             {
433                try
434                {
435                   index = Integer.parseInt (command);
436                   System.out.println
437                      (browseMessage (getMessageID (index)));
438                }
439
440                catch (NumberFormatException nfe)
441                {
442                   System.out.println (help ());
443                }
444             }
445          }
446
```

```
447          catch (ArrayIndexOutOfBoundsException abe)
448          {
449             System.out.println (abe.getMessage ());
450          }
451
452          catch (NamingException ne)
453          {
454             System.out.println ("Invalid destination");
455          }
456       }
457       while (!quit);
458
459       listenerReceiver.close ();
460       listenerSession.close ();
461    }
462
463    /**
464     * Displays a list of valid commands.
465     */
466
467    public String help ()
468    {
469       return "Commands: " +
470          "\n\t<n>: show message number n (sets index to n)" +
471          "\n\tl: list messages (resets index)" +
472          "\n\td: delete message at current index" +
473          "\n\tr: reply to message at current index" +
474          "\n\ts: send new message" +
475          "\n\tq: quit";
476    }
477
478    /**
479     * Terminates the connection to the JMS service provider.
480     */
481
482    public void disconnect () throws JMSException
483    {
484       session.close ();
485       connection.close ();
486    }
487 }
```

Chris Markes
HCI Architect
IBM UK Laboratories (UK)

An event queue monitor for OS/390 – part 2

This month's instalment concludes this article on an event queue monitor for MQ Series on OS/390.

MQSEVT.C (CONTINUED)

```
/*--------------------------------------------------------------------*/
/*   Finds handler for an event with MQCFH.Reason == Reason.          */
/*--------------------------------------------------------------------*/
EveMsgHandler*   FindEveMsgHandler (MQLONG Reason)
{
    static EveMsgHandler TabEveMsgHandler [] = {
                { MQRC_CHANNEL_STOPPED,         ChannelStopped       },
                { MQRC_Q_SERVICE_INTERVAL_HIGH, QServiceIntervalHigh }
    };
    int Count;
    int TotEveMsgHandler = sizeof (TabEveMsgHandler) / sizeof
                                                      (EveMsgHandler);
    for  (Count = 0; Count < TotEveMsgHandler; ++Count)
        if  (TabEveMsgHandler[Count].Reason == Reason)
            return (&TabEveMsgHandler[Count]);

    return ((EveMsgHandler*) NULL);
}

/*--------------------------------------------------------------------*/
/*   Handle events where MQCFH.Reason == MQRC_CHANNEL_STOPPED.        */
/*--------------------------------------------------------------------*/
void    ChannelStopped (
            MQHCONN hConn,
            MQMD dEveMsg,
            PMQBYTE EveMsg,
            long Action
        )
{
    MQLONG  FieldIds [] = {
                MQIACF_REASON_QUALIFIER,
                MQCACH_CHANNEL_NAME,
                MQIACF_ERROR_IDENTIFIER,
                MQIACF_AUX_ERROR_DATA_INT_1,
            };

    MQLONG  ReasonQualifier;
    MQCHAR  ChannelName [ MQ_CHANNEL_NAME_LENGTH + 1 ];
```

```
MQLONG  ErrorIdentifier;
MQLONG  AuxErrorDataInt1;

char    Msg [ MSG_LEN ];
char    cErrorIdentifier [ MSG_LEN ];
int     MsgSuffix;

char    Cmd [ CMD_LEN ];

/*------------------------------------------------------------*/
/*  Get fields from the event message.                        */
/*------------------------------------------------------------*/
ReasonQualifier = 0;
ChannelName [0] = '\0';
ErrorIdentifier = 0;
AuxErrorDataInt1 = 0;

GetField (
    EveMsg,
    FieldIds,
    sizeof (FieldIds) / sizeof (MQLONG),
    &ReasonQualifier,
    ChannelName,
    &ErrorIdentifier,
    &AuxErrorDataInt1
);

/*------------------------------------------------------------*/
/*  Gets channel failure message suffix from ErrorIdentifier,  */
/*  which has format xxxxxSSS (hexadecimal), with suffix SSS.  */
/*------------------------------------------------------------*/
sprintf (cErrorIdentifier, "%lx", ErrorIdentifier);
sscanf (
    cErrorIdentifier + strlen (cErrorIdentifier) - 3,
    "%d",
    &MsgSuffix
);

