53

# MQ update

*November 2003*

## In this issue

# MQ Update

**Published by**
Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38126
From USA: 01144 1635 38126
Fax: 01635 38345
E-mail: info@xephon.com

**North American office**
Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: (303) 410 9344
Fax: (303) 438 0290

## Disclaimer

**Editor**
Madeleine Hudson
E-mail: MadeleineH@xephon.com

## 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 ($33.75) each including postage.

## Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £100 ($160) per 1000 words and £50 ($80) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £20 ($32) per 100 lines. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

## *MQ Update* on-line

Code from *MQ Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at www.xephon.com/mq; you will need to supply a word from the printed issue.

## Fine-tuning WMQI performance

Organizations will always strive to maximize the performance of an application and this is particularly important where WMQI (WebSphere MQ Integrator) is concerned. Many WMQI implementations use the 'hub and spoke' type of architecture, where the WMQI server forms the hub and all other applications within the organization form the spokes. As a hub, processing many different types of message from different applications, WMQI becomes a single point for the provision of transformation and routing capabilities. Ensuring that WMQI provides maximum performance is, therefore, a key deliverable.

The performance of a WMQI implementation is dependent not only on the hardware capabilities but also on the solution architecture, which is key to getting the most out of the available resources. To illustrate these points I will discuss a specific customer implementation.

For this particular project at a customer's site we had one source application, which initiated XML messages that needed to be routed to seven other applications in flat file format specific to those applications. There were three different types of message from the source application. For this requirement the architecture comprised one main flow that could route the messages to other flows, depending on the type of message (see Figure 1).

The main flow was similar to that of a traffic director, routing the messages to relevant flows, which in turn processed the messages and routed the data to the relevant target application in the required flat file format. All the queues used by the message flows were kept persistent since the messages contained important data that we could not afford to lose.

This architecture gave us flexibility because each target application had its own flow and would process the messages independent of other flows. Each flow had a different processing logic since each target application required different data. For example, one of the target applications was SAP, which required 'Bill of Materials'

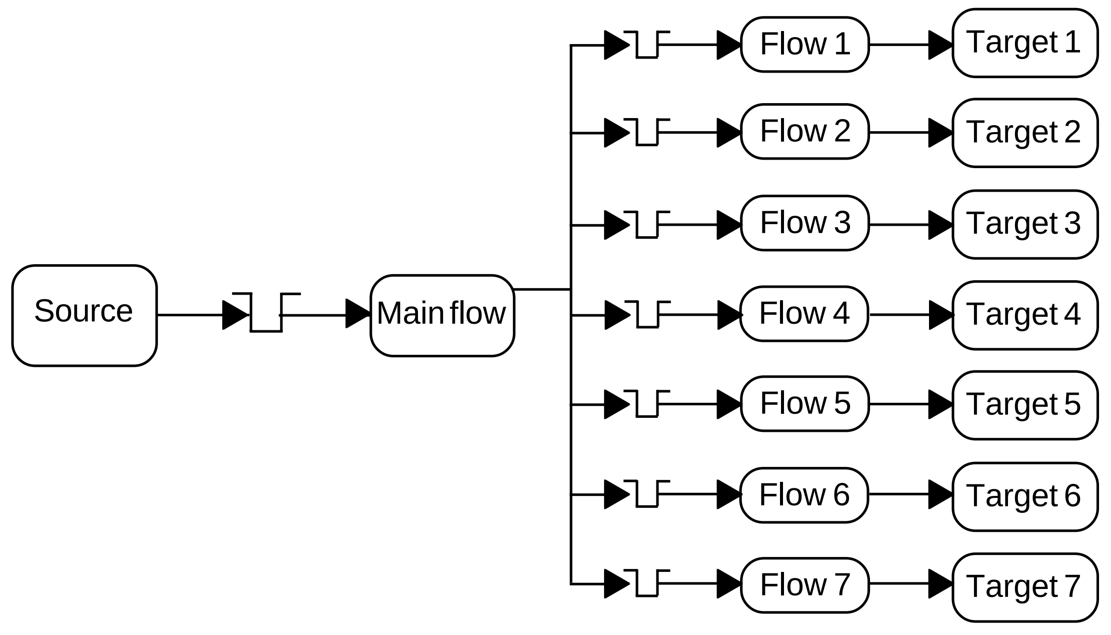

*Figure 1: Message flow architecture*

data. The message flow for processing SAP data was fairly complex. Another target application was a training management system. The message flow for processing training-related data was fairly straightforward and, as a result, training-related flows were able to process messages at a faster rate than the SAP flows. This architecture maintained the independence of each flow, allowing messages to be processed at different rates.

The architecture worked well and for messages from the source application of up to 800KB it gave a message-processing rate of 7.63 messages/minute during stress testing. This was the processing rate for the slowest of the seven message flows.

All the testing was done with two Sun Solaris 8 boxes with a 2GB RAM Broker with a DB2 database. But as the messages from the source application increased in size the message-processing rate decreased. For messages of between 800 KB and 2MB the same setup gave a processing rate of 5.56 messages/minute for the slowest message flow. This was expected because larger messages require more time for parsing and looping the processing logic but it created a problem with disk space.

The disk space for persistent logs filled up quite quickly. The main flow would get one copy of a message from the source application and send four copies of the same message to other flows on their input queues. Since all the queues were persistent all these messages are written to logs, thereby eating up the log space at a faster rate. At one point, message processing stopped completely because there was no disk space left to which to write the logs. Increasing the space allocated for logs was one solution but changing the architecture presented a possible alternative.

The main problem with the original architecture was that it used a greater number of message flows and, as a result, there were many persistent queues. Memory requirements, as well as those of disk space, are increased with this type of architecture.

We devised another architecture with only one message flow to process all the messages. We designated all seven target application flows as subflows in WMQI, which were then included in the main flow so we had one flow that did everything. This meant that instead of eight flows with their eight persistent input queues we had just one flow and one input queue.

This architecture produced significant improvements in performance. It gave a message-processing rate of 11.75 messages/minute for messages of up to 800KB and a rate of 8.37 messages/minute for the larger 2MB messages. In other words there was a performance improvement of about 50% over the previous solution.

The second architecture is better in terms of performance but it has a disadvantage in that all the target applications are dependent on a single flow, which becomes a single point of failure. If there is a problem with that flow all the target applications are deprived of the information they are supposed to receive. Each target application needs a different set of data; the processing logic is complex for some and for others it is straightforward but with only one flow all the target applications will receive the messages at the same rate, which is that of the slowest flow.

Each solution architecture has its own advantages and disadvantages. The ultimate choice of the solution depends on what the business side of the application really needs. In this case the business people wanted to have independence for all the applications so we decided to go with the first architecture even though it was costly in terms of the system resources required.

To summarize, the following points will help in achieving maximum performance from your WMQI solutions.

- Use as few persistent queues as you can get away with.
- Try to make message flows generic so that they can be used as subflows.
- One complex message flow is more efficient than having a number of message flows utilizing the same logic.
- Wherever possible, use smaller messages in WMQI. Memory requirements increase significantly as the message size increases beyond 4 MB.
- Try to use the Message-ID/Correlation-ID fields in message headers to identify messages. Populating these fields well help in identifying messages from various applications on queues, which will reduce the number of queues required.

*Kiran Kanetkar*
*G4 Technologies (USA)*

## Error event processing in a WMQI Broker domain

WMQI Broker is IBM’s product for processing WMQ messages. The processing is done by message flows that run in execution groups. A set of execution groups belongs to a broker. Brokers can work together in a broker network known as a broker domain. Each broker domain is managed by one WMQI Broker Configuration Manager.

Error detection in broker message flows is carried out by the broker. Errors during message processing are handled either via return codes (eg for database operations) or exceptions. If a return code during message flow processing is not expected the message flow can throw such an exception. If the exception is not handled by the message flow itself it can be reported to the system log of the operation system on which the broker is running.

Depending on the operating system this system log can be a console, the syslog, or the event viewer. The captured problem information can then be viewed at the system log by the owner of the broker or of the message flow. There is no way in WMQI Broker to access the problem description available in the various system logs centrally, eg at the Configuration Manager.

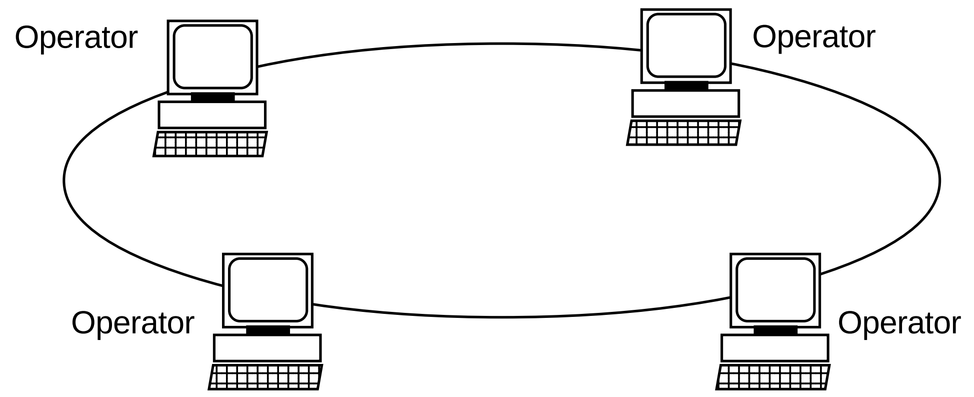

*Figure 1: Monitoring problems in a broker network*

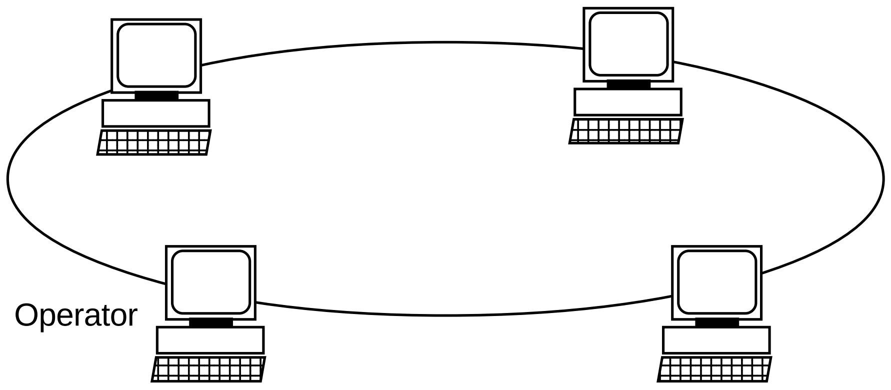

*Figure 2: Monitoring problems in a WBI for FN broker network*

When running one or two brokers this problem reporting is sufficient. In a domain containing many brokers an operator would need to monitor each broker's system log on different computers, as shown in Figure 1.

