27

MQ update

September 2001

In this issue

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*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at www.xephon.com/mqupdate.html (you will need to supply a word from the printed issue).

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 ($33.75) each including postage.

Printed in England.

How to include a local cluster queue in workload balancing

The problem of how to include a local cluster queue in workload balancing is of interest to many people. Figure 1 illustrates the difficulties.

Figure 1 shows that three queue managers have been connected in a single MQSeries cluster. The queue 'Fred' has been defined on each queue manager as a cluster queue with its 'default bind' option set to 'not fixed'. As we know, this means that the workload exit of the sending queue manager will distribute messages equally amongst the available queue managers. If Application 1, for example, is connected to queue manager QM1, then QM1's workload exit will try to send its messages to QM2 and QM3 in a round-robin fashion.

But … there is a big proviso, namely that, if there is a local instance of the cluster queue (as shown), the queue manager will select that queue, and the workload exit does not even get called. In fact, the *IBM Queue Manager Clusters* manual states this in its *Programming Considerations*, as reproduced here:

"If an application opens a target queue for output, the **MQOPEN** call chooses between all available instances of the queue. If there is a local

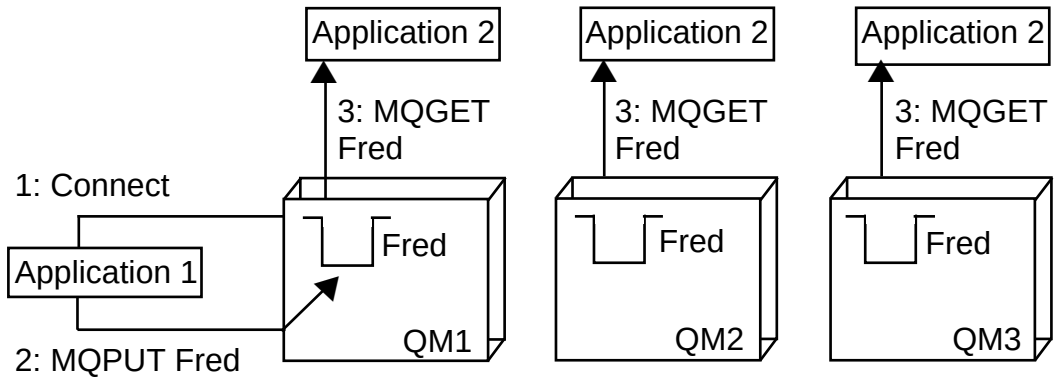

Figure 1: Difficulties in including a local cluster queue in workload balancing

version of the queue this is chosen in preference to the other instances. This may limit the ability of your applications to exploit clustering."

So how do we get round this problem? The solution is to use the old favourite – the 'Alias' queue – and carry out the following steps.

1 Define a new queue manager – QM4 – whose sole purpose is to re-route messages (or, if you already have a queue manager that doesn't host the cluster queue, use that one).

2 On QM4, define a QAlias – let's say 'Pete' – *as part of the cluster*, whose base or target queue is the original cluster queue 'Fred'.

```
DEFINE QALIAS(Pete) TARGQ(Fred) Cluster(CLUSTEST) DEFBIND(NOTFIXED)
```

Note that if you accidentally leave off the *DEFBIND* parameter it defaults to 'OPEN', and will result in a reason code *x'822'/2082* meaning *MQRC_UNKNOWN_ALIAS_BASE_Q*.

3 Alter Application 1 to MQPUT to the new alias queue 'Pete'.

Figure 2 illustrates how the cluster looks now.

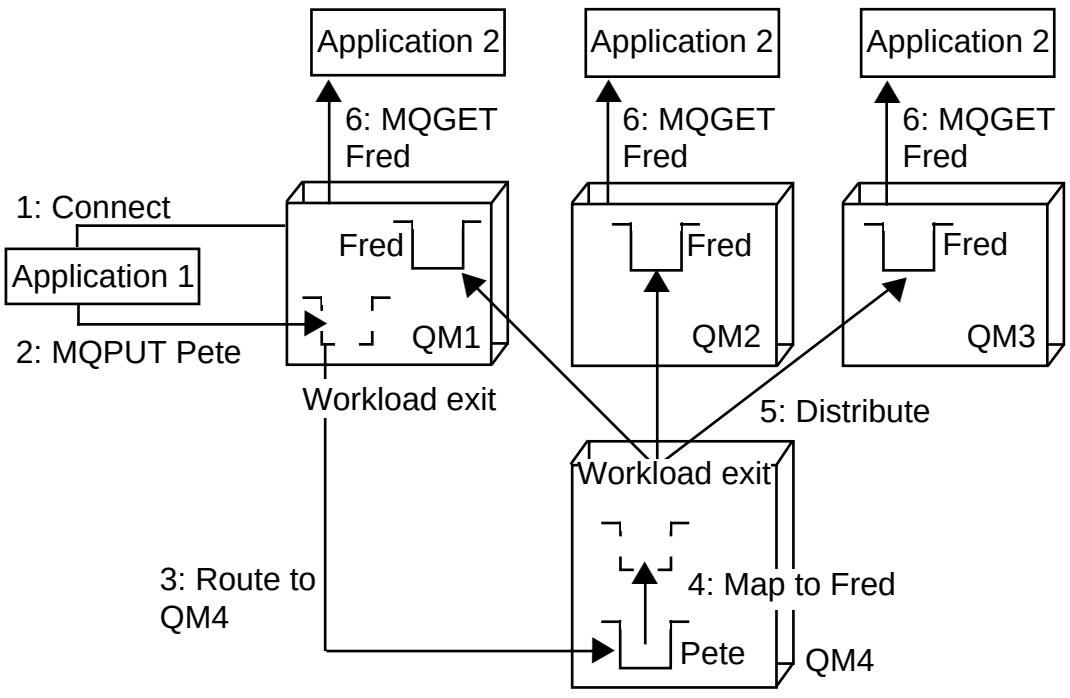

Figure 2: Using an 'Alias' queue to resolve the problem

All four queue managers could reside on different platforms or, in the case of OS/390, on different LPARs. The queue drawn with a dotted line is shown for reference only, and is not defined on that particular queue manager.

This particular configuration has been tested on MQSeries for Windows NT/2000 V5.1 at CSD6, as well as MQSeries for OS/390 V2.1, and should work on any of the platforms that support clustering. Please make sure you have all the latest maintenance applied.

If you feel that the 'router' queue manager could be a single point of failure then you could define more queue managers, as Figure 3 illustrates.

Although this diagram looks quite busy, remember that all these queue managers could reside on different machines, platforms, or LPARs.