/*------------------------------------------------------------*/
/*  Disconnect interval expired - ignores.                    */
/*------------------------------------------------------------*/
if  (ReasonQualifier == MQRQ_CHANNEL_STOPPED_OK)
    return;

/*------------------------------------------------------------*/
/*  Connection reset by peer - ignores.                       */
/*------------------------------------------------------------*/
if  (AuxErrorDataInt1 == 0x36)
    return;
```

```
    /*-------------------------------------------------------------*/
    /*  Channel is already active - ignores.                       */
    /*-------------------------------------------------------------*/
    if  (ReasonQualifier == MQRQ_CHANNEL_STOPPED_ERROR &&
          MsgSuffix == 514)
        return;

    /*-------------------------------------------------------------*/
    /*  Message sequence error:                                    */
    /*      Channel reset                                          */
    /*      Channel start.                                         */
    /*-------------------------------------------------------------*/
    if  (ReasonQualifier == MQRQ_CHANNEL_STOPPED_ERROR &&
          MsgSuffix == 506)   {

        /* RESET CHANNEL */
        sprintf (Cmd, "RESET CHANNEL(%s)", ChannelName);
        RunCommand (hConn, Cmd);

        /* START CHANNEL */
        sprintf (Cmd, "START CHANNEL(%s)", ChannelName);
        RunCommand (hConn, Cmd);

        return;
    }

    /*-------------------------------------------------------------*/
    /*  Other errors:                                              */
    /*      Records message at the system console.                 */
    /*-------------------------------------------------------------*/
    sprintf (Msg, "Channel %s with problems", ChannelName);
    /* Adds tcp/ip return code to the msg, if it exists */
    if  (AuxErrorDataInt1)
        sprintf (Msg, "%s (tcp=%lx)", Msg, AuxErrorDataInt1);

    OperatorPrintf (Action, Msg);
}

/*-------------------------------------------------------------------*/
/*   Handle MQCFH.Reason == MQRC_Q_SERVICE_INTERVAL_HIGH events.     */
/*-------------------------------------------------------------------*/
void   QServiceIntervalHigh (
           MQHCONN hConn,
           MQMD dEveMsg,
           PMQBYTE EveMsg,
           long Action )
{
    /*-------------------------------------------------------------*/
    /*  Some platforms put the queue name in field QName, others    */
```

```
    /*  put it in BaseQName, so both are extracted.                 */
    /*--------------------------------------------------------------*/
    MQLONG  FieldIds [] = {
                MQCA_Q_MGR_NAME,
                MQCA_BASE_Q_NAME,
                MQCA_Q_NAME
            };

    MQCHAR  QMgrName [ MQ_Q_MGR_NAME_LENGTH + 1 ];
    MQCHAR  BaseQName [ MQ_Q_NAME_LENGTH + 1 ];
    MQCHAR  QName [ MQ_Q_NAME_LENGTH + 1 ];

    char    Msg [ MSG_LEN ];

    /*--------------------------------------------------------------*/
    /*  Get fields from the event message.                          */
    /*--------------------------------------------------------------*/
    QMgrName [0] = '\0';
    BaseQName [0] = '\0';
    QName [0] = '\0';

    GetField (
        EveMsg,
        FieldIds,
        sizeof (FieldIds) / sizeof (MQLONG),
        QMgrName,
        BaseQName,
        QName
    );

    /*--------------------------------------------------------------*/
    /*  Put non-blank queue name into QName.                        */
    /*--------------------------------------------------------------*/
    if  (QName [0] == '\0')
        strcpy (QName, BaseQName);

    /*--------------------------------------------------------------*/
    /*  Records message at system console.                          */
    /*--------------------------------------------------------------*/
    sprintf (
        Msg,
        "Queue %s/%s is not being read",
        TrimRight (QMgrName, strlen (QMgrName)),
        TrimRight (QName, strlen (QName))
    );
    OperatorPrintf (Action, Msg);
}

/*--------------------------------------------------------------*/
```

```
/*   Extracts fields from the event message, storing them in the   */
/*   arguments variable list, according to the FieldIds vector.    */
/*-----------------------------------------------------------------*/
void    GetField (
            PMQBYTE EveMsg,
            MQLONG* FieldIds,
            int TotFields,
            ... )
{
    PMQCFH  pCfh;
    PMQBYTE pEventData;
    PMQCFST pCfst;
    PMQCFIN pCfin;
    MQLONG  Count;
    long*   pLong;
    char*   pChar;
    void*   pVoid;
    va_list Arg;

