MQ update

14

August 2000

In this issue

3 A JMS publisher application
12 MQSeries and 'implicit transaction processing'
18 MQSeries Integrator V2.0
34 Customizing CSQ4APPL and CSQ4INP2
44 MQ news

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.

A JMS publisher application

This article is a companion to the *MQSeries' JMS 'publish-and-subscribe'* article published in last month's issue of *MQ Update* (Issue 13, July 2000). That article described the use of the Java Message Service's (JMS) publish-and-subscribe API, illustrating this with an example pager application (JMSPager). This article complements the previous one with an example publisher application that can be used to send messages to a JMSPager.

The JMSPageSender application, like JMSPager, is an AWT-based application that extends the *Frame* class. It maintains a single *TopicConnection* and uses an anonymous publisher to publish messages to topics named at publication time, rather than being associated with a single topic from creation.

Much of the set-up code for this application is the same as that for the JMSPager application. However, I'll briefly summarize it here.

The MQSeries JMS provider classes are supplied in MQSeries SupportPac MA88, which is available for download for free from the IBM MQSeries Web site for Windows, AIX, HP-UX, and Solaris (the list of references at the end of this chapter tells you where to find it). 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 also in the SupportPac, and the MQSeries publish/subscribe component is in SupportPac MA0C (again, a reference to the Web site can be found at the end of this chapter).

To configure the environment, you need to set the following:

- 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.

- 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.bim.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* in the case of Windows NT) sub-directory of the MQSeries/MA88 installation.

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 *INITIAL_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.

JMSPAGESENDER

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

/**
 * A simple AWT GUI application demonstrating the basic capabilities
 * of JMS publish/subscribe. A JMSPageSender publishes short text
 * messages to a specified topic.
 */
```

```
public class JMSPageSender extends Frame
{
   static InitialContext context;

   TextComponent topicText;
   TextComponent messageText;
   Button clearButton;
   Button sendButton;
   Label statusLabel;

   TopicConnection connection;
   TopicSession session;
   TopicPublisher publisher;

   int index = 0;

   /**
    * Runs the JMSPageSender application.
    */

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

         new JMSPageSender ().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);
      }
   }

   /**
    * 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.
 */

public JMSPageSender () throws
   JMSException, NamingException
{
   super ("JMSPageSender");

   Panel p;

   setLayout (new BorderLayout (6, 6));
   setBackground (Color.lightGray);
   add ("North", p = new Panel (new BorderLayout (6, 4)));
   p.add ("West", new Label ("Pager ID:"));
   p.add ("Center", topicText = new TextField ());
   p.add ("South", new Label ("Enter text message: "));
   add ("Center", messageText = new TextField (30));
   add ("South", p = new Panel (new BorderLayout (6, 4)));
   p.add ("West", clearButton = new Button (" Clear "));
   p.add ("East", sendButton = new Button (" Send "));
   p.add ("South", statusLabel = new Label ());
   pack ();
   setResizable (false);

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

   clearButton.addActionListener (new ActionListener ()
   {
      public void actionPerformed (ActionEvent event)
      {
         messageText.setText ("");
      }
   });

   sendButton.addActionListener (new ActionListener ()
   {
      public void actionPerformed (ActionEvent event)
      {
         publish (topicText.getText (), messageText.getText ());
      }
   });

   connection = getConnection
      (System.getProperty ("jmspager.server.name", "jmspager"));
   session = connection.createTopicSession (false,
      Session.AUTO_ACKNOWLEDGE);
   publisher = session.createPublisher (null);
}

/**
 * Publishes a text message to the named channel.
 */

public void publish (String topicName, String messageText)
{
   try
   {
      Topic topic = getTopic (topicName);
      TextMessage message = session.createTextMessage ();

      message.setText (messageText);
      publisher.publish (topic, message);
      showStatus (++index + ": published to " + topicName);
   }

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

   catch (NamingException ne)
   {
      showError (ne);
   }
}

/**
 * Drops the connection to the JMS service provider.
 */

public void disconnect ()
{
   try
   {
      publisher.close ();
      session.close ();
      connection.close ();
   }

   catch (JMSException jmse)
   {
      ;
   }
}

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

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

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

public 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);
   }
}
```

A publisher with no specified topic name is created after the connection and session are established. The topic on which messages are published is taken from an entry field in the GUI. The publish method then publishes a message to the named topic.

RUNNING THE EXAMPLE

Starting the JMSPager and JMSPageSender

To run the JMSPager and JMSPageSender, use the **jms** invocation script or batch file:

```
$ jms JMSPager &
```

or, to specify an explicit topic, use the following command:

```
$ jms JMSPager pager-id &
```

The queue manager and broker must be available, and all JNDI objects must be defined to the topic connection factory and topic. When first invoked, the pager is 'off'. We also need to start a page sender:

```
$ jms JMSPageSender &
```

Sending messages

To send a message, enter the pager ID in the upper text field of the page sender (see Figure 1 overleaf), write a short message in the lower text field, and press 'Send'.

Figure 1: The page sender's GUI

Receiving messages

The pager application only receives messages when it's 'on'. To turn it on, select the checkbox marked 'active' on the GUI (see Figure 2). When turned on, the pager application displays the number of new and saved messages.

Figure 2: The pager's GUI

To display each message in turn, press the right-hand button (marked '>'). While active, the pager application will beep and display 'New message!' as soon as a new message is received.

MQSERIES-SPECIFIC TOPIC CONNECTIONS AND TOPICS

In the example application, references to JMS-administered objects – *Topic* and *TopicConnectionFactory* – are obtained through the JNDI. As with point-to-point messaging using MQSeries's implementation of JMS, provider-specific instances of these objects can be instantiated directly using the *MQTopic* and *MQTopicConnectionFactory* classes respectively. These are obtainable from the *com.ibm.mq.jms* package.

RESOURCES

- JMS information:

  http://java.sun.com/products/jms/index.html
- JNDI information:

  http://java.sun.com/products/jndi/index.html
- MQSeries JMS implementation (SupportPac MA88):

  http://www.ibm.com/software/ts/mqseries/txppacs/ma88.html
- MQSeries publish/subscribe component (SupportPac MA0C):

  http://www.ibm.com/software/ts/mqseries/txppacs/ma0c.html
- MQSeries home page:

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

Chris Markes
HCI Architect
IBM UK Laboratories Ltd (UK)

MQSeries and 'implicit transaction processing'

The use of MQSeries for transaction processing (TP) systems is often discussed in journals like *MQ Update*. In these discussions, it is usually pointed out that, while MQSeries is essentially for asynchronous communication, it is nevertheless suitable for systems that require synchronous communication, such as those handling TP. However, while the mechanics of using MQSeries in such systems is often discussed in detail (see, for instance, *MQSeries in an OS/390 client/ server environment*, *MQ Update* Issue 2, August 1999), readers are often left to figure out for themselves what are the advantages, if any, of using MQSeries in such systems. Indeed, it's often unclear that there are differences between the types of TP carried out by, say, CICS systems and those based on MQSeries. This is the gap that this article seeks to bridge – to discuss the fundamental differences between TP systems that rely on messaging and synchronous communication technologies.

TODAY'S TP ENVIRONMENT

The existence of distributed processing has changed the way everyone, including IT, thinks about computing. Connectivity has moved to centre stage. While the distributed world may not yet be fully connected (whether by LANs, WANs, or the Internet), the need to connect applications grows relentlessly, with profound implications for a company's software infrastructure.

To complicate matters, specialist IT skills are constantly being diluted by the sheer volume of EAI and similar projects currently being implemented, along with the arrival of distributed (but not necessarily connected) databases and their related applications. The success in the nineties of products from SAP, BAAN, PeopleSoft, and Oracle on many different platforms attests to this.

MQSeries has a major role to play here, as it enables different applications to be reliably coupled together without the need to change the way the applications work. Furthermore, MQSeries' transactional capabilities enable what can be described as 'implicit transaction processing' (as opposed to 'pessimistic TP'). In a world that's migrating to Web-enabled e-business, this capability will be critical.

FLAVOURS OF TRANSACTION PROCESSING

Traditional commercial transaction processing is about passing the transactional 'ACID' test (Atomicity, Consistency, Isolation, and Durability) to ensure that each transaction is processed once and once only, and that the processing is accurate and appropriate. Most of the successful, dependable, and robust applications of the past, many of which still support today's businesses, were built on these properties and their associated techniques. This is sometimes known as 'pessimistic TP' because of the assumption that all will fail unless the whole end-to end transaction succeeds.

But requiring the ACID properties of pessimistic TP solves neither the transaction processing problems of e-business nor the problem of integrating enterprise applications. What is needed is a different approach that is more flexible, so that the new breed of distributed transactions can be supported.

Four assumptions underpin this requirement:

- A distributed transaction system is one where two or more physical systems, almost certainly acting as peers, are involved. Each of the systems may have one or more applications running on it.
- The system could include two or more workstations or mainframes but may also include other platforms, such as PCs and minicomputers.
- A client can act as a server and a server can act as a client at different points during processing – they may even act as both client and server at the same time.
- The objective is to ensure that any given business transaction that is initiated either runs to completion or fails in a predictable and understandable manner that's commensurate with the needs of the business.

MQSeries is relevant here. It links multiple platforms with a reliable, once only, assured delivery protocol and implementation. It enables different applications to be linked in order to deliver application integration. The way it achieves this is by ‘implicit transaction processing’.

IMPLICIT TRANSACTION PROCESSING

Implicit transaction processing (I-TP) is a term that applies to systems whose design is underpinned by a series of assumptions that are sufficient, individually and in aggregate, to allow one to depend on transactions being processed accurately and completely by them. These systems avoid the formality inherent in ‘pessimistic TP’ – the type of transaction processing that underlies systems based on CICS or Tuxedo. Implicit TP can be best understood in terms of the concepts of queuing and of decoupling of application parts – both of which are illustrated by the example in Figure 1 (opposite), which is based on an MQSeries implementation (though it applies equally to any other reliable queueing system).

Decoupling occurs when you separate the different parts of an application across one or more systems, regardless of whether the application is transactional. In Figure 1, system *A* hosts application *A*, which communicates with application *B* on system *B*. This is a simple but representative example of a distributed application.

Traditionally the development of an application (the design, build, and installation) would have an overall single point of control. However, if systems *A* and *B* are on different platforms (say one is a Windows NT system and the other a Unix server or CICS/MVS system), the designer must understand both of the application platform environments and the method of communication between them. Usually the result is a monolithic design that is difficult to test, maintain, and change. In addition, few individuals have the breadth of skills to deliver it.

A decoupled approach changes this. The example shown has five components when decoupling is considered:

- The first component is application *A*.

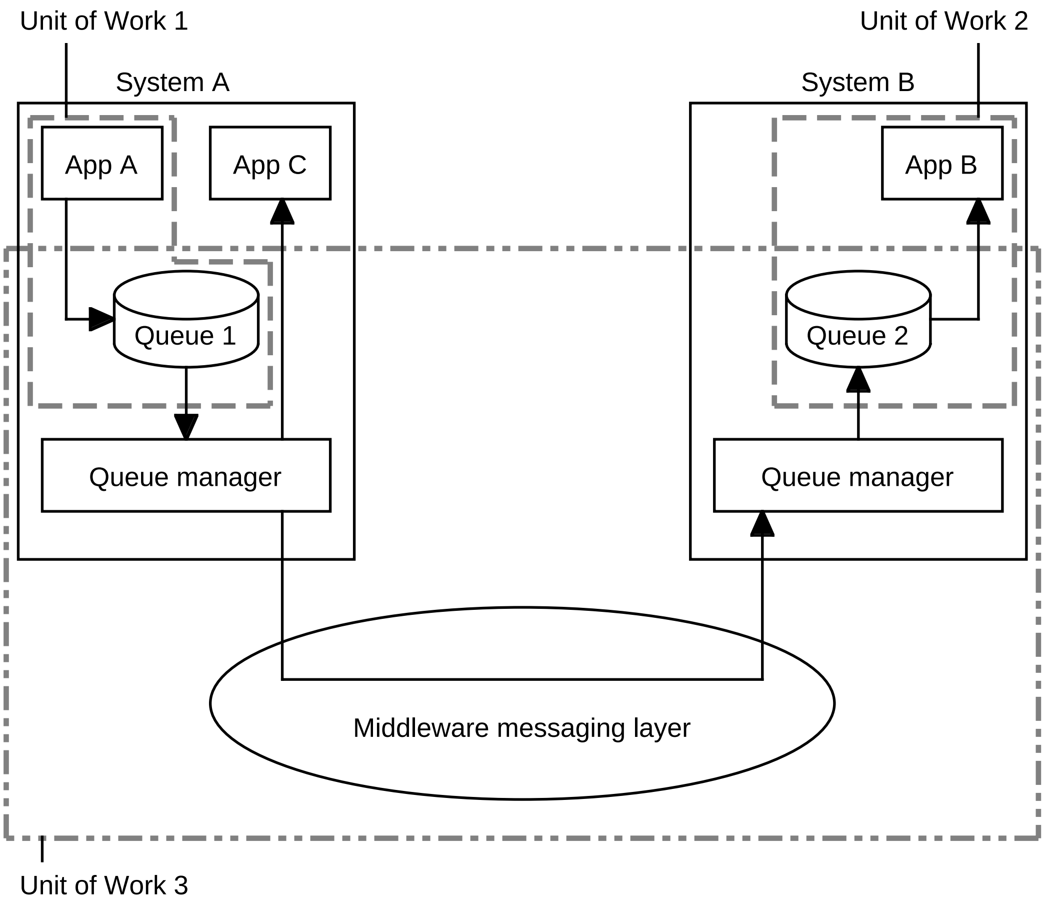

Figure 1: Units of work in a distributed processing system

- The second is queue *1* and its queue manager on system *A*, which communicate with other similar queues via an agreed mechanism (this could be messaging, RPC, 'conversational', or some other method).
- The third is queue *2* and its queue manager on system *B*. These communicate with other queues (including queue *1*).
- The fourth is application *B*.
- The fifth is application *C*.

It is assumed that:

- The queue managers (queue *1* and queue *2*) are able to communicate reliably with each other.

- Application *A*'s development tool (and, consequently, the completed application) can write to and read from queue *1*.
- Application *B*'s development tool (and, consequently, the completed application) can pick up information from queue *2*, as well as write any returns/responses back to queue *2*.

In this scenario, a developer can write application *A* as long as he or she has agreed the form of the dispatch between the application elements (application *A* and application *B*) with the developer of *B* (and the developer of *B* with the developer of *C*, and so on). The result is that the development of applications *A*, *B*, and *C* can be completed independently of each other, with developers using their own individual skills and without requiring them to know of each other's environment.

Applications *A*, *B*, and *C* also become components that can be swapped in or out so long as the dispatch structure (and a supporting queuing structure) remain in place.

Now, apply this type of development to the creation of applications for transaction processing and you have the essence of I-TP. As long as the following requirements are met:

- You can rely on one unit of work (*Unit of Work 1* in Figure 1) completing on the first platform.
- You can rely on a reliable transfer between a first and a second platform (*Unit of Work 3*), including rollback/recovery in the event of a failure in one of the communication links.
- You can rely on a process completing correctly on the second platform (*Unit of Work 2*).

Then you can implicitly rely on the business transaction (actually several transactions on different platforms), which is essentially what I-TP integrity is.

This is what MQSeries delivers. While implicit TP is ultimately not as reliable as pessimistic TP, which is what products like CICS, IMS, or Tuxedo deliver, it nevertheless offers a way to bridge heterogeneous systems in today's distributed environments. Furthermore, implicit

TP is also simple to understand and it enables originally isolated systems to be tied together.

The integrity of the transaction is, of course, dependent on the assumptions being observed. To strengthen these assumptions requires some additional development work on a coordinating application (see *Using MQSeries as a transaction coordinator* in *MQ Update* Issue 9, March 2000). This is especially the case if rollback and recovery are to be implemented across all the 'sub-transactions'. Furthermore, the queue managers on each system provide only queue-to-queue integrity – they do not provide application-to-queue integrity.

CONCLUSION

Transaction processing involving distributed systems that interoperate via message-oriented middleware technology differs in a number of subtle yet significant ways from traditional 'pessimistic' transaction processing systems, such as those based on CICS, Tuxedo, or IMS. For instance, a transaction in an implicit transaction processing system actually comprises a number of 'sub-transactions', each of which is based on a unit of work – the success of the whole transaction is based on the success of all sub-transactions. While traditional systems offer a more formal type of transaction processing, and also benefit from a greater success rate for transactions (MQSeries provides assured delivery of messages but is, nevertheless, constrained by the availability of the underlying network, which may be the Internet; this means that messages may not be delivered in time for transactions to succeed), implicit transaction processing systems also have their benefits. Among them is the ability to build systems from reliable components, which allows development projects to be rationalized into manageable parts, each of which requires a more limited skill set than would be the case if the project were to develop a 'traditional' distributed application based on, say, CICS. Another increasingly important benefit is that implicit transaction processing systems are especially suited to processing transactions over the Web.

Charles Brett
President
C3B Consulting

MQSeries Integrator V2.0

From a business standpoint, MQSI is used to integrate a number of applications and implement business processes. Users can define business rules using MQSI and use its built-in formatter to ensure that applications receive data in formats they support. MQSI also provides a framework that third-party vendors and resellers can use to provide connectivity between their products and those of other vendors. Resellers plug their products into MQSI by creating custom MQSI nodes (which are discussed later).

From a functional standpoint, MQSI provides a simple yet sophisticated way of processing messages en-route to their destination. MQSI functions as a 'message broker', also providing a message transport layer. A message broker acts as a hub, processing messages based on its own configuration and the contents of the messages themselves. The advantage of using a hub architecture is that it simplifies the task of application integration – systems are connected once to the hub rather than to each other system with which they need to interoperate. MQSI performs functions associated with processing, transforming, and distributing messages, while the base MQSeries product, which is supplied with MQSeries Integrator V2.0, handles message transport and queueing.

MQSI combines the message brokering function with publish/subscribe functionality, also adding a message dictionary and message warehouse. Message warehousing can be used for auditing message flows, replaying messages, and even "data mining", while message dictionaries make the process of transforming and reformatting messages much simpler by providing templates of message formats and structures.

Within the message broker, individual functions are assigned to a collection of interconnected nodes, and it is at the nodes where transformation and other processing actually takes place.

Nodes are implemented as Dynamically Linked Libraries (DLLs) in Win32 environments and as shared libraries in Unix environments, these being called by the message broker's execution environment.

‘Message processing nodes’ are – as you’d expect – responsible for processing messages. A message processing node is a ‘well-defined processing stage’, meaning that it performs a specific task or set of tasks on messages that pass through it in a message flow across a message broker. As well as being able to access data in the message payload and headers, message processing nodes can also access data outside the message flow, such as data stored on databases. Message processing nodes can also put messages on queues.

Nodes are ‘wired together’ to form a message flow. A message flow is initiated by an ‘input node’ (MQSI comes with a pre-defined input node – the MQInput node). The input and output of a node are referred to (and represented diagrammatically) as ‘terminals’, and terminals are joined by ‘connectors’ to implement a message processing framework. Note that connectors are there simply to represent message flows – messages are actually ‘transmitted’ by the passing of a reference (a pointer) to the message object in a method call. Terminals come in several varieties, including ‘in’ (for receiving messages), ‘out’ (for forwarding messages), and ‘failure’ (for sending messages in the event of an error or in response to an exception).

