13

MQ update

July 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

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.

MQSeries’ JMS ‘publish-and-subscribe’

INTRODUCTION

This article describes the use of the Java Message Service’s (JMS) publish-and-subscribe API in conjunction with the MQSeries JMS implementation supplied in SupportPac MA88. Also in this article is an example application that demonstrates the essential features of JMS’s publish-and-subscribe.

THE PUBLISH-AND-SUBSCRIBE MODEL

Publishers, subscribers, and topics

The publish-and-subscribe model provides an alternative to the conventional point-to-point messaging model. Instead of communicating through named queues, publishers (originators of messages) and subscribers (recipients of messages) rendezvous on named topics. Topics can be thought of as active destinations: propagation and retention of messages is affected by the presence or absence of client subscriptions. In particular:

- A message published to a single topic may be consumed by any number of subscribers
- A message published to a topic may never actually be delivered if there are no subscriptions to the topic.

The two models do not necessarily represent different levels of abstraction; in JMS, *Topic* and *Queue* are simply different types of *Destination*, and each can be used to represent application interactions in the same way.

Non-durable and durable subscriptions

JMS defines two types of subscription. A client with a *non-durable* subscription to a particular topic can receive messages from the published topic only when the client is active. A client with a *durable* subscription is able to receive published messages when the client is inactive, as the messages are retained by the service provider between invocations of the subscriber.

Publish-and-subscribe client operation

A JMS publish-and-subscribe client typically takes the following steps to prepare itself to send or receive messages. These steps are similar to those used for point-to-point messaging.

1 Use JNDI to locate a *ConnectionFactory* object (*TopicConnectionFactory*).

2 Use JNDI to find one or more *Destination* (*Topic*) objects.

3 Use the *ConnectionFactory* to create a JMS *Connection* object (*TopicConnection*).

4 Use the *Connection* object to create one or more *Session* objects (*TopicSession*).

5 Use a *Session* object and *Destination* objects to create the *MessageProducer* and *MessageConsumer* (publisher and subscriber, respectively) objects needed.

As with JMS's point-to-point messaging, provider-specific *ConnectionFactory* and *Destination* objects can be obtained by direct instantiation without using JNDI.

PUBLISH-AND-SUBSCRIBE WITH MQSERIES JMS

Brokers, streams and queues

Publish-and-subscribe services are provided in MQSeries by *message brokers*. A message broker is an MQSeries application associated with a single MQSeries queue manager. The queue manager maintains queues for the storage of topic messages and uses the communication links between queue managers to exchange messages with other message brokers.

When topic messages are published to an MQSeries broker, they are addressed to a stream queue, which acts as the message broker's input for one or more topics. A stream queue is associated in MQSeries' implementation of JMS with a *TopicConnection*, and it appears as an attribute of the *TopicConnectionFactory*-administered object. A corresponding MQSeries queue must be defined at the queue manager separately from the definition of the JMS-administered object before the *TopicConnectionFactory* can be used. The default stream queue is *SYSTEM.BROKER.STREAM.QUEUE*, which is typically created during the initial configuration of MQSeries' publish-and-subscribe environment.

The MQSeries queues in which topic messages ultimately reside are created automatically when subscriptions are established; they are not accessed explicitly either by clients or during administration tasks. Topic queues are removed automatically when their subscriptions are terminated.

Organization of topics

JMS does not define a scheme for naming or organizing topics – this is left up to the service provider. MQSeries uses a hierarchical scheme where levels in a hierarchy are delimited by the forward slash character ('/') in topic names, and wildcards can be used by subscribers to identify a collection of topics in a single subscription request.

Administration

Two types of JMS-administered object – *TopicConnectionFactory* and *Topic* – are used in JMS publish-and-subscribe. They are defined in a JNDI repository using a provider-specific administration tool (for MQSeries, the tool is the JMSAdmin program supplied with MQSeries SupportPac MA88).

JMS SUBSCRIBER EXAMPLE: A SIMPLE PAGER APPLICATION

About the example

This example is a JMS subscriber application. It provides a simple GUI interface (see Figure 1) representing a pager device and subscribes to a single topic that represents a pager ID (a user). The application subscribes when it is invoked. When running, a check box determines whether the pager is 'on' (receiving messages) or 'off', in which case messages remain on the subscriber queue. When turned on, the pager application first checks synchronously for any messages published to it that may have accumulated, and then switches to waiting for notification of new messages. The pager application stores all the messages it receives internally until the user erases them (or quits).

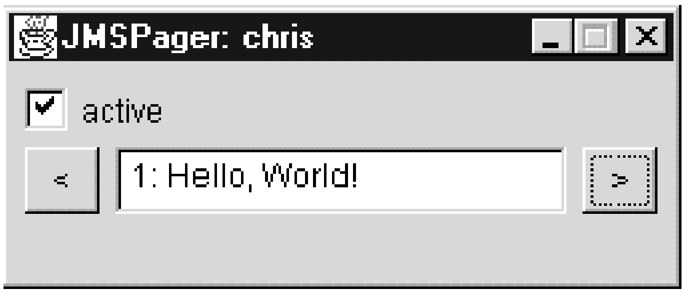
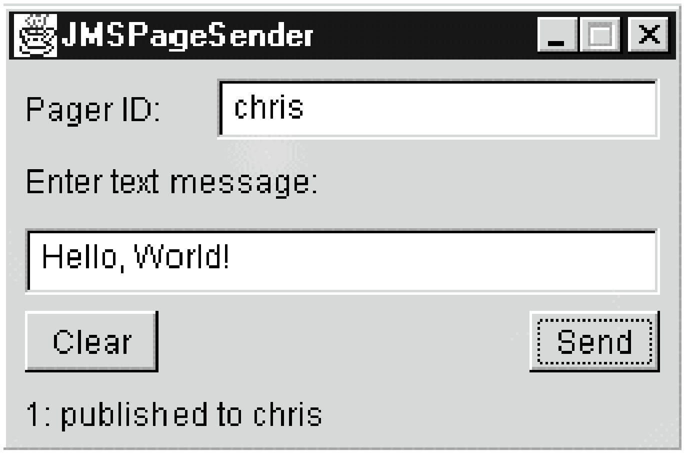


Figure 1: The pager's main interface

The pager has two buttons:

- The left-hand button steps through a menu allowing either a received message to be selected for display or all messages to be erased.
- The right-hand button either shows the selected message, scrolling it across the display, or confirms that messages were erased.



Pager ID: chris

Enter text message:

Hello, World!

1: published to chris

Figure 2: Pager showing Pager ID text field

When a new pager message is received, the pager beeps and the text ‘New message!’ appears on the display.

To publish messages to one or more *JMSPager* instances, the JMSPageSender application is provided.

The Pager ID text field (Figure 2) is used to enter a topic to which to send a message, and the message text is entered in the lower text field.

Requisites/environment

The MQSeries JMS provider classes are supplied in MQSeries SupportPac MA88, which is available for download free from the IBM MQSeries Web site for Windows, AIX, HP-UX, and Solaris. MQSeries 5.1 and JDK 1.1.6 or later are required (JDK 1.1.7 or later in the case of HP-UX). The MQSeries 5.1.1 Java classes, which are also required, are supplied in the SupportPac, and the MQSeries publish-and-subscribe component is supplied in SupportPac MA0C.

The description of the example application that follows assumes that the above components are installed.

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

- AIX 4.2.1 running MQSeries 5.1 and JDK 1.1.8
- Microsoft Windows NT 4.0 running 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* must be set to include the classes in the MQSeries, JMS, and JNDI archives.
- A JNDI-accessible repository must be configured and populated with one or more *ConnectionFactory* and *Destination* entries to enable JNDI to look up JMS-administered objects.
- Any MQSeries objects referred to by the JMS-administered object definitions must be defined in MQSeries and made available.