    /* Initializes arguments variable list */
    va_start (Arg, TotFields);

    /*-------------------------------------------------------------*/
    /*  Extract each FieldId from event message.                   */
    /*-------------------------------------------------------------*/
    for  (; TotFields > 0; --TotFields, ++FieldIds)   {

        /* Find beginning of event data */
        pCfh = (PMQCFH) EveMsg;
        pEventData = EveMsg + pCfh->StrucLength;

        /*---------------------------------------------------------*/
        /*  Searches for FieldId in event structures               */
        /*---------------------------------------------------------*/
        for  (Count = 0; Count < pCfh->ParameterCount; ++Count)   {

            /* Maps event structure */
            pCfin = (PMQCFIN) pEventData;

            /* Found the FieldId ! */
            if  (pCfin->Parameter == *FieldIds)   break;

            /* Steps to next event structure */
            pEventData += pCfin->StrucLength;
        }

        /* If the FieldId wasn't found, removes the argument pointer
           from the arguments variable list and steps to the next
           FieldId */
```

```
        if  (Count >= pCfh->ParameterCount)   {
            pVoid = va_arg (Arg, void*);
            continue;
        }

        /* Found the FieldId: extract & store */
        switch  (pCfin->Type)   {
            case MQCFT_INTEGER:
                pLong = va_arg (Arg, long*);
                *pLong = pCfin->Value;
                break;
            case MQCFT_STRING:
                pChar = va_arg (Arg, char*);
                pCfst = (PMQCFST) pEventData;
                memmove (pChar, pCfst->String, pCfst->StringLength);
                pChar [ pCfst->StringLength ] = '\0';
                break;
        }
    }
    va_end (Arg);
}

/*--------------------------------------------------------------------*/
/*   Runs MQSC command.                                               */
/*--------------------------------------------------------------------*/
void    RunCommand (MQHCONN hConn, char* Command)
{
    MQHOBJ      hReplyQ;

    MQOD        dOutQ       = { MQOD_DEFAULT };
    MQOD        dReplyQ     = { MQOD_DEFAULT };
    MQMD        dOutMsg     = { MQMD_DEFAULT };
    MQMD        dReplyMsg   = { MQMD_DEFAULT };

    MQLONG      oOpen;
    MQLONG      oClose;
    MQPMO       oPut        = { MQPMO_DEFAULT };
    MQGMO       oGet        = { MQGMO_DEFAULT };

    MQLONG      CompCode;
    MQLONG      Reason;

    char        Reply [ REPLY_LEN ];
    MQLONG      MsgLen;

    long        CmdCount;
    long        CmdReturn;
    long        CmdReason;
```

```
OperatorPrintf (ACTION_EVENTUAL, "Running: %s", Command);

/*-------------------------------------------------------------*/
/*  Opens tempdyn reply queue.                                 */
/*-------------------------------------------------------------*/
strncpy (
    dReplyQ.ObjectName,
    "SYSTEM.COMMAND.REPLY.MODEL",
    sizeof (dReplyQ.ObjectName)
);

oOpen   = MQOO_INPUT_SHARED
        + MQOO_FAIL_IF_QUIESCING;

MQOPEN (
    hConn,
    &dReplyQ,
    oOpen,
    &hReplyQ,
    &CompCode,
    &Reason
);

if  (CompCode != MQCC_OK)   {
    OperatorPrintf (
        ACTION_EVENTUAL,
        "MQOPEN %s: %ld",
        TrimRight (dReplyQ.ObjectName, sizeof
                                    (dReplyQ.ObjectName)),
        Reason
    );
    return;
}

/*-------------------------------------------------------*/
/*  Puts command into system command input queue.        */
/*-------------------------------------------------------*/
strncpy (
    dOutQ.ObjectName,
    "SYSTEM.COMMAND.INPUT",
    sizeof (dOutQ.ObjectName)
);
dOutMsg.MsgType = MQMT_REQUEST;

memmove (
    dOutMsg.Format,
    MQFMT_STRING,
    sizeof (dOutMsg.Format)
);
```