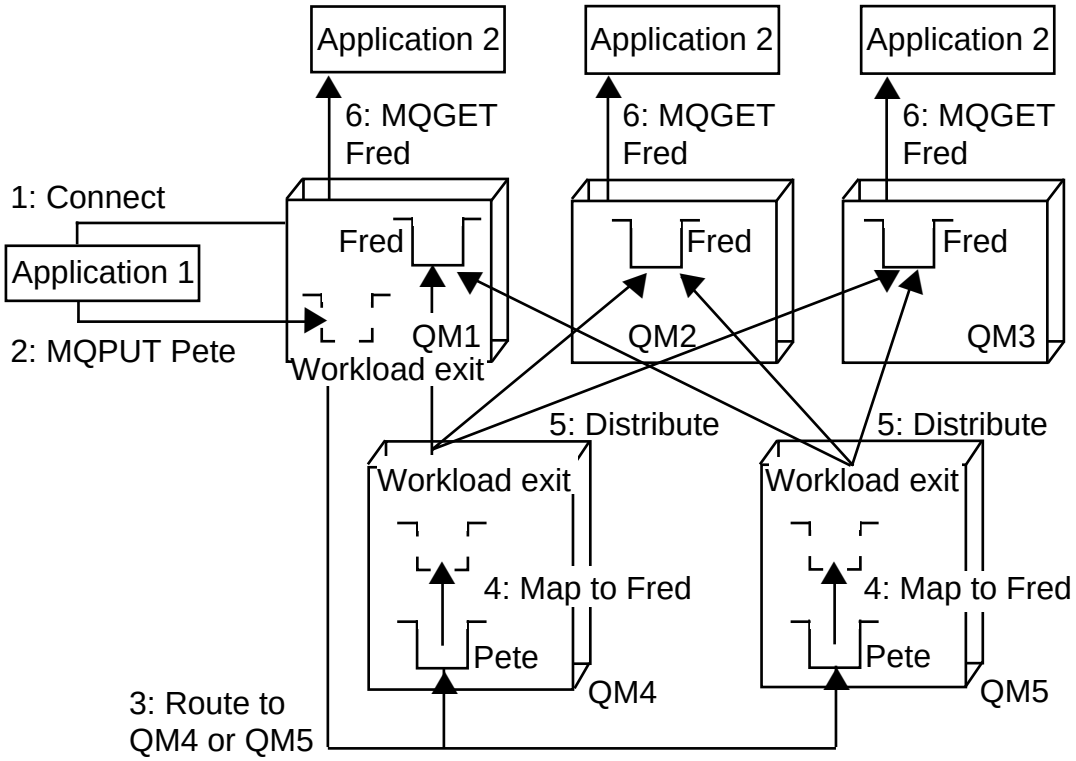

Figure 3: Avoiding a single point of failure

MQSI V2 exception processing (part 1)

INTRODUCTION

MQSI V2 exception processing lacks sufficient IBM support documentation, except for a section within the Appendix on ESQL reference.

This article serves as a starting point for anyone wanting to know more about implementing exception processing in an MQSI message flow. It can be elaborated upon for individual MQSI shops, as each shop will probably have different requirements for exception processing.

TO BEGIN

When an MQSI exception condition occurs message processing is suspended and an exception is thrown. Control is passed back to a higher level, that is, an enclosing catch block. An exception list is built to describe the failure condition and then the whole message, together with the destination list and the newly-populated exception list, is propagated through an exception-handling message flow path.

Exception handling paths are no different in principle from a normal message flow path, but they start at:

- A failure terminal (most message processing nodes have these).
- The catch terminal of an MQInput node or the catch terminal of a TryCatch node.

Normal message flows consist of a set of interconnected message flow nodes defined by the message flow designer. The exception handling paths differ in detail; for example, they might examine the exception list to determine the nature of the error and so be able to make an appropriate response.

The simplest way to implement exception processing of a message flow is to connect the failure terminals of the IBM node in your message flow to a failure queue. In this way, when exceptions happen, messages detailing the problem(s) will be captured in the failure

queue. These messages can then be re-routed to the message flow in the development environment in order to determine the cause of the problem.

Figure 1 illustrates a typical message flow with failure terminals connected to an output failure queue.

An MQ monitoring tool can also be set to monitor the failure queue and trigger some action if the 'curdepth' (current depth) of such a failure queue is not zero.

It may also be helpful to connect the different failure terminals to different failure queues instead of connecting all the failure terminals to the same failure queue. This enables us to determine the actual error by the presence of a message in a particular failure queue; for example, if it's in the failure queue connected to the compute node that does the transformation of CWF to XML format, we will know that the problem is within the message transformation section.

One setback of such an approach is that there will be a lot of queues associated with the message flow. As more projects buy into using MQSI, the growth in the number of queues associated with message flows will also be significant. This may create a problem with some shops that treat queues as overheads that take up space. Furthermore, it may make administration or maintenance of the system more difficult.

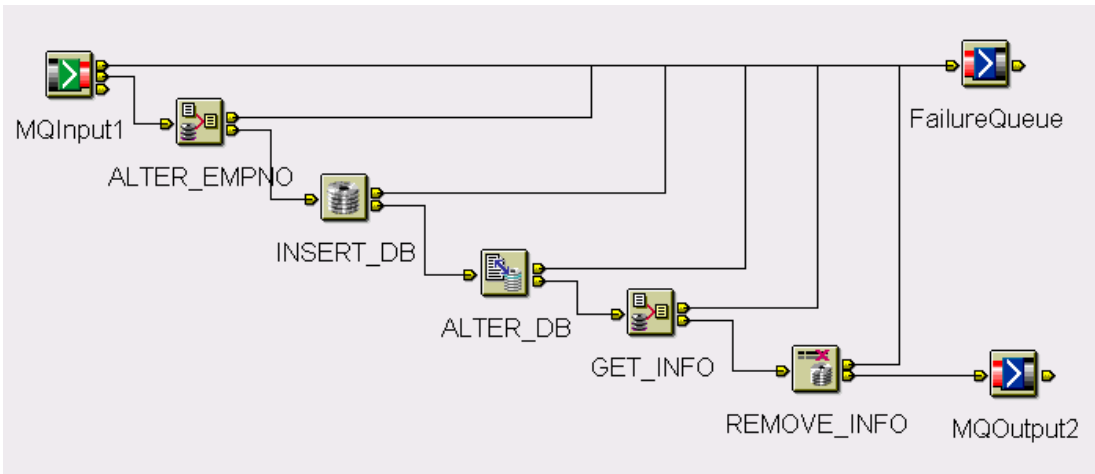

Figure 1: A typical message flow

THE QUIRK

Once the failure terminal of an IBM node is connected to an output queue MQSI makes the assumption that the exception processing will then be handled by the developer on their message flow. MQSI will then no longer issue error logs to the NT event logs (or the SYSLOG if the broker is hosted in a Unix environment). Many MQSI developers only become aware of this feature after working with MQSI for some time, because, as mentioned earlier, the MQSI documentation is limited. The only place where it is documented is on the *SupportPac IHI1, MQSI Version 2.1 problem determination, version 1.3*, as reproduced below.

