# 21

# MQ

*March 2001*

## In this issue





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

## Contributions

When Xephon is given copyright, articles published in *MQ Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. For more information about contributing an article you can download a copy of our *Notes for Contributors* from www.xephon.com/contnote.html.

## *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.

## Commissioning Editor

Peter Toogood
E-mail: PeterT@xephon.net

## Managing Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

## Disclaimer

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

## Subscriptions and back-issues

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

# BLOB-based message processing techniques for MQSeries Integrator Version 2

MQSeries Integrator Version 2 (MQSI) implements the ESQL message processing language. ESQL can be used to transform and augment messages in many ways, both on its own and in interaction with other components of MQSI or external DBMS data sources. In this article we look at some techniques for processing messages using ESQL that operate on the body of an incoming message as a single BLOB entity.

'BLOB' is an ESQL data type. An ESQL field or variable of type BLOB is considered to hold binary data, that is, it represents no value over and above the string of bytes that it contains as data. So a message body consisting of a single BLOB entity is considered to represent nothing more than its constituent bytes. There are two main situations where this might be useful:

- When a legacy-format message, which for some reason cannot be modelled satisfactorily with the MRM or other IBM-supplied parser, is being processed.
- When modifying the message body at a binary level evades or alleviates some problem that would otherwise be complex or time-consuming to solve.

Examples of both these situations, drawn from real life situations, are discussed below.

The ESQL, which implements a BLOB-based processing technique, will generally be contained in an MQSI Compute node. In the cases considered below, we will be assuming that the job of the ESQL, which does the actual processing, is to take the BLOB value held in an ESQL variable named *inputData* and transform it into another BLOB value to be placed in the ESQL variable named *outputData*. The body of the Compute node's input message will be placed in *inputData* before the transformation begins; the value of *outputData* will be used as the body of the Compute node's output message once the transformation is complete.

Using a temporary ESQL variable in this way is more efficient than repeatedly referencing the input and output messages themselves.

## GETTING THE INPUT MESSAGE BODY INTO *INPUTDATA*

This is easily achieved with the following ESQL (all ESQL in this article was created and tested using MQSI v2.0.1):

```
DECLARE inputData BLOB;
SET inputData = BITSTREAM(InputBody);
```

Note that here, and throughout the rest of this article, it is assumed that any message headers have already been dealt with, for example by selecting the 'copy message headers' radio button from the Compute node configuration panel. The **BITSTREAM** function returns the input message body's bitstream as a BLOB value, which is assigned to *inputData*.

This will work regardless of the domain of the input message (BLOB, MRM, XML, etc). For example, consider an XML message in an ASCII code page with the simple body *<Data>hello</Data>*. This would be processed by the above ESQL so as to result in the BLOB value **X'3c446174613e68656c6c6f3c2f446174613e'** (the binary values of the ASCII characters *<Data>hello</Data>*) being assigned to *inputData*.

## ASSIGNING *OUTPUTDATA* TO THE OUTPUT MESSAGE BODY

The ESQL from the previous section can easily be extended so that it performs the necessary steps:

```
DECLARE inputData BLOB;
DECLARE outputData BLOB;
SET inputData = BITSTREAM(InputBody);
SET outputData = inputData;
SET OutputRoot."BLOB"."BLOB" = outputData;
```

The final line creates an output message body using the IBM-supplied BLOB parser and assigns to it the value contained in *outputData*. It should be emphasized that the BLOB parser is a separate entity to the ESQL BLOB data type. The parser is so named because its action on parsing a message is to create a single field named BLOB, which has, as an ESQL BLOB data type value, the binary data of the message body.

Conversely, when building an output message it takes any binary data contained in an existing BLOB field and uses it as the bitstream of the

output message. Hence the field reference *OutputRoot."BLOB"."BLOB"* refers to (and creates if it does not already exist) a field of the output message named BLOB, which belongs to the BLOB parser – precisely what we want. The double quotes around both instances of BLOB are not necessary and have no particular significance; they are included purely to avoid a bug in the Compute node syntax checker – present in versions of MQSI to date – which causes it to report an error if an ESQL keyword identifier is included as part of a field reference. Using keyword identifiers in this way is not an error and will not cause a deploy-time failure; the only reason for putting quotes around the keyword identifier is that it mollifies the syntax checker.

The penultimate line, **SET outputData = inputData**, is perhaps the simplest possible example of a message transformation, copying the input message body, unchanged, to the output message body. It is this line that we will be replacing with some more complex algorithms to implement the various processing techniques discussed below. The other lines remain constant, and although they are implicitly present, will not be repeated in subsequent ESQL examples.

## THE RCD NODE

The end result of the above will be that your Compute node outputs a message in the BLOB domain with the transformed binary value from *outputData* as its body. It may be that a BLOB domain message is not what you want; for example, having tinkered with the bitstream directly in the Compute node you may want to treat the message as XML or MRM. The 'Reset Content Descriptor' node makes this possible.

An RCD node reserializes an incoming message, invoking the necessary parsers to turn its tree into a wire-format sequence of bytes – the binary data that is actually transmitted along the network 'wire'. It then changes the domain (and, if required, other associated attributes, such as message type and format) of the message and passes it down the message flow. The next time any subsequent node tries to parse it, the bitstream will be treated as belonging to whatever new domain the RCD specified. Consider the following 'transformation':

```
SET outputData = X'3c4e616d653e4a6f686e3c2f4e616d653e';
```

This assigns a binary literal representing the ASCII string *<Name>John</Name>* to *outputData*. If this is assigned to *OutputRoot."BLOB"."BLOB"* then the parser invoked by the RCD node to effect reserialization will be the BLOB parser, which simply uses whatever BLOB value it fields in the BLOB field as the wire-format byte sequence. If the RCD node changes the domain to XML then the next node to parse the message would see an XML message with the single field *<Name>* containing the value 'John'. This is because the wire format produced by the BLOB parser upon reserialization of the above value is such that it can also form the wire format of a valid XML message.

## THE TRANSFORMATIONS

Now that we can get data into and out of the Compute node it is time to look at what happens in between. BLOB-based transformations rely on the fact that an ESQL BLOB value is, in effect, a string of bytes and, as such, can be operated upon by the string manipulation functions, including **POSITION**, **LENGTH**, **TRIM**, **SUBSTRING**, **OVERLAY**, and the concatenation operator, ||. Each of the following techniques using these operations was developed to solve a real problem, the nature of which is discussed below. However, they can easily be adapted for use in other situations.

### Find and replace

*Scenario*

We needed to put a **NULL** value, consisting of the four-byte sequence X'FFFFFFFF', into a four-character field of an MRM message. The message was processed in an ASCII code page but then converted into EBCDIC. This caused a problem because the X'FFFFFFFF' value was treated as if it were ASCII data and itself 'converted' by the MRM parser. EBCDIC does not support any character with a binary value of X'FF', so it was not possible to preserve this value after conversion from ASCII – we just ended up with X'40404040'.

The solution we adopted was to allow conversion to take place, then process the message as a single BLOB entity, replacing the value

X'40404040' with X'FFFFFFFF'. Because this processing took place after conversion, it allowed us to put the necessary value into the EBCDIC message.

The sample ESQL copies *inputData* to *outputData* with all occurrences of the BLOB literal specified in **find replaced**, with the BLOB literal specified in **replace**.

ESQL

```
DECLARE find BLOB;
DECLARE replace BLOB;
DECLARE pos INTEGER;
SET find = X'40404040';
SET replace = X'FFFFFFFF';
SET outputData = X'';
SET pos = POSITION(find IN inputData);
WHILE (pos <> 0 AND inputData IS NOT NULL) DO
     SET outputData = outputData ||
         SUBSTRING(inputData FROM 1 FOR (pos-1)) || replace;
     SET inputData = SUBSTRING(inputData FROM pos+LENGTH(find));
     SET pos = POSITION(find IN inputData);
END WHILE;
SET outputData = outputData || inputData;
```

*Notes*

- The concatenation operator cannot accept **NULL** as one of its operands. This may be fixed in subsequent versions of MQSI, but, for now, we have to write the ESQL is such a way that **NULL** operands are avoided. This is why *outputData* is initialized to X'' before processing begins.
- Every **SUBSTRING** and concatenation operation takes up processing time. To maximize efficiency, **POSITION** is used to locate the next instance of the search string in *inputData*. This means that the number of **SUBSTRING**/concatenation operations is dependent on the number of occurrences of the search string, not the length of *inputData*. The size of the input message does not, therefore, have a direct linear relationship to the processing time, which may be an important consideration if you are processing large messages.

**Parsing a mixed-format message**

*Scenario*

We were trying to process a legacy-format message that consisted of a 16-byte header followed by XML data. This could not be processed by the XML parser as it stood, because the header portion was not valid XML. Initially, we solved this by processing the message as a BLOB, modifying the bitstream so that the header appeared to be just another XML field, then using an RCD node to turn it into an XML message.

The sample ESQL transforms a message with the format: *[16-byte header]<xml_data> ... </xml_data>*, into *<Msg><Header>[16 - byte header]</Header><xml_data> ... </xml_data></Msg>*, where *<xml_data> ... </xml_data>* represents any valid XML data.

ESQL

```
SET outputData = '3c4d73673e3c4865616465723e'
     || SUBSTRING(inputData FROM 1 FOR 16)
     || x'3c2f4865616465723e'
     || SUBSTRING(inputData FROM 17)
     || x3c2f4d73673e';
```

*Notes*

- The binary literals in the above represent the following ASCII values:

  X'3c4d73673e3c4865616465723e' ==> *'<Msg><Header>'*

  X'3c2f4865616465723e' ==> *'</Header>'*

  X'3c2f4d73673e' ==> *'</Msg>'*

  A quick way to work out the binary literal representation of a particular ASCII (or other code page) value is to put a message on an MQSeries queue containing the ASCII literal required. Investigating the contents of the queue, the MQSeries Explorer tool or the **amqsbcg** utility will then show the data portion of the message with the ASCII literal and its binary representation, next to each other.

**Interpreting the contents of fields**

*Scenario*

Another problem involved removing, rather than encapsulating, a non-XML header from an otherwise XML message. There was an additional difficulty because the header was not of fixed length; instead, it had a length field in a fixed position in the header. We needed to process the message, initially as a BLOB, because no other parser could handle its mixed format; however, this meant that direct access to the value of the header length field was not possible.