```
strncpy (
    dOutMsg.ReplyToQ,
    dReplyQ.ObjectName,
    sizeof (dOutMsg.ReplyToQ)
);
strncpy (
    dOutMsg.ReplyToQMgr,
    dReplyQ.ObjectQMgrName,
    sizeof (dOutMsg.ReplyToQMgr)
);
oPut.Options    = MQPMO_NO_SYNCPOINT
                + MQPMO_FAIL_IF_QUIESCING;
MQPUT1 (
    hConn,
    &dOutQ,
    &dOutMsg,
    &oPut,
    strlen (Command),
    Command,
    &CompCode,
    &Reason
);
if  (CompCode != MQCC_OK)   {

    OperatorPrintf (
        ACTION_EVENTUAL,
        "MQPUT1 %s: %ld",
        TrimRight (dOutQ.ObjectName, sizeof (dOutQ.ObjectName)),
        Reason
    );
    oClose = MQCO_DELETE_PURGE;

    MQCLOSE (
        hConn,
        &hReplyQ,
        oClose,
        &CompCode,
        &Reason
    );
    return;
}

/*-------------------------------------------------------------*/
/*  Reads all replies                                          */
/*-------------------------------------------------------------*/

for  ( ; ; )   {

    /*---------------------------------------------------------*/
```

```
/* Each set of replies starts with message CSQN205I,        */
/* whose structure is:                                      */
/*    CSQN205I COUNT=count, RETURN=rc, REASON=reason.       */
/*----------------------------------------------------------*/

memmove (
    dReplyMsg.MsgId,
    MQMI_NONE,
    sizeof (dReplyMsg.MsgId)
);
memmove (
    dReplyMsg.CorrelId,
    MQCI_NONE,
    sizeof (dReplyMsg.CorrelId)
);
oGet.Options    = MQGMO_NO_SYNCPOINT
                + MQGMO_WAIT
                + MQGMO_FAIL_IF_QUIESCING;

oGet.WaitInterval = TIME_OUT;

MQGET (
    hConn,
    hReplyQ,
    &dReplyMsg,
    &oGet,
    sizeof (Reply),
    Reply,
    &MsgLen,
    &CompCode,
    &Reason
);
if  (CompCode != MQCC_OK)   {

    OperatorPrintf (
        ACTION_EVENTUAL,
        "MQGET %s: %ld",
        TrimRight (dReplyQ.ObjectName, sizeof
                                       (dReplyQ.ObjectName)),
        Reason
    );
    oClose = MQCO_DELETE_PURGE;

    MQCLOSE (
        hConn,
        &hReplyQ,
        oClose,
        &CompCode,
        &Reason
```

```
    );
    return;
}
sscanf (
    Reply, "CSQN205I COUNT=%ld, RETURN=%lx, REASON=%lx",
    &CmdCount,
    &CmdReturn,
    &CmdReason
);
/* ReturnCode != 0 => problem processing the command */
if  (CmdReturn)   {
    OperatorPrintf (
        ACTION_EVENTUAL,
        "Failed to run %s: return %lx reason %lx",
        Command,
        CmdReturn,
        CmdReason
    );
    oClose = MQCO_DELETE_PURGE;

    MQCLOSE (
        hConn,
        &hReplyQ,
        oClose,
        &CompCode,
        &Reason
    );
    return;
}

/*-------------------------------------------------------------*/
/*  Reads a set of replies.                                    */
/*-------------------------------------------------------------*/

/* Msg CSQN205I counts towards the msgs count, so remove it */
--CmdCount;

while  (CmdCount-- > 0) {

    memmove (
        dReplyMsg.MsgId,
        MQMI_NONE,
        sizeof (dReplyMsg.MsgId)
    );
    memmove (
        dReplyMsg.CorrelId,
        MQCI_NONE,
        sizeof (dReplyMsg.CorrelId)
    );
```

```
    oGet.Options    = MQGMO_NO_SYNCPOINT
                    + MQGMO_WAIT
                    + MQGMO_FAIL_IF_QUIESCING;

    oGet.WaitInterval = TIME_OUT;

    MQGET (
        hConn,
        hReplyQ,
        &dReplyMsg,
        &oGet,
        sizeof (Reply),
        Reply,
        &MsgLen,
        &CompCode,
        &Reason
    );
    if  (CompCode != MQCC_OK)   {

        OperatorPrintf (
            ACTION_EVENTUAL,
            "MQGET %s: %ld",
            TrimRight (
                dReplyQ.ObjectName,
                sizeof (dReplyQ.ObjectName)
            ),
            Reason
        );
        oClose = MQCO_DELETE_PURGE;