"Answer 4.3b: if the 'out' terminal of the MQInput node is correctly wired up, check in the local error log for the broker for a message indicating that message processing has been terminated due to problems. Additional messages will give more detailed information.

"Note: if the failure terminal of the MQInput node has been wired to an MQOutput node, for example, these messages will not appear. Wiring a node to a failure terminal of any node indicates that you have designed the message flow to deal with all error processing. If you connect a failure terminal to an MQOutput node your message flow ignores any errors that occur."

THE QUICK FIX

To fix the quirk we can connect the failure terminal to a trace node first, before wiring it to the MQOutput failure queue. In the trace node we specify the trace node to print out a trace of the *ExceptionList* of the message (see Figure 2).

```
**** ExceptionList generate by messageflow exceptionhandlingl
**** exception generated at ${CURRENT_TIMESTAMP}
${ExceptionList}
**** Message Properties is
${Properties}
**** Message descriptor is
${Root.MQMD}
```

The following is a sample output of the trace file captured with *ExceptionList* trace.

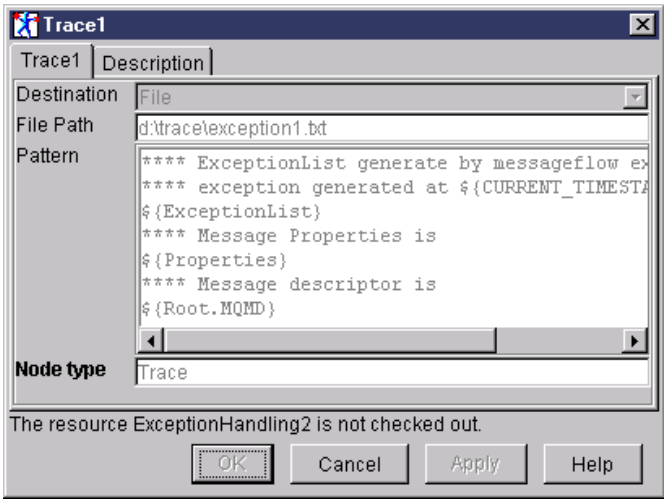

Figure 2: Specifying where to print a trace of the message ExceptionList

SAMPLE OUTPUT

```
**** ExceptionList generate by messageflow exceptionhandlingl
**** exception generated at 2001-06-16 17:26:38.809001
(
  (0x1000000)RecoverableException = (
    (0x3000000)File                 = 'F:/build/S201_P/src/
DataFlowEngine/ImbDataFlowNode.cpp'
    (0x3000000)Line                 = 538
    (0x3000000)Function             =
'ImbDataFlowNode::createExceptionList'
    (0x3000000)Type                 = 'ComIbmMQInputNode'
    (0x3000000)Name                 = 'b0ab6922-e700-0000-0080-
ca4d18331a10'
    (0x3000000)Label                = 'ExceptionHandling1.MQInput1'
    (0x3000000)Text                 = 'Node throwing exception'
    (0x3000000)Catalog              = 'MQSIv201'
    (0x3000000)Severity             = 3
    (0x3000000)Number               = 2230
    (0x1000000)RecoverableException = (
      (0x3000000)File            = 'F:/build/S201_P/src/DataFlowEngine/
ImbComputeNode.cpp'
      (0x3000000)Line               = 354
      (0x3000000)Function           = 'ImbComputeNode::evaluate'
      (0x3000000)Type               = 'ComIbmComputeNode'
      (0x3000000)Name               = 'b9d46922-e700-0000-0080-
ca4d18331a10'
      (0x3000000)Label              = 'ExceptionHandling1.Compute1'
      (0x3000000)Text               = 'Caught exception and rethrowing'
      (0x3000000)Catalog            = 'MQSIv201'
```

```
      (0x3000000)Severity           = 3
      (0x3000000)Number             = 2230
      (0x1000000)ParserException    = (
        (0x3000000)File             = 'F:/build/S201_P/src/MTI/
MTIforBroker/GenXmlParser/XmlImbParser.cpp'
        (0x3000000)Line             = 803
        (0x3000000)Function         = 'XmlImbParser::parseFirstChild'
        (0x3000000)Type             = ''
        (0x3000000)Name             = ''
        (0x3000000)Label            = ''
        (0x3000000)Text             = 'XML Parsing Errors have occurred'
        (0x3000000)Catalog          = 'MQSIv201'
        (0x3000000)Severity         = 3
        (0x3000000)Number           = 5009
        (0x1000000)ParserException  = (
          (0x3000000)File     = 'F:/build/S201_P/src/MTI/MTIforBroker/
GenXmlParser/XmlBrokerAsgardParser.cpp'
          (0x3000000)Line     = 1995
          (0x3000000)Function = 'XmlBrokerAsgardParser::error'
          (0x3000000)Type     = ''
          (0x3000000)Name     = ''
          (0x3000000)Label    = ''
          (0x3000000)Text     = 'An error has been reported by the
     BIPXML4C component.'
          (0x3000000)Catalog  = 'MQSIv201'
          (0x3000000)Severity = 3
          (0x3000000)Number   = 5004
          (0x1000000)Insert   = (
            (0x3000000)Type   = 2
            (0x3000000)Text   = '65551'
          )
          (0x1000000)Insert   = (
            (0x3000000)Type   = 5
            (0x3000000)Text   = ''
          )
          (0x1000000)Insert   = (
            (0x3000000)Type   = 2
            (0x3000000)Text   = '1'
          )
          (0x1000000)Insert   = (
            (0x3000000)Type   = 2
            (0x3000000)Text   = '2'
          )
          (0x1000000)Insert   = (
            (0x3000000)Type   = 5
            (0x3000000)Text   = 'Invalid document structure'
          )
          (0x1000000)Insert   = (
            (0x3000000)Type   = 5
            (0x3000000)Text   = 'XML'
```

```
          )
        )
      )
    )
  )
)
**** Message Properties is
(
  (0x3000000)MessageSet       = ''
  (0x3000000)MessageType      = ''
  (0x3000000)MessageFormat    = ''
  (0x3000000)Encoding         = 546
  (0x3000000)CodedCharSetId   = 437
  (0x3000000)Transactional    = TRUE
  (0x3000000)Persistence      = FALSE
  (0x3000000)CreationTime     = GMTTIMESTAMP '2001-06-16 22:26:38.610'
  (0x3000000)ExpirationTime   = -1
  (0x3000000)Priority         = 0
  (0x3000000)Topic            = NULL
)
**** Message descriptor is
(
  (0x3000000)SourceQueue      = 'ALEX_TEST_IN'
  (0x3000000)Transactional    = TRUE
  (0x3000000)Encoding         = 546
  (0x3000000)CodedCharSetId   = 437
  (0x3000000)Format           = 'MQSTR   '
  (0x3000000)Version          = 2
  (0x3000000)Report           = 0
  (0x3000000)MsgType          = 8
  (0x3000000)Expiry           = -1
  (0x3000000)Feedback         = 0
  (0x3000000)Priority         = 0
  (0x3000000)Persistence      = 0
  (0x3000000)MsgId            =
X'414d51204d5153492020202020202020ccc02b3be3620000'
  (0x3000000)CorrelId         =
X'0000000000000000000000000000000000000000000000000000'
  (0x3000000)BackoutCount     = 0
  (0x3000000)ReplyToQ         = '
'
  (0x3000000)ReplyToQMgr      = 'MQSI
'
  (0x3000000)UserIdentifier   = 'Alexau      '
  (0x3000000)AccountingToken  =
X'16010515000000e551800cb8a4a40ae558d40aea030000000000000000000000b'
  (0x3000000)ApplIdentityData = '                                '
  (0x3000000)PutApplType      = 11
  (0x3000000)PutApplName      = 'C:\WINNT\System32\MMC.EXE   '
  (0x3000000)PutDate          = DATE '2001-06-16'
  (0x3000000)PutTime          = GMTTIME '22:26:38.610'
```

```
  (0x3000000)ApplOriginData   = '    '
  (0x3000000)GroupId          =
X'000000000000000000000000000000000000000000000000'
  (0x3000000)MsgSeqNumber     = 1
  (0x3000000)Offset           = 0
  (0x3000000)MsgFlags         = 0
  (0x3000000)OriginalLength   = 4
)
```