Setting the classpath

The following classes from the MA88 *jar* archives must be accessible in the *classpath* (in addition to the base JDK classes) to run any MQSeries JMS program:

- *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 the *java/lib* (or *java\lib*) sub-directory 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. The two entries required are *INTIAL_CONTEXT*, the JNDI service provider class name that determines the mechanism by which the repository is accessed, and *PROVIDER_URL*, the URL that locates the repository. Both are configured in the *JMSAdmin.config* properties file. For example, the following entries can be used for a filesystem-based repository:

```
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 the JMSPager application, we need definitions for:

- A topic connection factory

- One or more topics corresponding to pager users.

To create a queue connection factory *jmspager* corresponding to the local queue manager *LYCHEE.QM* and topics *chris* and *bob*, the JMSAdmin commands would be (following the prompt 'InitCtx>'):

```
InitCtx> define tcf(jmspager) qmanager(LYCHEE.QM)
InitCtx> define t(chris)
InitCtx> define t(bob)
```

Here, the topic connection factory definition refers to a local queue manager by name, and this results in the use of the MQSeries Java bindings. Alternatively, a topic connection factory can be defined as an MQSeries client connection (note the use of the continuation character, '➤', below to indicate a formatting line break):

```
InitCtx> define tcf(jmspager) qmanager(LYCHEE.QM)
➤  hostname(lychee) port(1414) channel(CLIENT)
➤  transport(client)
```

The transport of 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 an **inetd** entry (on Unix) or active listener must exist on the specified port of the target host.

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, *jms* (for Unix environments), to invoke an arbitrary JMS application is shown below:

SAMPLE UNIX SCRIPT JMS

```
#!/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...

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

For Windows, the equivalent batch file, *jms.bat*, is:

SAMPLE WINDOWS BATCH FILE

```
@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...

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

Note that the *SET JMS_CLASSPATH* statement is all on one line, as is *java -classpath....*

THE JMSPAGER CODE

The JMSPager application extends the *Frame* top-level window class. It establishes a subscription to a single topic that's named either on the command line or through the *jmspager.user.name* system property. A *Vector* is used to store messages as they are received; the two buttons allow the user to select and display individual messages and erase them after reading. The *index* instance variable keeps track of which message is selected.

The *main* method initializes the JNDI naming context and instantiates a pager subscribed to the specified topic name (if specified as a command-line argument) or the topic name obtained from either the *jmspager.user.name* or *user.name* property of the *System* object.

The next two methods (*getConnection* and *getTopic*) return references to JMS-administered objects from the JNDI, using the context set-up in the *main* method.

The constructor (*JMSPager*) configures the AWT layout of the JMSPager's child components and connects listeners to the buttons and checkbox (the checkbox is the on/off switch for the pager). Next, a connection to the JMS service provider is opened, a session created, and subscription to the named topic established. The application registers itself as a JMS *ExceptionListener* for asynchronous notification of JMS errors. Finally, the JMS connection is started to enable messages to be received over it.

The *activate* method is invoked by the *ItemListener* attached to the checkbox when the pager is 'turned on'. It uses the synchronous *receive* method to retrieve any waiting messages from the topic and adds them to the *Vector* store. By registering the application as a *MessageListener* to the subscription, it then requests notification of any further messages as they appear using the *onMessage* method.

The *deactivate* method is invoked by the checkbox *ItemListener* when the pager application is 'turned off'. By setting the subscriber's message listener to 'null', it cancels the application's registration as a message listener, so it is no longer notified of new messages.

The *select* method is invoked by the *ActionListener* on the left-hand button. It manipulates the *index* variable, determining which message is displayed when the right-hand button is pressed. If the *index* is set to '-1', stored messages are erased when the right-hand button is pressed.

The *display* method is invoked by the *ActionListener* on the right-hand button. It steps through the stored messages by invoking the *scroll* method for each one and incrementing *index*. If *index* is negative, it empties the *Vector* of stored messages.

The *scroll* method displays a stored message by loading its contents into the text area and incrementing the caret position across the string.

The *onMessage* method – defined by the JMS *MessageListener* interface – is invoked by the *Session* to which the listener is registered. Incoming messages are passed directly to this method while the application is 'turned on'. The method stores each message and sets *index* to refer to it so it can be displayed with a single click.

The *onException* method is invoked by the JMS connection if an error occurs in the environment rather than as a direct result of a JMS method invocation by the client.

Finally, the *disconnect* method closes the JMS resources used by the application. *showText*, *showStatus*, and *showError* are 'convenience' methods for displaying text in the application.

For neatness, the *getInsets* method – inherited from *Container* via the application's parent class *Frame* – establishes a border between the edge of the frame and the components within.

JMSPAGER.JAVA

```
import javax.naming.*;
import javax.jms.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

/**
 * A simple AWT GUI application demonstrating the basic capabilities of
 * JMS publish-subscribe. A JMSPager subscribes to a topic that
 * represents a user and receives messages both synchronously and
 * asynchronously.
 */

public class JMSPager extends Frame implements
   MessageListener, ExceptionListener
{
   static InitialContext context;

   final static int COLUMNS = 20;
```

```
Vector vector = new Vector ();

Checkbox onOffButton;
TextField messageText;
Button leftButton;
Button rightButton;
Label statusLabel;

TopicConnection connection;
TopicSession session;
Topic topic;
TopicSubscriber subscriber;

int index = 0;
/**
 * Runs the JMSPager application.
 */

public static void main (String [] args)
{
  try
   {
      context = new InitialContext (System.getProperties ());

      try
      {
         new JMSPager (args [0]).show ();
      }

      catch (ArrayIndexOutOfBoundsException abe)
      {
         new JMSPager (System.getProperty ("jmspager.user.name",
            System.getProperty ("user.name"))).show ();
      }
   }

   catch (JMSException jmse)
   {
      System.err.println ("JMS error: " + jmse +
         " (" + jmse.getLinkedException () + ')');
      System.exit (0);
   }

   catch (NamingException ne)
   {
      System.err.println ("JNDI error: " + ne);
      System.exit (0);
   }

   catch (ArrayIndexOutOfBoundsException abe)
   {
         System.err.println ("Usage: java JMSPager user-name");
      }
   }
  /**
    * Returns a named TopicConnection using the JNDI. The
    * TopicConnectionFactory is a JMS-administered object.
    */

   static TopicConnection getConnection (String name) throws
      JMSException, NamingException
   {
      TopicConnectionFactory factory = (TopicConnectionFactory)
         context.lookup (name);

      return factory.createTopicConnection ();
   }

   /**
    * Returns a named Topic handle using the JNDI. The Topic is a
    * JMS-administered object.
    */

   static Topic getTopic (String name) throws
      JMSException, NamingException
   {
      return (Topic) context.lookup (name);
   }

   /**
    * Default constructor. Builds a simple AWT layout and then
    * establishes a connection to the JMS service provider. The
    * connection must be explicitly started before messages can
    * be received.
    */
   public JMSPager (String id) throws
      JMSException, NamingException
   {
      super ("JMSPager: " + id);

      setLayout (new BorderLayout (6, 2));
      setBackground (Color.lightGray);
      add ("North", onOffButton = new Checkbox ("active", false));
      add ("Center", messageText = new TextField (COLUMNS));
      add ("West", leftButton = new Button (" < "));
      add ("East", rightButton = new Button (" > "));
      add ("South", statusLabel = new Label ());
      pack ();
      setResizable (false);
      messageText.setEditable (false);

      // WindowListener to deal with quitting the app.
```

```
addWindowListener (new WindowAdapter ()
{
   public void windowClosing (WindowEvent event)
   {
      disconnect ();
      System.exit (0);
   }
});

// ActionListener attached to left ('select') button.