If these brokers run on different platforms the operator must be familiar with the handling of different system logs. To avoid this, error detection should be possible from any one machine within the network, as shown in Figure 2. This article describes a mechanism to achieve this. The mechanism is implemented in WebSphere Business Integration for Financial Networks (WBI for FN), a set of products that extends WMQI Broker. The base product provides common functionality, such as event handling and dynamic configuration for message flows or auditing. Extensions to the base product can be delivered by IBM, independent software vendors, or end users.

## EVENT HANDLING CONCEPT

There are two stages to problem detection and error handling:

- Detecting and reporting.
- Viewing or monitoring.

For both parts, the WBI for FN event concept uses standard WMQI Broker functionality, exceptions, publish /subscribe functionality, and database access. All errors detected can be reported as WBI for FN events. These events are stored in a database and are published within the broker network. WBI for FN also provides viewers for both methods, publications, and database entries.

### WNI for FN events

A WBI for FN event is similar to a WMQI Broker exception. An event comprises:

- Identification of the problem.
- Referencing the event to the appropriate message text.

- Location of the problem.
- Studying in more detail the parameters that describe the problem.

Problem identification and referencing will be familiar from WMQI Broker exceptions. The main difference within WBI for FN is that the catalogues used to store information on event text formatting are the same on every platform. Instead of platform-dependent implementations, resource bundles of the International Components for Unicode (ICU) are used to hold the message text.

When looking at an exception reported in the system log the exception provides only information about the location in the message flow where the problem occurred, eg the node name. If the message flow is used infrequently this is usually sufficient to identify the cause of the problem.

When this message flow is used to process many messages in a short time it is difficult to determine which of these messages caused the problem. To support identification of the erroneous message, a WBI for FN event can also store the identification of the message that caused the problem. For WMQ messages the identification of a message contained in the WBI for FN event is its Message-ID and Correlation-ID.

A WMQI Broker exception refers to the node within the message flow that caused the exception. This is helpful when the developer knows the nodes in the message flow and can control which ones are used. Products that use WMQI Broker usually structure common functionality in subflows that can be embedded in more than one message flow. Such subflows are also provided by WBI for FN. For message flows that use these subflows the identification of the nodes would not provide any meaning to the developer as he doesn’t know which nodes perform what operations or how to fix a defect in them.

To help identify whether or not the problem is caused in such a subflow and to identify the functionality, a WBI for FN event can contain a subcomponent code. This subcomponent code is a short identification of the function that has the problem, eg the auditing or warehousing function.

When looking at the system log for problems in a message flow it is obvious on which computer the problem occurred. Server identification is added to WBI for FN events so the information available is consistent when looking at a set of events that occurred somewhere in a network.

The parameters of an event that describe the problem in more detail are the same as the parameters for an exception in WMQI Broker. They can contain variable information about the event, such as which resources were involved in the problem.

## Issuing events

If a problem is detected during the processing of a message flow, a WBI for FN event can be issued. There are two ways to achieve this:

- A WBI for FN-provided node.
- A WBI for FN-provided message.

The WBI for FN-provided node is the Event Store node. Within a message flow all necessary information can be prepared to call this Event Store node. The node then stores all information in a WBI for FN database table and publishes the information with a WBI for FN-defined topic.

The WBI for FN Event Store node performs its work under transactional control. That's why this node can be used only when the message flow ends successfully with the message flow processing being committed.

When a problem occurs a rollback is often required. For these cases WBI for FN offers a message flow for storing the event information that can be invoked via a WMQ queue. So if there is a problem or an exception the developer of a message flow, when necessary, can format a new request message with all required information and use a standard WMQI Broker MQOutput node to send this event information to a message flow outside the transactional control of the message flow.

Because this kind of processing is often required if an exception occurs in a message flow, WBI for FN provides a node that can handle WMQI Broker exceptions. This node reformats the exceptions into WBI for FN events. This reformatted exception than can be directed to the event-handling message flow. After these actions the exception in the message flow needs to be rethrown to force a rollback.

The event processing message flow consists mainly of the Event Store node. It writes the event into the event database and publishes the event information. The event processing message flow only stores the event information so there is no other customer logic that could force a rollback. Because the event-handling message flow processes WMQ messages in WBI for FN format, any other WMQ program that has something to report can use the same event-reporting functionality.

Where publications are concerned most of the information on the WBI for FN event is included in the publication message. To facilitate subscriptions to specific events part of the information, eg the component code, is also used to build a subtree in the topic tree.

## Monitoring

There are two different monitoring models:

- The push model.
- The pull model.

For the pull model it must be possible to view events after they occur. This is similar to what is already available with the WMQI Broker event log. With WBI for FN event functionality, this is achieved using an event administration message flow. A WMQ message can be formatted and sent to this message flow. The message flow gets the message and retrieves the event information from the database. The request message to the message flow can contain parameters that specify which events the user wants to see. They limit the listed events to specific criteria, eg a time frame or the system where the event occurred.

For inserting and formatting the request messages to the event administration message flow, WBI for FN provides a command line interface program (CLI). It receives responses from the message flow and formats the events using the catalogues. The CLI program can be used to access any administration message flow that is part of WBI for FN.

The event administration message flow can run on any broker within the WMQI Broker domain. This allows events to be listed centrally from any workstation that is connected to the MQ network. There is no need to connect to each broker within the domain and to analyse each of the system logs. Another advantage is that several people can run this program and list different events without interfering with each other.

In contrast to the pull method, where a user actively requests the event information, the WBI for FN event concept also supports a push model. This is done with the WMQI Broker publish/subscribe event facility. WBI for FN provides two different monitoring options:

- Monitoring all events; as events occur each is formatted via the appropriate catalogue and then displayed to the user.
- Monitoring specific events as determined by the user-defined filter criteria. For example, one operator might look for IT-specific problems while another looks for component-specific problems.

For operators who want to view all events centrally on their usual system log, WBI for FN also provides a console notifying message flow. This message flow is subscribed for all WBI for FN events. Every time an event is issued this message flow gets the corresponding messages as publication information. The message flow formats the event information and writes it to the system log.

## Using the error event infrastructure

The WBI for FN event functionality can be used to report problems detected during message flow processing. This functionality cannot be used to report internal broker problems, eg at startup and shutdown.

Through the use of open interfaces when sending WMQ messages it is possible to monitor problems centrally in a network of brokers. It is very easy to connect to external monitoring programs, specifically, when publishing the event information.

In addition to error detection this infrastructure can also be used to monitor business events. An example is implemented in the WBI for FN Extension for SWIFTNet, which provides access to the SWIFT financial network. A connection to the network is represented by a SWIFT logical terminal. For the business it is mandatory to know whether the connection is active, so, WBI for FN events are issued for each state change of a logical terminal that occurs. An operator can subscribe to such events and is notified when anything happens to the network connection.

The publish/subscribe mechanism can be used not only by the monitoring programs provided by WBI for FN but also by customer programs, thereby enabling them to react automatically to specific events.

## SUMMARY

WMQI Broker provides a means to detect problems that are local to brokers. WBI for FN shows that it is possible to extend the problem detection in a WMQI Broker domain from a distributed operation model to a powerful error detection system for an entire broker domain, where problems can be monitored centrally. To implement this, WBI for FN events handling is used. WMQI Broker provides all the necessary base functionality, eg publish/subscribe, database access, exceptions, and message flows.

The event functionality can be used to detect problems during message flow processing in a broker network. It’s also possible to use this function to connect external monitors or to monitor business events.

*Michael Groetzner*
*IBM (Germany)*

# Using WMQ in J2EE: part two – JMS and the WAS

## INTRODUCTION

This article describes the use of WMQ in the Java 2 Enterprise Environment (J2EE) and is the second of two parts. In the previous article, published last month, the J2EE standard interface for messaging was described, as was its WMQ implementation. Here we discuss the J2EE in more detail along with the integration of WMQ with the messaging options available in IBM's J2EE platform, the WebSphere Application Server (the WAS).

## THE J2EE

The J2EE is an umbrella specification that references many other Java applications. It defines the standard runtime environment for enterprise applications written in Java; the current version that has industry acceptance is J2EE 1.3.

Some of the J2EE specifications define the programming model for writing applications for the J2EE platform, such as Java Servlets, Java Server Pages (JSPs), and Enterprise Java Beans (EJBs). Other component specifications of the J2EE platform define the runtime services that are available for J2EE applications, such as JNDI, JDBC, and (of course) JMS.

There is a paradigm shift between writing applications for the J2EE platform and writing applications in, for example, the C programming language, although both these applications will require services and characteristics such as transactions, resource management, concurrency, and lifecycle.

In the J2EE these services and characteristics are held in components in such a way that the applications require no specific coding to use or include them. This is in marked difference to applications written in C, where the programmer must have knowledge of all these services and must code or compile to use them. Together with the set of APIs supplied in the J2EE, this is what is meant by the J2EE Programming Model. It allows applications to be built more easily, quickly, and cheaply because the only requirement on the application programmer is to have an understanding of the behaviour that applications are required to have.