One problem with adding the trace node before the failure queue is the maintenance of the trace file captured.

Most MQSI shops have a requirement that, before a message flow is rolled out to production, all trace nodes need to be disconnected from the message flow. This is because administration problems can arise if trace files accumulate in the production system.

A BETTER APPROACH

A better approach to handling exception processing is to implement an exception flow logic in the catch terminal of the MQInput node, or put the TryCatch node into the message flow.

When an exception condition arises in the message flow, and if the message flow is processing messages in a unit of work (UOW) and getting the message under syncpoint, an error in your message flow will cause the UOW to be backed out. This results in that message being reinstated on the input queue and, therefore, being processed again. If the error condition persists, the message will continue to be passed through the message flow and backed out, causing a processing loop. The message will continue to be processed until the MQMD *BackoutCount* equals or exceeds the value of the backout threshold (*BOTHRESH*) attribute of the input queue. MQSeries Integrator will then attempt backout processing by attempting to propagate the message to the following, in order:

1 The failure terminal of the MQInput node.

2 The queue's backout requeue (*BOQNAME*).

3 The dead-letter queue (DLQ).

Since we are not connecting the failure terminal in this approach, it is good practice to define a backout queue for the input queue and

specify a value for the backout threshold count. In this way we can avoid the processing loop, capture the roll-back message in the backout queue, and investigate the cause of failure.

The backout queue can be defined by a 'right-click' on the queue, followed by selecting Storage on the Property tab.

In the following sections we will set up some exception processing logic to help determine the problem with the message exceptions.

A SAMPLE EXCEPTION HANDLING FLOW

Figure 3 shows a very simple exception processing flow connected to the catch terminal of the MQInput node.

When an exception happens – either on the MQInput node or the compute node – the message will be rolled back into the MQInput node and will be processed via the flow connected to the catch node of the MQInput node. In this flow a trace of the ExceptionList will be recorded in a text file and the problem message will be thrown back to the backout queue.

A LOOK AT THE EXCEPTION LIST

This section will look at the *ExceptionList* structure and how to get the most out of it.

As Figure 4 illustrates, the *ExceptionList* tree has a definite structure.

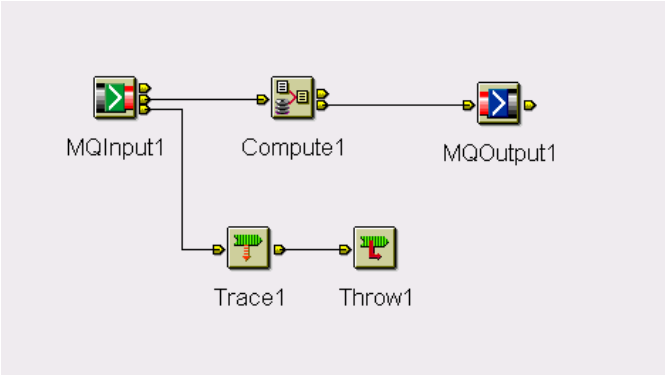

Figure 3: A simple exception processing flow

The root of the tree is called *ExceptionList*, and the tree itself consists of a set of one or more exception descriptions. Each exception description consists of one of the following name elements:

- *RecoverableException.*
- *ParserException.*
- *ConversionException.*
- *DatabaseException.*
- *UserException.*

These name elements contain children that take the form of a number of name-value elements that give details of the exception, and zero or more name elements whose name is *Insert*.

THROWING A USEREXCEPTION

Of the above listed exceptions, the type *UserException* can be inserted within the *MessageFlow* in anticipation of an erroneous situation, and thereby redirect the flow to the exception handling flow.

Figure 4 depicts a sample message flow where user exceptions can be thrown within a compute node and trapped by a filter node. Then a throw node redirects the message to the exception handling flow, which, in this case, is the flow attached to the catch node of the MQInput node.

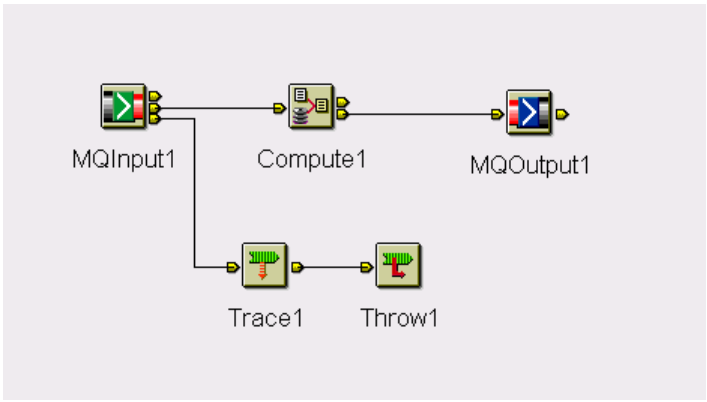

Figure 4: A sample message flow

In this example, to trap the error situation in the compute node when a field on the input message should be present but is not, the exception can be set with the following ESQL statement:

```
SET OutputExceptionList.UserException.Text = 'The mandatory field
of AddOrder.xml is missing!!';
```

The *UserException* can then be trapped by a filter node attached to the output terminal of the compute node with the statement:

```
Cardinality(ExceptionList.UserException[])>0
```

In order for this to work, one key requirement is to make sure the output of the compute node contains the *ExceptionList*, together with the message itself. This can be done by selecting the Exception and Message option from the pull-down list of the Compute Mode, in the Advanced tab of the compute node.