leftButton.addActionListener (new ActionListener ()
{
   public void actionPerformed (ActionEvent event)
   {
      select ();
   }
});

// ActionListener attached to right ('ok') button.

rightButton.addActionListener (new ActionListener ()
{
   public void actionPerformed (ActionEvent event)
   {
      display ();
   }
});

// Item listener attached to Checkbox ('on/off button').

onOffButton.addItemListener (new ItemListener ()
{
   public void itemStateChanged (ItemEvent event)
   {
      if (event.getStateChange () == ItemEvent.SELECTED)
      {
         activate ();
      }
      else
      {
         deactivate ();
      }
   }
});

// Establish a connection to the JMS service provider.

connection = getConnection
   (System.getProperty ("jmspager.server.name", "jmspager"));
connection.setExceptionListener (this);
```

```
    session = connection.createTopicSession
       (false, Session.AUTO_ACKNOWLEDGE);
    topic = getTopic (id);
    subscriber = session.createSubscriber (topic);   // non-durable

    // Enable the connection to receive messages.

    connection.start ();
 }
 /**
  * Turns the pager on. First checks synchronously for available
  * messages (delivered while turned off), then switches to
  * asynchronous receipt mode by establishing the JMSPager as a
  * MessageListener. In asynchronous mode, JMS calls the onMessage
  * method when new messages arrive.
  */

 public synchronized void activate ()
 {
    try
    {
       Message message = subscriber.receive (0);
       int old = vector.size ();
       int arrived = 0;

       // Add any received messages to our local store (the Vector).

       while (message != null)
       {
          vector.addElement (message);
          arrived++;
          message = subscriber.receive (0);
       }

       showText (arrived + " new " +
          (arrived == 1 ? "message" : "messages") +
          " (" + old + " old)");

       // Switch to asynchronous mode.
       //
       // Note: after attaching a message listener, attempts to
       // receive synchronously on the same session would fail.

       subscriber.setMessageListener (this);
    }

    catch (JMSException jmse)
    {
       showError (jmse);
    }
 }
```

```
/**
 * Turns the pager off by disconnecting the message listener on the
 * session. The subscription and connection to the JMS service
 * provider is retained.
 */

public synchronized void deactivate ()
{
   try
   {
      // Remove the message listener.

      subscriber.setMessageListener (null);

      showText ("");
      index = -1;
   }

   catch (JMSException jmse)
   {
      showError (jmse);
   }
}
/**
 * Steps through the message selection menu.
 */

public synchronized void select ()
{
   if (vector.size () == 0)
   {
      showText ("No messages.");
   }
   else if (index < vector.size () - 1)
   {
      showText ("Message " + (++index + 1) + '?');
   }
   else
   {
      showText ("Erase all?");
      index = -1;
   }
}

/**
 * Displays the message from the local store (the Vector), as
 * identified by the current index (a negative index means
 * 'erase all messages').
 */
public synchronized void display ()
{
   if (vector.size () == 0)
   {
      showText ("No messages.");
   }
   else if (index < 0)
   {
      showText (vector.size () + " messages erased.");
      vector.removeAllElements ();
   }
   else
   {
      try
      {
         scroll ((TextMessage) vector.elementAt (index++));
      }

      catch (ArrayIndexOutOfBoundsException abe)
      {
         showText ("No more messages.");
      }

      catch (JMSException jmse)
      {
         showError (jmse);
      }
   }
}
/**
 * Displays the specified message content by scrolling across the
 * text field.
 */

public synchronized void scroll (TextMessage message) throws
   JMSException
{
   String text = index + ": " + message.getText ().trim ();

   showText (text);

   for (int i = COLUMNS; i < text.length (); i++)
   {
      messageText.setCaretPosition (i);

      try
      {
         Thread.sleep (200);
      }

      catch (InterruptedException ie)
      {
         ;
      }
   }
}
/**
 * MessageListener interface method for asynchronous message
 * delivery. The pager application is passed newly arrived
 * messages by the invocation of this method.
 */

public synchronized void onMessage (Message message)
{
   System.out.print ((char) 0x07);   // beep
   vector.addElement (message);
   showText ("New message!");
   index = vector.size () - 1;
}
/**
 * ExceptionListener interface method for asynchronous exception
 * notification.
 */

public void onException (JMSException e)
{
   showError (e);
}
/**
 * Drops the JMS service provider connection.
 */

public synchronized void disconnect ()
{
   try
   {
      subscriber.setMessageListener (null);
      subscriber.close ();
      session.close ();
      connection.close ();
   }

   catch (JMSException jmse)
   {
      ;
   }
}
/**
 * Convenience method for displaying text in the central text field.
 */

synchronized void showText (String text)
{
   messageText.setText (text);
   }

   /**
    * Convenience method for displaying a message in the status area.
    */

   void showStatus (String message)
   {
      statusLabel.setText (message);
   }

   /**
    * Convenience method for displaying an error.
    */

   void showError (Exception e)
   {
      if (e instanceof JMSException)
      {
         System.err.println (e +
            " (" + ((JMSException) e).getLinkedException () + ')');
      }
      else
      {
         System.err.println (e);
      }

      showStatus (e.getMessage ());
   }
   /**
    * Container method establishes a border margin (cosmetic).
    */

   public Insets getInsets ()
   {
      Insets insets = super.getInsets ();

      return new Insets (insets.top + 6, insets.left + 6,
         insets.bottom, insets.right + 6);
   }
}
```

In next month's *MQ Update* we publish a follow-up article on a JMS publisher application that complements this month's pager application.

Chris Markes
HCI Architect
IBM UK Laboratories

MQSeries programming with ActiveX

If you read my previous article (published in *MQ Update* November 1999), then you should have a good idea of how to use the procedural method of accessing MQ-related functions via Visual Basic. In this follow-up article, I'm going to discuss how to use the MQSeries ActiveX Automation Classes (MQAX) to allow applications that are based on Microsoft's Component Object Model (COM) to access all the functions and features of the MQSeries API, allowing full interconnectivity to other MQSeries platforms and environments.

COM is an object-based programming model that provides a framework for integrating components. Using COM-capable development tools and environments, such as Visual Basic, PowerBuilder, Delphi, Internet Explorer, and Active Server Pages (ASP), applications can access MQSeries services.

PROPERTIES, METHODS, AND EVENTS

ActiveX is a component development architecture based on COM that provides interfaces that can be accessed dynamically by applications. It supports properties, methods, and events that define its behaviour. A class's data (its settings or attributes) are called *properties*, while the procedures that can operate on a class and actions it can perform are called *methods*. An *event* is an action or response recognized by a class, such as a mouse click or a key press, and you can write code that responds to the event. *Exceptions* are also part of an ActiveX class. When unexpected events interrupt the normal processing of an application, an exception is usually raised to inform the class that an error has occurred.

MQSERIES CLASSES

MQAX provides sets of classes to enable applications to access other applications running in non-ActiveX environments. These classes are the familiar objects that you usually encounter and use in developing MQSeries applications. The following describes each class and how to use it.

MQSession class

This is the root class, which provides an object that contains the last action performed on any MQAX object, including the status of the operation (whether it was successful). Properties such as *CompletionCode*, *ReasonCode*, and *ReasonName* can be derived through this class. Another property, *ExceptionThreshold*, defines the circumstances under which the MQAX classes throw exceptions, the default being *MQCC_FAILED*. A value greater than *MQCC_FAILED* effectively prevents exception processing, leaving it up to the programmer to implement code that checks the *CompletionCode* and *ReasonCode*.

The code below shows how to create an *MQSession* object in your Active Server Page using VBScript. The code attempts to connect to a queue manager – as the class doesn't throw exceptions, code to handle errors should be implemented.

CREATING AN MQSESSION