In addition to the Programming Model, the J2EE also defines a Deployment Model, which specifies how applications are packaged and deployed into a J2EE Application Server.

## JMS APPLICATION SERVER FACILITIES

The JMS Application Server Facilities, as the name suggests, are various advanced functions that are implemented by expert JMS Providers, such as Application Servers. They are split into two main parts, concurrent message delivery and distributed transactions.

### Distributed transactions

Distributed transactions, as described in the X/Open document *Distributed Transaction Processing: The XA Specification*, involve the coordination of updates of multiple resources in an atomic unit of work, providing support for robustness using a two-phase commitment protocol. The specification describes two roles:

- The Resource Manager has control of a single resource that will be updated as part of the transaction. The WMQ queue manager is an XA-compliant resource manager.
- The Transaction Coordinator communicates with, and controls the operation of, all resource managers associated with the transaction context. It uses the two-phase commitment protocol to ensure atomic updating of all resources, in other words, ensuring that all operations happen, or none. The WAS EJB Container is an XA-compliant transaction coordinator.

The J2EE specification mandates the JTA specification to support distributed transactions in the J2EE. A JMS Provider must expose its JTA support through the following special JMS objects:

- *XAConnectionFactory*: an XA-aware ConnectionFactory.
- *XAConnection*: an XA-aware Connection.
- *XASession*: an XA-aware Session. The XASession holds an XAResource object, which provides the facilities for allowing the Session to participate in distributed transactions controlled by an XA transaction coordinator.

The transactional context does not flow across Sessions – messages can be sent and received from the same Session under a single transaction but the same does not apply for different Sessions.

The J2EE specifications support the demarcation of transactions by making use of API calls (application or bean-managed transactions) but it is preferable to use the Application Server to control the transaction (container-managed transactions). This is accomplished by specifying the transactional behaviour required when the application is deployed into the Application Server.

## The JMS ConnectionConsumer

In part one of this article asynchronous (or passive) message delivery was described. This may at first glance appear to be only a convenient manner of negating the need for applications receiving messages to block indefinitely whilst waiting for a message. However, this section describes how this method can be used to drive very powerful, scalable, concurrent message delivery. The JMS defines a flexible method of concurrent message delivery, with the following participant roles:

- The *JMS Provider* has responsibility for delivering the messages.
- The *Application Server* has responsibility for creating the consumer and managing the threads used by the concurrent MessageListeners.
- The *Application* has responsibility for defining an interest in a JMS Destination (with an optional message selector) and for providing a single-threaded MessageListener. The Application

Server will construct multiple instances of the MessageListener to process messages concurrently.

The following JMS objects provide the facility for concurrent message delivery:

- The *JMS Session* provides methods to both determine and retrieve a MessageListener as well as a 'run' method, which causes messages (loaded on the Session by the ConnectionConsumer – see below) to be serially processed in its MessageListener.
- The *JMS ServerSession* is implemented by the Application Server to wrap the Session object for concurrent message delivery.
- The *JMS ServerSessionPool* is also implemented by the Application Server and its function is to manage ServerSessions. The implementation of the ServerSessionPool is not defined – it could be very simple, with a defined maximum number of ServerSessions, or expand and contract dynamically as required. If a ServerSession is requested from the pool and one is not available, the calling method will block until the ServerSessionPool can satisfy the request.
- The *JMS ConnectionConsumer* pulls all the above together. It holds a reference to a ServerSessionPool and, as messages arrive at the Destination that it is consuming messages from (with an optional selector), it retrieves ServerSessions from its ServerSessionPool. It then loads them with messages (to be processed by their Sessions' MessageListeners) and starts them.

  When traffic is light the Sessions are loaded with one message at a time but when traffic becomes heavier the Sessions are loaded with multiple messages. The ConnectionConsumer has a parameter limiting the maximum number of messages that can be loaded into a Session at any one time.

The Application Server owns the ConnectionConsumers and

ServerSessionPools. It must create these and any other JMS objects that it needs to satisfy concurrent message delivery requirements. The application supplies the MessageListener code that will be registered with the Sessions held in the ServerSessionPool (in the J2EE these are in the form of Message-Driven Beans – see below for a description of J2EE Messaging).

## The WMQ JMS ConnectionConsumer

If a JMS Destination is served by ConnectionConsumers with a non-maximal set of selectors, unmatched messages will not be retrieved from the Destination. The WMQ JMS ConnectionFactory object can be configured to discard these by setting the Message Retention parameter to MQJMS_MRET_NO. This highlights a conflict between WMQ robustness and JMS concurrent message delivery because this situation can cause messages to be discarded. The following scenarios will cause messages to be left on the Destination:

- The arrival of a badly formatted message that WMQ JMS cannot process.
- The arrival of a ‘poison’ message. This type of message is well-formed but cannot be processed by the application’s MessageListener and is, therefore, continually processed and then rolled back. See below for more details on poison messages.
- There being no ConnectionConsumer interested in the message.

If any of these situations occur the messages will be removed according to the report options in the MQMD (the last one also requires MQJMS_MRET_NO to qualify for this discussion). These can be either:

- MQRO_DEAD_LETTER_Q, in which case the message will be requeued to the dead letter queue.
- MQRO_DISCARD_MSG, in which case the message will be discarded.

Report messages are also generated if the report options are set to include these. If the message cannot be put to the dead letter queue, further processing is based upon the message's persistence:

- If the message is not persistent it is discarded.
- If the message is persistent it is rolled back to the underlying queue and all further processing is stopped. It is therefore vital to monitor the dead letter queue to prevent messages stacking up.

Poison messages are messages that are continually read from the underlying queue and backed out. This can be caused by application failure, transactional backing out, or Session recovery for client-acknowledged Sessions.

WMQ keeps a record of the number of times a message is backed out in the MQMD, which is known as the BackoutCount. The WMQ JMS ConnectionConsumer verifies the BackoutCount against the BackoutThreshold set on the underlying WMQ queue and, if the threshold is reached, the message will be requeued to the BackoutQueue specified on the underlying WMQ queue. If the BackoutQueue is not accessible the message will be requeued to the dead letter queue.

## MESSAGING IN THE J2EE

This section will describe how to access the messaging function in the J2EE programming environment. There are two requirements in the J2EE 1.3 specification that are directly related to messaging:

- An Application Server must provide an implementation of the JMS 1.0.2b specification. The JMS function must be available to application code that is executing in any of the three Containers: the Web Container (for JSPs and Servlets), the EJB Container, and the Client Container (for J2SE). The implementation must allow applications running in the three Containers to communicate using JMS messaging.
- An Application Server must support the Message-Driven

Bean (MDB) programming model as defined in the EJB 2.0 specification. The EJB 1.*x* specification defined Entity Beans (for managing persistent objects) and Session Beans (for representing services that can be invoked).

The EJB 2.0 specification introduced the MDB, which is a stateless component that is invoked by the Application Server's Listener component when a message arrives at a designated JMS Destination. The execution of the MDB can be viewed as being triggered by the arrival of the message.

The programming model for MDBs is a subset of the EJB programming model and is very simple. A normal EJB has a home and a remote interface, which are used by the Application Server to manage the EJB and to advertise the business methods provided by the EJB respectively.

An MDB has neither a home nor a remote interface as its management is predefined and there is only one business method that can be invoked: the onMessage method (the fact that this looks exactly like a JMS MessageListener is no coincidence). When a message arrives at the JMS Destination stating that the MDB is serving, the onMessage method is called with the message as a parameter. The code inside the MDB (which must be written by a developer) contains the JMS logic, which will extract the payload of the message and then trigger its processing.

The recommended system design where MDBs are involved is to have them process the message to extract the data and then call a normal EJB to process the data within its business logic. In a sense the MDB is used as an adapter to the EJB. This decoupling of the messaging logic from the business logic has several advantages:

- The business logic can be driven asynchronously by the arrival of messages.
- The business logic can be driven from other sources, such as direct IIOP client programs or Web applications. This provides a greater degree of ubiquity for the business logic code.
- The skills required for developing EJBs (containing business

logic) and MDBs (containing messaging logic) are cleanly separated, allowing better definition between application programmer roles.

It is recommended that, because MDBs are meant to be relatively short-lived, the EJB methods called by them do not tend towards long-running transactions. This means that any functions called from within the onMessage method of the MDB should return fairly quickly so that the thread being used by the MDB can be recycled in a timely manner.

Not only is the MDB programming model a simplified subset of the EJB programming model, but also the MDB transactional attributes are a subset of those of an EJB. An MDB can have one of two transactional attributes:

- TX_NOT_SUPPORTED, which implies that the message processing and any other work done by the MDB does not require a transaction.
- TX_REQUIRED, which implies that the receipt of the message by the Application Server's Listener component, the processing of the message by the MDB, and the invocation of any other EJB function by the MDBs are all contained within the same transactional context.

It is also possible for the onMessage method of the MDB to contain calls that demarcate the transaction (a bean-managed transaction), but in this case there is no way to associate the message receipt by the Listener with the transaction – the transaction starts only when the appropriate call is made within the MDB.

## MESSAGING IN THE WAS 4.*x*

Although the WAS 4.*x* implements most of the J2EE specifications (EJBs, Servlets, JSPs, JNDI, etc) it does not claim J2EE compliance. Despite this, it offers several functions that facilitate the implementation of solutions involving messaging:

- Global (XA) transactions that include message sending or synchronous message receipt under two-phase commit control with other coordinated resources.

- Pooling of JMS Session objects (corresponding to the underlying base WMQ Java Connection Pool).

JMS support in the WAS 4.*x* (and 3.5.3) requires the separate installation of messaging products. For point-to-point messaging either of the following can be used:

- MQSeries 5.2 plus the MA88 Product Extension.
- WMQ 5.3.