To work around this, we interrogated the bytes making up the length field directly, and converted their individual binary values into an integer value for the field as a whole. Once this was obtained we could remove the correct number of bytes to excise the header, leaving a pure XML bitstream for an RCD node to convert to XML domain.

The sample ESQL assumes an incoming message with a variable-length header followed by XML. Bytes five to eight of the header are assumed to contain an ASCII field representing the length of a header.

ESQL

```
DECLARE lengthAsBlob BLOB;
DECLARE lengthAsInt INTEGER;
DECLARE index INTEGER;
SET lengthAsBlob = SUBSTRING(inputData FROM 5 FOR 4);
SET index = 1;
SET lengthAsInt = 0;
WHILE (index < 5) DO
     SET lengthAsInt = lengthAsInt +
          ((CASE SUBSTRING(lengthAsBlob FROM index FOR 1)
              WHEN x'30' THEN 0
              WHEN x'31' THEN 1
              WHEN x'32' THEN 2
              WHEN x'33' THEN 3
              WHEN x'34' THEN 4
              WHEN x'35' THEN 5
              WHEN x'36' THEN 6
              WHEN x'37' THEN 7
              WHEN x'38' THEN 8
              WHEN x'39' THEN 9
              END)
          *
```

```
          (CASE index
              WHEN 1 THEN 10000
              WHEN 2 THEN 1000
              WHEN 3 THEN 10
              WHEN 4 THEN 1
              END));
     SET index = index + 1;
END WHILE;
SET outputData = SUBSTRING(inputData FROM lengthAsInt+1);
```

*Notes*

- For all index purposes, ESQL starts counting at one, not zero. This can sometimes cause confusion if you are used to C-style indexing, where the count starts at zero.
- **CASE** is used twice, first to translate the binary value of each ASCII digit into the corresponding integer value, and second to multiply that integer by a second integer representing the position in the ASCII value. In other words, the first digit is multiplied by 10,000, the second by 1,000, and so on, down to one for the last digit. This is a fairly crude casting mechanism, and in subsequent versions of MQSI some additional function may be added to ESQL, which will render such an approach unnecessary.

## CONCLUSION

The three scenarios discussed here are just a sample of the many possible applications of BLOB-based message processing techniques. Although the aim of MQSI is to offer all the tools necessary to process messages as pre-defined formats, it will always be the case that there are some situations where what's on offer isn't quite what you need. In those situations, an understanding of how to use ESQL to modify the message bitstream directly can be invaluable.

---

*Rafael Jay*
*Developer, IBM UK*

# MQSeries channel security exits

## AUTHENTICATION ON THE WIDER NETWORK

The increasing use of MQSeries within organizations has been shown to pay dividends in facilitating the development of distributed networked applications. It enables companies to leverage their often huge investments in legacy applications and brings them into the distributed world without having to code the networking layer and manage session-based connections.

As the level of sophistication builds up, and MQSeries is used to connect to external nodes for B2B or B2C applications, so the security implications multiply. Security control for messages revolves around four key criteria:

- Authentication: are the users who they say they are?
- Integrity: on receipt, is the message content exactly the same as it was when sent?
- Privacy: can the contents be viewed by anyone other than the intended recipient?
- Non-repudiation: can a sender of a message deny that they sent the message?

MQSeries security exits can help with the first of these issues.

For many years, secure access to mainframes has been tightly controlled by the use of user-ids and passwords, together with the proximity of terminals to the machine. This verification allows access to various system resources as set up by the systems administrators. With distributed systems connected by MQSeries channels, it becomes more difficult to ensure that the node requesting the start of a channel is given access to mission-critical data.

It is a well known fact that any security system is only as strong as its weakest link. With MQSeries, the ability to put a message to a queue is based on applications such as RACF, ACF2, Top Secret, etc on OS/390, and through the OAM on other platforms. An application's

security level is checked when the application attempts to connect to a queue manager and access its resources, such as opening a queue or getting a message.

The request succeeds or fails depending on the permissions granted to the user. The user is normally verified by the operating system via some form of password control before access to the queue manager can be attempted and the user allowed to put a message on a queue. There is an excellent discussion of how to use RACF and the OAM for end-to-end security by Sam Garforth in *MQ Update* Issue 1 (July 1999).

As soon as that message goes outside the domain of that operating system, however, the user-id in the message header is all that remains of the verification process. In a tightly-secured Windows network it is possible to verify that the user-id should have access to resources within that network, but relying on the internal IS team to maintain permissions for systems based outside their normal domain of operation may be difficult. Asking the team to do that for systems outside the corporation is definitely unreasonable. Some other system of authentication must be employed.

If some basic information is known about the target system it is possible to spoof the receiver into thinking that you are somebody else. With a user-id (password not required), channel name, and TCP/IP address of the receiver, you can send messages to a remote queue manager. On a stand-alone Windows system with admin authority you can set up a logon id in the **mqm** group and set-up an MQSeries sender channel to the target system. You can then format the message appropriately and send it to the receiving system, which will accept the message as being from that user-id. Even if requester/sender channels are used to implement a callback system, it is possible, within the Windows environment, to set a specific TCP/IP address to simulate the legitimate system, which the sender channel will call back to.

There is, however, a simple method of implementing additional security to the initial connection without resorting to encryption techniques and the administration tasks of key maintenance, in the form of an MQSeries security exit.

## MQSERIES CHANNEL SECURITY EXITS

MQSeries provides a means of establishing the identity of the partner channel during initialization by means of a security exit. This exit allows the exchange of information through user-written code, which can determine whether the channel should actually start.

The channel security exit is first called at MCA initiation with an initialization instruction. The standard initial data negotiation then takes place, first to synchronize message sequence numbers and then to initiate channel start-up. This is followed by an opportunity for the exits to execute an exchange of data. In a sender/receiver configuration, the receiver has the first opportunity to initiate the exchange by providing a message buffer to be delivered to the sender end. If it declines the opportunity, the sender may initiate the conversation. An exchange may then take place, described in detail later.

When both ends have finished sending and processing messages and returned a successful completion code, the channel will be started for normal operation.

The exit is re-invoked on termination of the channel. The exchange of information may be any type of data. An example security exit for DCE authentication is supplied by IBM as a supportpac. I will show a much simpler example as a skeleton, to demonstrate the mechanics and structure of how the channel security exit operates.

## EXIT PROGRAM OPERATION BASICS

The exit is designed to operate such that data can be passed in both directions for cross-validation. There are two entry points: **MY_CHANNELEXIT_SDR** and **MY_CHANNELEXIT_RCV**. The **_RCV** entry point is designed to operate on an MQSeries receiver connection channel, while the **_SDR** entry point handles the required functionality for a matching sender channel.

The exit may be invoked one or more times by the Message Channel Agent (MCA), each time passing a reason code (**ExitReason**) indicating why the exit is being called. When the exit finishes processing, it passes back a response code (**ExitResponse**) to the MCA indicating what it should do next, together with an optional

information response (**ExitResponse2**). The **ExitResponse** can determine whether to continue as normal (**MQXCC_OK**), to send a message to the partner MCA (**MQXCC_SEND_SEC_MSG**), to send a message and wait for a reply **(MQXCC_SEND_AND_REQUEST_SEC_MSG**), or to not go any further and close the channel (**MQXCC_CLOSE_CHANNEL**).

For a channel security exit the reason codes occur in the following order:

- **MQXR_INIT**. The exit is invoked with this reason code having been first called by the MCA. The code to process this event can be used to allocate any storage for use by the exit.
- **MQXR_INIT_SEC**. This event is only called for channel security exits. The receiver's security exit is always invoked with this reason immediately after being invoked with **MQXR_INIT**, to give it the opportunity to initiate a security exchange. If it declines the opportunity, the sender's security exit is invoked with **MQXR_INIT_SEC**. If the receiver's security exit does initiate a security exchange, however, the sender's security exit is never invoked with **MQXR_INIT_SEC**; instead it is invoked with **MQXR_SEC_MSG** to process the receiver's message. The security exchange must complete at the side that initiated the exchange, so if a security exit is invoked with **MQXR_INIT_SEC** and it does initiate an exchange, the next time the exit is invoked it will be with **MQXR_SEC_MSG** (see below).
- **MQXR_SEC_MSG**. Again, this event is only called for channel security exits, indicating that a security message has been received. If it is a 'real' message (see below), the **pAgentBuffer**, **pDataLength** parameters passed to the exit by the MCA contain the message data and its length sent by the partner MCA. For the side which invoked the exchange (see **MQXR_INIT_SEC**), the exit is invoked with this reason code even if no message exchange takes place, but in this case there is no message and **pDataLength** is set to zero. Checking for **pDataLength==0** can give an indication that no matching security exit has been invoked on the partner MCA.

- **MQXR_TERM**. This event indicates that the channel is terminating and should be used to clean-up any memory allocated in the exit.

## THE CODE

### Platforms

The attached code has been tested on Windows 2000 and Solaris. Slight modifications may be required when porting to OS/390. When defining structures, it is important either to align all members on 16-byte boundaries or, as in this case, set matching compilation options that ensure correct byte alignment.

### Entry points

There are two entry points to the program:

- **MY_CHANNELEXIT_SDR()**.
- **MY_CHANNELEXIT_RCV()**.

They are identical in that they invoke the function **MyChannelExit()**, but each sets a Boolean value **isSender** to indicate on which side of the channel it is working. This could in fact be determined by the program looking at the value of **ChannelType**, passed into the function as part of the MQCD structure pointed at by **pChannelDefinition**, the second input parameter of the entry point definition, but this way allows the user to specify which end initiates the exchange. In this implementation, the receiver side initiates the exchange.

### MyChannelExit()

Once in the function, **MyChannelExit()** first makes sure it has cleaned-up any previously allocated memory pointed at by the **pExitBufferAddr** parameter. It then checks that this exit is being invoked by a security exit by checking the **pParms->ExitId** value. If an MQSeries administrator mistakenly defined the exit to be a channel message exit or any other type, then the channel is stopped and no further action is taken.

Next, the **ExitReason** is used to switch between execution paths:

- **MQXR_INIT**: simply invokes a status **printf()** message. It can be used to initialize any storage required by the exit code.
- **MQXR_INIT_SEC**: this checks to see if we are a receiver (thereby having the first chance to initiate a message exchange), and, if so, invokes the **sendSecurityInfo()** function to send data to the partner. If the receiver processes this entry the equivalent **ExitReason** is not called on the sender.
- **MQXR_SEC_MSG**: this indicates that we have received a message from the partner and invokes the function **processSecurityMessage()** to check the message contents. If this function returns a **FALSE** code the channel is closed. The same function is called for both sender and receiver.