```
<%@ Language="VBScript"%>
<HTML>
<BODY>
<%
Set mqsess = Server.CreateObject ("MQAX200.MQSession")
'Set a high exception so we could handle errors
mqsess.ExceptionThreshold  = 3
'Connect to default queue manager
Set QMgr = mqsess.AccessQueueManager("")
   If mqsess.CompletionCode <> 0 Then
      Response.Write "Unable to connect to Queue Manager. "
   Else
      Response.Write "Connected to Queue Manager"
   End If
%>
</BODY>
</HTML>
```

If the creation of an *MQSession* object is successful, and the connection to the queue manager also succeeds, a reference to the *MQQueueManager* is returned along with the corresponding *CompletionCode* and *ReasonCode*.

You should be aware that there is only one *MQSession* object per client process. However, MQAX supports a free-threading model that allows objects to be shared between threads. Generally, COM follows an apartment threading model in which a process is either a single-threaded apartment (STA) or a multi-threaded apartment (MTA). STAs contain only one main application thread, which belongs to the apartment itself, and calls can be received only from that thread. MTAs can have more than one application thread and the application can receive calls from any of the apartment's threads. COM is responsible for managing threads, apartments, and the creation of objects. It also marshals interface pointers across thread boundaries.

MQQueueManager class

This class represents a connection to a queue manager. Applications must create an object of this class and connect it to a queue manager that's running either locally on an MQSeries server or remotely, with access provided by an MQSeries client. The connection to the queue manager is automatically terminated when the *MQQueueManager* object is destroyed. This class, like other MQAX classes such as *MQMessage*, also has its own *CompletionCode*, *ReasonCode*, and *ReasonName* properties that you can check after making a method call on the object.

Using Visual Basic, we'll establish connection to a queue manager and then try to check the status of our operation using the properties of the *MQQueueManager* object.

CONNECTING TO A QUEUE MANAGER AND CHECKING RESULTS

```
Dim QMgr as MQQueueManager         ' queue manager object

Set QMgr = New MQQueueManager    'Create an instance of a queue manager
QMgr.Connect                     'Connect to the default queue manager
MsgBox Str$(QMgr.CompletionCode) & " " & QMgr.ReasonName, vbOKOnly, _
       "Connect Result"
```

There can be only one *MQQueuManager* object connected to a queue manager per ActiveX instance. Once we are connected to a queue manager, the next class in the process enables us to access a queue.

MQQueue class

Access to an MQSeries queue is encapsulated in this class. An associated *MQQueueManager* object provides the connection to the queue manager. When an object of this class is destroyed, the queue that was opened is closed automatically. In order to access the queue, you must use the *AccessQueue* method of the *MQQueueManager* class.

We'll try to open a queue using PowerBuilder. In the code below, the queue *ProdLocQ* is opened with some appropriate options and is then associated with the queue manager *ProdQueueMgr*. The corresponding result is checked. A queue object must always associate itself with a queue manager object. Note the use of the continuation character, '➤', in the code below to indicate a formatting line break.

ACCESSING A QUEUE

```
olenvQMgr = Create OLEObject
ll_result = olenvQMgr.ConnectToNewObject("MQAX200.MQQueueManager")

olenvQueue=olenvQMgr.AccessQueue("ProdLocQ",MQOO_OUTPUT+ &
MQOO_INPUT_AS_Q_DEF,"ProdQueueMgr")

MessageBox ("Open results", "Completion Code: " + string
➤  (olenvQueue.CompletionCode) + ", &
Reason Code: " + string (olenvQueue.ReasonCode))
```

Normally, after opening a queue, we would *PUT* messages on or *GET* messages from the queue – the next class allows us to do just that.

MQMessage class

In order to send and receive messages to and from a queue, an object of this class must be used. This class represents an MQSeries message. The properties of an MQSeries message descriptor (MQMD) are contained in this class, which also provides a buffer to hold the message data. The code below illustrates how to create an MQMessage object and assign it data using Visual Basic.

CREATING A MQMESSAGE OBJECT

```
Set msgPut = New MQMessage
```

```
msgPutStr = "This is a sample message"
msgPut.MessageData = msgPutStr
```

Methods, such as those to copy data from an ActiveX application to an MQMessage object (the *Write* method) and from an MQMessage object to an ActiveX application (the *Read* method), are included in this class. Unlike procedural MQ API calls, which require you explicitly to declare the size of the data buffer (otherwise the buffer will contain no data), this class automatically manages the allocation and de-allocation of memory for the buffer. This makes programming much easier. The buffer size need not be declared as it grows to accommodate data written to it. However if the buffer size exceeds the *MaximumMessageLength* of a queue, then the message cannot be placed on the queue.

When trying to put or get different data types to and from a queue, you can use the available methods of the *MQMessage* object. For example, if you want to write a string message to a queue, you use the *WriteString* method:

```
    msgPut.WriteString(msgPutStr)
```

If you want to read a numeric value, such as a *Long* data type, from the message's data buffer, you can use the *ReadLong* method:

```
    LongMsg = msgGet.ReadLong
```

Some of *MQMessage* methods maintain a cursor position in the message buffer, which makes it easier to read or write incremental data. This is done using the *DataOffset* property of the *MQMessage* object.

Data conversion is performed by the *Read* and *Write* methods of the *MQMessage* class. Conversion is between ActiveX's internal formats and the MQSeries message format defined by the *CharacterSet* and *Encoding* properties of the message descriptor. Before issuing a *Write* method, you should set the values of the properties above, if possible, to match the characteristics of the receiver of the message. You don't normally have to do this when you are reading a message because the characteristics are set by the incoming MQMD.

After constructing a message, you can put an *MQMessage* object on an MQSeries queue using the *Put* method of the *MQQueue* class. This method takes a copy of the message's message descriptor (MQMD) and the data portion of the object being *PUT* and places the copy on the queue. Applications can delete or modify an *MQMessage* object after the *Put* method without affecting the message on the queue. The queue manager may adjust some fields in the MQMD when copying messages bound for an MQSeries queue.

The next example illustrates how to put a message on a queue using VBScript in an Active Server Page. The message string is assigned to the *MessageData* property of the *MQMessage* object and put on the queue *ProdLocQ*.

PUTTING A MESSAGE ON A QUEUE

```
<%@Language="VBScript"%>
<HTML>
<BODY>
<%
Set QMgr = Server.CreateObject("MQAX200.MQQueueManager")
Set Queue = QMgr.AccessQueue("ProdLocQ", MQOO_OUTPUT, "ProdQueueMgr")
Set msgPut = Server.CreateObject("MQAX200.MQMessage")
msgPutStr = "This is a sample text sent at:" & Time
msgPut.MessageData = msgPutStr
Queue.Put msgPut
%>
</BODY>
</HTML>
```

Using the *Get* method of the *MQQueue* class, messages can be read into an *MQMessage* object. Incoming messages completely replace any message descriptor (MQMD) or message data that may already exist in the *MQMessage* object with their own values, as long as the method call is successful. As mentioned earlier, the *MQMessage*'s data buffer automatically adjusts its size to match that of the incoming message data.

We'll try to get the message data that was sent in our previous code example from the queue using JavaScript in an Active Server Page.

GETTING A MESSAGE FROM A QUEUE

```
<%@ Language="Javascript"%>
<HTML>
<BODY>
```

```
<%
QMgr = Server.CreateObject("MQAX200.MQQueueManager")
Queue = QMgr.AccessQueue("ProdLocQ", MQOO_INPUT_AS_Q_DEF,
"ProdQueueMgr")
msgGet = Server.CreateObject("MQAX200.MQMessage")
msgGet.MessageId = msgPut.MessageId
Queue.Get msgGet
msgGetStr = msgGet.ReadString(msgGet.MessageLength)
%>
</BODY>
</HTML>
```

After receiving our message, we can check the various MQMD properties of the associated *MQMessage* object and examine such attributes as the *MessageId*, *GroupId*, *UserId*, and *CorrelationId* that we use to identify the messages received.

MQPutMessageOptions class

We could add *PUT* message options to our code by using the *MQPutMessageOptions* class. Options that control the action of putting a message on a queue are all included in this class, which contains the MQPMO data structure. When you create an object of this class, all its properties are initialized.

So, before putting our message on the queue using the section of code headed ‘Putting a message on a queue’, we could insert the following code:

```
    Set msgPutOpt = Server.CreateObject("MQAX200. _
                    MQPutMessageOptions")
    msgPutOpt.Options = MQPMO_FAIL_IF_QUIESCING
```

MQGetMessageOptions class

Likewise, if we want to include options when getting a message from a queue, we can use the *MQGetMessageOptions* class. This class is used to set various *GET* options for a message. The MQGMO data structure is encapsulated in this class. As with the *PUT* option class, all properties are initialized when you create an instance of this class.

To set message options within our Visual Basic application, we just need to create an *MQGeMessagetOptions* object and set the property *Options* to any acceptable set of values to control message retrieval.

In this example, we set the *Options* property to wait ten seconds for a message to arrive.

SETTING GET OPTIONS

```
Dim Queue As New MQQueue
Dim msgGet As New MQMessage
Dim msgGetOpt As New MQGetMessageOptions

'code for opening the queue is omitted

msgGetOpt.Options = MQGMO_WAIT
msgGetOpt.WaitInterval = 1000

Queue.Get msgGet, msgGetOpt
```

The following classes are available only in MQSeries Version 5.1 (both Client and Server) and not in the ActiveX SupportPac (MA7B). There are differences between the classes in Version 5.1 and those included in the SupportPac, including the additional features mentioned in the former. You should be aware of this and not install one over the other, as this may cause problems with your application.

MQDistributionList class

This is the class that contains a collection of different queues such as local, remote, and alias. You can use this class to send messages to several destinations with a single *PUT*.

MQDistributionListItem class

This class is used to manipulate elements of an ‘object record’ (MQOR), ‘put message record’ (MQPMR), and ‘response record’ (MQRR). These structures are included in this class, which associates them with an owning distribution list. This class is represented as instances of *MQDistributionList* objects.

Another feature that is found only in Version 5.1 is that the coordinating of units of work by queue managers may involve external resource managers. For instance, we may want messages received to be coordinated with database updates in a unit of work. Thus, if the operation succeeds, the work is committed, the database is updated, and the message is removed from the queue. However, if an error occurs, the work is rolled back, the database is not updated, and the message remains in the queue.

COORDINATING UNIT OF WORK

```
Qmgr.Begin
Queue.Get msgGet, msgGetOpt
'Database Transaction
EXEC SQL UPDATE TABLE EMPLOYEE SET EMP_NO= :hEmpNo
'Check for errors, if none commit transaction
Qmgr.Commit
```

The following example illustrates the MQAX classes in action. It is written for an Active Server Page using VBScript. You need an MQSeries client if Internet Information Server (IIS) is not on the same system as the MQSeries server or the server install if the two are on the same machine. The installation comprises two files: *MQForm.html*, where you enter the queue name and the message to send, and *MQPut.asp*, which processes the data from the HTML form. (Note the use of the continuation character, '➤', in the code below to indicate a formatting line break that's not present in the original source.)

MQFORM.HTML

```
<HTML>
  <BODY>
    <!--'====================
        'create the HTML form
        '====================-->
    <FORM ACTION="mqput.asp" METHOD="POST">
      Queue Name: <INPUT TYPE="TEXT" NAME="QueueNm"></INPUT><BR>
      Message: <INPUT TYPE="TEXT" NAME="PutMsg"></INPUT>
      <INPUT TYPE="SUBMIT" VALUE="Put Message"></INPUT>
      <INPUT TYPE="RESET" VALUE="Reset"></INPUT>
    </FORM>
  </BODY>
</HTML>
```

MQPUT.ASP

```
<%@ Language="VBScript"%>
<HTML>
<TITLE>Simple Put Message</TITLE>
<BODY>
<%
```

```
'===========================
'retrieve values from the HTML Form
'===========================
QueueNm = Request.Form("QueueNm")
PutMsg = Request.Form("PutMsg")

'=====================
'create an MQSession object
'=====================

Set mqsess = Server.CreateObject ("MQAX200.MQSession")
On Error Resume Next
'=================================
'Set a high exception so we can handle errors
'=================================

mqsess.ExceptionThreshold  = 3

'=========================
'Connect to default queue manager
'=========================
Set QMgr = mqsess.AccessQueueManager("")

'============================================
' check for failures, then write appropriate message to the page
'============================================

   If mqsess.CompletionCode <> 0 Then
          Response.Write "Unable to connect to Queue Manager. "
   End If

'===================================
' declare constant, you can put this in an include file
'===================================
MQOO_OUTPUT = 16
MQOO_INPUT_AS_Q_DEF = 1
MQPMO_FAIL_IF_QUIESCING = 8192

'==========================
' access the queue that was specified
'==========================
Set Queue = QMgr.AccessQueue(cstr(QueueNm), MQOO_OUTPUT +
          ➤  MQOO_INPUT_AS_Q_DEF)
If mqsess.CompletionCode <> 0 Then
          Response.Write "Unable To Open Queue."
End If

'====================================================
' create a new message and put option object using the MQSession method
'====================================================
Set MsgPut = mqsess.AccessMessage()
Set msgPutOpt = mqsess.AccessPutMessageOptions()
msgPutOpt.Options = MQPMO_FAIL_IF_QUIESCING

'============================
' write the message , then put to the queue
'============================

MsgPut.WriteString cstr(PutMsg)
Queue.Put MsgPut, msgPutOpt

If mqsess.CompletionCode = 0 then
    Response.Write "The message has been successfully sent!"
End If