For publish/subscribe messaging, any of the following can be used (together with one of the above WMQ JMS implementations):

- The MA0C Product Extension (this goes out of service at the end of 2004).
- WMQ Event Broker.
- WMQ Integrator Broker.

The WAS 4.*x* Enterprise Edition adds support for Message Beans. This is provided in the additional JMS Listener component. Message Beans behave exactly like MDBs because the intention behind the implementation was to provide support for concurrent message delivery ahead of WAS 5.0 and J2EE 1.3. This was due to a loophole in J2EE 1.2, where there was no transactional responsibility defined between the Listener receiving the message and the invocation of the MDB.

## MESSAGING IN THE WAS 5.0

The WebSphere Application Server 5.0 is fully compliant with J2EE 1.3 and delivers much improvement in the area of messaging when compared with previous releases. These include the points mentioned above, and the new ones are summarized as follows (to be discussed later in this article):

- Complete integration of a JMS Provider (Embedded Messaging), which is optionally installed with WebSphere and managed as part of the WebSphere runtime through the standard administration consoles. WebSphere Embedded Messaging is built on WMQ technology.

- The choice of the integrated WebSphere JMS Provider, using WMQ as an external JMS Provider, or using a third party JMS Provider.
- Full support for MDBs through the integral MessageListener component.
- Greater flexibility in the configuration of the various pool of threads and objects for messaging – both in terms of basic JMS and for MDBs.

## WebSphere Embedded Messaging

The Embedded Messaging JMS Provider is (optionally) installed as part of the WebSphere installation and is managed from the WebSphere administration consoles. It is built upon WMQ technology and the concept behind it is to provide messaging function to J2EE programmers who have no knowledge of WMQ. Because of this requirement there are limitations on its usage:

- Use of the WebSphere JMS Provider is possible only from the WAS containers. It can be used to communicate by any applications running within them but not by any applications running outside.
- The WebSphere Embedded Messaging queue manager and broker cannot communicate with external WMQ queue managers or WMQ Event Broker instances.
- The WebSphere JMS Provider can be administered only from the WebSphere administration consoles, not from any external WMQ tools.

A WMQ practitioner, when observing the WebSphere Embedded Messaging component, will recognize running processes as those of a standard WMQ queue manager. However, it should be noted that any direct configuration of the underlying queue manager (not using the WebSphere administration consoles) is not supported and will cause problems for the Application Server. This is because the Application Server verifies the configuration of the queue manager against its own internal configuration.

There are several parts to the WebSphere Embedded Messaging component:

- The JMS Server manages the Embedded Messaging constituent parts. When run on a single server instance of WAS it is run as part of the Application Server itself. In a WebSphere Network Deployment environment it runs as a separate server.
- The JMS Wrappers provide an interface between the underlying WMQ JMS objects and JCA-managed connections in WebSphere.
- The Embedded Messaging component is installed with specialized scripts, which check for existing versions of WMQ on the machine and install or upgrade the required components.
- Point-to-point messaging is provided by WMQ technology at V5.3.0.1. The JMS Server creates and manages a queue manager for the Embedded Messaging function. There can only be one instance of WMQ software installed on any one machine so Embedded Messaging will replace any previous version of MQSeries.
- Publish/subscribe messaging is provided by WMQ Event Broker technology. The JMS Server creates and manages a broker instance for Embedded Messaging . In contrast to the above, multiple copies of the Event Broker technology can be installed on a single machine, so the Application Server installs its own version of the broker code, which is entirely separate from any existing Event Broker or Integrator Broker installations that may exist.

  The broker used for Embedded Messaging is substantially different from Event Broker in that it runs inside the JMS Server and does not require a relational database.

The Embedded Messaging Server is isolated from other Embedded Messaging Servers or external WMQ queue managers. This does not mean that it is impossible to have more than one Embedded Messaging Server in a cell but it does mean that they cannot communicate. Any applications that are required to communicate via JMS should agree on one Embedded Messaging Server to use, which should be accessed in one of the following two ways:

- If the application and the Embedded Messaging Server are on the same machine the application will use a direct bindings connection.
- If the application is not on the same machine as the Embedded Messaging Server the application will use a TCP/IP client connection. It should be noted that the Embedded Messaging Client component should always be installed with the Application Server because the Java code within offers an XA-aware protocol.

One reason for having multiple Embedded Messaging Servers in a cell is for availability purposes.

## The WebSphere JMS Provider *versus* WMQ

Although the JMS Provider delivered in the WebSphere Embedded Messaging component is robust and fully JMS-compliant it is expected that there will often be a preference for using WMQ as a separate external JMS Provider for the following two reasons:

- The Embedded Messaging component can be accessed only from within the WebSphere containers. In order to interface with existing applications that will not run in a J2EE framework (such as COBOL applications running on a host system), an external WMQ queue manager, which can be accessed from any WMQ-aware application, should be used.
- WMQ offers advanced facilities, such as queue manager clustering, configurability in high availability technologies, publish/subscribe broker collectives, and z/OS shared queues, which cannot be accessed from Embedded Messaging. External WMQ queue managers should be used when accessing any of these features.

It should be stressed that Embedded Messaging and WMQ are not meant to compete with JMS Providers – Embedded Messaging is designed to provide access to J2EE messaging for programmers who are not familiar with WMQ. Conversely, if the application designers are competent in the use of WMQ, it can be frustrating to know that WMQ is driving the JMS messaging with no means to configure it directly.

What is gained by using Embedded Messaging on the one hand – ease of administration and seamless integration – is lost on the other if workload balancing or availability are required as part of the JMS architecture.

## MDB support in the WAS 5.0

The WAS 5.0 provides full support for MDBs as required by the EJB 2.0 specification (and hence the J2EE 1.3 specification). This is provided in both the Advanced and Enterprise editions of the product (rather than only the Enterprise edition as in the WAS 4.*x*) and is fully integrated into the core runtime. Configuration and management is done with the standard WebSphere consoles and MDBs in the WAS 5.0 can be used with any compliant JMS Provider.

Several new WebSphere administered objects are introduced as part of the MDB support, as described below:

- The Listener Service, which manages the whole MDB structure within an instance of the Application Server.
- The ListenerPort, which is associated with a ConnectionFactory and Destination and is used to drive the MDBs. In JMS terms the ConnectionConsumer is located here, as is the ServerSessionPool.
- The MDB, which is associated with a ListenerPort and hence a ConnectionFactory and Destination.

There is a fine granularity on the configuration and management of these objects. They can be stopped and started at all levels (a single MDB up to the whole Listener Service) and they can be configured to the same degree of detail. The configuration is centred on the number of threads available at the various levels, as required by the MDB thread pools and the ServerSession threads required by the ServerSessionPools owned by the ConnectionConsumers.

## WebSphere Studio Application Developer 5.0

WebSphere Studio Application Developer 5.0 includes a single-server copy of the WAS 5.0. It can be controlled and administered from within WebSphere Studio, in other words the standard WebSphere administration consoles do not need to be used. This integrated tooling facilitates faster application development. There is also fully integrated support for the development and deployment of MDBs.

There is also special support included for unit testing JMS applications. This JMS 'simulator' implements the full JMS API but does not provide the robustness behind the scenes that is expected of WMQ. It is provided solely for the unit testing of JMS applications (which should, of course, also be tested in the fully robust JMS environment) and should not be confused with WebSphere Embedded Messaging. Its advantage is that it further reduces the time to start the Application Server within the development environment because the JMS Simulator turns around much more quickly than the full JMS Provider.

*Ewan Withers*
*IBM Hursley (UK)*

# MQMON

MQMON is a REXX utility for monitoring and operating WMQ channels on OS/390. It is intended to assist helpdesk operators and MQ administrators.

On execution the program searches the MQ subsystems defined on OS/390 (the vectors are read from storage), which are then displayed. This allows the user to choose the required MQ subsystem. Subsequently the main panel is displayed, which appears as follows:

```
MQSeries CHANNELS                                                   Row 1 to 4 of 4
COMMAND ===>                                                        SCROLL ===> CSR
Opc Channel              Xmitq                   Conname    Status   Type  Seqno
    MQP2.TO.MQP3_AIX.P   MQP3_AIX.XMIT.P.QUEUE   MQP3_AIX   RUNNING  SDR   820723
    MQP5.TO.MQP6         QP6.XMIT.P.QUEUE        MQP6       RUNNING  SDR   60339
    MQP3_AIX.TO.MQP2.P                           MQP3_AIX   RUNNING  RCVR  799989
    MQP6.TO.MQP5                                 MQP6       RUNNING  RCVR  63750
```

Pushing 'enter' will refresh the values on the panel; normally you will see an increment of the 'Seqno' for a channel in 'running' status.

Operation is by means of the following options:

- 'S' to start a channel.
- 'T' to stop a channel.
- 'R' to reset a channel.
- 'D' to display the channel definition with the date and hour of the last Start channel. This information is shown in an ISPF window.

With the 'T' (stop) and 'R' (reset) options a window is displayed before execution so the user can confirm the action.

The program is supplied with help panels (use the F1 key). It also has explanatory and error messages in two versions: brief and detailed. The more detailed messages are accessed through the F1 key when the short messge is displayed. The message library used is ISPMLIB.

This tool minimizes much of the intervention required from MQ administrators when problems are presented. It is also easy to use and consumes few system resources. At present the program is executed with OS/390 V2.9 and MQSeries V1.2 and uses the RXMQVC interface to connect TSO and MQSeries. I installed MQMON as a TSO option on the main panel.