  If this is a sender, it needs to respond to the security message received above, and so invokes the **sendSecurityInfo()** function.
- **MQXR_TERM**: this simply invokes a status **printf()** message. It can be used to clean-up any storage required by the exit code.

Figure 1 illustrates the program flow.

**sendSecurityInfo()**

This function firstly allocates storage to hold the **SECEXITDATA** structure, which holds the user-id and password data picked up from the operating system. In this program, static strings are placed into this buffer. Also included in this example buffer is the **SCYDATA** value taken from the channel definition.

The **ExitResponse** and **ExitResponse2** values are now set to indicate to the MCA what is expected to happen next. If it is running on a sender the **ExitResponse** is set to **MQXCC_SEND_SEC_MSG** to indicate that we want the MCA to transport the data pointed to by the **pExitBufferAddr** to the partner channel. A sender will have already received the message from the receiver, so it does not expect a further reply after sending this message. A receiver node is using this function to initiate a message exchange and thus uses **ExitResponse = MQXCC_SEND_AND_REQUEST_SEC_MSG** to indicate that it wishes to send the data and expects to receive a reply.

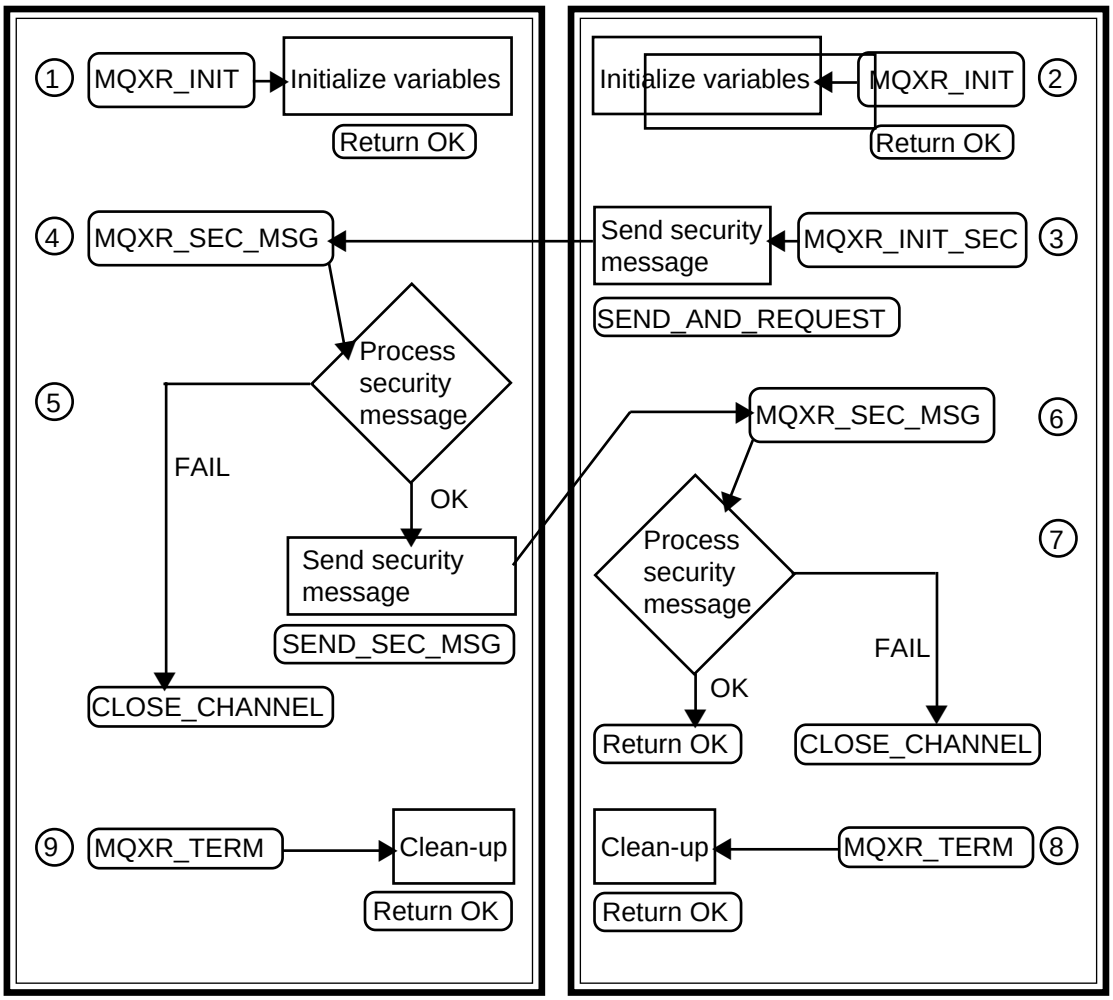


*Figure 1: Program flow*

The function returns a pointer to the newly allocated buffer.

**ProcessSecurityMessage()**

This function will be used to validate the message from the partner MCA.

Firstly, it checks that the size of the incoming message is exactly the size of the expected structure. Then a further check is made to make sure it is a recognized message type and that it is not receiving a message from a channel exit of the same type. (There is potential for looping if both sides send and expect a reply.) At this point, the

message contents can be examined to get an indication that they came from a partner exit using the same algorithm.

The server should verify the message contents at this point, and the receiver can verify that it is communicating with the correct server if that functionality is required. As well as having the hostname passed as part of the **SECEXITDATA** structure, the exit has access to the partner connection name, which, depending on the network, is either a TCP/IP address or host name.

This function returns either **TRUE** or **FALSE**, depending on whether the security check passed successfully. If it returns **FALSE** the exit closes with **ExitResponse** of **MQXCC_CLOSE_CHANNEL**, which causes the channel communication to stop.

## COMPILING THE EXIT PROGRAM

The program can be compiled on Solaris using the following command script:

```
cc -c -KPIC MySecExit.c
ld -G MySecExit.o -o MySecExit
```

To compile it on NT (or Windows 2000) using MSVisual C++, the project should be created as an empty WIN32 Dynamic Link Library. The *MySecExit.c* file should be added to the project, and structure member alignment set to one byte. The additional library module *MQMVX.LIB* should be added to the link options. The headers can be found in *{mqm_root}\tools\c\include* while the libraries are located at *{mqm_root}\tools\lib.*

When compiled, the exit should be copied to */var/mqm/exits* on Unix and *{mqm_data_root}\exits* on NT/Windows 2000.

## MQSERIES SECURITY CHANNEL EXIT DEFINITION

Channel exits must be named in the channel definition. You can do this when you first define the channels, or you can add the information later, using for example the MQSC command **ALTER CHANNEL**.

A sample definition for the skeleton exit provided follows:

```
ALTER CHANNEL(MYCHLNAME) SCYEXIT('MySecExit(MY_CHANNELEXIT_SDR)')
```

If the channel definition does not contain a user exit program name the user exit is not called.

A single exit library can contain a number of entry points that can execute code for any MQSeries exit. The example *MySecExit* library contains two entry points, one which is designed to initiate the message exchange from the receiver end, and one designed for the sender end, which responds. The exit implements the sending and processing of security information messages from both partners in the exchange. Either end can choose not to continue the conversation and close the channel.

## VIEWING THE OUTPUT

If you wish to see the output from the channel security exit stand out from the **printf()** statements, a different command needs to be executed, depending on whether you are looking at the sending or receiving end of the channel.

### Sender

To see the output from the sender exit it is possible to start the channel directly, even if no messages are waiting to be moved across the channel, by executing **runmqchl**, eg:

```
runmqchl -c MYCHL.TO.YOURCHL -m MYQMGR
```

### Receiver

Receiver channel output can be viewed by starting an MQSeries listener for the receiving queue manager:

```
runmqlsr -m MYQMGR -t TCP -p 1415
```

## SETTING UP CLIENT CHANNELS

This exit can be used to verify certain types of client connection, which can be set up in a number of ways.

### Environment

The easiest to use of the client connection definitions is the environment variable **MQSERVER**. This defines the connection name, the queue

manager, and connection type, in the form:

```
export MQSERVER=ChannelName/TransportType/ConnectionName
```

This method does not allow specification of an exit so cannot be secured by an MQSeries security exit.

### Channel table

The next method is to use a client channel table. This is a table set up by an MQSeries queue manager and referenced by the client machine. The client will automatically look in the directory */var/mqm* for the file *AMQCLCHL.TAB*. If the client machine has the environment variable **MQCHLLIB** set, this defines the directory in which the client channel table should be located. Setting the environment variable **MQCHLTAB** defines the name of the client channel definition table. The default file name is *AMQCLCHL.TAB*.

To define a client connection (for example, to the queue manager *TSTQMGR* on host HOST1, port 1416, over the server connection channel *TSTQMGR.CLT1*) you would need to define on the server:

```
DEFINE CHANNEL(TSTQMGR.CLT1) CHLTYPE(CLNTCONN) +
TRPTYPE(TCP) DESCR('Client 1 connection')QMNAME(TESTQMGR)  +
SCYEXIT('MySecExit(MY_CHANNELEXIT_SDR)') +
SCYDATA('MySecData') CONNAME('HOST1(1416)'
```

This will add the client connection definition to the file channel table file located in the directory */var/mqm/qmgrs/{QUEUEMANAGERNAME}/@ipcc/amqclchl.tab*, in this case:

```
/var/mqm/qmgrs/TSTQMGR/@ipcc/amqclchl.tab
```

This file should then either be made accessible to the client, or copied onto the client's machine to a file name and directory matched by the above environment variables.

### MQCONNX

The third method of client connection is carried out programmatically, by setting up the required parameters to make a dynamic client connection. The **MQCONNX** call (and associated C++ version) allows the caller to specify an MQCD structure, which contains all the parameters required to connect to the server. If the source code can be

modified this method can be used by specifying the security exit in the MQCD structure **SecurityExit** and **SecurityUser** data elements.

### Java

For Java clients there are no MQSeries channel libraries on the client machine, but the security exit must be included in the client code. To provide your own security exit, define a class that implements the **MQSecurityExit** interface. Create a new instance of your class and assign the **MQEnvironment.securityExit** variable to it before constructing your MQQueueManager object.