When an exception occurs, the *UserException* is inserted into the *ExceptionList*. The filter node condition should be set to true. The exception is thrown by the throw node attached to the true terminal of the filter node.

The message will then be routed to the exception processing flow connected to the catch terminal of the MQInput node. This flow will capture the exception trace on file indicating the detail of the *UserException*, and the message will be rolled back to the backout queue.

```
**** ExceptionList generated by messageflow exceptionhandlingl
**** exception generated at 2001-06-16 21:39:55.387001
(
  (0x1000000)RecoverableException = (
    (0x3000000)File          = 'F:/build/S201_P/src/DataFlowEngine/
ImbDataFlowNode.cpp'
    (0x3000000)Line          = 538
    (0x3000000)Function      = 'ImbDataFlowNode::createExceptionList'
    (0x3000000)Type          = 'ComIbmMQInputNode'
    (0x3000000)Name          = 'b0ab6922-e700-0000-0080-ca4d18331a10'
    (0x3000000)Label         = 'ExceptionHandling2.MQInput1'
    (0x3000000)Text          = 'Node throwing exception'
    (0x3000000)Catalog       = 'MQSIv201'
    (0x3000000)Severity      = 3
    (0x3000000)Number        = 2230
    (0x1000000)UserException = (
      (0x3000000)File     = 'F:/build/S201_P/src/DataFlowEngine/
BasicNodes/ImbThrowNode.cpp'
```

```
      (0x3000000)Line     = 229
      (0x3000000)Function = 'ImbThrowNode::evaluate'
      (0x3000000)Type     = 'ComIbmThrowNode'
      (0x3000000)Name     = '29ae1123-e700-0000-0080-ca4d18331a10'
      (0x3000000)Label    = 'ExceptionHandling2.Throw2'
      (0x3000000)Text     = 'User exception thrown by throw node'
      (0x3000000)Catalog  = 'MQSIV201'
      (0x3000000)Severity = 1
      (0x3000000)Number   = 3002
      (0x1000000)Insert   = (
        (0x3000000)Type = 5
        (0x3000000)Text = 'The ADDOrder.XML has a mandatory field
missing'
      )
    )
  )
)
```

The error encountered will also be written out as an NT event log, indicating the problem encountered by the throw node, as shown in Figure 5.

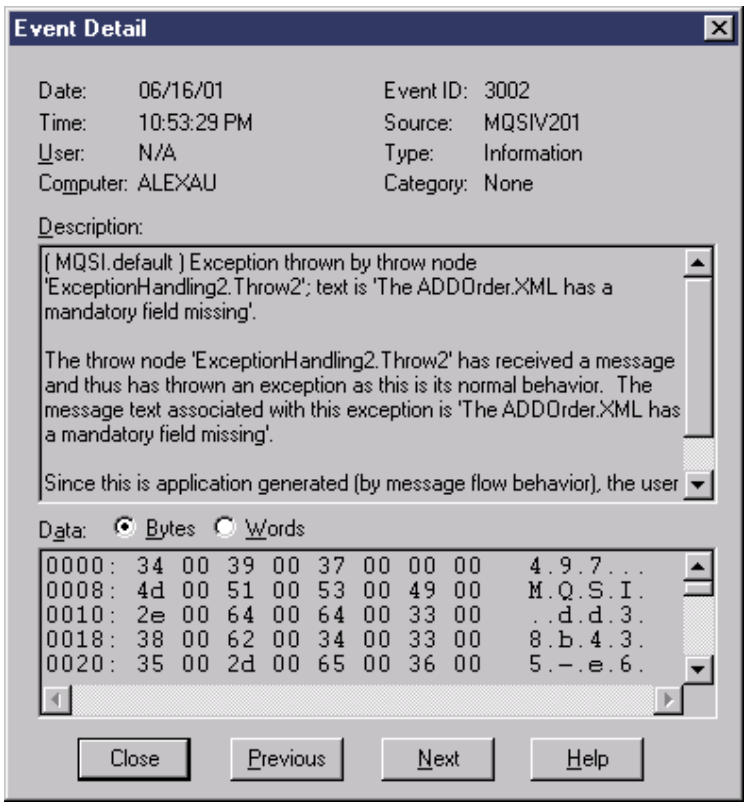

Figure 5: An NT event log

ADDING A TRYCATCH NODE

Sometimes, when an exception occurs at a certain point on the message flow you may not want to roll back the message all the way to the MQInput node, but would prefer to examine the *ExceptionList* to determine the nature of the error and thereby make an appropriate response.

Figure 6 illustrates the use of a TryCatch node in the message flow.

A message (*M1*) and destination list (*D1*) are being processed by a message flow. They are passed through the *Compute1* node to the *TryCatch1*, and then onto the *Compute2* node.

Compute1 updates the message and destination list and propagates a new message (*M2*) and destination list (*D2*) to the next compute node, *Compute2*.

An exception is thrown in *Compute2*. The exception is propagated back to theTryCatch node, but the message and destination list are not. Therefore, the exception handling path starting at point *B* has access to the intermediate message and destination list, *M2* and *D2*.

If there had been no TryCatch node in the message flow, the message and destination list *M1* and *D1* would have been propagated to the catch terminal connected to that MQInput node. The intermediate information of the message *M2* that may be of interest to solve the message flow problem will be lost.

BEST PRACTICE

A useful point to remember is that, when a messaging exception occurs, MQ can generate an error message that includes all the

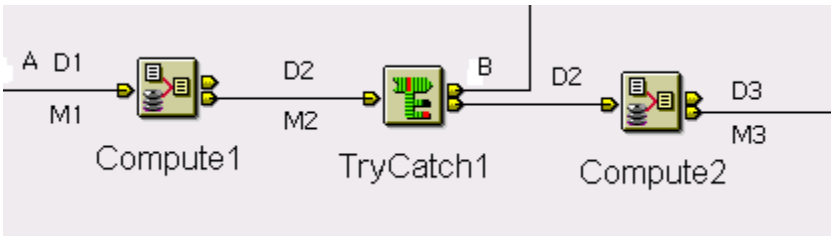

Figure 6: The use of a TryCatch node in the message flow

information that will help in debugging the problem. The error message generated will be propagated with the same Message-ID and Corel-ID as the failure message. In this way, the message that failed and was sent to the backout queue can be referenced by the error message generated in the error queue.