Users and vendors can create additional MQSI nodes that are then ‘plugged in’ to the system. Nodes should be written in C; if they’re written in another language, they’ll need a C wrapper. Sample code for plug-in nodes is provided with MQSI. In order to be fully backward-compatible with Version 1.0, Version 2.0 is also supplied with NEON Rules and NEON Formatter nodes, which allow access to the NEONRules and NEONFormatter engines of MQSeries Integrator Version 1.0. These nodes and other standard nodes that ship with MQSeries Integrator and are available via the design palette are described later.

MQSI also includes Control Center, a graphical tool for constructing and managing message flows. This tool can be used to ‘wire together’ message flows through MQSI nodes. According to IBM, Control Center “allows the fast creation, deployment, and control of message-based business solutions”. Message flows can be customized by using Control Center to alter the properties of individual nodes. Typical properties that can be customized are the filter statement of message processing nodes and the input queue of MQInput nodes.

MULTITHREADING

MQSeries Integrator offers multithreading for message processing, meaning that more than one message can be processed simultaneously. However, the way multithreading is handled needs a little explanation. Think of a message broker that implements a particular flow, comprising nodes and connectors, as being in some ways analogous to a class in an object-oriented programming language. When a flow is initiated, an instance of the message broker that implements it is created and given a thread from a thread pool. (A message flow is initiated by a message being placed on an MQSeries application queue, and is terminated when the last message in the flow is put on another MQSeries application queue.)

As a message progresses through a broker, it may come to a node that has more than one output terminal, such as the node marked ‘A’ in Figure 1, which has one ‘in’ terminal and two ‘out’ terminals. At this point, the process handling the message flow does not fork – the processing of each route through the system is handled “independently and sequentially”, meaning that the message broker executes Branch 1 in Figure 1 and then, when it has completed Branch 1 by putting a message on output queue Q2, it executes Branch 2. When this branch terminates by putting a message on Q3, the entire flow terminates, and the thread returns to the thread pool. The entire flow is thus handled by one thread, which means that no multithreading is available inside an MQSeries Integrator flow: multithreading is provided by offering multiple instances of the same flow, each of which can handle a separate input message.

Given that the processing of Branch 2 happens only once the processing of Branch 1 terminates, it is in the interest of those designing flows and developing nodes to ensure that the processing carried out by individual nodes is kept to a minimum, to ensure the rapid passage of messages through nodes. Particular attention should be paid to processing that requires I/O, which may block the processing of other branches in the flow.

Each message broker that implements a flow is allocated a pool of up to 256 threads, and each message that arrives in the input queue is allocated one thread.

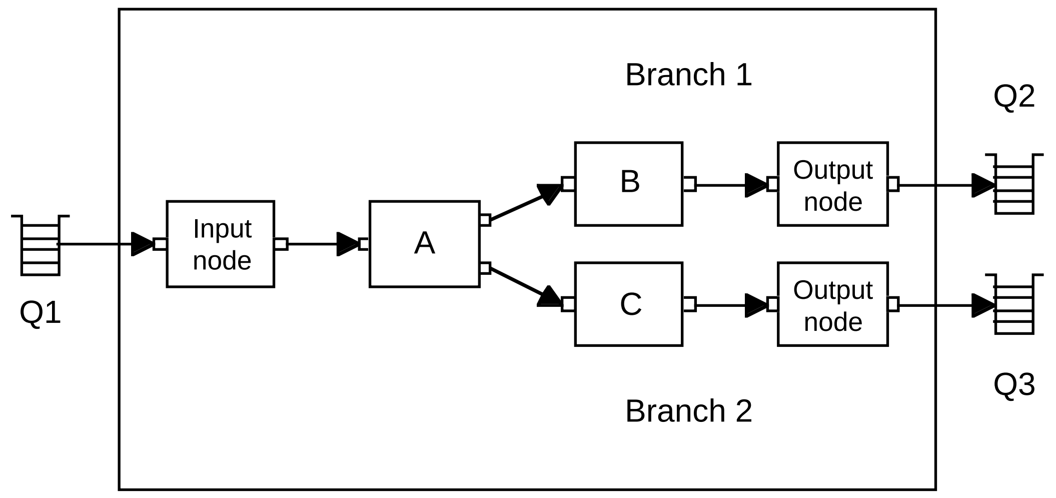

Figure 1: A message flow through a message broker

MQSI'S NODE PALETTE

The following message processing nodes ship with MQSI. Each has input and failure terminals, and may have a variety of output terminals depending on their function. The nodes that ship with MQSI are sometimes referred to as 'primitive nodes'.

Standard message processing nodes

- *MQInput*

  This node has already been discussed; it offers basic triggering and initiation functionality for a message flow. The node uses an MQGET call to receive a message from a designated MQSeries queue and passes the message on to the next node in the flow. The node has the regular input, output, and failure terminals, and also has a 'catch' terminal, whose function is to enable the node to catch an exception that occurs later in the message flow. This is necessary, for instance, to roll back a flow that is part of a transaction.

- *Check node*

  The check node is essentially for validating a message format in a message flow. It has three terminals: 'in', 'match', and 'failure'.

When a message arrives at the ‘in’ terminal, the ‘domain’, ‘set’, and ‘type’ attributes of the message type specification are compared with pre-set values. If they match, the message is deposited in the ‘match’ terminal; otherwise, messages are deposited in the ‘failure’ terminal.

- *Filter node*

  The filter node has five terminals: ‘in’, ‘true’, ‘false’, ‘unknown’, and ‘failure’. When a message arrives at the ‘in’ terminal, an SQL expression is applied to its content. The result of the SQL expression is used to decide whether to send the message to the ‘true’ or ‘false’ terminal; if the result is indeterminate, the message goes to ‘unknown’, and if an error occurs during processing, it goes to the ‘failure’ node.

- *Compute node*

  The compute node has the ability to receive a message of one type and output one of a different type. The content and attributes of the new message may be based on both the content of the message received and on values retrieved from an external relational database.

- *Extract node*

  The extract node produces an output message based on the content of the input message. Selected elements of the input message may simply be copied to the output message, though the elements may also be transformed in the process.

- *ResetContentDescriptor node*

  The ResetContentDescriptor node allows a message to be processed by another parser within the same message flow (the section on message dictionaries explains the way message parsing is handled within a message flow).

- *NEON Formatter node*

  The NEON Formatter node is somewhat different from other nodes mentioned so far in that it uses the NEONFormatter engine to transform messages. When a message arrives at the node, the node uses the NEON Repository to look up the message's format and a NEON format definition that's then used to transform the message, after which the message is passed on to the next node in the flow.

External database nodes

There are five basic nodes that are provided for modifying an external database: DataInsert, DataUpdate, DataDelete, Database, and Warehouse. Each of these nodes has three terminals: 'in', 'out', and 'failure'. While all database nodes alter the state of an external database in response to the arrival of a message, none of them alters the message itself (compare this with the Compute node, which is able to access an external database and use the information returned to process a message). The message types of the DataInsert, DataUpdate, and DataDelete nodes must be specified, though the input of the Database node can be generic XML. If any of the nodes is part of a transactional flow, then transactional integrity may be maintained either by an external agent or by the node independently committing the transaction.

- *DataInsert, DataDelete, and DataUpdate nodes*

  The DataInsert node performs an insert operation on the database specified. The insert is performed using an SQL statement generated by the node. While the statement generated may depend on data in the message itself, the message may simply act as a trigger for the insert.

  The DataUpdate node performs a similar function to that of the DataInsert node – one or more rows in a database are updated in response to the arrival of a message at the node. As with the insert generated by a DataInsert node, the update may be based on data in the message and is performed by an SQL statement generated by the node.

  The DataDelete node deletes one or more rows from a database in response to the arrival of a message at a node; again, the choice of rows may be independent of the content of the message, in which case the message simply triggers the action, or they may be chosen in some way that's based on the message's content.

Given that the DataInsert, DataUpdate, and DataDelete nodes expect messages of a certain type as their input, it's not unusual for their input to be the output of a filter node.

- *Database node*

  The Database node is more generic in function than the three database nodes discussed so far – it simply performs a database operation in response to the arrival of a message at the node. The database operation is performed using an SQL statement and the statement may be dependent or independent of the message content.

- *Warehouse node*

  The Warehouse node is somewhat different from the other database nodes discussed so far. While it also alters an external database in response to the arrival of a message, leaving the message itself unchanged, the database in this case is a 'message warehouse', which is used to store messages for auditing, batch processing, or deferred processing by the message broker. As with the other database nodes, an SQL statement generated by the node is used to insert messages in the database, and the messages themselves may be stored either as attachments to an index record built from the message schema or as a 'BLOB' attached to the index. Note that the Warehouse node may perform a calculation or transformation on the message before passing it on to the message warehouse.

Decision and routing nodes

Four of the nodes supplied belong in this category: MQOutput, MQReply, NEONRules, and Publication.

- *MQOutput and MQReply nodes*

  The MQOutput node is the counterpart of the MQInput node – it acts as the end-point of a message flow, so that messages leaving this node are put on an MQSeries queue. The actual queue to which a message is sent may be determined from the content of the message itself. An MQReply node is a type of MQOutput node in which the message is put on a queue specified by the ReplyTo field of the message header.

- *NEONRules node*

  As mentioned earlier, the NEONRules node is provided for backward compatibility with MQSeries Integrator Version 1. This node is able to pass messages to the NEONRules engine for evaluation. It is, therefore, a driver for the rules engine and is analogous to the daemon supplied with MQSeries Integrator Version 1. The NEONRules engine evaluates the message based on its stored rules and fires one or more terminals on the node depending on the outcome of the evaluation. The NEONRules node includes a number of terminals: 'PutQueue' is intended to be connected to the 'in' terminal of an MQOutput node for output to a named queue; 'noHit' fires when a rule could not be found to handle the message, which could be handled by further processing or output to a queue via an MQOutput node; and 'propagate' fires when the message was successfully evaluated with respect to a rule in NEONRules but no output queue was specified in the message. Also provided in the node is a failure terminal. Note that MQSI's 'destination list' feature allows messages leaving the node to be written to more than one MQSeries queue.

- Publication node

  The Publication node is part of MQSI's publish/subscribe functionality. A message arriving at this node is evaluated for topic and content and then sent either to message brokers or subscribers that have registered an interest in the topic and content. MQSI's Control Center is used to manage both published topics and subscribers.

Error handling and tracing

Three nodes are supplied with MQSI to handle errors and tracing: Throw, TryCatch, and Trace.

- *TryCatch and Throw nodes*

  The Throw node has only one terminal: 'in'. Its function within the message flow is to throw an exception that will either be caught by a TryCatch node earlier in the message flow or, if the exception is unhandled, cause a message flow to cease and (if necessary) a transaction to be rolled back. Note that the exception thrown may depend on the content of the message. The counterpart of the Throw node is the TryCatch node, which is used to catch exceptions thrown by a Throw node (if the exception isn't caught by the TryCatch node, it propagates back to the MQInput node and fires the node's 'catch' terminal). The message is received by TryCatch's 'in' terminal and is propagated to the 'catch' terminal unprocessed. If 'catch' is connected, further processing may then occur.

- *Trace node*

  The Trace node is simply for producing a trace for debugging. It has one 'in' and one 'out' terminal, and messages pass from 'in' to 'out' without being processed, though the arrival of a message triggers Trace to write a trace message to a specified destination to allow the tracking of messages.

Custom plug-in nodes

As mentioned earlier, MQSI allows users and third-party developers to create their own plug-in nodes. Plug-in nodes are either written in C or have C wrappers. Another requirement is that they conform with the "Message Flow Framework". Put simply, this ensures that new nodes are compatible with the MQSeries Integrator Design Tool, which is used to create message flows visually. This requirement is fulfilled by supplying an XML signature template.

PERSISTENCE

One point that needs clarification relates to persistent data. MQSI V2 maintains an external store of persistent data using a standard relational database (at the moment the choice is limited to DB2 and Microsoft SQL Server). This is used to hold configuration data for the message broker, and also stores information on publications, subscriptions, and the operational state of the broker when it was last running. It's tempting to think that this means that the persistent data store also holds state information on the state of message flows in particular instances of the broker. This is not the case; any persistent data relating to message flows needs to be stored by using an external relational database accessed via database nodes (an alternative is to use MQSeries Workflow to manage state information on message flows).

TRANSACTIONAL INTEGRITY

The ability to deliver transactional integrity is a feature of MQSeries. As MQSI uses the base MQSeries product as its messaging component, MQSeries can be used to provide transactional integrity to MQSI-based message brokers. (The ability to use MQSeries as a transaction coordinator, including the coordination of external XA-compliant databases via the XA interface, was discussed in the article *Using MQSeries as a transaction coordinator* in *MQ Update* March 2000).