## MQMON

```
/* REXX   MQMON   carlos-osorio@excite.com  */
TRACE OFF
NUMERIC   DIGITS 12;
CVT     = C2X(STORAGE(1Ø,4))                 /* ADDR DE CVT          */
CVTJESCT= D2X((X2D(CVT))+296)                /* POINTER A CVTJESCT   */
JESCT   = C2X(STORAGE(CVTJESCT,4))           /* ADDR DE JESCT        */
JESSSCT = D2X((X2D(JESCT))+24)               /* POINTER A JESSSCT    */
SSCVT   = C2X(STORAGE(JESSSCT,4))            /* ADDR DE SSCVT        */
J = Ø
DO I = 1  WHILE (SSCVT <> ØØØØØØØØ)
          SSCTSNAM=D2X((X2D(SSCVT))+8)       /* POINTER A SSCTSNAM   */
          ERLY  = D2X((X2D(SSCVT))+2Ø)       /* POINTER A ERLY       */
          ERLYAD= C2X(STORAGE(ERLY,4))       /* ADDR DE ERLY         */
          ERLYSCOM= D2X((X2D(ERLYAD))+56)    /* POINTER A ERLYSCOM   */
          IF SUBSTR(STORAGE(ERLYSCOM,64),29,8) = 'CSQ3EPX ' THEN
             DO
             J = J + 1
             SSIDMQ.J = STORAGE(SSCTSNAM,4)  /* ADDR DE SSCTSNAM     */
             END
          SSCTSCTA = D2X((X2D(SSCVT))+4)     /* POINTER AL SGTE SSCVT*/
          SSCVT    = C2X(STORAGE(SSCTSCTA,4))
END
"ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PANELI')"
"ISPEXEC TBCREATE TSSIDMQ",
"NAMES(O SMQ)",
"NOWRITE REPLACE"
O= '';
DO J = 1 TO J
SMQ  = SSIDMQ.J
"ISPEXEC TBADD   TSSIDMQ"
END
"ISPEXEC TBTOP   TSSIDMQ"
"ISPEXEC TBDISPL TSSIDMQ PANEL(MQØCHNSP)"
IF RC = 8 THEN EXIT;
"ISPEXEC LIBDEF ISPPLIB"
"ISPEXEC LIBDEF ISPMLIB"
"ISPEXEC TBEND  TSSIDMQ"
SSID = SMQ
SELECT
  WHEN SSID = 'MQP5' THEN
       DO
      "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.PANELLIB.USER')"
      "ISPEXEC LIBDEF ISPMLIB DATASET ID('YOUR.PROD.MSGLIB')"
      "ISPEXEC LIBDEF ISPLLIB DATASET ID('YOUR.MA19LOAD')"
       END
  WHEN SSID = 'MQD5' THEN
       DO
      "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.DEVE.PANELLIB.USER')"
      "ISPEXEC LIBDEF ISPMLIB DATASET ID('YOUR.DEVE.MSGLIB')"
      "ISPEXEC LIBDEF ISPLLIB DATASET ID('YOUR.MA19LOAD')"
       END
  OTHERWISE
       DO
      "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.PANELLIB.USER')"
      "ISPEXEC LIBDEF ISPMLIB DATASET ID('YOUR.PROD.MSGLIB')"
      "ISPEXEC LIBDEF ISPLLIB DATASET ID('YOUR.MA19LOAD')"
       END
END
ADDRESS  TSO   "ALLOC  FI(SYSTSPRT) DA(*)"
RCCI = RXMQVC('INIT')
DO FOREVER
ADDRESS ISPEXEC "TBCREATE TBMQCH",
"NAMES(CHANNEL XMITQ CONNAME STATUS TYPE SEQNO CHSTATI CHSTADA)",
"NOWRITE REPLACE"
O = ''
OUT.Ø = Ø
M = OUTTRAP('MQCMD.')
RCC1 = RXMQVC('COMMAND',SSID,'DIS CHS(*) ALL','OUT.')
M = OUTTRAP('OFF')
INTERPRET 'FCS = RXMQVC.RCMAP.'WORD(RCC1,1)
IF  WORD(RCC1,1) <> Ø
    THEN DO
         SAY WORD(RCC1,1)
         SAY FCS
         EXIT
         END
I = STRIP(WORD(RCC1,1),'B')
IF (OUT.Ø <> Ø)
    THEN DO I = 1 TO OUT.Ø
         IF WORD(OUT.I,1) = 'CSQM42ØI'
            THEN DO
                 CHANNEL = STRIP(SUBSTR(WORD(OUT.I,2),1Ø,2Ø))
                 XMITQ   = STRIP(SUBSTR(WORD(OUT.I,4),7,21))
                 CONNAME = STRIP(SUBSTR(WORD(OUT.I,6),9,12))
                 STATUS  = STRIP(SUBSTR(WORD(OUT.I,9),8,8))
                 TYPE    = 'SDR '
                 SEQNO   = STRIP(SUBSTR(WORD(OUT.I,19),1,6))
                 SEQNO   = STRIP(SEQNO,'T',')')
                 CHSTATI = SUBSTR(OUT.I,441,17)
                 CHSTADA = SUBSTR(OUT.I,459,18)||')'
                 ADDRESS ISPEXEC "TBADD TBMQCH"
                 END
         IF WORD(OUT.I,1) = 'CSQM422I'
            THEN DO
                 CHANNEL = STRIP(SUBSTR(WORD(OUT.I,2),1Ø,2Ø))
                 CONNAME = STRIP(SUBSTR(WORD(OUT.I,4),9,12))
                 STATUS  = STRIP(SUBSTR(WORD(OUT.I,7),8,8))
                 TYPE    = 'RCVR'
                 SEQNO   = STRIP(SUBSTR(WORD(OUT.I,17),1,6))
                 SEQNO   = STRIP(SEQNO,'T',')')
                 CHSTATI = SUBSTR(OUT.I,385,17)
                 CHSTADA = SUBSTR(OUT.I,4Ø3,18)||')'
                 ADDRESS ISPEXEC "TBADD TBMQCH"
                 END
         CHANNEL=''
         XMITQ=''
         CONNAME=''
         STATUS=''
         TYPE= ''
         SEQNO=''
         CHSTATI=''
         CHSTADA=''
         END
ADDRESS ISPEXEC "TBTOP   TBMQCH"
ADDRESS ISPEXEC "TBDISPL TBMQCH PANEL(MQCHP)"
IF RC = 8
   THEN DO
        ADDRESS ISPEXEC "TBEND TBMQCH"
        CALL FINAL
        END
   ELSE DO
        IF ZTDSELS > Ø THEN CALL RESUELVE_OPCION_INGRESADA
        O = ' '
        END
ADDRESS ISPEXEC "TBEND TBMQCH"
END
EXIT
/*-- R U T I N A S  ---  carlos-osorio@excite.com    ---------------*/
RESUELVE_OPCION_INGRESADA:
IF O = 'S'                                                 /* START */
   THEN SELECT
        WHEN STATUS = 'RUNNING' THEN
             ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø2)"
        OTHERWISE
             DO
             CMD_START = 'START CHANNEL('||CHANNEL||')'
             M = OUTTRAP('MQCMD.')
             RCCS = RXMQVC('COMMAND',SSID,CMD_START,'OUTSTART.')
             M = OUTTRAP('OFF')
             IF RC = Ø
                THEN DO
                     ADDRESS ISPEXEC "SETMSG MSG(MQS1ØØ)" /* GRABALG */
                     END
                ELSE
                     ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø1)"
             END
        END
IF O = 'T'                                                 /* STOP  */
   THEN SELECT
        WHEN STATUS = 'STOPPED' THEN
             ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø7)"
        OTHERWISE
             DO
             CALL CONFIRMT;
             IF CONFIRMT_RC <> Ø THEN RETURN;
             CMD_STOP  = 'STOP CHANNEL('||CHANNEL||')'
             M = OUTTRAP('MQCMD.')
             RCCS = RXMQVC('COMMAND',SSID,CMD_STOP,'OUTSTART.')
             M = OUTTRAP('OFF')
             IF RC = Ø
                THEN DO
                     ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø8)" /* GRABALG */
                     END
                ELSE
                     ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø9)"
             END
        END
IF O = 'R'                                                   /* RESET   */
   THEN DO
        CALL CONFIRMR;
        IF CONFIRMR_RC <> Ø THEN RETURN;
        CMD_STOP  = 'STOP CHANNEL('||CHANNEL||')'
        M = OUTTRAP('MQCMD.')
        RCCS = RXMQVC('COMMAND',SSID,CMD_STOP,'OUTSTOP.')
        M = OUTTRAP('OFF')
        CMD_RESET = 'RESET CHANNEL('||CHANNEL||')'
        M = OUTTRAP('MQCMD.')
        RCCS = RXMQVC('COMMAND',SSID,CMD_RESET,'OUTRESET.')
        M = OUTTRAP('OFF')
        IF RC = Ø
           THEN DO
                CMD_START = 'START CHANNEL('||CHANNEL||')'
                M = OUTTRAP('MQCMD.')
                RCCS = RXMQVC('COMMAND',SSID,CMD_START,'OUTSTART.')
                M = OUTTRAP('OFF')
                ADDRESS ISPEXEC "SETMSG MSG(MQS1ØØ)"
                IF RC = Ø
                   THEN DO
                        ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø6)"
                        END
                   ELSE
                        ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø1)"
                END
           ELSE
                ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø5)"
        END
IF O = 'D'                                                   /* DISPLAY */
   THEN DO
       "NEWSTACK"
        CALL LIBDEFWINDJ;
        DSNDISCH = USERID() || '.MQDISCH' || '.CMD'  || TIME('S')
        ADDRESS TSO "ALLOC FILE(DISCH) DATASET('"DSNDISCH"') " ,
                    "NEW CAT REUSE UNIT(SYSDA)" ,
                    "LRECL(8Ø) BLKSIZE(2792Ø) RECFM(F B) SPACE(1,1)
CYL"
                     IF RC <> Ø THEN
                        DO
                        ADDRESS ISPEXEC "SETMSG MSG(DBCØ21)"
                        RETURN
                        END
        CMD_DIS = 'DIS CHANNEL('||CHANNEL||')'
        M = OUTTRAP('MQCMD.')
        RCCD = RXMQVC('COMMAND',SSID,CMD_DIS,'OUTDIS.')
        M = OUTTRAP('OFF')
        IF RC = Ø
           THEN DO
                N = 1
                DO UNTIL N >= LENGTH(OUTDIS.1)
                POS_CHAR = POS(')',OUTDIS.1,N)
                CHP = STRIP(SUBSTR(OUTDIS.1,(N+1),(POS_CHAR-N-1)))||')'
                    SELECT
                      WHEN WORD(CHP,1) = '-'       THEN NOP
                      WHEN WORD(CHP,1) = 'SQM41ØI' THEN NOP
                      WHEN WORD(CHP,1) = 'SQM412I' THEN NOP
                      OTHERWISE                         QUEUE CHP
                    END
                N = POS_CHAR + 1
                END
                QUEUE ""
                ADDRESS TSO "EXECIO * DISKW DISCH (FINIS"
                ADDRESS TSO "EXECIO * DISKR DISCH (STEM DISCHQ. FINIS"
               "ISPEXEC TBCREATE TBMQDCH",
               "NAMES(TBMQDCH1)",
               "NOWRITE REPLACE"
                   TBMQDCH1 = CHSTADA
                  "ISPEXEC TBADD  TBMQDCH"
                   TBMQDCH1 = CHSTATI
                  "ISPEXEC TBADD  TBMQDCH"
                   TBMQDCH1 = LEFT('-',49,'-')