        MQCLOSE (
            hConn,
            &hReplyQ,
            oClose,
            &CompCode,
            &Reason
        );
        return;
    }
}  /* 'while' reads a set of replies */

/* Are there more sets of replies ??? */
/*-------------------------------------------------------------*/
/*  If there are more sets of replies, ReturnCode = 0 and   */
/*  Reason = 4.                                               */
/*-------------------------------------------------------------*/
if  (CmdReason == 4)
    continue;    /* Let's read the other set of replies */
else
    break;    /* That's all folks */
```

```
    }  /* for: Reads all replies */

    /*-----------------------------------------------------------------*/
    /*  Closes reply queue.                                            */
    /*-----------------------------------------------------------------*/
    oClose  = MQCO_DELETE_PURGE;

    MQCLOSE (
        hConn,
        &hReplyQ,
        oClose,
        &CompCode,
        &Reason
    );
    return;
}

/*-------------------------------------------------------------------*/
/*  printf at system console.                                        */
/*-------------------------------------------------------------------*/
void    OperatorPrintf (long Action, char* Fmt, ...)
{
    char        Msg [ MSG_LEN ];
    char        OperatorMsg [ MSG_LEN ];
    va_list     Arg;

    va_start (Arg, Fmt);
    vsprintf (Msg, Fmt, Arg);
    va_end (Arg);

    sprintf (OperatorMsg, "%s %s", MQS_PREFIX, Msg);

    exec cics   write operator
                text (OperatorMsg)
                textlength (strlen (OperatorMsg))
                action (Action)
    ;
}

/*-------------------------------------------------------------------*/
/*  Converts C-string to lowercase                                   */
/*-------------------------------------------------------------------*/
char*   StrLower (char* Str)
{
    for  (; *Str; ++Str)
        *Str = tolower (*Str);
}

/*-------------------------------------------------------------------*/
/*   Removes trailing blanks                                         */
```

```
/*---------------------------------------------------------------*/
char*   TrimRight (char* Str, int Len)
{
    char*   pCh;

    for (pCh = Str + Len - 1; pCh >= Str && *pCh == ' '; --pCh)
        ;
    *(pCh+1) = '\0';