## RELATED SECURITY ISSUES

### Channel auto-definition

In MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, Windows NT, and OS/390 (cluster-receiver and cluster-sender channels only), if there is no appropriate channel definition for a receiver or server-connection channel that has auto-definition enabled, a definition is created automatically. The definition is created using:

- The appropriate model channel definition:
  - **SYSTEM.AUTO.RECEIVER**
  - **SYSTEM.AUTO.SVRCONN**

  The model channel definitions for auto-definition are the same as the system defaults, **SYSTEM.DEF.RECEIVER** and **SYSTEM.DEF.SVRCONN**, except for the description field, which is 'Auto-defined by' followed by 49 blanks. The systems administrator can choose to change any part of the supplied model channel definitions.
- Information from the partner system. The partner's values are used for the channel name and the sequence number wrap value.
- A channel exit program, which you can use to alter the values created by the auto-definition.

The description is then checked to determine whether it has been altered by an auto-definition exit or because the model definition has

been changed. If the first 44 characters are still ‘Auto-defined by’ followed by 29 blanks, the queue manager name is added. If the final 20 characters are still all blanks, the local time and date are added. Once the definition has been created and stored the channel start proceeds as though the definition had always existed. The batch size, transmission size, and message size are negotiated with the partner.

This auto-definition allows for connection by remote users who have not been specifically set up by the receiving queue manager. There are two ways to manage this. Firstly, there is the option to disallow auto-definition of channels by setting the queue manager attribute for auto channel definition to disabled:

```
ALTER QMGR CHAD(DISABLED)
```

Alternatively, the definition templates can be altered so that any automatically-defined channels are defined to have the security exit defined. This can be achieved by the following **runmqsc** commands:

```
ALTER CHANNEL(SYSTEM.AUTO.RECEIVER)
      SCYEXIT(MySecExit(MY_CHANNELEXIT_RCV))
ALTER CHANNEL(SYSTEM.AUTO.SVRCONN)
      SCYEXIT(MySecExit(My_CHANNELEXIT_RCV))
```

The first of these options allows for stricter control of connections to the queue manager. The second allows simpler administration while still ensuring that new connections are controlled by the security exit.

### Usage examples

There are any number of ways that this security exit could be used to help authenticate an attempted channel connection. The IBM supportpac mentioned earlier shows how to implement a channel security exit with DCE security. It would be possible to implement much simpler custom solutions.

Here is a very brief list of suggestions:

- The sender could maintain its own id/password list and the receiver could authenticate them against its own set of valid users (held securely). This would allow the sending system to authenticate itself independently of the user-id of the application sending the messages.

- By checking the partner's channel parameters a channel exit can verify that the hostname or TCP/IP address is a valid partner.
- The sender could simply define a special passcode in the **SCYDATA** parameter on the channel definition. Obviously, this is a very basic check, but using cross-validation techniques can ensure that the partner channel at least has a matching channel security exit, which in itself is a (basic) form of authentication.
- On OS/390, systems user-ids are held in upper case: Unix systems administrators hate to use upper case and Windows administration systems don't care. The channel security exit could be used to match the case of the target system before running a native security check.

MYSECEXIT.C

```
/* Module Name: MySecExit                                          */
/* Description: Channel Security Exit                              */
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <cmqc.h>
#include <cmqxc.h>
typedef short BOOL;
#ifndef TRUE
#define FALSE ((BOOL)0)
#define TRUE ((BOOL)1)
#endif
enum { MSGT_INV_LOWER=100, MSGT_SENDER_ID, MSGT_RECEIVER_ID,
    MSGT_INV_HIGHER } MSGTYPES;
typedef struct
{
    MQBYTE  msgType;
    char    user-id[16];
    char    password[16];
    char    channelExitData[16];
} SECEXITDATA, MQPOINTER PSECEXITDATA;
/* Function Name: processSecurityMessage                           */
/* Function:Process a security message                             */
/* Input Parameters: pInMsgAddr                                    */
/*                   pInMsgAddr                                    */
/*                   inMsgLength                                   */
/* Output Parameters:None                                          */
/* Returns: TRUE (non-zero) if success                             */
```

```
/*                FALSE (zero) if security not validated          */
BOOL MQENTRY processSecurityMessage(BOOL isSender, PMQCXP pParms, PMQCD
                  pChDef, PMQVOID pInMsgAddr, MQLONG inMsgLength)
{
    PSECEXITDATA pSecData;
printf("Received message of length %d\n",inMsgLength );
    if (inMsgLength != sizeof(SECEXITDATA))
     return FALSE; /* The wrong data has been sent back - don't start */
pSecData = (PSECEXITDATA)pInMsgAddr;/* Map structure onto the message */
  if ( (pSecData->msgType == MSGT_RECEIVER_ID && !isSender) ||
       (pSecData->msgType == MSGT_SENDER_ID && isSender) ||
       (pSecData->msgType <= MSGT_INV_LOWER || pSecData->msgType >=
                            MSGT_INV_HIGHER) )
     return FALSE; /* We are talking to the same type of channel exit */
/* ——— Validate the pSecData->user-id and pSecData->password here ———*/
    printf("pSecData->user-id = %.16s\n",pSecData->user-id);
    printf("pSecData->password = %.16s\n",pSecData->password);
    printf("pSecData->channelExitData = %.16s\n",pSecData
                            >channelExitData);
    /* —— Validate the pSecData->user-id and pSecData->password here --
*/
/* ——— Validate the pChDef->ConnectionName (remote host) here -——*/
    printf("pChDef->ConnectionName = %.48s\n",pChDef->ConnectionName);
    printf("pChDef->ShortConnectionName = %.48s\n",pChDef
                            >ShortConnectionName);
    /* ———— Validate the pParms->PartnerName (remote host) here ————*/
return TRUE;
}
//* Function Name: sendSecurityInfo                                   */
/* Function: Send a security info message                            */
/* Input Parameters:  isSender                                       */
/*                    pExitBufferLength                              */
/* Output Parameters: None                                           */
/* InOut Parameters:  pParms                                         */
/* Returns: Pointer to allocated memory block                        */
PMQVOID sendSecurityInfo(BOOL isSender, PMQCXP pParms, PMQLONG
    pExitBufferLength)
{
    PSECEXITDATA pSecData;
pSecData = calloc(1, sizeof(SECEXITDATA));
    pSecData->msgType = isSender ? MSGT_SENDER_ID : MSGT_RECEIVER_ID;
/* ——— Populate the pSecData->user-id and pSecData->password here ——*/
    strcpy(pSecData->user-id, isSender ? "SenderUser-id":"ReceiverUser-
id");
    strcpy(pSecData->password, isSender ?
"SenderUserPwd":"ReceiverUserPwd");
    /* —— Populate the pSecData->user-id and pSecData->password here --
*/
/* ———— Use any of the channel parameters to pass across here ————*/
    strncpy(pSecData->channelExitData, pParms->ExitData,
                  sizeof(pSecData->channelExitData)-1 );
```

```
    /* ——— Use any of the channel parameters to pass across here ——*/
pParms->ExitResponse = isSender ? MQXCC_SEND_SEC_MSG :
                                    MQXCC_SEND_AND_REQUEST_SEC_MSG;
    pParms->ExitResponse2 = MQXR2_USE_EXIT_BUFFER;
    *pExitBufferLength = sizeof(SECEXITDATA);
    return pSecData;
}
/* Function Name: MyChannelExit                                        */
/* Function:                                                           */
/*       Main entry point for channel security exit                    */
/*       Check to see that this has been called as a security exit     */
/*       Then switch to whatever is defined by pParms->ExitReason      */
/* Input Parameters:  pAgentBufferLength    UNUSED                     */
/*                    isSender    Execute the sender-side function     */
/* Output Parameters: None                                             */
/* InOut Parameters:  pChannelExitParms Channel exit parameter block   */
/*                    pChannelDefinition   Channel definition          */
/*                    pDataLength          Length of data              */
/*                    pAgentBuffer         Agent    buffer             */
/*                    pExitBufferLength    Length of exit buffer       */
/*                    pExitBufferAddr      Address of exit buffer      */
/* Returns: None
#ifdef WIN32
__declspec( dllexport ) void MY_CHANNELEXIT_SDR();
__declspec( dllexport ) void MY_CHANNELEXIT_RCV();
#endif
void MyChannelExit(
    BOOL      isSender,      /* Label to determine if it is a sender*/
    PMQVOID   pChannelExitParms,  /* Channel exit parameter block   */
    PMQVOID   pChannelDefinition, /* Channel definition             */
    PMQLONG   pDataLength,        /* Length of data                 */
    PMQLONG   pAgentBufferLength, /* Length of agent buffer         */
    PMQVOID   pAgentBuffer,       /* Agent buffer                   */
    PMQLONG   pExitBufferLength,  /* Length of exit buffer          */
    PMQPTR    pExitBufferAddr)    /* Address of exit buffer         */
{
    PMQCXP    pParms = (PMQCXP)pChannelExitParms;
    PMQCD     pChDef = (PMQCD)pChannelDefinition;
if (pExitBufferAddr != NULL)    /* Free previously allocated buffer */
    {
        free(*pExitBufferAddr);
        *pExitBufferAddr = NULL;
        *pExitBufferLength = 0;
    }
/* Invoked as security exit? */
    if ( !(pParms->ExitId==MQXT_CHANNEL_SEC_EXIT) )
    {
        printf("Wrong exit: ExitId == %d\n", pParms->ExitId);
        pParms->ExitResponse = MQXCC_SUPPRESS_FUNCTION;
        return;
    }
```

```
    else
        printf("My MQSeries channel security exit entry\n");
       /* now switch to whatever function we were called to do     */
    pParms->ExitResponse = MQXCC_OK;
    switch ( pParms->ExitReason )
    {
        case MQXR_INIT:        /* Any initialization can be done here */
            printf("Exit reason = MQXR_INIT\n");
            break;
case MQXR_INIT_SEC: /* Initialize security sequence - receiver sends
                                                       first */
printf("%s: Exit reason = MQXR_INIT_SEC\n",
    isSender?"Sender":"Receiver");
            if (!isSender)
            {
*pExitBufferAddr = sendSecurityInfo(isSender, pParms,pExitBufferLength);
            *pDataLength = *pExitBufferLength;
            }
            break;
case MQXR_SEC_MSG:           /* Received a security message */
            printf("Exit reason = MQXR_SEC_MSG\n");
            if (!processSecurityMessage(isSender, pParms, pChDef,
pAgentBuffer,*pDataLength))
            {
                /* First part of security failed, close channel */
                pParms->ExitResponse = MQXCC_CLOSE_CHANNEL;
                break;
            }
            if (isSender)         /* Now the sender's turn to reply */
            {
*pExitBufferAddr = sendSecurityInfo(isSender, pParms,pExitBufferLength);
            *pDataLength = *pExitBufferLength;
            }
            break;
case MQXR_TERM:              /* Termination code goes here */
            printf("Exit reason = MQXR_TERM\n");
            break;
default:
            /* Unrecognized exit function */
            printf("Exit reason = Unrecognised\n");
            break;
    } /* endswitch */
}
/* Exported entry points to allow functions to be visible          */
void MQENTRY MY_CHANNELEXIT_SDR(
    PMQVOID    pChannelExitParms,  /* Channel exi parameter block   */
    PMQVOID    pChannelDefinition, /* Channel definition            */
    PMQLONG    pDataLength,        /* Length of data                */
    PMQLONG    pAgentBufferLength, /* Length of agent buffer        */
    PMQVOID    pAgentBuffer,       /* Agent buffer                  */
```

```
    PMQLONG    pExitBufferLength,  /* Length of exit buffer         */
    PMQPTR     pExitBufferAddr)    /* Address of exit buffer        */
{
   MyChannelExit(1,pChannelExitParms,pChannelDefinition,  pDataLength,
   pAgentBufferLength, pAgentBuffer,pExitBufferLength,pExitBufferAddr);
}
void MQENTRY MY_CHANNELEXIT_RCV(
    PMQVOID    pChannelExitParms,   /* Channel exi    parameter block*/
    PMQVOID    pChannelDefinition,  /* Channel definition            */
    PMQLONG    pDataLength,         /* Length of data                */
    PMQLONG    pAgentBufferLength,  /* Length of agent buffer        */
    PMQVOID    pAgentBuffer,        /* Agent buffer                  */
    PMQLONG    pExitBufferLength,   /* Length of exit buffer         */
    PMQPTR     pExitBufferAddr)     /* Address of exit buffer        */
{
    MyChannelExit(0,pChannelExitParms,pChannelDefinition, pDataLength,
    pAgentBufferLength, pAgentBuffer,pExitBufferLength,pExitBufferAddr);
}
/* Known entry point to allow it to be loadable on Unix      */
void MQStart(void) { ; }
```