                  "ISPEXEC TBADD  TBMQDCH"
                DO J = 1 TO DISCHQ.Ø
                   TBMQDCH1 = SPACE(DISCHQ.J,Ø)
                  "ISPEXEC TBADD  TBMQDCH"
                END
                ADDRESS TSO "FREE DDNAME(DISCH)"
                ISPEXEC 'TBTOP   TBMQDCH'
                ISPEXEC 'ADDPOP  ROW(3)  COLUMN(16)'
                ISPEXEC 'TBDISPL TBMQDCH PANEL(MQTDISCH)'
                ISPEXEC 'REMPOP'
                ISPEXEC 'TBEND   TBMQDCH'
                END
           ELSE
                ADDRESS ISPEXEC "SETMSG MSG(MQS1Ø4)"
```

```
       "DELSTACK"
        CALL LIBDEFPANEL
        W = OUTTRAP(DELDSN.)
        ADDRESS TSO "DELETE ('"DSNDISCH"')"
        W = OUTTRAP('OFF')
        END
RETURN
LIBDEFWINDJ:
SELECT
  WHEN SSID = 'MQP5' THEN
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.WINDOWSJ.LIB')"
  WHEN SSID = 'MQD5' THEN
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.DEVE.WINDOWSJ.LIB')"
  OTHERWISE
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.WINDOWSJ.LIB')"
END
RETURN
LIBDEFWINDO:
SELECT
  WHEN SSID = 'MQP5' THEN
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.WINDOWS.LIB')"
  WHEN SSID = 'MQD5' THEN
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.DEVE.WINDOWS.LIB')"
  OTHERWISE
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.WINDOWS.LIB')"
END
RETURN
/*--------------------  carlos-osorio@excite.com  --------------*/
LIBDEFPANEL:
SELECT
  WHEN SSID = 'MQP5' THEN
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.PANELLIB.USER')"
  WHEN SSID = 'MQD5' THEN
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.DEVE.PANELLIB.USER')"
  OTHERWISE
 "ISPEXEC LIBDEF ISPPLIB DATASET ID('YOUR.PROD.PANELLIB.USER')"
END
RETURN
CONFIRMT:
C = ''
"ISPEXEC ADDPOP POPLOC(O)"
CALL LIBDEFWINDO;
"ISPEXEC DISPLAY PANEL(MQTCHNC)"
IF C = 'S' THEN CONFIRMT_RC = Ø
ELSE CONFIRMT_RC = 1;
"ISPEXEC REMPOP"
CALL LIBDEFPANEL;
RETURN
CONFIRMR:
C = ''
"ISPEXEC ADDPOP POPLOC(O)"
```

```
CALL LIBDEFWINDO;
"ISPEXEC DISPLAY PANEL(MQRCHNC)"
IF C = 'S' THEN CONFIRMR_RC = Ø
ELSE CONFIRMR_RC = 1;
"ISPEXEC REMPOP"
CALL LIBDEFPANEL;
RETURN
/*-----------------------------------------------------------------*/
FINAL:
M = OUTTRAP('MQCMD.')
RCCT = RXMQVC('TERM')
M = OUTTRAP('OFF')
"ISPEXEC LIBDEF ISPPLIB"
"ISPEXEC LIBDEF ISPMLIB"
"ISPEXEC LIBDEF ISPLLIB"
EXIT
RETURN
/* -----  FIN  END  ---- carlos-osorio@excite.com ---------------*/
MEMBER  'YOUR.PANELI(MQØCHNSP)' :
)ATTR
 % TYPE(TEXT)   INTENS(HIGH) SKIP(ON)
   color(turquoise)
 + TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)            HILITE(USCORE)
 ! TYPE(OUTPUT) INTENS(LOW)  CAPS(ON) JUST(LEFT) HILITE(REVERSE)
   color(turquoise)
 @ TYPE(TEXT)   INTENS(HIGH) CAPS(ON)            HILITE(REVERSE)
   color(turquoise)
)BODY EXPAND(\\)
@                       SUBSISTEM(S) MQSeries
@
%COMMAND ===>_ZCMD
%
%                         Opc  Subsistem MQ
+                        %---  -----------
)MODEL
+                         _O+ !SMQ +
)INIT
 .help   = mqØchnsh
 .cursor = O
 &SCRO   = PAGE
)REINIT
 .cursor = O
)PROC
if (.PFKEY = 'PFØ1')
    &pfkeyin = 's'
IF (&ZTDSELS > ØØØØ)
    VER(&O,LIST,S)
)END
MEMBER  'YOUR.PANELI(MQØCHNSH)' :
)attr
```

```
 % TYPE(TEXT)   INTENS(HIGH) SKIP(ON)
   color(turquoise)
 + TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)            HILITE(USCORE)
 ! TYPE(OUTPUT) INTENS(LOW)  CAPS(ON) JUST(LEFT) HILITE(REVERSE)
   color(turquoise)
 @ TYPE(TEXT)   INTENS(HIGH) CAPS(ON)            HILITE(REVERSE)
   color(turquoise)
)BODY EXPAND(\\)
@                        SUBSISTEM(S) MQSeries
%   Opc (OPTIONS) valids :+
+  %S+        - To select the subsystem MQSeries.
%  %Field description :+
+  %Subsistem + Name/Identification of subsystem MQseries
+             defined in MVS - OS/39Ø (SYS1.PARMLIB).
)END
MEMBER  'YOUR.xxxx.panellib.user(MQCHP)' :
)ATTR
 % TYPE(TEXT)   INTENS(HIGH) SKIP(ON)
   color(turquoise)
 + TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)             HILITE(USCORE)
 ! TYPE(OUTPUT) INTENS(LOW)  CAPS(ON) JUST(LEFT)  HILITE(REVERSE)
   color(turquoise)
 ? TYPE(OUTPUT) INTENS(LOW)  CAPS(ON) JUST(RIGHT) HILITE(REVERSE)
   color(turquoise)
 @ TYPE(TEXT)   INTENS(HIGH) CAPS(ON)             HILITE(REVERSE)
   color(turquoise)
)BODY EXPAND(\\)
@                          MQSeries CHANNELS
%COMMAND ===>_ZCMD                                        +%SCROLL
===>_SCRO+
%
%Opc Channel      Xmitq            Conname      Status  Type  Seqno
+--- ------------ --------------  -----------  -------  ----  -----
)MODEL
+_O+!CHANNEL      !XMITQ           !CONNAME     !STATUS !TYPE ?SEQNO +
)INIT
 .cursor = o
 &SCRO   = CSR
 .help   = mqchh
)REINIT
 .cursor = o
)PROC
if (.PFKEY = 'PFØ1')
    &pfkeyin = 'S'
IF (&ZTDSELS > ØØØØ)
    VER(&O,LIST,D,S,T,R)
)END
MEMBER  'YOUR.xxxx.panellib.user(MQCHH)' :
)attr
```

```
 % TYPE(TEXT)   INTENS(HIGH) SKIP(ON) color(turquoise)
 ! TYPE(TEXT)   INTENS(HIGH) SKIP(ON) color(red)
 + TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)            HILITE(USCORE)
 @ TYPE(TEXT)   INTENS(HIGH) CAPS(ON)            HILITE(REVERSE)
   color(turquoise)
)BODY EXPAND(\\)
@                           MQSeries CHANNELS
%  !Opc (OPTIONS) valids :+
+  %D+      - Display the channel definition.
+  %R+      - RESET the message sequence number of the channel.
+  %S+      - START a channel.
+  %T+      - STOP  a channel.
%  !Field descriptions :+
+  %Channel+- Channel name.+
+  %Xmit+   - Transmission channel name.+
+  %Conname+- Connection name.+
+  %Status+ - Channel status. Can have the following values:
%             STARTING,BINDING,INITIALIZING,RUNNING,STOPPING,RETRYING,
%             PAUSED,STOPPED,REQUESTING.
+  %Type+   - Type of the channel. Can be %SDR/RCVR (SENDER/RECEIVER)+
+  %Seqno+  - Sequence number of the last message sent/received.
)END
MEMBER  'YOUR.XXXX.WINDOWSJ.LIB(MQTDISCH)' : (lrecl 5Ø blksize 5ØØØ)
)ATTR
 % TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
                  COLOR(TURQUOISE)
 @ TYPE(TEXT)   INTENS(HIGH) CAPS(ON)
   HILITE(REVERSE) COLOR(BLUE)
 $ TYPE(TEXT)   INTENS(HIGH) CAPS(ON)
   HILITE(REVERSE) COLOR(BLUE)
 ¿ TYPE(OUTPUT) INTENS(LOW)  CAPS(ON)
   HILITE(REVERSE) COLOR(WHITE)
 ! TYPE(OUTPUT) INTENS(LOW)  CAPS(ON)
   HILITE(REVERSE) COLOR(TURQUOISE)
)BODY WINDOW(5Ø,16)
%COMMAND ===>_ZCMD
$  DESCRIPTION OF THE CHANNEL¿CHANNEL           $
$-------------------------------------------------
)MODEL
!TBMQDCH1
)INIT
)PROC
)END
MEMBER  'YOUR.XXXX.WINDOWS.LIB(MQTCHNC)' : (lrecl 4Ø blksize 4ØØØ)
)attr
 % TYPE(TEXT)   INTENS(HIGH) SKIP(ON)
 + TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
   HILITE(REVERSE) color(blue)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)
   HILITE(REVERSE) color(RED)
 ! TYPE(OUTPUT) INTENS(HIGH) CAPS(ON)
   HILITE(REVERSE) color(turquoise)
 @ TYPE(TEXT)   INTENS(HIGH) SKIP(OFF)
   HILITE(REVERSE) color(turquoise)
)BODY WINDOW(4Ø,1Ø) ASIS
@   CONFIRMATION OF THE STOP CHANNEL
@ +Confirm Stop Channel      @
@ +      !CHANNEL         +==>_C@(S|N)
)INIT
 .CURSOR = C
)PROC
VER(&C,NONBLANK,LIST,S,N)
)END
MEMBER  'YOUR.XXXX.WINDOWS.LIB(MQRCHNC)' : (lrecl 4Ø blksize 4ØØØ)
)attr
 % TYPE(TEXT)   INTENS(HIGH) SKIP(ON)
 + TYPE(TEXT)   INTENS(LOW)  SKIP(ON)
   HILITE(REVERSE) color(blue)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)
   HILITE(REVERSE) color(RED)
 ! TYPE(OUTPUT) INTENS(HIGH) CAPS(ON)
   HILITE(REVERSE) color(turquoise)
 @ TYPE(TEXT)   INTENS(HIGH) SKIP(OFF)
   HILITE(REVERSE) color(turquoise)
)BODY WINDOW(4Ø,1Ø) ASIS
@   CONFIRMATION OF THE RESET CHANNEL
@
@ +Confirm Reset Channel     @
@ +      !CHANNEL         +==>_C@(S|N)
)INIT
 .CURSOR = C
)PROC
VER(&C,NONBLANK,LIST,S,N)
)END
MEMBER  'YOUR.XXXX.MSGLIB(MQS1Ø)' :
MQS1ØØ   'CHANNEL STARTED'         .ALARM=YES .WINDOW=LNORESP
'START CHANNEL HAS BEEN COMPLETED SUCCESSFULLY'
MQS1Ø1   'START CHANNEL FAILURE'   .ALARM=YES .WINDOW=LNORESP
'REVIEW RACF PERMISSIONS FOR MQSERIES CL(MQCMDS  )'
MQS1Ø2   'CHANNEL ALREADY RUNNING' .ALARM=YES .WINDOW=LNORESP
'START CHANNEL NOT EXECUTED. THE CHANNEL IS ALREADY RUNNING'
MQS1Ø3   'CHANNEL IS NOT STOPPED'  .ALARM=YES .WINDOW=LNORESP
'START CHANNEL NOT EXECUTED. THE CHANNEL NOT IN STOPPED STATUS'
MQS1Ø4   'CHANNEL DISPLAY FAILURE' .ALARM=YES .WINDOW=LNORESP
'REVIEW RACF PERMISSIONS FOR MQSERIES CL(MQCMDS  )'
MQS1Ø5   'CHANNEL RESET FAILURE'   .ALARM=YES .WINDOW=LNORESP
'REVIEW RACF PERMISSIONS FOR MQSERIES CL(MQCMDS  )'
MQS1Ø6   'CHANNEL RESETED'         .ALARM=YES .WINDOW=LNORESP
'RESET CHANNEL HAS BEEN COMPLETED SUCCESSFULLY'
MQS1Ø7   'CHANNEL ALREADY STOPPED' .ALARM=YES .WINDOW=LNORESP
'STOP  CHANNEL NOT EXECUTED. THE CHANNEL IS ALREADY STOPPED'
```

```
MQS1Ø8   'CHANNEL STOPPED'         .ALARM=YES .WINDOW=LNORESP
'STOP  CHANNEL HAS BEEN COMPLETED SUCCESSFULLY'
MQS1Ø9   'STOP CHANNEL FAILURE'    .ALARM=YES .WINDOW=LNORESP
'REVIEW RACF PERMISSIONS FOR MQSERIES CL(MQCMDS  )'
```