    return (Str);
}
```

Paulo Marcel Coelho Aragão (Brazil)

MQSeries coding standards and guidelines

The move by many large users from MVS to production systems on heterogeneous platforms including Unix, OS/2, and NT has necessitated the implementation of reliable and secure heterogeneous system communication. The ability to access and integrate systems across platforms is now more critical to organizations than ever before. Thankfully, IBM's MQSeries has made this task relatively simple. It's eliminated the need to connect applications directly or make them understand underlying networking protocols.

MQSeries handles the networking calls, freeing the programmer from the need to code at this level. The system's messaging paradigm enables applications to communicate synchronously (conversationally) or asynchronously. With a consistent Application Programming Interface (API) across all platforms, it allows programmers to be more productive. Once you understand the MQ API, you have the skills to get applications communicating on any platform supported by MQSeries.

While MQSeries brings with it many intrinsic benefits, you must develop and enforce standards in order to use it effectively. I've found that, while naming standards for MQSeries objects are typically in place before application development begins, little or no thought is

often given to standards or guidelines for constructing MQSeries applications until after coding has started. This leads to many architects and programmers duplicating each others' work. By proactively consolidating this work, building applications can be made easier, as the groundwork is already laid. Another benefit is that it reduces the need to educate developers, as key concepts are documented more concisely.

This article, which is in two parts (the second part is in next month's issue of *MQ Update*), provides background information on messaging models, then presents eleven standards for successful application construction. It also explains why the standards are justified and how to use each one. Next month's instalment explores ten guidelines for constructing MQSeries applications.

STANDARD VERSUS GUIDELINE

The difference between a standard and a guideline is that a standard is a hard-and-fast rule that you must follow, while a guideline is a recommendation that you should follow in most cases. If you fail to follow a standard, negative consequences are almost certain. But if you don't follow a guideline, there's probably little or no risk to the organization. Most organizations allow programmers to deviate from a standard only after a thorough review of all the implications. In most cases, deviation requires permission from a senior programmer or analyst who thoroughly understands the implications. For guidelines, the good judgement of the programmers themselves is sufficient to allow deviation – no higher-level approval is required.

MQSERIES MESSAGING MODELS

Knowing the common messaging models will help you understand the requirements for constructing these models using MQSeries. With this knowledge, you will see characteristics that naturally lead to standards and guidelines for constructing applications. There are two basic messaging models:

- Datagram
- Request/reply.

The datagram model, also known as 'fire-and-forget', is the least complex messaging model. In it, an application builds a message and introduces it to the network simply by putting it on a queue. In most cases, the application takes advantage of the 'once-and-once-only' assured delivery that MQSeries provides. Hence, it needs to know only that the receiving application will, in due course, process the message, and doesn't need to know when processing will take place. Datagram messages are generally persistent, since some form of update occurs from message creation.

The request/reply model is more complex and, consequently, its construction requires more effort. You must consider such questions as:

- How long should the sending application wait for a reply?
- Should another request be generated if no reply is received?
- Will database updates occur as a result of a reply?
- Does session or state information need to be maintained (in other words, is this application conversational in nature)?

Answers to these questions, and many others based on business requirements, will significantly impact application construction and your choice of MQSeries options.

ELEVEN STANDARDS TO CONSIDER

The standards listed below are important because they help to ensure that MQSeries applications function predictably and are easy to maintain. Consideration is given to MQSeries features and options that enhance performance management, portability, scalability, and reliability. These standards are by no means exhaustive; rather, they address some of the typical application construction issues that arise when installation begins. To determine whether a standard applies to your organization, ask yourself: must programmers follow it for the application to function properly?

1 Applications must check reason and completion codes after all MQSeries API calls and take appropriate action when a non-zero

reason or completion code is returned.

Overview:

All MQSeries API calls return completion and reason codes indicating the success or failure of calls.

Justification:

This ensures that applications are functioning according to their design. Without checking completion and reason codes, applications cannot ensure that MQSeries messages are produced or consumed correctly.

2 Applications must check the back-out count after all *MQGET*s that are performed under syncpoint.

Overview:

When a message is retrieved under syncpoint and then backed out (explicitly or implicitly), the *BackoutCount* field in the message descriptor is incremented. This indicates whether the message was read prior to being backed out. If *BackoutCount* is greater than a pre-defined threshold, the message needs special processing.

Justification:

Without this check, applications may get stuck in an infinite loop of reading and backing out of messages.

3 Applications must use the *FAIL_IF_QUIESCING* option on *MQOPEN*, *MQPUT*, and *MQGET* calls.

Overview:

The MQSeries subsystem is independent of applications that use the services it provides. It is sometimes necessary to shut it down; when this occurs, any connected applications should quiesce and no additional applications may connect. Applications that are connected should terminate immediately. For applications to be notified of a pending shutdown, the *FAIL_IF_QUIESCING* attribute must be specified on the appropriate MQSeries API calls.

Justification:

Applications that do not specify the appropriate *FAIL_IF_QUIESCING* attribute continue to run after the MQSeries subsystem has started to quiesce. This state may persist for a considerable time and prevent the MQSeries subsystem from terminating. This, in turn, may compromise processing time set aside for maintenance and back-up. It may ultimately be necessary to cancel or kill MQSeries applications that don't use the *FAIL_IF_QUIESCING* option. If applications aren't killed, it increases the likelihood of data integrity problems, which are usually costly to recover from.

4 Use queue-based priority to prioritize requests; applications must not set the priority of request messages.

Overview:

Priority is a message attribute that states the priority of the message relative to others in the MQSeries network. As priority can change as new applications are introduced or existing ones are eliminated, it's impossible for an application to hardcode a message's effective priority. By using queue-defined priority, network administrators can prioritize messages by altering the queue's *DEFPRTY* attribute.

Justification:

Controlling message priority allows greater flexibility in managing the network.

5 Requesting applications must not use *MQOO_OUTPUT* when opening a request queue.

Overview:

Request queues may be local or remote to the application making the request. Local queues may be written to (*MQPUT*) or read from (*MQGET*); remote queues may only be written to. However, requesting applications need only to put messages on the request queue. So, the requesting application needs to open the request queue only for output (*MQPUT*). This means it's immaterial

whether the queue is local or remote, as both can be opened for output. If a requesting application opens a request queue for input, then the application must be local to the queue.

Justification:

This ensures that services can be moved around the MQSeries network without impact on requesting applications.

6 Update messages must be *MQPUT* and *MQGET* under syncpoint.

Overview:

Applications can control when messages become visible to the MQSeries network by using syncpoint options on *MQPUT* and *MQGET* calls. When database updates have to be coordinated with MQSeries messages, the messages must be controlled using two MQSeries syncpoint options:

– *MQPMO_SYNCPOINT*

– *MQGMO_SYNCPOINT*.

Justification:

Using syncpoint ensures that no database updates occur when MQSeries messages are backed out. MQSeries messages may be backed out if there are problems with the message or the database. Syncpoint options allow messages to be saved for further processing in the event of a failure.

7 Update messages' persistence attribute must be set to 'yes'.

Overview:

Messages resulting in database updates must be logged, and the messages themselves should survive queue manager restarts. Only persistent messages meet these criteria.

Justification:

Non-persistent messages could be lost if a queue manager fails.

8 Applications that modify and forward messages or reply to requests must preserve identity context information.

Overview:

Every message in an MQSeries system has an identity. This identity represents the user who created or introduced the message into the MQSeries network. As messages flow through the network, their identity must be preserved. If the original message is modified and forwarded, or another message, such as a reply, is created from it, then the modified or new message should also be attributable to the original user through its identity context.

Justification:

For security and auditing, it must be possible to trace all messages back to the user who originated them. Refer to the *MQSeries Application Programming Guide* for more details on context security and the *Intercommunication Guide* for details on the channel attribute *PUTAUT*.

9 Update messages must not expire.

Overview:

Messages aren't always processed in the time expected by the applications that created them. In an asynchronous environment, all that can be said is that a message will be processed sometime in the future, though exactly when is unknown. Because of this, update messages should not expire, as expiration may occur before the message is processed.

Justification:

This ensures no update messages are lost. Also see standard 7 on the persistence of update messages.

10 The name of permanent dynamic reply queues must include a *userid* or *taskname* that is predictable across multiple invocations of the same application.

Overview:

Applications that require dynamic reply queues can create temporary or permanent versions of these queues. In most cases,

the permanent queues hold recoverable messages (those that are most likely to be persistent). This implies that the application using the permanent reply queue must be able to restart and open the queue if the application abends. To facilitate this restart, the dynamic queue name must be predictable across multiple invocations of the application.

Justification:

This ensures that persistent replies can be processed across multiple invocations of an application.

11 Applications that use temporary dynamic queues for replies must ensure that the applications that handle the requests do not set the persistence of their replies to *MQPER_PERSISTENT*.

Overview:

Temporary dynamic reply queues cannot hold persistent messages. Because of this restriction, applications that respond to requests must know whether the queue that receives the reply is a temporary or permanent dynamic reply queue. Requesting and replying applications must coordinate the setting of the reply's persistence to ensure that only non-persistant messages are put on temporary dynamic reply queues.

Justification:

This prevents replies from ending up on the system's undeliverable message queue.

ENFORCEMENT

As you can see, there are many standards that come into play when developing MQSeries applications, and many more are possible. Organizations may define others based on their business requirements. One question applies to all organizations, however: given that resources are often limited, how do you ensure that standards are being followed?

There are two ways to enforce standards. The first is to conduct

technical code reviews. This option tends to be costly, as it requires experienced programmers to go through code line-by-line. The second option is to create wrappers for MQSeries API calls that application programmers must use. You then build the organization's standards into the wrapper. This method guarantees that MQSeries calls conform with your organization's preferences and also requires a code review to ensure the wrapper is being used. But since the wrapper typically simplifies the API calls, code reviews take less effort, thereby saving your organization's resources. Note that a wrapper will not allow your organization to avoid code reviews completely – reviews should be carried out whether native MQSeries API calls or wrappers are used.

CONCLUSION

Standards and guidelines are essential for developing reliable and efficient MQSeries messaging applications. They ensure that your applications function as expected. You must be able to justify the standards and guidelines that you choose to adopt, as programmers tend to deviate from those that don't make sense or offer no apparent benefit to them or the code they write. Some of the standards presented in this article may make sense for your organization, while others may not. Research the options and carefully choose the standards and guidelines that reflect the goals of your organization.

I intend to discuss the issue of guidelines in an article to be published in a future issue of *MQ Update*.

Mark Verhiel
Candle Corporation (USA)

QM definition scripts from the BSDS – a reply

The November 1999 and December 1999 editions of *MQ Update* contain an article for a program that outputs a list of *DEFINE* commands that can be used to recreate all the MQSeries object

definitions for a queue manager. However, such a program is not needed in an MVS/ESA environment, as MQSeries for MVS/ESA provides the same function via its *CSQUTIL* utility using the *MAKEDEF* operand of the *COMMAND* function. Details of this can be found in the *MQSeries for MVS/ESA System Management Guide.* Sample JCL to run this utility is shown below.

SAMPLE JCL