%>
</BODY>
</HTML>
```

Rommel K Abdon
Senior Systems Engineer
Client Server Technologies Inc (The Philippines)

MQSeries Everyplace configuration

One of the challenges of discussing MQSE is that it uses terms that are similar to those used with 'big' MQSeries, but with slightly different meanings and implications. To understand some of the differences, as well as highlighting how MQSE works, consider the following example configurations (Figures 1 to 3 on pages 32 to 34).

'Big' MQSeries has the concept of:

- An MQSeries server (the square in Figure 1)
- An MQSeries client (the triangle in Figure 1).

An MQSeries server possesses at least one queue manager and queue. In contrast, an MQSeries client possesses no queue or queue manager; instead it can 'put to' and 'get from' a queue managed by a queue manager on an MQSeries server.

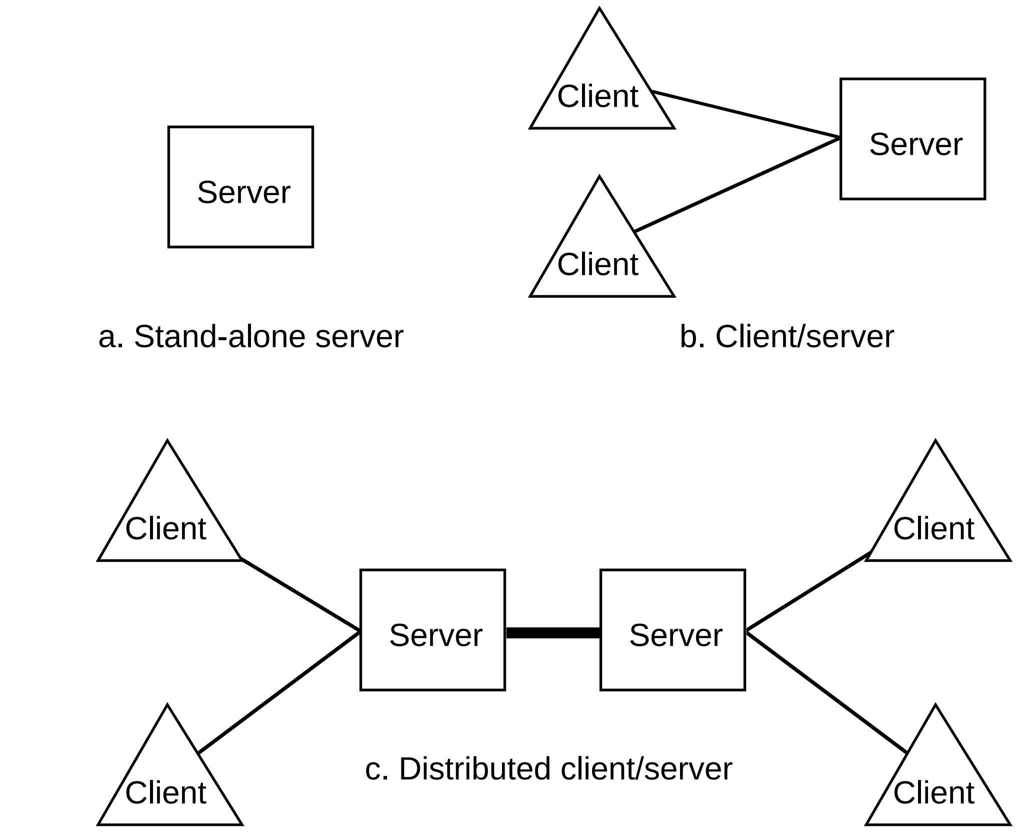


Figure 1: Stand-alone versus distributed client/server

While an MQSeries server can stand alone and/or talk to one or more other MQSeries servers (or clients), an MQSeries client cannot talk directly to another MQSeries client. In Figure 1 the regular lines represent synchronous links (client channels) between an MQSeries client and an MQSeries server, while the heavier lines represent full MQSeries asynchronous links (messaging channels).

Figure 2 depicts MQWin – the lightweight form of ‘big’ MQSeries originally available on Windows 3.1 and Windows 95/98. This has a queue manager that can stand alone or use radio or messaging channels to talk to an MQSeries server (which, in turn, can communicate with other MQSeries servers and/or MQSeries clients). MQSE will replace MQWin.

In the MQSE environment (Figure 3), the picture changes from that described for ‘big’ MQSeries. The PDA or other end point system is now called a ‘device’. In addition, MQSE introduces what IBM calls

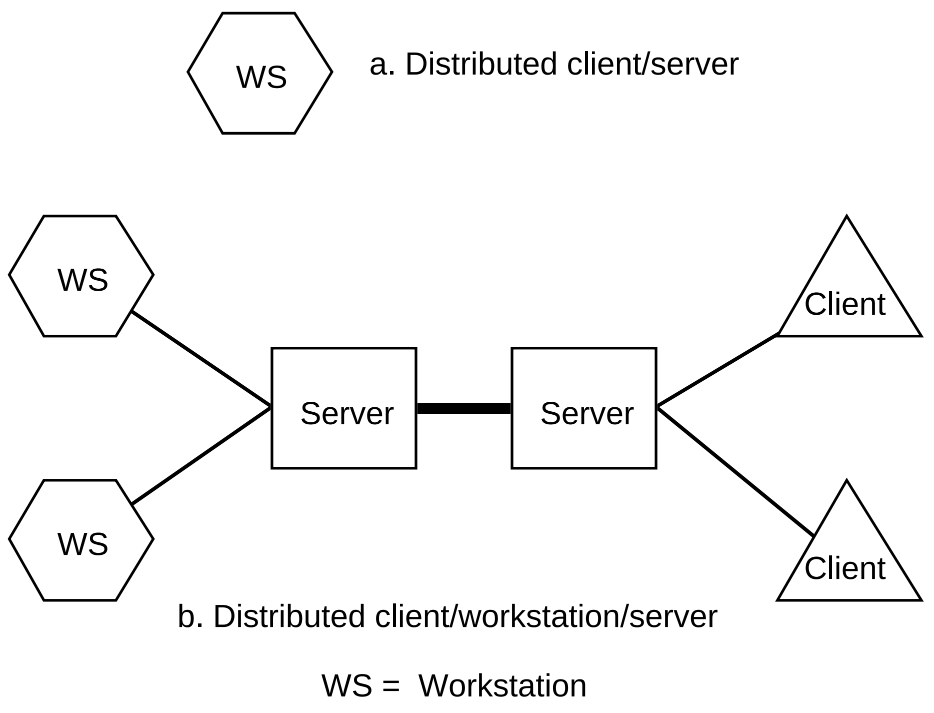

WS
a. Distributed client/server
WS
Client
Server
Server
WS
Client
b. Distributed client/workstation/server
WS = Workstation

Figure 2: Distributed client/workstation/server model

'gateways'. (While this is not an original term, it's the most 'descriptive' one that IBMers have yet been able to agree on.)

An MQSE device can stand alone, with its own queue managers, queues, and messages, or one device can talk directly to another, with no MQSeries server in between.

This is a major difference between MQSE and 'big' MQSeries – it's this that enables a Palm Pilot to send messages to an EPOC phone, assuming that both have MQSE-enabled applications. The dashed line that joins devices in Figure 3 is a new MQSE-specific channel that IBM calls a 'dynamic channel'.

A device can also talk to a gateway, which will initially be a Windows NT or Windows 2000 system. In most instances, a gateway has all the facilities of a device, but with the additional abilities to:

- Offer device-to-device connection via a hub (the gateway itself),
- Provide server capabilities for MQSE (for example home gateway capabilities,

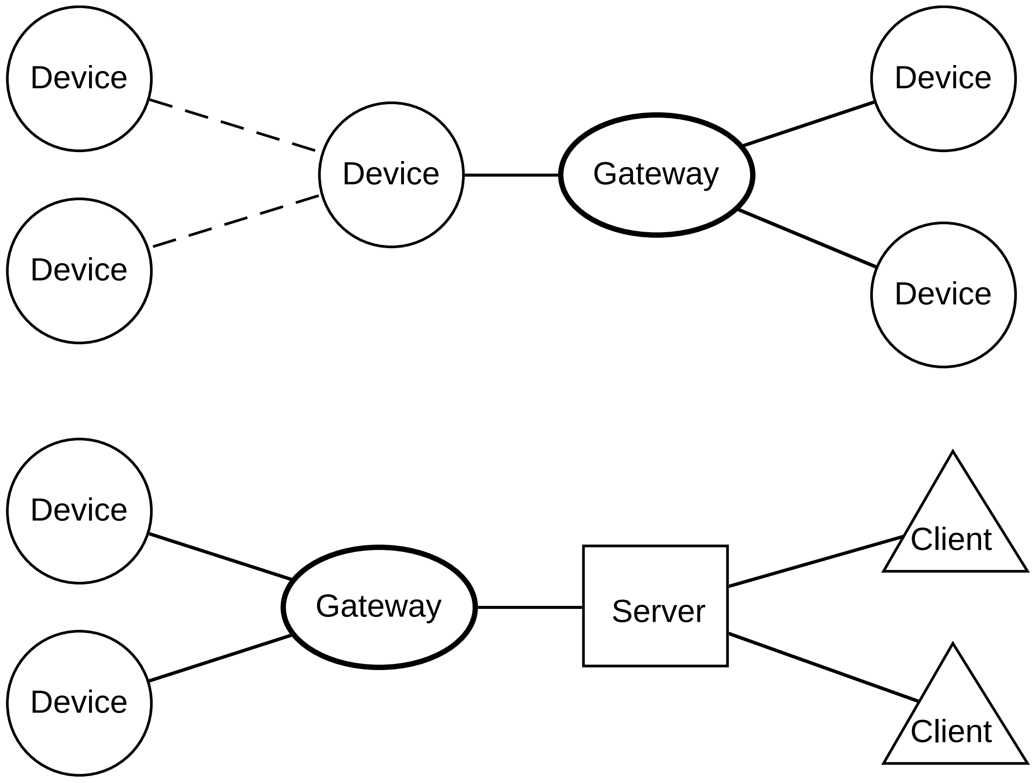


Figure 3: Device/gateway interaction

- Connect MQSE to conventional 'big' MQSeries message channels.

The home gateway queue manager is where the device looks for messages that may be awaiting delivery to it. After being disconnected for a period, the device contacts the home gateway queue manager to collect outstanding messages.

A device in an MQSE deployment can be:

- Stand-alone
- Connected to other MQSE-equipped devices
- Connected to a gateway.

An MQSE gateway provides:

- Device-to-device connections

- Device-to-gateway-to-device support (the MQSE gateway is a server for multiple devices)
- MQSE-to-MQSeries connections.

In the third instance, the gateway system hosts both the MQSE gateway itself and the MQSeries server. MQSE sends messages from its own queue managers/queues to/from the 'big' MQSeries queue managers/queues. Using this mechanism, messages can then be sent to or received from other MQSeries systems.

ASYNCHRONOUS AND SYNCHRONOUS POSSIBILITIES

One of the key differences between 'big' MQSeries and MQSE is that the latter offers the ability to know when a message arrives. In asynchronous messaging (at least with the 'big' MQSeries family), the sending application knows that the receiving application will receive the message – but not when. By contrast, MQSE enables you to store the message either asynchronously or synchronously on MQSE queues (and/or 'big' MQSeries queues via a gateway) anywhere in the 'MQ' network.

With MQSE asynchronous messaging:

- Control returns after the local 'put'.
- Delivery is assured.
- Messages are queued until delivery (the sending application doesn't know when the receiving application takes delivery of the message).
- Sending and receiving applications are decoupled.
- Network independence is introduced.
- MQSE and 'big' MQSeries work together.

By contrast, MQSE synchronous messaging is slightly different:

- Control returns to the application as you 'put' the message on a destination queue (after the remote 'put').

- The source application, which is responsible for error handling, knows that the message has arrived on the target system/queue.
- There is still a decoupling of the sending and receiving applications via the receiving queue.
- The network must be working (from source to destination).

The only real disadvantage is that all the communication links have to be available. On the other hand, the originating application on an MQSE-equipped device knows for certain that the message has arrived.

Synchronous messaging works between MQSeries Everyplace queues and also ‘big’ MQSeries queue managers that are directly attached to the MQSE gateway.

Another difference between MQSE and ‘big’ MQSeries is that the former was designed using object-orientated programming principles. Everything is an object – even messages are objects, so that you’re moving objects around the network. In this context, MQSE messages are objects that inherit properties from field objects. Understanding field objects is, therefore, important to the successful use of MQSE. A field object is a collection of fields where each field has a name, type, and value. The types include ASCII, Boolean, byte, integer, and many others. Strings, fixed arrays, dynamic arrays, etc are all supported.

What you ship across the network is not, therefore, just a string of bytes – it is a collection of fields whose names tell you what they’re shipping. Furthermore, these properties can be nested, so that you can build composite and complex messages. Just as importantly, when you receive a message, you know what it contains, what to do with it, and how to convert it – all because the message is an object.

Besides simplifying the use of MQSE, the field object approach can reduce what has to be sent over the network. If the message object can describe where some data is to be found at the destination, then this data does not itself need to be sent, thus reducing the payload and network requirements. Furthermore, a message object can, when it arrives, initiate other actions, such as starting an FTP session, invoking a process, making a database query, or even configuring a machine.

The only attribute that an MQSE message has to possess is a unique identifier. To make MQSE easy to use, this is created automatically (as a function of the time the object was created and the originating queue manager). All other fields in the object are determined by the application.

Unlike 'big' MQSeries, there are no 'manager' fields. MQSE does not possess the concept of a header and message data: there are only the fields and the message ID. If you want a message to have a priority, *CorrelId*, or target queue manager name, then you or the application must give it these attributes, which are optional.

The net result is a highly compact message, which can be made still more compact and secure via compression and encryption. Indeed, MQSE is so flexible that you can even choose which fields need to be encrypted or compressed – if that's what you need. Everything, including compression and security, is field object based.

QUEUE MANAGER AND QUEUE OPERATIONS

Now that the concepts of field objects, messages, queues, and queue managers have been outlined, the operations that can be applied should be fairly self-evident. An application talking to a queue manager can 'browse', 'get', 'put', 'delete', 'listen', and 'wait'.