*Carlos Montoya (Peru)*

# Auditing remote administration of WMQ

## INTRODUCTION

Keeping track of what changes have been made to a WMQ (WebSphere MQ) queue manager and who made those changes can be a difficult task. This article and its accompanying source code show how you can create an audit trail of all changes that have been submitted through the remote administration service – the Command Server.

## HOW ADMINISTRATION IS DONE

WMQ provides two main interfaces for its administrative operations, available across all of the distributed platforms. These are the **runmqsc** program, which allows you to type in MQSC commands, and the **Command Server**, which reads its commands from a message queue.

With very minor exceptions the capabilities of both interfaces are the same. However, the Command Server works with messages in a structured binary form known as PCF. Because these messages are just regular MQ messages on a regular MQ queue, they could have been sent from another queue manager on another machine. The Command Server reads these PCF messages (using MQGET), processes them, and then sends any responses to the ReplyToQ. This allows queue managers to be administered from a central site.

There is today, however, no built-in way that allows someone to track what PCF commands have been submitted to change the configuration of a queue manager; being able to monitor this can be useful for audit trails and problem diagnosis.

There are some other aspects of administration that I'm not going to explore in any detail here, but will mention for completeness. For example, updates to the ini files or Windows registry can affect the behaviour of a queue manager. These updates are made outside the control of WMQ itself and any logging, auditing, or access control is under the auspices of the operating system. In order to get the Command Server to process PCF messages it must be running, and that is not done automatically by WMQ on most platforms. You need local procedures that will start the Command Server when the queue manager is started.

## AUDITING **RUNMQSC** OPERATIONS

The **runmqsc** program has a simple text-based command line. It works with the normal stdin, stdout, and stderr devices. Keeping track of what has been entered here could be done fairly easily by getting all users to invoke it via a wrapper program that logs terminal input and output.

SupportPac MS0E is a more sophisticated wrapper, which can also apply further access control checks and will give an audit trail, showing what has been done locally. The Unix version of MS0E is modelled after the **sudo** command; it allows you to have WMQ administrators who are not in the mqm group. In fact there should be very little need for any use of the mqm group or mqm User-ID on systems with MS0E installed.

## AUDITING PCF OPERATIONS

The Command Server is conceptually a very simple program. It reads messages from the SYSTEM.ADMIN.COMMAND.QUEUE, acts on the instructions inside the message, and puts any responses to the named reply queue. It continues to loop through this until it is stopped or the queue manager is ended.

Note that only one Command Server is permitted for a given queue manager and the name of its input queue is hard-coded.

Because the Command Server is using MQGET and MQPUT the messages can be intercepted using the API Exits interface that was first made available with MQSeries V5.2, but formally supported across all the distributed platforms on WebSphere MQ V5.3. My article on API Exits was published in the July 2002 edition of *MQ Update* – please refer to that for more information on what can be done with this very useful extension to WMQ.

One of the nice parts of the API Exit interface is that the queue manager tells the exit code something about the environment in which it is running. In this case we can tell immediately whether the API Exit has been loaded into the Command Server; if it has not, then no further action is taken.

The cmdlog API Exit module dumps information about every message that is read from the input queue by the Command Server. It is formatted to show all the commands and parameters. It also shows the User-ID that will be used for authorization checks and where the response is going to be sent to. Knowing the ReplyToQMgr will allow you to track down who submitted the command.

## THE SOURCE CODE AND HOW TO BUILD IT

The source for this program is contained in a single file, which can be found at www.xephon.com/extras/cmdlogic.txt.

To compile the program I used the following Makefile. My earlier article on API Exits gave an example based on Unix systems; this time I've used Windows. The module should compile (with suitably minor changes) on any of the distributed platforms.

## MAKEFILE.NT

```
#!include <ntwin32.mak>
TARGET_DIR=c:\mqm\exits
# This is where we find the MQSeries header files and libraries
MQM_INC=c:\mqm\tools\c\include
```

```
MQM_LIB=c:\mqm\tools\lib\mqm.lib
# The C compiler
CPP=cl.exe
CFLAGS=-c -W3 -Gs- -Z7 -Od -nologo -LD -D_X86_=1
CDEFINES=-DWIN32 -D_WIN32 -D_MT -D_DLL
all: inst clean
inst: cmdlog.dll
        -@rename $(TARGET_DIR)\cmdlog.dll cmdlog.old
        copy cmdlog.dll $(TARGET_DIR)
cmdlog.dll: $*.obj
        link -nod -nologo -dll @cmdlog.link -out:cmdlog.dll -
   def:cmdlog.def
cmdlog.obj: cmdlog.c
        $(CPP) $(CFLAGS) $(CDEFINES) cmdlog.c
# Get rid of files created during the build steps
clean::
        -@erase *.obj *.pdb
        -@erase cmdlog.exp
        -@erase cmdlog.lib
        -@erase *.bin *.res
```

On Windows you also need *cmdlog.def* and *cmdlog.link* to complete the build process; these files are not needed on the Unix platforms.