*Chris Howarth*
*Senior Systems Engineer, CommerceQuest (UK)* 

## Architect your MQSeries environment on Unix

INTRODUCTION

As your MQ Unix environment grows, it is very important to have a common architecture. Naming standards and security form a significant part of setting up MQSeries, but establishing a common architecture is essential, too. It will make MQSeries administrators' and system administrators' jobs easier.

This article illustrates some of the choices I made in structuring our system and the reasoning behind them, and may serve as a useful guide for your own organization.

INSTALLATION GUIDE FOR SYSTEM ADMINISTRATORS

You probably want to create a common document for the AIX, Sun

 

Solaris, HP-UX, etc, system administrator. You will need to install MQSeries on a particular Unix platform, then you can send them a document that outlines the steps that will be needed not only to install MQSeries but to maintain it as well. For instance, detailed below is the likely progression for an HP-UX system administrator.

**MQSeries installation guide for HP-UX system administrators**

- Install the MQSeries for HP Server (Unix system administrative/ MQSeries administrative).
- See chapter three in *Quick Beginnings – MQSeries for HP-UX V5.1*
  - see *http://www-4.ibm.com/software/ts/mqseries/library/manualsa/amqcac/amqcac0s.htm#HDRAMQ3712*
  - except create at least 4 GB for */var/mqm/log*
  - except create at least 3 GB for */var/mqm*
  - except create at least 1 GB for */var/mqm/errors*
  - create */var/mqm/log* on a separate disk volume from */var/mqm*
  - install MQSeries patches (Unix system administrative)
  - install the latest patch for HP-UX V10 only. See *ftp://ftp.software.ibm.com/software/mqseries/fixes/hp51/v10/U469692/*
  - install the latest patch for HP-UX V11 only. See *ftp://ftp.software.ibm.com/software/mqseries/fixes/hp51/v11/U469693/*
  - see *readme.txt* for installing the patches.
- Create */mqadmin* directory (Unix system administrative)
  - create 2 GB */mqadmin* directory for user **mqm.**
- Authorize *mqm* to use cron (Unix system administrative)
  - authorize *mqm* to create/edit *crontab*.

- Kernel configuration (Unix system administrative)
  - the kernel configuration must be at the minimal kernel parameters (see Figure 1).
- Reboot production server (Unix system administrative)
  - Reboot HP server.
- Create S99mqm under */etc/rc2.d* for starting and shutting down MQSeries whenever the server is bounced (Unix system administrative). (Note: MQSeries administrator will give you the queue manager name.)

```
case "$1" in
'start')
        # Start MQSeries
        echo "starting audit daemon"
         su - mqm /mqadmin/util/mqm_startup.ksh  <Queue Manager Name> ;;
'stop')
        # Stop MQSeries
        su - mqm /mqadmin/util/mqm_shutdown.ksh <Queue Manager Name>  ;;
esac
```

| | |
|---|---|
| shmmax | 4194304 |
| shmseg | 1024 |
| shmmni | 1024 |
| shmem | 1 |
| sema | 1 |
| semaem | 16384 |
| semvmx | 32767 |
| semmns | 16384 |
| semmni | 1024 (semmni < semmns) |
| semmap | 1026 (semmni +2) |
| semmnu | 2048 |
| semume | 256 |
| msgmni | 50 |
| msgtql | 256 |
| msgmap | 258 (msgtql +2) |
| msgmax | 4096 |
| msgmnb | 4096 |
| msgssz | 8 |
| msgseg | 1024 |
| maxusers | 32 |

*Figure 1: Suggested kernel parameter values*

## SETTING UP COMMON DIRECTORIES FOR THE MQSERIES ADMINISTRATOR

It is important to separate your customized administration from the actual MQSeries software. Instead of putting your scripts, programs, configuration files, etc in */var/mqm* and */opt/mqm* or */usr/mqm*, you should put them under a separate directory. It is preferable to put them in a unique directory other than a home directory, such as */mqadmin* or */mqm* or */mqmuser*. The following list shows some possible sub-directories that you could use under your own directory:

- *bin*: binaries.
- *lib*: libraries/shared objects.
- *src*: sources/make files.
- *util*: utility scripts.
- *log*: logs.
- *config*: MQSeries administrator configurations.
- *exit*: for any exit routines.
- *bkup*: backups of MQSeries definitions (queue manager, queues, processes, channels, etc), *mqs.ini*, *qm.ini*, etc.
- *man*: man pages.

## CREATING COMMON UTILITIES FOR ALL UNIX PLATFORMS

You may want to create common start-up and shut-down scripts that can be re-used for all Unix platforms. These scripts will need to log any information to your */log* directory and you will need to pass a queue manager name. See the following start-up script:

### MQM_STARTUP.KSH

```
#    Description:    start-up queue manager, command server,
#                    channel init, channels
#    Last modified:  mm/dd/YYYY
#    Frequency:      on request and reboot of the server
#    Parameters:
#       MQ_MANAGER:  MQSeries Queue Manager
MQ_MANAGER=$1
```

```
MQ_LOGFILE=/mqadmin/log/'date "%Y%m%d"'.$MQ_MANAGER.log
MQ_CONFIG=/mqadmin/config
echo "############################################################ " >>
$MQ_LOGFILE
echo "#'date +%D %T' - Starting MQSeries " >> $MQ_LOGFILE
echo "############################################################ " >>
$MQ_LOGFILE
echo "Starting MQSeries QManager " $MQ_MANAGER >> $MQ_LOGFILE
strmqm $MQ_MANAGER >> $MQ_LOGFILE 2>> $MQ_LOGFILE
wait
strmqcsv $MQ_MANAGER >> $MQ_LOGFILE 2>> $MQ_LOGFILE
wait
echo "Starting MQSeries Channel Init & Channels " >> $MQ_LOGFILE
nohup runmqsc $MQ_MANAGER < /mqadmin/config/$MQ_MANAGER.start.channels
>> $MQ_LOGFILE 2>> $MQ_LOGFILE
echo "Starting MQSeries Listener " >> $MQ_LOGFILE
nohup runmqsc $MQ_MANAGER < /mqadmin/config/$MQ_MANAGER.start.listener
>> $MQ_LOGFILE 2>> $MQ_LOGFILE
echo "Re-set crontab " >> $MQ_LOGFILE 2>> $MQ_LOGFILE
crontab $MQ_CONFIG/crontab.config
exit Ø
```

The script above lists the following information as comments:

- Title.
- Author.
- Description.
- Last modified.
- Frequency.
- Parameters.

It helps the whole team in the long run.

If you review the document *MQSeries Installation Guide for System Administrator* you will see that this same script is listed under the */etc/rc2.d* for starting MQSeries at boot-up. If you have a need to modify the script, you have one place to change it, and then you can export this script to all the Unix servers.

Also, you will notice that the log files all start with *YYYYMMDD.<Queue Manager Name>.log*. You can create your own standard for log file names. In the case above, I have broken down the logs by queue manager and date/time.

Furthermore, this script includes unique configuration parameters for starting the channels and listener. It will pull in the */config/<Queue Manager Name>.start.channels* file and start the channels unique to that queue manager. Also, it will populate *crontab* for any MQSeries administrator scheduled jobs.

Some of the common scripts that I have created are: start-up, shut-down, back-up, clean-up, record image, purge syslogs, create queue manager, delete queue manager, etc.

## CREATING COMMON PROGRAMS FOR ALL UNIX PLATFORMS

You may want to create common programs, such as browsing the message on a particular queue and queue manager, saving queue manager, queues, processes, channels definitions to a file, channel exits, etc. These programs can be re-used for all Unix platforms except for the *make* files. You will have different *make* files for AIX, Sun Solaris, and HP-UX. You may want to create your binaries in your */bin* directory and your shared objects in your */lib*.

## CREATING UNIQUE CONFIGURATIONS FOR A PARTICULAR QUEUE MANAGER

You may want to create unique configurations for the queue manager, such as the names of channels to be started-up in a start-up script, or names of channels to be stopped in a shut-down script. You may want a copy of your queue manager definitions.