Every MQSI message flow is initiated by an input node performing an MQGET to read a message from an MQSeries queue and is terminated when an output node performs an MQPUT to put a message on a queue. MQSeries can, therefore, provide transactional integrity at these two points. If an error occurs within a message flow resulting in an unhandled exception, the MQInput node will catch the exception via its ‘catch’ terminal, and the flow can be terminated at this point, if the MQInput node is suitably configured, and the transaction rolled back to the MQGET. Transactional integrity within MQSeries Integrator is thus maintained “within the bounds of the message flow”, so that transactional integrity is assured at the start and end of a message flow in a message broker. Regular message processing nodes are not responsible for maintaining transactional integrity, which is the responsibility of the MQInput node (which has to be configured to terminate the flow when an unhandled exception occurs) and database nodes (which must also be configured to roll-back uncommitted transactions involving changes to external databases).

The MQInput node includes a ‘transaction’ attribute that determines whether messages passing through the node are handled transactionally, and the MQOutput node includes a ‘persistence’ attribute that specifies the persistence of outgoing messages. Database nodes have a ‘coordination’ attribute that specifies whether database operations are part of a transaction. The attribute can also specify how and when a transaction is committed or rolled back.

MESSAGE DICTIONARIES

MQSI is able to process messages and external databases (and perform other work that's external to the message flow) based on the content of messages. A pre-requisite of this functionality is the ability to 'parse' messages – that is, to resolve message fields and other elements. This, in turn, requires a rapid, efficient, and manageable way of handling message formats, which is the aim of MQSI Version 2's message dictionaries.

Messages coming from an MQSeries queue are in 'wire message format'. The wire format is described in message's MQSeries message descriptor. MQSI processes messages in its own 'logical format', in which message fields are resolved and directly accessible by nodes. MQSI's Message Services Component is responsible for (among other things) parsing and deconstructing messages from 'wire' to 'logical' format. Thus, when an MQSI input node retrieves a message from a queue, it uses the message's wire format descriptor to obtain a logical message descriptor from the message dictionary. This is then used to parse and deconstruct the message, ready for processing by MQSI nodes. At the end of a message flow, which may include operations that transform the message from its original format, MQSI needs to reconstruct the message and obtain a wire message format descriptor that allows MQOutput and Publish nodes to put the message on MQSeries queues. This is, again, done with the aid of the message dictionary. Thus the message dictionary's role is to assist in the translation between MQSI's logical message formats and MQSeries wire message formats during the parsing, deconstruction, and reconstruction of messages.

Message dictionaries can handle messages comprising XML or byte structures, such as those created by C or COBOL programs, and they also handle MQMD message descriptors, RFH and RFH2 format headers, and NEONFormatter definitions. (RFH headers carry information used by MQSI, NEONRules, NEONFormatter, and MQSeries V5's publish/subscribe component. RFH2 headers extend this by carrying information about publish/subscribe flows and message sets. RFH2 headers can be further extended to carry proprietary data required by specific applications.)

In an MQSI message dictionary's database, message descriptions are known as 'message type definitions', and these can be grouped together as 'message sets' for deployment to relevant message brokers (each message set actually defines a message dictionary). A message type definition comprises a message model and a message template, where the message model identifies fields and other elements in a message template. All user interaction with message dictionaries is conducted via the Control Center GUI, which is used to perform such tasks as defining and maintaining message models.

Internally, the message dictionary comprises three components: the Message Repository Manager (MRM), the Resource Manager, and the Message Translation Interface (MTI). When a message dictionary is deployed on a broker, it is implemented locally by a Runtime Dictionary (RTD), which provides local message brokers and parsers with format definitions (this improves performance – the RTD acts as a locally-cached copy of that part of the message dictionary that's required by the local message broker). One limitation of RTDs is that their message definition store cannot be updated – if the message type definitions in the message dictionary on which they're based change, a new RTD must be defined at the message broker. Note that MQSI uses XML for storing all its configuration data.

MQSI handles the process of exchanging messages with queues using the four message attributes below, which are stored in the message header and also have corresponding definitions in the Control Center. Note that these attributes are not necessary for well-formed XML messages, which are 'self-describing'.

- The 'message domain', which identifies whether the message definition is managed by Control Center or its NEON counterpart.
- The 'message set' (also known as the 'project'), which groups message types within a specified domain.
- The 'message type', which identifies the exact structure of the message in terms of the number, location, and size of the message's fields.

- The ‘message format’, which identifies the wire format of the message.

These attributes allow MQSI to determine which message dictionary to use to obtain the message type definition, and the right message parser to use to disassemble the message, when a message from an MQSeries queue enters a flow at an MQInput node. The attributes are also used to reassemble messages and obtain wire message formats when putting messages on queues at the end of a message flow. For this reason, message type definitions and Control Center definitions are needed not only for messages that enter message brokers but also for those created in brokers as a result of transformations and flowing out of brokers and on to MQSeries application queues.

As indicated, the actual task of disassembling a message and making the contents of its fields available for processing in message processing nodes is the job of message parsers. Parsers are specific to particular message types, and those available include ones that ship with MQSI, those created by the user, and those supplied by third-party vendors. Messages aren’t always processed by parsers before they enter an MQSI flow – for instance, the message may belong to the NEON domain, which means that it is processed by NEONFormatter, being routed to the formatter via a NEONFormatter node.

MESSAGE WAREHOUSES

A message warehouse is a store of messages in a standard relational database. As outlined earlier, there are many reasons for using a message warehouse, including logging messages, building an audit trail, providing a facility for deferred or batch processing, and creating a message store for data mining or analysis. Messages are stored in a message warehouse using the Warehouse node discussed earlier. Messages may be stored in their original state or in a state resulting from some transformation that is performed on them.

One action that a Warehouse node can perform on a message is to parse it. Whether this is necessary depends on how the message warehouse is used. For instance, if the message warehouse is no more than a temporary store for messages, then it’s sufficient that messages are stored as BLOBs, even though this means that the database has little access to fields in the message body. In this instance, should the message require content-based processing, it's handed back to the message broker where processing can take place. On the other hand, if processing is to be carried out while the message is in the message warehouse, then it is necessary for the message to be parsed, so that its fields can be identified and stored in a table.

PUBLISH/SUBSCRIBE

MQSI can bring additional benefits over those offered by a standard publish/subscribe system. For a start, it supports 'content-based filtering', which ensures that subscribers receive only relevant messages (or the relevant parts of messages) and irrelevant ones are discarded, even when their topic matches that of current subscriptions (contrast this with 'standard' publish/subscribe systems, which generally filter messages by topic alone). Another benefit of MQSI is its ability to output material in a format that's convenient to the recipient. MQSI's content filters, which take the form of SQL statements, are stored in the Dynamic Subscription Table. Publish/subscribe functionality is implemented via the Publication node described earlier – when a Publication node receives a message, it evaluates the message for topic and content and deposits the message on one or more MQSeries queues based on the result.

MQSI Version 2 supports two types of subscription: 'dynamic subscription' and 'static subscription'. For those familiar with programming, static subscriptions are 'early bound', meaning that the list of recipients is known in advance and message routing can be set up from the outset. Dynamic subscriptions are 'late bound', meaning that the recipients are not known in advance and routing is set up only when a publication is ready to be sent to subscribers.

MQSI allows users to publish information at one message broker and subscribers to receive it at another. This is a key requirement to create a scalable publish/subscribe system – if messages can be collected by subscribers only from the brokers at which they are published, then information must be published at more than one broker. As the number of brokers increases, so does the burden of publishing at multiple brokers. By allowing information to be published at one broker and propagated automatically to others, brokers can be added to the system without impact on the management of subscriptions. However, a mechanism must be put in place to allow brokers to propagate messages. Essentially, the mechanism is for a broker to subscribe to a publication on another broker on behalf of its subscribers (which may be other brokers). Updates are then sent from the publisher to its broker, then from the publisher's broker to the subscriber's broker (possibly via other intermediate brokers), and finally from the subscriber's broker to the subscriber.

The method of routing subscriptions may organize brokers either in a hierarchical tree structure or a point-to-point system. In installations with a large number of brokers, the use of a tree structure may result in the passing of messages between many intermediate brokers before they arrive at their destination, which is inefficient and unnecessarily delays messages. A point-to-point system also has drawbacks at large installations, where an essentially flat structure may be difficult to implement, manage, and use.

MQSI V2 offers a third alternative to these two organizational structures – 'collectives'. A collective is a group of brokers that form a point-to-point publish/subscribe network. Within a collective, messages never require sending to intermediate brokers, as every broker is connected to every other broker. Collectives may be joined in a network of collectives, in which a broker in one collective is joined to a broker in another collective. Individual collectives in a network of collectives are organized in a tree structure. This, according to IBM, brings the benefits of a hierarchical structure without the drawback of tortuously long routes.

As with just about every MQSI administrative task, the management of collectives is carried out using the Control Center. The process is very straightforward, simply requiring that a broker is assigned to a collective in Control Center's Topology dialogue (this is actually just a tab on the main GUI). As stated, more than one collective may be defined, and a network of collectives is also created using Control Center's Topology tab.

MQSI's publish/subscribe component has a security mechanism controlled by Access Control Lists (ACLs). Topics that are available for both publication and subscription are organized in hierarchical tree structures. Each ‘node’ or ‘leaf’ in this structure may have an ACL attached to it; if no ACL is specified, the node inherits an ACL from its parent node. ACLs determine who may either publish material on a topic or subscribe to it, also identifying those who are allowed to receive persistent messages. The whole mechanism works fairly seamlessly, so that (for instance) if a user attempts to subscribe to a number of topics, using wildcard characters to select the topics, the user is subscribed only to those topics to which he or she has appropriate permission.

There are some subtle (and, perhaps, not-so-subtle) differences in the way publish/subscribe functionality is implemented in MQSeries Integrator V2.0 and MQSeries V5.0. Perhaps the two most important differences (other than the provision of content-based filtering) are in the naming of topics and the formats that the two can handle. In both MQSI V2 and MQSeries V5, publish/subscribe topics may be organized in a hierarchical structure. However, in MQSI V2, topics also use a hierarchical name space, starting with the top-level topic and separating topics in the hierarchy with a forward slash (‘/’). In MQSeries V5, topics use a flat name space comprising arbitrary strings – this effectively hides the hierarchy, so that it’s not clear to users (and administrators) how one topic relates to others in the hierarchy.

The main difference in the formats supported by MQSI V2’s and MQSeries V5’s publish/subscribe systems is that MQSI V2 does not handle command messages in PCF (Programmable Command Format). There are other minor differences between the two systems, with some functions in MQSI V2 being available only for compatibility with MQSeries V5 and (therefore) ‘deprecated’ (that is, not advised for new implementations) or supported only for migration purposes. It’s worth pointing out that it’s IBM’s intention that MQSI V2’s publish/subscribe mechanism should be used in preference to MQSeries V5’s. To this end they’ve produced a tool, **migmqbrk**, to aid the migration process, though it should be stated that the two systems can co-exist and interoperate, and that it is not necessary to choose one in preference to the other (though the overhead of using two systems to implement the same functionality may make it desirable to do so).

Industry Analyst (UK) © Xephon 2000

Customizing CSQ4APPL and CSQ4INP2

CSQ4APPL is an example of how the version of CSQ4DISX and CSQINP2 (the sample for distributed queueing without CICS) supplied in the MQM.SCSQPROC dataset can be customized to meet the requirements of a specific application. This sample contains queue definitions dedicated to an application called *APPL*, which is set up to communicate with a Unix environment. Additional members can be set up and dedicated to new applications – isolation at application level reduces housekeeping and maintenance for all definitions.

AN EXAMPLE OF CUSTOMIZED CSQ4DISX(CSQ4APPL)