Because features like the *CorrelId* are not built into MQSE (you have to implement them yourself, as described above), all the fields in a message can be used in retrieval operations. You can browse or add a filter, whichever you prefer. And you can browse either local queues or remote synchronous queues, assuming that you have the right permissions. You can even browse with a lock, only reaching those messages to which you have the correct key. Similarly you can 'get', 'put', or 'delete' in the same fashion.

If you choose to 'listen', you can wait until you receive an event that's triggered by a message appearing on an appropriate queue. You can then browse the queue or take whatever other action is appropriate.

Where MQSE is to work with 'big' MQSeries, certain types of field are expected, *CorrelId* being the obvious one. If these are present in a message from 'big' MQSeries, MQSE will generate indices as messages are stored. You can then retrieve messages based on their *CorrelId*, using indices to speed up access.

When accessing remote queues, the relevant authentication, encryption, and compression information is needed. To obtain this you need access to the remote queue definition that keeps this information – which explains why IBM incorporated remote queue definitions. MQSE also uses remote queue definitions to control whether access is synchronous or asynchronous, for routing, roaming, and dial-out support.

The new 'dynamic channel' – used for queue-manager-to-queue-manager connection – also possesses authentication, compression, and encryption. This is based on what you are trying to do. For example, if you are putting an encrypted message on an encrypted queue, MQSE automatically sends the message over an encrypted channel.

All configuration data is held in a registry along with queue definitions, queue manager definitions, remote definitions, etc. In addition, security certificates are held in the registry. This is stored locally, accessed through the gateway, or shared across several gateways.

MQSERIES AND MQSE MESSAGE INTERCHANGE

Three basic scenarios serve to illustrate how 'big' MQSeries works with MQSeries Everyplace.

In the first, an MQSE network is connected to an MQSeries network, which is then connected to another MQSE network. If you send a message all the way across this infrastructure, your MQSE-sourced message will arrive unchanged at the destination MQSE network, even though it was passed, via MQSE gateways, through the MQSeries network. In other words, MQSE can be used both around and across an existing MQSeries installation.

The same is true if you have two MQSeries networks with an MQSE network in between – MQSeries messages can flow across MQSE networks unchanged. This could be used, for example, as a way to pass through Internet firewalls.

In the second scenario, an MQSE queue manager wants to send a message to an existing MQSeries application that expects a specific string of bytes to arrive on an MQSeries queue. MQSE enables you to construct any message you desire and then send the message to an MQSeries queue manager. The key here is the ability to build the correct message in MQSE, and there are tools that allow you to do just this.

As with the previous example, the reverse is also true, so that an MQSeries message can flow to an MQSE device.

The third scenario involves conversion between MQSE and ‘big’ MQSeries. MQSE provides facilities for converting messages between ‘big’ MQSeries and MQSE. For example, MQSE can construct ‘big’ MQSeries headers, etc, as needed. Similarly you can construct a message body as required.

The net result of all this is that there are comprehensive facilities to exchange messages between every version of ‘big’ MQSeries and MQSE. In due course, there will also be a common set of tools and utilities for the two MQSeries ‘families’, and also a common set of addressing schemes.

On the API front, MQSE uses the ‘SPI’, which is a subset of the AMI (with some extensions). MQI and CMI support (for those who want it) will come later. One reason for the MQI coming later is that it at least doubles the memory requirement! Similarly, clustering support, already in ‘big’ MQSeries, will follow in a later version of MQSE.

AVAILABILITY AND ROLLOUT

Recent statements suggest that the NT version of MQSE will be out in June 2000, with support for AIX and Solaris gateways coming later in the year. In any case, the first batch of MQSE systems will mostly be packaged within larger solutions, which could be sold by system vendors or by system integrators creating specific solutions.

In the first release, dynamic channels will have support for both TCP/IP alone and HTTP over TCP/IP (though the latter will be measurably slower than the former). The Wireless Application Protocol will be added at some later time.

The initial devices to be supported include EPOC, WinCE, Windows 95/98, Windows NT, Windows 2000, OTI's Neutrino, the Palm OS, and Synergy (for Sharp's Zaurus machines). The MQSE Gateway will initially be Windows NT/Windows 2000-based (with AIX, Solaris, HP-UX, and Linux to follow).

From the start XOR/RLE compression is available. Operating system, private file, and digital certificate authentication will also be there. Encryption will include XOR, table substitution, 55-bit DES, 1024-bit RSA, triple DES, RC4, and RC6.

BUILDING BLOCKS

Finally, to summarize, a different way to understand MQSeries Everyplace is to think of it as Lego building blocks with different kinds of connector:

- A device can spawn multiple connectors (though there will be a limit on their number). These are 'dynamic channels'.
- A gateway can spawn connectors to 'big' MQSeries message channels, as well as dynamic channels.

Using the building block approach, you can build whatever configurations you like and connect them whichever way you choose. Devices can:

- Talk to each other without a gateway or talk to 'big' MQSeries via a gateway.
- Support more than one queue manager, and also communicate via both synchronous and asynchronous messaging.
- Exchange messages in real-time without queuing (the default).
- Optionally be configured to support asynchronous messaging with local queues, though this will increase the size of MQSE's footprint on the device.

By using these building blocks, MQSeries Everyplace enables the power and flexibility of generations of mobile devices (and dispersed controllers) to be brought into the wider IT environment.

CONCLUSION

MQSE is not the same as 'big' MQSeries. It is designed for smaller devices – PDAs, mobile phones, and the like – operating in mobile and even hostile conditions with appropriate security, but without losing the primary attraction of the MQSeries family – namely reliable messaging and queuing.

While MQSE is not an 'end-user application', it is the underpinning that can make such applications usable by providing the dependable infrastructure on which businesses can rely. With the relentless increase in the number of mobile devices deployed, applications to exploit this hardware are needed. Before MQSE, this was difficult in data processing terms because the communications were exposed and insecure. With the arrival of MQSE this changes, and applications on the device – and their users – can be knitted into the broader information processing and communications infrastructure, including existing 'big' MQSeries.

At base the real attraction of MQSE is its ability to broaden the range of devices that can reliably be part of the business, in spite of hostile communication environments.

Charles Brett
President
C3B Consulting (UK)

Configuring distributed queuing without CICS

The version of *CSQ4INPX* supplied in the *MQM.SCSQPROC* dataset can be customized to meet the requirements of a specific application. The sample version below contains a set of commands that could be issued whenever distributed queuing without CICS is started. It is used to start a listener for each communication protocol that you use. Define the correct LU name for the listener, which must be unique to each MQ subsystem. TCP/IP port 1414 is reserved for MQSeries, but additional ports can also be defined. This is very useful for distinguishing individual subsystems. You can also display the status of all the channels that are current.

AN EXAMPLE OF CUSTOMIZED CSQINPX