## CMDLOG.DEF

```
 LIBRARY CMDLOG
DESCRIPTION 'API Exit for logging remote commands'
EXPORTS
   EntryPoint
```

## CMDLOG.LINK

```
cmdlog.obj
mqm.lib mqmzf.lib
msvcrt.lib oldnames.lib kernel32.lib user32.lib
```

## SAMPLE OUTPUT

```
[--------] metaylor           Fri Jul 25 15:22:45 2003
  Command     : Start Command Server
[--------] metaylor           Fri Jul 25 15:23:11 2003
  Command     : INQUIRE QMGR
  Reply QMgr  : 'apix                                     '
  Reply Queue: 'MQAI.REPLY.3F213D3401020020               '
  Parameters :
```

```
  Response   : Success
[--------] metaylor           Fri Jul 25 15:23:11 2003
  Command    : INQUIRE CHANNEL
  Reply QMgr : 'apix                                            '
  Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
  Parameters :
    CHANNEL_NAME                 '*'
    CHANNEL_TYPE                 8
  Response   : Success
[--------] metaylor           Fri Jul 25 15:23:14 2003
  Command    : INQUIRE QUEUE
  Reply QMgr : 'apix                                            '
  Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
  Parameters :
    Q_NAME                       '*'
    CLUSTER_INFO                 1
    Q_ATTRS
      [0]               1009
  Response   :  22 Success(es)
          0 Warning(s) (Last RC 0)
          0 Error(s)   (Last RC 0)
[--------] metaylor           Fri Jul 25 15:57:01 2003
  Command    : INQUIRE QUEUE
  Reply QMgr : 'apix                                            '
  Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
  Parameters :
    Q_NAME                       'SYSTEM.DEFAULT.LOCAL.QUEUE'
    Q_TYPE                       1
    Q_ATTRS
      [0]               1009
  Response   : Success
[--------] metaylor           Fri Jul 25 15:57:04 2003
  Command    : INQUIRE NAMELIST
  Reply QMgr : 'apix                                            '
  Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
  Parameters :
    NAMELIST_NAME                '*'
  Response   :  2 Success(es)
          0 Warning(s) (Last RC 0)
          0 Error(s)   (Last RC 0)
[--------] metaylor           Fri Jul 25 15:57:04 2003
  Command    : INQUIRE PROCESS
  Reply QMgr : 'apix                                            '
  Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
  Parameters :
    PROCESS_NAME                 '*'
  Response   : Success
[--------] metaylor           Fri Jul 25 15:57:08 2003
  Command    : CHANGE QUEUE
  Reply QMgr : 'apix                                            '
```

```
   Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
   Parameters :
     Q_NAME                      'SYSTEM.DEFAULT.LOCAL.QUEUE'
     Q_TYPE                      1
     USAGE                       0
     DEF_PERSISTENCE             1
     DEF_PRIORITY                0
     INHIBIT_GET                 0
     INHIBIT_PUT                 0
     HARDEN_GET_BACKOUT          1
     DEF_INPUT_OPEN_OPTION       2
     MAX_MSG_LENGTH              4194304
     MAX_Q_DEPTH                 5000
     BACKOUT_THRESHOLD           0
     TRIGGER_MSG_PRIORITY        0
     TRIGGER_DEPTH               1
     TRIGGER_TYPE                1
     TRIGGER_CONTROL             0
     RETENTION_INTERVAL          999999999
     MSG_DELIVERY_SEQUENCE       0
     SHAREABILITY                1
     SCOPE                       1
     Q_SERVICE_INTERVAL_EVENT     0
     Q_DEPTH_LOW_LIMIT           20
     Q_DEPTH_LOW_EVENT           0
     Q_DEPTH_HIGH_LIMIT          80
     Q_DEPTH_HIGH_EVENT          1
     Q_DEPTH_MAX_EVENT           1
     DIST_LISTS                  0
     Q_SERVICE_INTERVAL          999999999
     DEF_BIND                    0
     Q_DESC                      ' '
     PROCESS_NAME                ' '
     INITIATION_Q_NAME           ' '
     BACKOUT_REQ_Q_NAME          ' '
     TRIGGER_DATA                ' '
     CLUSTER_NAMELIST            ' '
     CLUSTER_NAME                ' '
   Response   : Success
 [--------] metaylor          Fri Jul 25 15:57:08 2003
   Command    : INQUIRE QUEUE
   Reply QMgr : 'apix                                            '
   Reply Queue: 'MQAI.REPLY.3F213D3401020020                     '
   Parameters :
     Q_NAME                      'SYSTEM.DEFAULT.LOCAL.QUEUE'
     Q_TYPE                      1
     Q_ATTRS
       [0]               1009
   Response   : Success
```

## INSTALLATION AND CONFIGURATION

For Unix systems you need to update the qm.ini file for the queue manager on which the exit will be used. For Windows you can use the WMQ Services panels to update the registry.

For both styles of configuration you need to give the name of the module (cmdlog), a sequence number (I used 100), and the function to be invoked when it is loaded (EntryPoint). On Windows, the module is a DLL.

Here is a stanza suitable for the qm.ini file on Unix.

```
ApiExitLocal:
   Sequence=100
   Function=EntryPoint
   Module=/var/mqm/exits/cmdlog
   Name=PCFLogger
```

## SECURITY

It's worth mentioning a couple of aspects of security that are related to how remote administration is controlled.

First, you need to be careful about whom you allow to put messages to the command queue. Anyone who can get messages onto this queue can potentially modify your configuration in serious ways. Only allow well-controlled User-IDs to put messages to this queue.

Second, the majority of applications that put messages to this queue are going to be coming over a remote connection. Make sure your channels are secure. Do not allow clients to connect into a channel where the MCAUSER attribute is blank. Use SSL and/or channel exits to protect the connections. Remember that, when putting to a target queue, the PUTAUT attribute of a channel can be used to perform access control based on the message's User-ID field instead of the MCA's User-ID.

There are some products available that can encrypt message data, decrypting it during the MQGET operation. These can be very useful for additional authentication and authorization of who put the PCF message to the command queue. If such products are used, the decryption operation needs to be called before the cmdlog API Exit is invoked. When the security module is implemented as an API Exit, this can be done easily by modifying the sequence number in the qm.ini file.

## POSSIBLE ENHANCEMENTS

As always there are plenty of developments that could be done with this code but which I've omitted here. Here are some of the ideas I thought about while writing it.

- Better logging of responses and error codes. The cmdlog exit only gives details of the command issued with a summary of the response codes. Problem diagnosis might be easier if the responses were also expanded along with the detailed error structures that might be returned. However, I did not feel this information was needed just for an audit trail.
- There are some operations that you might like to control remotely but for which there are currently no defined PCF equivalents. Two examples are the **setmqaut** and **dspmqaut** programs, which must be run on the local machine; they cannot be executed from a central point.

  It would be possible for you to define your own PCF commands and, when they are recognized by the cmdlog exit, to execute them directly instead of giving them to the Command Server. As the API Exit is able to call MQI functions it could send responses itself to the requesting program. This gives you extended function while fitting inside the overall WMQ administration framework.
- You could write tools to summarize the log file generated by the exit, for example:
  - how many remote commands were issued and by whom
  - how many errors there were.
- It would also be possible to generate automatically the FormatCommand and FormatParm functions from the WMQ

header files as part of the build process. I didn’t do that this time but some fairly simple perl or awk/grep code could used in the Makefile. That would make it easier to update the exit code whenever there is a new release of WMQ.

## SUMMARY

I hope this article has given something that is immediately useful for audit trails as well as prompting more ideas about what you can do with API Exits. Perhaps one day WMQ will include some of these functions in the base product. But until then please use this code and feel free to modify it.

*Mark E Taylor*
*IBM Hursley (UK)*

## MQ news

DataPower Technology, a provider of intelligent XML-Aware Network (XAN) infrastructure, has recently announced that the latest firmware release for its DataPower XS40 XML Security Gateway and XA35 XML Accelerator network devices includes support for WebSphere MQ.

This enables direct access to the XML acceleration and Web services security features of DataPower's XAN products for WMQ applications.

Support for WMQ also makes it possible to use the XS40 XML Security Gateway as a wirespeed trusted gateway between internal WMQ-based Web services and external HTTP Web services.

*For more information contact:*
DataPower Technology, One Alewife Center, 4th Floor, Cambridge, MA 02140, USA.
Tel: +1 617 864 0455.
Fax: +1 617 864 0458.
Web: http://www.datapower.com

* * *

MQSoftware is this month releasing Version 3.1 of its Q Pasa! middleware management and monitoring solution.

Version 3.1 adds a Web-enabled management console that will facilitate anytime, anywhere authorized access to real-time data through Q Pasa!'s Business Dashboard, which is claimed to provide a consolidated view of what is happening in the middleware environment in real-time.

*For more information contact:*
MQSoftware,1660 South Highway 100, Suite 400, Minneapolis, Minnesota 55416, USA.
Tel: +1 952 345 8720.
Fax: +1 952 345 8721.
Web: http://www.mqsoftware.com

MQSoftware, Surrey Technology Centre, 40 Occam Road, Surrey Research Park, Guildford, Surrey, GU2 7YG, UK.
Tel: +44 1483 295400.
Fax: +44 1483 573704.

* * *

IBM has announced the release of V4.2.4 of WebSphere Business Integration Modeller and Monitor, products that are claimed to improve business visibility for fact-based decision making and optimized business processes.

New features of the WebSphere Business Integration Modeller include the WebSphere Business Integration Workbench, which enables users to model, and also to represent, WebSphere MQ Integrator Broker V2.1 and WebSphere Business Integration Message Broker V5 flow activities.

Amongst a number of new features in the latest release of the WebSphere Business Integration Monitor is the facility to monitor data and events from WebSphere Business Integration Message Broker V5 and WebSphere MQ Integrator Broker V2.1.

*For more information contact your local IBM representative.*

* * *

xephon