```
******************************************************************
*
*                IBM MQSeries for MVS/ESA
* CSQAPPL (copy of CSQINP2)
*
******************************************************************

DEFINE QLOCAL( 'QL.APPL.FROM.UNIX.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( 'APPL records from unix' ) +
         STGCLASS(DATAQ01) +
         MAXMSGL( 16384 )
*
DEFINE QALIAS( 'RT.APPL.DATA.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.FROM.UNIX.QUEUE' )
*
**************************************************
DEFINE QLOCAL( 'QL.APPL.ERROR.HANDLER.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ04) +
         MAXMSGL( 9520 )
*
DEFINE QALIAS( 'RT.ERROR.HANDLER.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.ERROR.HANDLER.QUEUE' )
*
```

```
************************************************
DEFINE QLOCAL( 'QL.APPL.DEBLOCK.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ03) +
         MAXMSGL( 152 )
*
DEFINE QALIAS( 'RT.DEBLOCK.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.DEBLOCK.QUEUE' )
*
************************************************
DEFINE QLOCAL( 'QL.APPL.DUPS.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ03) +
         MAXMSGL( 16384 )
*
DEFINE QALIAS( 'RT.APPL.DUPS.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.DUPS.QUEUE' )
*
************************************************
DEFINE QLOCAL( 'QL.APPL.EAN.BUILD.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ04) +
         MAXMSGL( 152 )
*
DEFINE QALIAS( 'RT.EAN.BUILD.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.EAN.BUILD.QUEUE' )
*
************************************************
DEFINE QLOCAL( 'QL.APPL.PROD.CONV.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ05) +
         MAXMSGL( 150 )
*
DEFINE QALIAS( 'RT.PROD.CONV.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.PROD.CONV.QUEUE' )
```