```
*****************************************************************
*
* @START_COPYRIGHT@
*   Statement:     Licensed Materials - Property of IBM
*
*                  5695-137
*                  (C) Copyright IBM Corporation. 1993, 1996
*
*   Status:        Version 1 Release 1
* @END_COPYRIGHT@
*
*****************************************************************
*
*               IBM MQSeries for MVS/ESA
* CSQINPX sample
*
*****************************************************************
*
* This sample data set contains an example of a set of commands
* that could be issued whenever distributed queuing without CICS
* is started.
*
*****************************************************************
* You must supply your own values for items shown thus:
*
*     ++value++
*
*****************************************************************
*
*
*****************************************************************
* Start Listeners
*****************************************************************
*
* You must start a listener for each communications protocol that
* you use.
*
******
* The lu name below will need to change for each QMGR on each LPAR
START LISTENER TRPTYPE( LU62 ) LUNAME( MVSMQLU1)
START LISTENER TRPTYPE( TCP ) PORT( 1414 )

*
*****************************************************************
* Start and Stop Channels
*****************************************************************
*
* Sender channels normally start automatically when a trigger
* message is put on the channel initiation queue.  Similarly,
* receiver and server channels start automatically when a message
* is received from a remote queue manager.  They stop
* automatically when there is no further work for them.
*
* Starting such channels manually is necessary only when they have
* stopped because of an error, or they have been stopped manually.
* Both these conditions are cleared when the channel initiator is
* started, so there is no need to issue any START CHANNEL
* commands.
*
* However, if there are certain channels that you do NOT want to
* start automatically, then you should issue STOP commands for
* them when the channel initiator starts.  Later, when you want
* them to start, you will have to issue START commands for them.
*
******
*STOP CHANNEL( '++channel-name++' )

*
*
*****************************************************************
* Display channel status
*****************************************************************
*
* Show the status of all the channels that are current.
*
******
DISPLAY CHSTATUS(*) CURRENT ALL

*
*
*****************************************************************
* End of CSQ4INPX
*****************************************************************
_
```

Saida Davies
IBM (UK)

MQ news

IBM has released MQSeries Everyplace for Windows V1.0 and MQSeries Adapter Offering V1.0, plus MQSeries V5.1 for Compaq Tru64 Unix and a link to R/3 for Compaq Tru64 Unix V1.2.

MQSeries Everyplace for Windows V1.0 (formerly ‘MQSeries Lite’) extends the software to PDAs and other mobile devices used by remote workers. It allows access to and exchange of data across public networks and supports Java applications.

The MQSeries Adapter Offering uses Open Applications Group standards for business object documents. It provides applications with a common language they can use to talk to each other through standard MQSeries messages, without a transformation engine. Adapter Kernel provides the run-time environment for adapters built using an adapter builder toolkit.

MQSeries for Compaq Tru64 Unix V5.1 ties into Tru64 Unix and claims to offer functional parity with the other MQSeries V5.1 products. The link for R/3 for Tru64 Unix V1.2 is functionally equivalent to the SAP R/3 links already available, which allow the exchange of data between ERP systems and all the platforms on which MQSeries runs.

For further information, contact your local IBM representative.

* * *

Prolifics has announced a RAD tool for IBM’s WebSphere. Panther for IBM WebSphere, the new product, is the latest member of the company’s existing line of Panther products. It’s a framework for building WebSphere-ready transactional applications; it includes visual tools, an object repository, and application building blocks.

Panther also provides facilities for building server-side EJBs, and includes a library of pre-built classes, methods, and components for application development. The product also supports Rational Rose for database design and modelling.

Out now, details on pricing are available on request from the vendor.

For further information contact:
Prolifics, 116 John Street, New York, NY 10038, USA
Tel: +1 212 267 7722
Fax: +1 212 608 6753
Web: http://www.prolifics.com

Prolifics, 4 Chiswell Street, London EC1Y 4UP, UK
Tel: + 44 20 7786 9555
Fax: + 44 20 7786 9556

* * *