```
//jobname JOB etc
//*****************************************************************
//*
//*      Generate definition statements for MQSeries objects
//*
//*****************************************************************
//CSQUTIL EXEC PGM=CSQUTIL,REGION=4096K,PARM='qmgr'
//STEPLIB  DD DSN=SYS1.SCSQANLE,DISP=SHR
//         DD DSN=SYS1.SCSQAUTH,DISP=SHR
//DEFINE   DD DSN=your.output.dataset
//SYSPRINT DD SYSOUT=*
//CSQUCMD  DD *
*****************************************************************
*
* These DISPLAY commands are used to generate a set of DEFINE
* commands to be able to recreate MQSeries objects.
*
*****************************************************************
*
DISPLAY STGCLASS( * ) ALL
DISPLAY QUEUE( * ) TYPE( QLOCAL ) ALL
DISPLAY QUEUE( * ) TYPE( QMODEL ) ALL
DISPLAY QUEUE( * ) TYPE( QALIAS ) ALL
DISPLAY QUEUE( * ) TYPE( QREMOTE ) ALL
DISPLAY PROCESS( * ) ALL
DISPLAY NAMELIST( * ) ALL
DISPLAY CHANNEL( * ) ALL
//SYSIN    DD *
 COMMAND DDNAME(CSQUCMD) MAKEDEF(DEFINE)
//
```

Eric Judd
Technical Consultant
Metropolitan Life (South Africa)

IBM, i2, and Ariba have announced an alliance to deliver an end-to-end system for B2B e-commerce and collaboration. The system is to handle product integration, global marketing, and selling of targeted solutions. As part of the agreement, IBM is to make an undisclosed equity investment in both Ariba and i2, which, in turn, will extend their software to work with IBM's, including MQSeries, WebSphere, WebSphere Commerce Suite, and DB2, and will optimize their software for IBM's servers.

Hot on the heels this announcement, IBM announced WebSphere B2B Integrator, which integrates businesses' operational systems with those of their trading partners. It's built on WebSphere, MQSeries, and an IBM XML technology called tpaML, which is designed for handling contract exchange.

IBM also announced that extensions were added to MQSeries Workflow to support B2B processes, and that it intends to add more functions to WebSphere Commerce Suite to support B2B and marketplace commerce, including negotiation and dynamic price discovery models, RFP/RFQ, and workflow.

The first release of WebSphere B2B Integrator will be available this summer.

For further information contact:
Ariba, 1565 Charleston Road, Mountain View, CA 94043, USA
Tel: +1 650 930 6200
Fax: +1 650 930 6300
Web: http://www.ariba.com

Ariba United Kingdom Ltd, 1000 Great West Road, Brentford, Middlesex TW8 9HH, UK
Tel: +44 181 261 4431
Fax: +44 181 261 4558

i2 Technologies Inc, One i2 Place, 11701 Luna Road, Dallas, TX 75234, USA
Tel: +1 469 357 1000
Fax: +1 214 860 6060
Web: http://www.i2.com

i2 Technologies , 10th Floor, Market Square House, St James's Street, Nottingham, Nottinghamshire NG1 6FG, UK
Tel: +44 115 959 0880
Fax: +44 115 941 2849

* * *

New Era of Networks has enhanced its distribution relationship with IBM, which allows it to provide components of IBM's MQSeries Integrator. The two firms have extended the platform coverage of MQSI now that Version 1.1 is available on MVS and OS/390. In addition, NEON has worked with IBM to port MQSI to the AS/400, adding to existing products for AIX (and other versions of Unix) and NT. NEON is also to support Tivoli monitoring and management capabilities in its integration servers.

For further information contact:
NEON, 7400 East Orchard Road, Englewood, CO 80111, USA
Tel: +1 303 694 3933
Fax: +1 303 694 3885
Web: http://www.neonsoft.com

New Era of Networks Ltd, Aldermary House, 15 Queen Street, London EC4N 1TX, UK
Tel: + 44 171 329 4669
Fax:+ 44 171 329 4567