```
*
************************************************
DEFINE QLOCAL( 'QL.APPL.EAN.BUILD.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ02) +
         MAXMSGL( 152 )
*
DEFINE QALIAS( 'RT.EAN.BUILD.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.EAN.BUILD.ERROR.QUEUE' )
*
************************************************
DEFINE QLOCAL( 'QL.APPL.CONV.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ02) +
         MAXMSGL( 152 )
*
DEFINE QALIAS( 'RT.CONV.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.CONV.ERROR.QUEUE' )
*
************************************************
DEFINE QLOCAL( 'QL.APPL.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ03) +
         MAXMSGL( 16384 )
*
DEFINE QALIAS( 'RT.APPL.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.ERROR.QUEUE' )
*
************************************************
DEFINE QLOCAL( 'QL.APPL.HEARTBEAT.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ03) +
         MAXMSGL( 20 )
*
DEFINE QALIAS( 'RT.HEARTBEAT.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.HEARTBEAT.QUEUE' )
*
**************************************************
DEFINE QLOCAL( 'QL.APPL.UNIX.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ02) +
         MAXMSGL( 100 )
*
DEFINE QALIAS( 'RT.UNIX.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.UNIX.ERROR.QUEUE' )
*
**************************************************
DEFINE QLOCAL( 'QL.APPL.UNIX.PROCESSING.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ05) +
         MAXMSGL( 600 )
*
DEFINE QALIAS( 'RT.UNIX.PROCESSING.ERROR.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.UNIX.PROCESSING.ERROR.QUEUE' )
*
**************************************************
DEFINE QLOCAL( 'QL.APPL.DEPTH.REFRESH.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DEFAULT) +
         MAXMSGL( 100 )
*
DEFINE QALIAS( 'RT.DEPTH.REFRESH.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.DEPTH.REFRESH.QUEUE' )
*
**************************************************
DEFINE QLOCAL( 'QL.APPL.HEART.REFRESH.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DEFAULT) +
         MAXMSGL( 100 )
*
DEFINE QALIAS( 'RT.HEART.REFRESH.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.HEART.REFRESH.QUEUE' )
*
**************************************************
DEFINE QLOCAL( 'QL.APPL.INTO.DEBLOCK.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         DESCR( ' ' ) +
         STGCLASS(DATAQ02) +
         MAXMSGL( 16384 )
*
DEFINE QALIAS( 'RT.INTO.DEBLOCK.QUEUE' ) +
         REPLACE +
         DEFPSIST( YES ) +
         TARGQ( 'QL.APPL.INTO.DEBLOCK.QUEUE' )
*
```

CSQ4CHNL is an example of how CSQ4INP2 and CSQ4DISX can be customized. This example contains channel definitions for remote locations that communicate with the *MQxx* sub-system on the mainframe via a *CHLTYPE(SVRCONN)* connection.

AN EXAMPLE OF CUSTOMIZED CSQ4INP2 (CSQ4CHNL)