## BACK-UP CRITICAL FILES

It is very important to back-up critical files such as *mqs.ini*, *qm.ini*, queue manager's definitions, etc. You may have to back-out a change or research when a change was made, or you may want to create common back-up scripts that can be re-used for all Unix platforms.

See the following back-up script, which is scheduled to run daily, after midnight.

## MQM_BACKUP.KSH

```
Description:  backing-up queue manager configurations
```

```
#    Last modified:  mm/dd/YYYY
#    Frequency:      daily
#    Parameters:
#    MQ_MANAGER:     MQSeries Queue Manager
MQ_MANAGER=$1
MQ_LOGFILE=/mqadmin/log/'date "+%Y%m%d"'.$MQ_MANAGER.log
MQ_CONFIG=/mqadmin/config
MQ_BACKUP=/mqadmin/bkup
echo "############################################################ " >>
$MQ_LOGFILE
echo "#'date +%D %T' - Backing up Queue Manager Configurations " >>
$MQ_LOGFILE
echo "############################################################ " >>
$MQ_LOGFILE
cp $MQ_CONFIG/$MQ_MANAGER.config $MQ_BACKUP/$MQ_MANAGER.config.'date
+%Y%m%d'
cp $MQ_CONFIG/$MQ_MANAGER.DLQ.handler.rules $MQ_BACKUP/
$MQ_MANAGER.DLQ_handler.rules.'date +%Y%m%d'
cp $MQ_CONFIG/$MQ_MANAGER.start.channels $MQ_BACKUP/
$MQ_MANAGER.start.channels.'date +%Y%m%d'
cp $MQ_CONFIG/$MQ_MANAGER.start.listener $MQ_BACKUP/
$MQ_MANAGER.start.listener.'date +%Y%m%d'
cp $MQ_CONFIG/$MQ_MANAGER.stop.channels $MQ_BACKUP/
$MQ_MANAGER.stop.channels.'date +%Y%m%d'
cp $MQ_CONFIG/$MQ_MANAGER.security.config $MQ_BACKUP/
$MQ_MANAGER.stop.channels.'date +%Y%m%d'
/mqadmin/bin/mqm_saveqmgr -m $MQ_MANAGER -c > $MQ_CONFIG/
$MQ_MANAGER.config
cp /var/mqm/mqs.ini $MQ_BACKUP/$MQ_MANAGER.mqs.ini.'date +%Y%m%d'
cp /var/mqm/qmgrs/$MQ_MANAGER/qm.ini $MQ_BACKUP/$MQ_MANAGER.qm.ini.'date
+%Y%m%d'
exit Ø
```

You will notice that the back-up files all start with <*Queue Manager Name*>.<*File Name* >.*YYYYMMDD*. You may create your own standard for back-up file names.

## CLEANING FILES

It is very important to keep your system clean by removing any old logs, backups, *FDC, etc. See the following clean-up script, which is scheduled to run daily, after midnight:

## MQM_CLEANUP.KSH

```
# Description: Remove logs, errors, traces, back-ups, over five days old
# Last modified:  mm/dd/YYYY
# Frequency:   Daily at midnight
```

```
# Parameters:
# MQ_MANAGER:  MQSeries Queue Manager
MQ_MANAGER=$1
MQ_LOGFILE=/mqadmin/log/'date "+%Y%m%d"'.$MQ_MANAGER.log
echo "############################################################ " >>
$MQ_LOGFILE
echo "#'date +%D %T' - Cleaning up old logs, traces, errors, backups "
>> $MQ_LOGFILE
echo "############################################################ " >>
$MQ_LOGFILE
find /var/mqm/errors/AMQ[0-9]* -mtime +5 -print >> $MQ_LOGFILE -exec rm
-f {} \;
find /var/mqm/trace -mtime +5 -print >> $MQ_LOGFILE -exec rm -f {} \;
find /mqadmin/log -mtime +5 -print >> $MQ_LOGFILE -exec rm -f {} \;
find /mqadmin/bkup -mtime +5 -print >> $MQ_LOGFILE -exec rm -f {} \;
echo "# Done cleaning old logs, traces, errors, bkups " >> $MQ_LOGFILE
exit 0
```

INSTALLATION GUIDE FOR MQSERIES ADMINISTRATORS

In order to create a common document for the MQSeries administrator you will need to configure and maintain MQSeries. The following outlines the likely progression for an MQSeries administrator.

**MQSeries configuration guide for an MQSeries administrator**

- Update the *.profile* for user *mqm* (MQSeries administrative)
  - for Sun Solaris, update the *.profile* for the following information.

```
stty istrip
TERM=vt100; export TERM
EDITOR=/usr/bin/vi; export EDITOR
stty erase ^H
PATH=/opt/SUNWspro/bin:$PATH:/usr/local/bin:/usr/bin:/usr/sbin/:/
mqadmin/bin:/mqadmin/util; export PATH
MANPATH=/opt/SUNWspro/man:/usr/man:/mqadmin/man:$MANPATH; export MANPATH
set -o vi
HOST='uname -n'
PS1=$HOST'${PWD}->'
LD_LIBRARY_PATH=/opt/SUNWspro/lib:/$OPENWINHOME/lib:/opt/mqm/lib:/
mqadmin/lib:$LD_LIBRARY_PATH; export LD_LIBRARY_PATH
```

For HP-UX, update the *.profile* for the following information.

```
stty istrip
TERM=vt100; export TERM
EDITOR=/usr/bin/vi; export EDITOR
```

```
stty erase ^H
PATH=$PATH:/usr/local/bin:/usr/bin:/usr/sbin/:/mqadmin/bin:/mqadmin/
util; export PATH
MANPATH=/usr/man:/mqadmin/man:$MANPATH; export MANPATH
set -o vi
HOST='uname -n'
PS1=$HOST'${PWD}->'
```

- Create directory files under */mqadmin* (MQSeries administrative)
  - under */mqadmin* directory:

    ```
    mkdir bin lib src util log config exit bkup man
    chmod 775 *
    ```

  - *bin*: binaries
  - **lib**: libraries/shared objects
  - *src*: sources/make files
  - *util*: utility scripts
  - *log*: logs
  - *config*: MQ, crontab, stop/start, dead letter handler configurations
  - *exit*: for any exit routines
  - *bkup*: backups
  - *man*: man pages.
- Copy generic scripts to */mqadmin/util* (MQSeries administrative).

| | |
|---|---|
| **mqm_cleanup_ipcs.ksh** | Clean up IPCS resources for user *mqm*. |
| **mqm_cleanup.ksh** | Remove any sprint logs, MQ errors, MQ traces, sprint back-ups over five days old. |
| **mqm_startup.ksh** | Start-up queue manager, command server, channels. |
| **mqm_shutdown.ksh** | Shutdown channels, command server, queue manager. |

**mqm_record_image.ksh** Record images for linear logging.

**mqm_purge_syslogs.ksh** Purge MQSeries linear logs that are not needed any more.

**mqm_backup.ksh** Back-up MQ configurations.

**mqm_handle_DLQ.ksh** Handle the dead letter queue.

**mqm_expired_message.ksh**
Handle expired messages.

**mqm_create_qmanager.ksh**
Create queue manager.

**mqm_delete_qmanager.ksh**
Delete queue manager.

**mqm_set_security.ksh** Set security.

**mqm_start_channel.ksh** Start channel.

**mqm_stop_channel.ksh** Stop channel.

**mqm_message_performance.ksh**
Application message performance.

**mqm_transfer_message.ksh**
Transfer message from queue to another.

- Copy generic configurations to */mqadmin/config* and upgrade the configurations for the specific queue manager (MQSeries administrative).

**crontab.config** Use to populate the *crontab* if the queue manager is up and running.

**crontab.empty.config** Use to populate the *crontab* if the queue manager is down.

**<Queue Manager>.start.channel**
Use to start the channel initiator and channels.

**<Queue Manager>.stop.channel**
Use to stop the channels.

**<Queue Manager>.DLQ.handler.rules**
Use to handle messages on the dead letter queue.

**<Queue Manager>.config**
A copy of queue manager configurations.

**<Queue Manager>.start.listener**
Use to start the listener.

- Copy generic sources/*make* files to */mqadmin/src* (MQSeries administrative) and make the binaries in */mqadmin/bin.*

**mqm_expired.c, mqm_expired.mak**
Handle expired messages.

**mqm_get_message.c, mqm_get_message.mak**
Get messages from the queue.

**mqm_put_message.c, mqm_put_message.mak**
Put messages from the queue.

**mqm_browse_message.c, mqm_browse_message.mak**
Browse messages from the queue.

**mqm_trigger_monitor.c, mqm_trigger_monitor.mak**
Trigger monitor.

**mqm_channel.c, mqm_channel.mak**
Save channel configurations.

**mqm_evmon.c, mqm_evmon.mak**
Event monitoring.

**mqm_inquire_queue.c, mqm_inquire_queue.mak**
Inquire queue information.

**mqm_message_performance.c, mqm_message_performance.mak**
Application message performance.

**mqm_namelist.c, mqm_namelist.mak**
Save namelist configurations.

**mqm_process.c, mqm_process.mak**
Save process configurations.

**mqm_qmgr.c, mqm_qmgr.mak**
Save queue manager configurations.

**mqm_queue.c, mqm_queue.mak**
Save queue configurations.

**mqm_saveqmgr.c, mqm_saveqmgr.mak**
Save all queue manager configurations.

**mqm_transfer_message.c, mqm_transfer_message.mak**
Transfer message from one queue to another.

**mqm_utils.c, mqm_utils.mak**
Utilities.

- Create the queue manager (MQSeries administrative).

  Note: it will create a log entry under */mqadmin/log* directory.

```
Execute  mqm_create_qmanager.ksh <Queue Manager>
```

- Update *mqs.ini* file in */var/mqm* (MQSeries administrative).

```
Update mqs.ini:
     DefaultQueueManager:
          Name=<Queue Manager>
```

- Update *qm.ini* file in */var/mqm/qmgrs/<QMgr Name>* (MQSeries administrative).

```
Update qm.ini:
     Log:
          LogBufferPages=32
```

- Start the queue manager (MQSeries administrative).

  Note: it will create a log entry under /mqadmin/log directory.

```
Execute mqm_startup..ksh <Queue Manager>
```

- Create man pages for every script and binaries (MQSeries administrative).