The same practice can be implemented with MQSI in the exception handling flow.

Figure 7 depicts the message flow that will propagate the information on the *ExceptionList* to an error message and output it to the error queue.

The essence of the *ExceptionList* can be extracted and propagated into the error message. This can be achieved with the following ESQL statement in the *WriteErrorMsg* compute node.

```
/* Error number extracted from exception list */
DECLARE Error INTEGER;
/* Current path within the exception list */
DECLARE Path CHARACTER;
DECLARE ErrorType  CHARACTER;
```

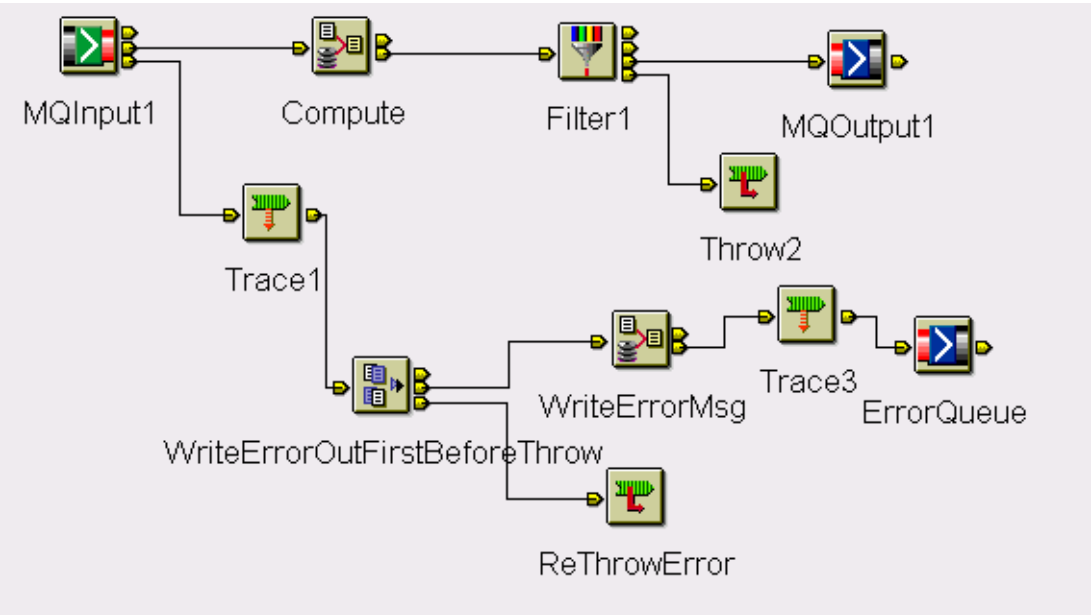

Figure 7: Message flow propagating ExceptionList information to error message

```
DECLARE ErrorTrans CHARACTER;
DECLARE ErrorText  CHARACTER;
SET ErrorType  = '';
SET ErrorTrans = '';
SET ErrorText  = '';
/* Start at first child of exception list */
SET Path ='InputExceptionList.*[1]';

/* Loop until no more children */
WHILE EVAL( 'FIELDNAME(' || Path || ') IS NOT NULL' ) DO

  /* Check if error number is available */
  IF EVAL( 'FIELDNAME(' || Path || '.Number) IS NOT NULL' ) THEN
    /* Remember only the deepest error number */
    SET Error      = EVAL( Path || '.Number' );
    SET ErrorType  = EVAL('FIELDNAME('||Path||')');
    SET ErrorTrans = EVAL(Path || '.Label');
    SET ErrorText  = EVAL(Path || '.Text');
    IF ErrorType   = 'UserException' THEN
     SET ErrorText = ErrorText || ' (' || (EVAL(Path || '.Insert.Text'))
|| ')';
    END IF;
  END IF;
  /* Step to last child of current element (usually a nested exception
  list) */
  SET Path = Path || '.*[LAST]';
END WHILE; /* End loop */
SET OutputRoot.XML.Exception.Type       = ErrorType;
SET OutputRoot.XML.Exception.Number     = CAST(Error AS CHAR);
SET OutputRoot.XML.Exception.SuspenseTimestamp = CAST(CURRENT_TIMESTAMP
  AS CHAR);
SET OutputRoot.XML.Exception.InboundQueue  = InputRoot.MQMD.SourceQueue;
SET OutputRoot.XML.Exception.Transaction   = ErrorTrans;
SET OutputRoot.XML.Exception.Reason        = ErrorText;
SET OutputRoot.XML.Exception.DetailException = InputExceptionList;
```

THE KEY TO SUCCESS

Two important steps need to be included in this error message generating flow. Firstly, in order for the error message to be written out successfully to the output error queue, a FlowOrder node named *WriteErrorOutFirstBeforeThrow* is implemented to ensure that the error message generated in the compute node *WriteErrorMsg* is written out to the error queue before we re-throw the exception and backout the input message to the backout queue.

Secondly, in the MQOutput node *ErrorQueue*, the action to put the error message into the error queue must be committed immediately.

Otherwise, when we re-throw the message to the backout queue, the uncommitted error message in the error queue will also be backed out.

This can be accomplished by selecting No from the drop down list in the Transaction Mode of the Advanced tab in the MQOutput node.

THE ERROR MESSAGE

The error message generated in the error queue will contain all the useful information from the *ExceptionList* generated from the message flow – a sample output is shown below.