```
********************************************************************
*
* THIS MEMBER HAS ONLY CHANNEL DEFINITIONS REQUIRED FOR REMOTE LOCATIONS
*
* COPY OF MEMBER CSQ4DISX
*
******************************************************************
* This definition is for Client Attachment Feature using LU62 and TCP/IP
* SVRCONN connection defined for the OS/2, UNIX MQ system and other
* locations for MQxx subsystem.
* The channel name must be unique hence the suffix of LUC for LU62 connection
******************************************************************
*
DEFINE CHANNEL( 'CH.20000.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for OS/2 via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 786 )
*
DEFINE CHANNEL( 'CH.68500.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for UNIX via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 34380 )
*
DEFINE CHANNEL( 'CH.62411.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 543210 )
*
DEFINE CHANNEL( 'CH.67140.TO.MQxx' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via TCPIP' ) +
         TRPTYPE( TCP ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 7654321 )
*
DEFINE CHANNEL( 'CH.64277.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4536180 )
*
DEFINE CHANNEL( 'CH.68477.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' 100 via LU62' )
+
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 6342312 )
*
DEFINE CHANNEL( 'CH.60778.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 7254420 )
*
DEFINE CHANNEL( 'CH.WASV017.TO.MQxx' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'WASV017' via TCPIP' ) +
         TRPTYPE( TCP ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 1234567 )
*
DEFINE CHANNEL( 'CH.62371.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 1172606 )
*
DEFINE CHANNEL( 'CH.64810.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4013442)
*
DEFINE CHANNEL( 'CH.64898.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4194304 )
*
DEFINE CHANNEL( 'CH.69710.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4104943)
*
DEFINE CHANNEL( 'CH.MQUNIX.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'UNIX box'  via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4419430)
*
DEFINE CHANNEL( 'CH.61811.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 1940443)
*
```