## CONCLUSION

The environment architecture outlined in this article is just an example,

and you may want to create a different one, more appropriate for your own organization, but there is no doubt that having a standard environment architecture will make the MQSeries administrator's job much easier. I have used this architecture for over a hundred Unix servers, and it saves a condiserable amount of time, not to mention headaches!

*Roberta M Carroll*
*Systems and Software Support, Sprint (USA)* 

# Clustered queue managers

INTRODUCTION

Today, most types of system can be grouped together to form a cluster, the objective being to provide increased availability and, sometimes, increased throughput as well. This is true for MQSeries queue managers, and I will describe three approaches that may be adopted when aiming to achieve these objectives and discuss the merits of each. The three methods for clustering queue managers are:

- Built-in MQSeries clustering support, introduced with V5.1 (distributed products) and V2.1 (OS/390).
- Shared queue support, introduced in V5.2 of MQSeries for OS/390.
- Independent standby management software.

BUILT-IN MQSERIES CLUSTERING SUPPORT

This is a standard feature of MQSeries, introduced with V5.1 (distributed products) and V2.1 (OS/390). The basic idea is that a group of queue managers advertise the existence of some of their channels and queues to other members of the cluster. There is no command to create the cluster, the cluster is formed by including the queues and channels belonging to the various queue managers.

The improvements in availability and increased throughput come

 

from being able to define equivalent queues, with identical names, on several of the queue managers in the cluster. A message **PUT** on one queue manager may then be moved to any one of these clustered queues and be processed (see Figure 1). All systems in the diagram are queue managers; the system where the message is put is a queue manager, or an MQSeries client attached to a queue manager. If one of the queue managers or communication to it fails, it is excluded from the choice of destinations for the messages.

Besides the increased availability and throughput, the administration needed to maintain the definitions may actually be less than when MQSeries clustering is not used. The reason is that each queue or

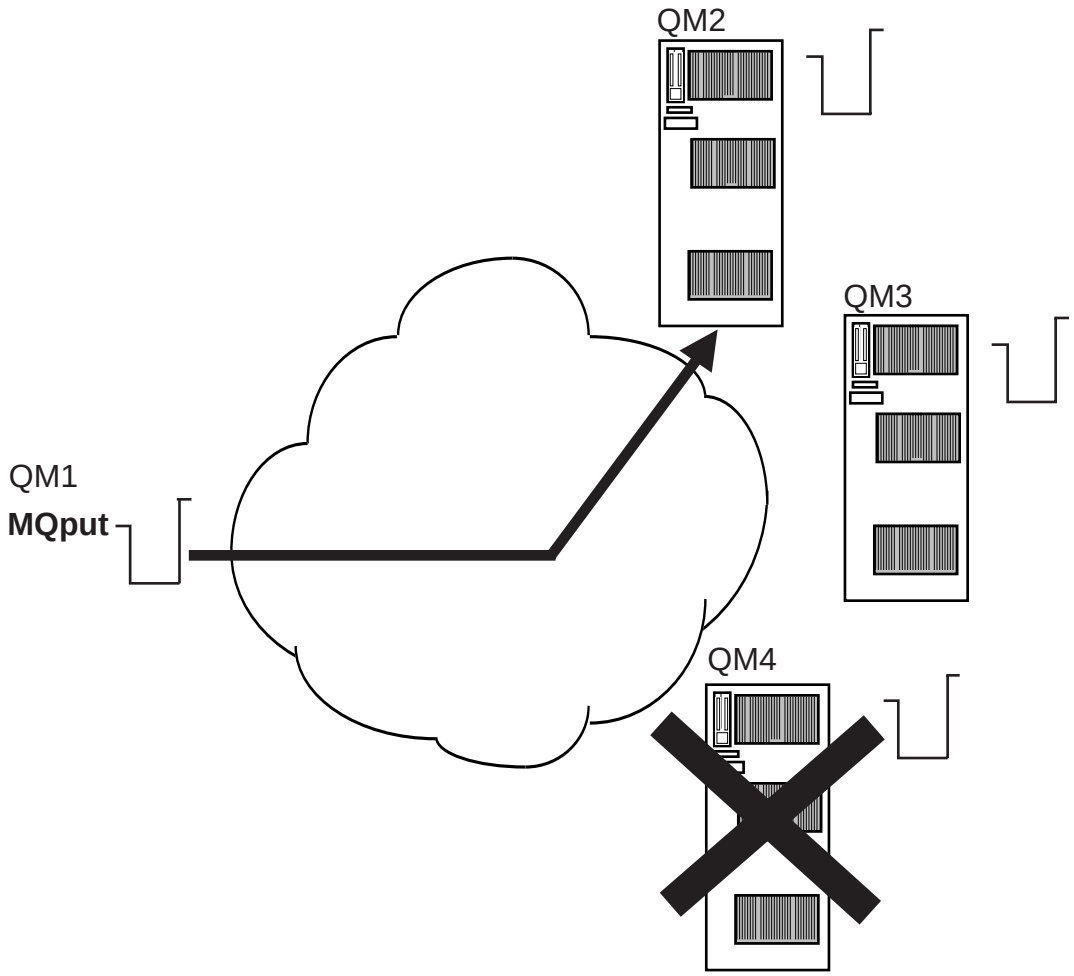


*Figure 1: Built-in MQ cluster support*

channel is administered only on the queue manager that hosts it, and the effects of any changes are advertised to the other members of the cluster. The amount of administration increases linearly in proportion to the number of queue managers on the cluster, and avoids the combinatorial explosion that might happen as more and more queue managers are added. This is a major benefit of MQSeries clustering, even if there is no immediate need to exploit it for availability and scalability reasons.

There is almost no performance overhead for using this form of clustering. Each queue manager in the cluster detects its own ability to reach the others by trying to send application messages. Cluster information itself flows when a definition change is made. This means that clusters built this way can be very large, perhaps consisting of thousands of queue managers.

Failure of a network or the destination queue manager manifests itself as a channel failure, with the channel performing its normal retry cycle. At the same time, any messages waiting on the *SYSTEM.CLUSTER.TRANSMIT.QUEUE* intended to move along the failed channel will be reprocessed and another destination found, if this is appropriate. The channels between each pair of queue managers in the cluster are protected independently, without any global monitoring of the cluster as a whole, and because there is no controlling agent these clusters are very robust.

By tuning the value of **HBINT** on the **CLUSRCVR** channels the detection time for channel failures can be made short, and the delay in re-routing a message to an already active alternate queue manager is also short, making it feasible to detect and recover from failures without the failure being noticeable to interactive users. Making the channel failure detection times too small leads to false failures, where a slow-down or a delay in network traffic is treated like a failure.

Applications need to be examined carefully to exploit MQSeries clustering fully. Message affinity is one such pitfall. This is where a sequence of messages must be processed in the same queue manager, usually because the application depends on some transient state that the earlier messages in the sequence create. Fortunately, it is not particularly common, and MQSeries provides a straightforward way to deal with it. The administrator or application writer can specify that

all messages using an **OPEN** handle must go to the same queue instance, by specifying **BIND_ON_OPEN**. This cures the affinity problem but means that any failure that occurs after the **OPEN** handle has been created will delay the processing of later messages until the failure is corrected.

If a request message reaches its destination queue manager, or at least gets in doubt on the channel to the destination, and then the destination queue manager fails, MQSeries clustering will not be able to recover the message (because of the possibility of duplicating it) without recovering the destination queue manager itself. If it is important to protect against this type of 'marooned message' problem, then one of the following techniques needs to be used in conjunction with MQSeries clustering.

**OS/390 shared queue support**

Shared queue support exploits the Coupling Facility hardware running under OS/390 and was introduced as a standard part of MQSeries for OS/390 V5.2. Messages reside in queues stored in the shared Coupling Facility. All the queue managers that are in a queue-sharing group and have access to the Coupling Facility can **GET** and **PUT** messages to these queues, as illustrated in Figure 2. In the current release, the messages must be non-persistent and less than 63 KB in size.

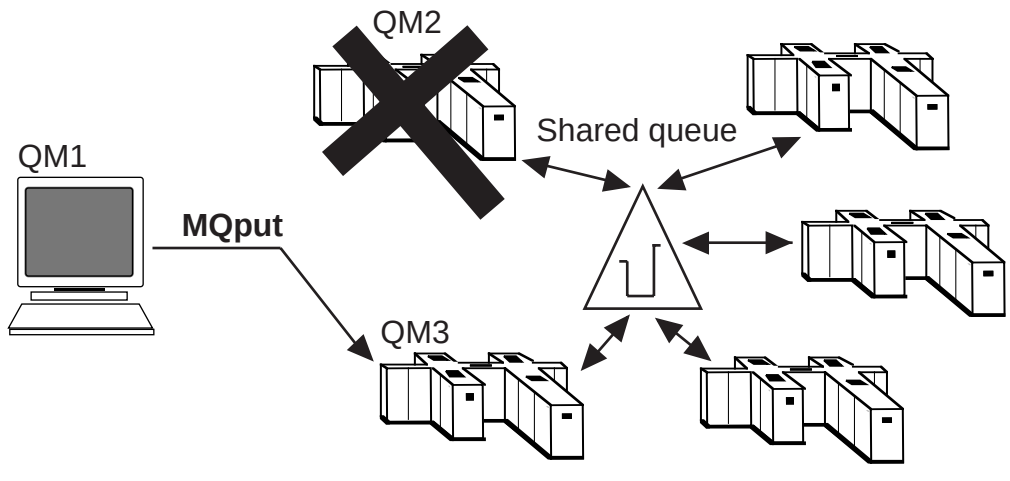


*Figure 2: MVS shared queue*

Availability and scalability derive from the ability of any of the group of queue managers to process a message. A client-driven request reply using **PUT** and **GET** operations typically takes between 10 and 30 per cent more CPU than for the same operation to a local queue.

Queue managers in a queue-sharing group can be administered as a group, with queues being defined as shared and the definition applied to the group by including the **CMDSCOPE** keyword. Distributed queue managers can send messages to queues in the group, rather than the individual queue managers. A channel defined in the group will be attached by VTAM or dynamic DNS to one of the queue managers in the group. If an instance of a channel fails and restarts it may be attached to a different queue manager in the group.