SAMPLE OUTPUT

```
- <Exception>
  <Type>UserException</Type>
  <Number>3002</Number>
  <SuspenseTimestamp>TIMESTAMP '2001-06-16 23:17:28.143999'</
   SuspenseTimestamp>
  <InboundQueue>ALEX_TEST_IN</InboundQueue>
  <Transaction>ExceptionHandling2.Throw2</Transaction>
  <Reason>User exception thrown by throw node (The ADDOrder.XML has a
   mandatory field missing)</Reason>
- <DetailException>
- <RecoverableException>
  <File>F:/build/S201_P/src/DataFlowEngine/ImbDataFlowNode.cpp</File>
  <Line>538</Line>
  <Function>ImbDataFlowNode::createExceptionList</Function>
  <Type>ComIbmMQInputNode</Type>
  <Name>b0ab6922-e700-0000-0080-ca4d18331a10</Name>
  <Label>ExceptionHandling2.MQInput1</Label>
  <Text>Node throwing exception</Text>
  <Catalog>MQSIv201</Catalog>
  <Severity>3</Severity>
  <Number>2230</Number>
- <UserException>
  <File>F:/build/S201_P/src/DataFlowEngine/BasicNodes/ImbThrowNode.cpp</
   File>
  <Line>229</Line>
  <Function>ImbThrowNode::evaluate</Function>
  <Type>ComIbmThrowNode</Type>
  <Name>29ae1123-e700-0000-0080-ca4d18331a10</Name>
  <Label>ExceptionHandling2.Throw2</Label>
  <Text>User exception thrown by throw node</Text>
  <Catalog>MQSIV201</Catalog>
  <Severity>1</Severity>
  <Number>3002</Number>
```

```
- <Insert>
  <Type>5</Type>
  <Text>The ADDOrder.XML has a mandatory field missing</Text>
  </Insert>
  </UserException>
  </RecoverableException>
  </DetailException>
  </Exception>
```

CONCLUSION

This concludes our introduction to exception handling for MQSI. Using these techniques we can further improve the exception handling of the message flows to filter out more granularity of the exception, and perform the appropriate action. This may include actions such as stopping the message flow when a system error (such as a DB2 authorization failure on a user table) occurs. This will be covered in part 2 of this article, which will appear in next month's *MQ Update*.

Alex Au
I/T Architect, IBM Global Services (USA)

Add-on for BMC PATROL to measure a channel's message rate

The add-on KM (Knowledge Module) 'ChannelShob' for BMC PATROL defines parameters for monitoring the MQ channels' message rate.

The KM is provided as it stands and is for educational purposes only, with no warranties. Although the supplied files were used without errors, rigorous testing was *not* performed. It is recommended that you use it in a test environment first. The addition and modification of KMs to PATROL is a complicated task and while detailed instructions are supplied it is strongly recommended that you don't modify your PATROL environment without proper training. Furthermore, when performing modifications, users are encouraged to backup all their files first.

(Editor's note: the ChannelShob.KM, Collectors_addition.txt, and NT_Collectors.KM subroutines supporting this article are available from Xephon's Web site at http://www.xephon.com/extras/ChannelShob.KM.)

NOTES

- The KM is independent of BMC's MQOperator product, but can be used with it.
- The KM collects the data from **runmqsc DISPLAY CHS** commands that are being executed in the monitored hosts.
- If MQ Operator V1.2 and upwards is used, the KM may be modified to collect the data from channel parameters that its MQ Operator supplies.

INSTALLATION INSTRUCTIONS

On the host(s) running PATROL Agent, make sure that PATROL's login has MQ permissions to run **DISPLAY MQSC** commands.

Carry out the following steps on the host running PATROL Console in developer mode.

1 Backup the folder *$PATROL_HOME/lib/knowledge* before proceeding.

2 Copy *ChannelShob.KM* to *$PATROL_HOME/lib/knowledge*.

3 Using a text editor, change *myhostname* and *myqmgrname* in the file *ChannelShob.KM* to the hostname and queue manager name to be monitored. If multiple hosts or queue managers are to be monitored, these code segments need to be repeated for every host and queue manager, respectively.

4 If Windows NT hosts are to be monitored, copy *NT_COLLECTORS.KM* to the same folder.

5 If Unix hosts are to be monitored, close the developer console and insert the contents of the file *collectors_addition.txt* to the file *COLLECTORS.KM at $PATROL_HOME/lib/knowledge* after the following lines:

```
INFO_BOX = {
       { NAME = "Unix KM Version", AVAILABILITY = AVAILABLE_ALWAYS,
  SECURITY = SECURITY_INHERIT,
            BASE_COMMAND = {
            { COMPUTER_TYPE = "ALL_COMPUTERS", COMMAND_TYPE = "PSL",
            COMMAND_TEXT = LOAD "unixkm_version.psl"}
            }
        }
    },
    PARAMETERS = {
```

And before the following line:

```
{  NAME = "PSColl", PARAM_TYPE = COLLECTOR, ACTIVE = False, MONITOR
   = False, CHECK = False,
```

6 The module will be activated once you restart the console, select File->Load KM from the menu, and select the files that were copied. The channel tree will be built automatically.

7 The default behaviour is to show all channels under every queue manager of the monitored hosts. To change that, at the developer console, ‘right-click’ ChannelShob and select KM commands->edit list (see Figure 1).

8 The parameters can be configured to warn or alarm at certain values like any other PATROL parameters (see Figure 2).

9 The channels will be monitored as long as the console remains connected. To continue monitoring while not connected, the KM can be defined as 'preloaded' at the target agents (see Figure 3).