```
DEFINE CHANNEL( 'CH.67077.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 3044194)
*
DEFINE CHANNEL( 'CH.68177.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4304419)
*
DEFINE CHANNEL( 'CH.64210.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4143049)
*
DEFINE CHANNEL( 'CH.41693.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4194304 )
*
DEFINE CHANNEL( 'CH.06611.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 441943)
*
DEFINE CHANNEL( 'CH.10646.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4143904 )
*
DEFINE CHANNEL( 'CH.64273.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL(43 41904 )
*
DEFINE CHANNEL( 'CH.76747.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 3044194)
*
DEFINE CHANNEL( 'CH.67746.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4304419)
*
DEFINE CHANNEL( 'CH.76111.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 1943404 )
*
DEFINE CHANNEL( 'CH.65256.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4194304 )
*
DEFINE CHANNEL( 'CH.76755.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 1944304 )
*
DEFINE CHANNEL( 'CH.87673.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4341904 )
*
DEFINE CHANNEL( 'CH.MQGAMSTON.TO.MQxx' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for Gamston via TCPIP' ) +
         TRPTYPE( TCP ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 3414094 )
*
DEFINE CHANNEL( 'CH.56779.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4419304 )
*
DEFINE CHANNEL( 'CH.16103.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 4341904 )
*
DEFINE CHANNEL( 'CH.MQREADING.TO.MQxx' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for Reading via TCPIP' ) +
         TRPTYPE( TCP ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 61921314 )
*
DEFINE CHANNEL( 'CH.10641.TO.MQxx.LUC' ) +
         REPLACE +
         CHLTYPE( SVRCONN ) +
         DESCR( 'Server connection for 'name of remote location' via LU62' ) +
         TRPTYPE( LU62 ) +
         MCAUSER( ' ' ) +
         MAXMSGL( 3041944 )
*
*****************************************************************
* End of CSQ4CHNL
*****************************************************************
```

Saida Davies
IBM (UK)

MQ news

Information Builders has launched Mobile Computing Server, a hosting and integration platform for mobile and wireless applications. It allows sites to make use of existing systems and provides IT professionals with the tools they need to manage, deploy, and connect Palm/ Windows CE PDAs and WAP- and Internet-enabled mobile devices. The company provides connectors to integrate mobile applications with the rest of the enterprise, including databases, transaction systems, and other software via MQSeries.

Out now, prices start at US$45,000 per 100 users on NT or Unix.

For further information contact:
Information Builders, 1250 Broadway, 30th Floor, New York, NY 10001, USA
Tel: +1 212 736 4433
Fax: +1 212 268 7470
Web: http://www.ibi.com/e-mcsrelease/

Information Builders (UK) Ltd, Wembley Point, Harrow Road, Wembley, Middlesex HA9 6DE, UK
Tel: +44 20 8982 4700
Fax: +44 20 8903 2191

* * *

MQSoftware has released QPasa! Version 2.2.1, the latest version of its management tool for MQSeries, including MQSeries Integrator Versions 1 and 2, MQSeries Workflow, and MQSeries Everyplace. It provides configuration, performance, problem, and operations analysis and management.

The Message Manager allows users to browse messages on queues, including local queues, dead-letter queues, and alias queues. It also allows users to edit messages and message headers. Application developers can use Message Manager to simulate messages from the real application during development. It works with AIX, HP-UX, Linux, Solaris, Windows NT, and Windows 2000. Details on pricing are available on request from the vendor.

For further information contact:
MQSoftware Inc, 7575 Golden Valley Road, Suite 140, Minneapolis, MN 55427, USA
Tel: +1 612 546 9080
Fax: +1 612 546 9082
Web: http://www.mqsoftware.com/products/prodsum/qpasa.html

MQSoftware Europe Ltd, The Surrey Technology Centre, 40 Occam Road, Surrey Research Park, Guildford, Surrey GU2 5YH, UK
Tel: +44 1483 295400
Fax: +44 1483 573704

* * *

IBM has also re-branded WebSphere as an ‘e-business platform’. The WebSphere brand now includes not only the WebSphere application server but also a number of other IBM products, including MQSeries.