While the Coupling Facility itself is extremely reliable, if it is stopped, or it fails, then all the messages within it at that time are lost, hence the restriction to non-persistent messages. Even though a non-persistent message in the Coupling Facility may be relatively safe, it is still a non-persistent message, and when moved outside the sysplex retains its non-persistent characteristics. For instance, if it happens to be in a distributed queue manager waiting to be moved to a sysplex, it will be lost when the distributed queue manager shuts down. This same restriction means that transmission queues holding messages to be moved outside the sysplex should not be defined in the Coupling Facility because they could contain non-persistent messages.

All the queue managers in a queue-sharing group take messages, in order, from a shared queue when a **GET** request is executed. This makes a self-balancing form of workload management known as 'pull-workload balancing'. The system that is able to execute the most **GET** commands processes the most messages.

As with MQSeries clustering, applications need to be capable of working in this environment. Message affinity is still an important consideration because there is no guarantee that a particular queue manager will process a given message.

## INDEPENDENT STANDBY MANAGEMENT SOFTWARE

Most systems can be managed by a failover product, examples of which are listed in Table 1. Generally, they allow a number of

| Operating system | Failover product | MQSeries support |
|---|---|---|
| OS/390 | Automatic Restart Manager | http://www-4.ibm.com/software/ts/mqseries/library/manualsa/csqsaw00/csqsaw00tfrm.htm (Chapter 13) |
| Microsoft Windows NT | Microsoft Cluster Server (MSCS) * | http://www-4.ibm.com/software/ts/mqseries/txppacs/mc74.html |
| AIX | HACMP Version 4.3.1 or 4.4.0 or HACMP/ES Version 4.4.0.3* | http://www-4.ibm.com/software/ts/mqseries/txppacs/mc63.html |
| Solaris | Sun Cluster 2.1 or 2.2 * | http://www-4.ibm.com/software/ts/mqseries/txppacs/mc69.html |
| Compaq | Compaq TruCluster Available Server V1.6.* | http://www-4.ibm.com/software/ts/mqseries/txppacs/mc68.html |
| HP-UX | HP Service Guard 10.05* | http://www-4.ibm.com/software/ts/mqseries/txppacs/mc66.html |

* Provides network address take-over

*Table 1: Failover products*

processors to be attached to a shared network and a shared disk, as illustrated in Figure 3. These clusters provide failure protection but do not allow any increase in throughput beyond that which can be obtained from a single system. The overhead of running the failover software is usually negligible.

The systems are configured so that a queue manager runs on one of the systems and the messages and the transaction log are written to the shared disk. All of the systems monitor each other, usually by sending a request, conceptually asking the question ‘are you dead?’. If no response is received the system is assumed to have failed. The failover software must then make sure that the failed queue manager cannot

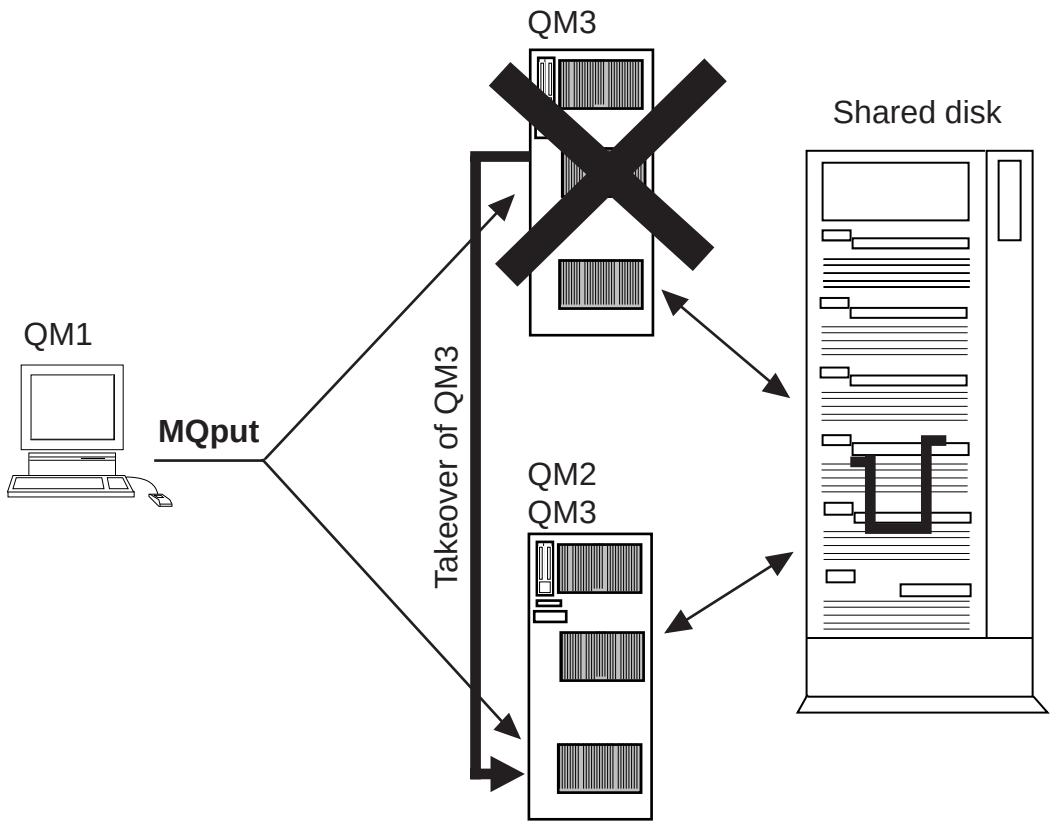


*Figure 3: Queue manager failover*

restart and runs a script or program on a standby processor to restart the queue manager there. On the standby processor, the queue manager reads the transaction log and recovers to the point where the failure occurred.

Many of the failover products provide an additional feature, to take over a floating IP address. The address is associated with the queue manager currently running, and means that, when the failure occurs, there is no need to change the configuration of other systems to reflect that the take-over has happened.

If this facility is not provided, the failover script can alter the IP address in the **CONNAME** field of a **CLUSRCVR** in the failing system to advertise the change to an MQSeries cluster. This technique is used by the ARM support provided with MQSeries for OS/390.

These systems require the provision of access to shared disks, which must themselves be highly available. In the other clustering techniques normal operation utilizes all of the queues and channels – to some degree, they are self-testing. All highly-available systems need frequent testing, which is crucial in a failover cluster as the standby system may otherwise have long periods of no usage, and fail itself at the very moment it is most needed.

These cold standby clusters do not suffer from the marooned message problems described earlier because the messages are still accessible on the shared disk. If IP address take-over is used, or the **CONNAME** is altered, 'in-doubt' messages can also be recovered when the communications restart to the new system.

Take-over times are usually too long to be invisible to interactive systems, the detection time can be significant because the failover support has to be certain that the failing system has permanently failed. Restarting the queue manager on the alternate system will require reprocessing of the transaction log. In the situation where the queue manager fails but the operating system is still viable, the failover scripts can ensure that the failed queue manager does not restart and speed up that step.

## AND FINALLY

None of the three forms of clustering mentioned above protect against an errant application. One insidious problem that can arise is where a server application is apparently functioning well, taking messages from a queue, and providing responses. However, if the responses are just indicating that the request cannot be processed because a database is unavailable, for instance, the system is unaware that anything is wrong.

This situation is often referred to as a 'storm drain' problem and has to be dealt with by the application code itself, by taking itself off-line, perhaps.

The principal features of each form of clustering are summarized in Table 2. Built-in clustering provides good protection for the collection of queue managers and its network, especially for interactive workloads. Shared queue and independent standby management enable recovery

| Feature | Built-in MQSeries clustering | OS/390 shared queue | Independent standby management |
|---|---|---|---|
| Maximum number of systems | Over 1000 | 32 queue mgrs per queue-sharing group | Usually 2-8 |
| Geographic limitation | None, wide area network | Connection to Coupling Facility, maximum 27Km | Connection to shared disk, usually same machine room, but can be further |
| Usual time to recover | Seconds | Sub-second | Minutes |
| Types of system | Heterogeneous | OS/390 | Homogeneous |
| Failover of network links | Yes | No | No |
| Recovery of previously delivered messages | No | Yes | Yes |
| Scalable throughput | Yes | Yes | No |
| Applications must be free from message affinity to exploit scalability | Yes, unless **BIND_ON_OPEN** is used | Yes | Not applicable |

*Table 2: Principal features of each clustering option*

of messages once they have been delivered to the target queue manager with shared queue allowing faster recovery, but only if OS/390 and non-persistent messages can be used.

*Andrew Banks*
*MQSeries Development, IBM (UK)*

# MQ news

Talarian has recently announced the availability of its Talarian SmartMQ 2.0 message queueing product. SmartMQ is one of the components designed to deliver a unified messaging environment when combined with the publish-subscribe architecture of Talarian's SmartSockets product. SmartMQ provides data persistence for messaging transactions.

The company claims that SmartMQ 2.0 will provide users with significantly increased performance, a new browser-based interface for easier administration and configuration, accessibility via the Internet, and full compatibility with MQSeries, allowing conversion between SmartSockets and MQSeries.

SmartSockets, Talarian's flagship product, is an infrastructure solution that is claimed to enable processes to communicate quickly, reliably, and securely across multiple operating systems and platforms.

*For further information contact:*
Talarian, 333 Distel Circle, Los Altos, CA 94022-1404, USA
Tel: +1 650 965 8050
Fax: +1 650 965 9077
Web: http://www.talarian.com

Talarian, 68 Lombard Street, London, EC3V 9LJ, UK
Tel: +44 20 7868 1630
Fax: +44 20 7868 1752

* * *

Compuware has started shipping the E-Business Edition of its Abend-AID fault management tool, designed to help save time in test and production environments and speed the integration of legacy systems and e-business applications.

It provides developers with diagnostic information that helps pinpoint problems and suggests corrective actions to resolve those problems.

It's said to function as "built-in expertise" for programmers for MQSeries in batch, IMS, and CICS environments. It enables programmers and developers to detect, analyse, and diagnose problems in applications that use MQSeries CICS Web Interface.

It maps out MQSeries so developers can identify any errors that might occur on those applications being integrated with MQSeries.

*For further information contact:*
Compuware, 31440 Northwestern Highway, Farmington Hills, MI 48334-2564, USA
Tel: +1 248 737 7300.
Web: http://www.compuware.com

Compuware, 163 Bath Road, Slough, Berkshire, SL1 4AA, UK
Tel: +44 1753 444000
Fax: +44 1753 444 900

* * *