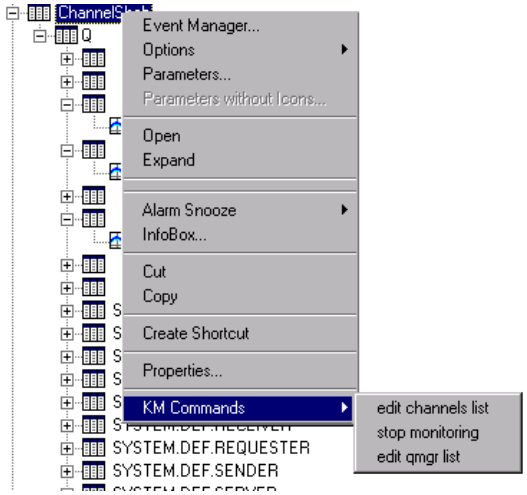

Figure 1: Changing the default behaviour

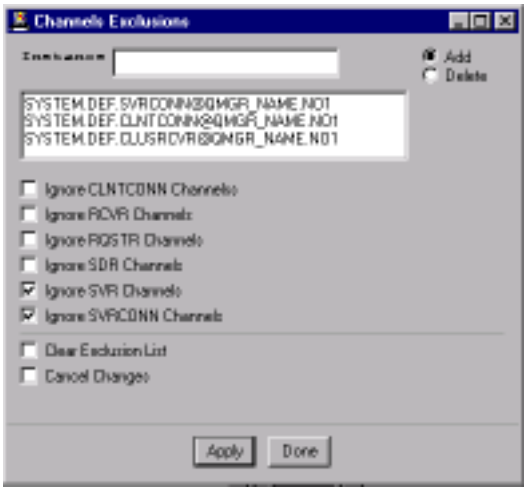

Figure 2: Configuring the parameters

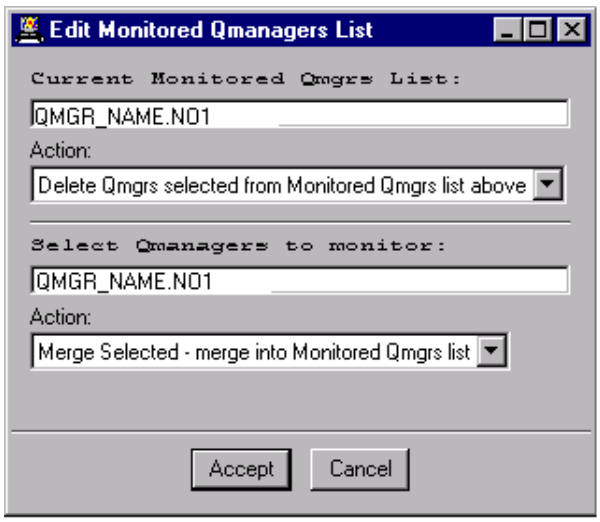

Figure 3: Monitoring the channels

Roy Razon, MQSeries Consultant (Israel) and
Liat Sabag, PATROL Consultant (Israel)

Simplifying journal management in MQSeries for AS/400 V5.2

INTRODUCTION

This article briefly discusses the way in which MQSeries for AS/400 uses journalling and how the journals can be managed. Source code is included for a command and CL program, which provides a single step to compute which journal receivers are required by one or all queue managers on a system.

MQSeries for AS/400 uses OS/400 journalling support to record message data flows and changes to queue manager objects. The data stored in the journals is used by MQSeries to ensure transactional integrity on queue manager restart, and to allow recovery of damaged objects.

Each queue manager has its own journal (*AMQAJRN*), which is stored in a queue manager-specific library, and each journal has a single attached journal receiver. The data that is being journalled is appended to the currently-attached receiver until that receiver reaches a predefined threshold size (defaulting to 65,536 KB in MQSeries for AS/400 V5.2).

The operating system detaches the attached receiver when it reaches the threshold size and attaches a new one. Detached receivers remain on disk until they start to become a significant disk overhead and are manually deleted.

Freeing disk space is just one reason for deleting redundant journal receivers; another important reason is to improve performance. MQSeries reads back through the journal receivers when a queue manager starts, and periodically, when the queue manager is running. Reading the journal receivers is an expensive operation so minimizing the ‘chain’ of receivers you keep on your system improves the performance of MQSeries.

DELETING RECEIVERS

Once detached, journal receivers cannot necessarily be deleted

immediately, because they may still contain journal entries that are required by MQSeries. Journal entries fall into two categories:

- Startup journal entries – required to restart the queue manager.
- Media recovery journal entries – required to rebuild objects.

For the normal operation of MQSeries for AS/400 you do not need to keep all of the media recovery journal receivers available on your disks. You can back up and delete media recovery journal receivers and keep only the journal receivers required for startup. In this way, your journal receivers will consume less disk space, but if an object is damaged you will need to restore the media recovery journal receivers from the backup in order to recreate the object.

To help determine which receivers are required MQSeries writes two messages to the queue manager's message queue (*QMQMMSG* in the queue manager library). The first of these messages is *AMQ7460*, which reports the time stamp and receiver name of the journal entry that is required for a queue manager to be restarted. The second message is *AMQ7462*, which reports the time stamp and receiver name of the oldest journal entry that is required to recover any object. See the *System Administration Guide* for more details about these messages.

In order to calculate which receivers can be deleted for a queue manager you must find the *AMQ7460* and *AMQ7462* messages in the queue manager's *QMQMMSG* message queue, compare their values, and only delete receivers earlier than those shown as still required. This can become a time-consuming process when you have installed MQSeries for AS/400 on several machines with multiple queue managers on each one.

MQSERIES V5.2 ENHANCEMENTS

In MQSeries for AS/400 V5.2 a new option was added to the **RCDMQMIMG** command, which forces it to issue the *AMQ7460* and *AMQ7462* messages to the current job log. This allows the messages to be generated on demand and simplifies the first step in automated journal management – establishing which journal receivers can be safely deleted.

THE **QRYMQMJRN** COMMAND

The **QRYMQMJRN** command accompanying this article can be used to report the journal receivers you need to keep on the system for each queue manager. The command bases its recommendations on the cautious policy of keeping all of the media recovery journal receivers online.

Note that information is only reported for active queue managers.

The program demonstrates the use of **AMQXLIST** to retrieve a list of queue manager names, and the use of **RCDMQMIMG** to retrieve journal receiver information for each queue manager. It also shows how the journal receiver date and time information can be compared to determine the oldest receiver that should be kept.

This command does not make any attempt to automate the backup and deletion of journal receivers, though the program could be modified to include that function. A program to determine the names of journal receivers for backup and deletion will be the subject of a later article.

Compiling the QRYMQMJRN program

The CL program is dependent on a physical file in *QTEMP*, so you must create that file before compiling the CL program. These commands assume that the source has been placed in a member called **QRYMQMJRN** in file *QCLSRC* in a library (*YOURLIB*).

To compile the CL program run these two commands:

```
CRTPF FILE(QTEMP/AMQQMGRS) RCDLEN(48)
CRTCLPGM PGM(YOURLIB/QRYMQMJRN) SRCFILE(YOURLIB/QCLSRC)
```

Compiling the QRYMQMJRN command

Assuming that the source has been placed in a member called **QRYMQMJRN** in file *QCMDSRC* in a library (*YOURLIB*), you can create the **QRYMQMJRN** command by running the following command:

```
CRTCMD CMD(YOURLIB/QRYMQINF) PGM(YOURLIB/QRYMQINF)
       SRCFILE(YOURLIB/QCMDSRC) SRCMBR(QRYMQMJRN)
```

Running the QRYMQMJRN command

Once you have built the **QRYMQMJRN** command you can query journal receiver information for a single queue manager with the following command:

```
QRYMQMJRN MQMNAME(YourQmgrName)
```

...or you can query all queue managers with the command:

```
QRYMQMJRN MQMNAME(*ALL)
```

The **QRYMQMJRN** command produces the following output:

```
> QRYMQMJRN MQMNAME(*ALL)
For Queue manager:"MP" the oldest startup journal entry is dated:
"01/05/31 15:29:13" in receiver: "AMQA000006". The oldest media recovery
journal entry is dated: "01/05/30 09:47:20" in receiver:AMQA000002".
ACTION: delete receivers earlier than media journal receiver
"AMQA000002".
Unable to get data for Queue Manager MQBLD
For Queue manager:"INVOICING" the oldest startup journal entry is dated:
"01/06/01 09:47:37" in receiver: "AMQA000068". The oldest media recovery
journal entry is dated: "01/05/31 14:10:54" in receiver:AMQA000034".
ACTION: delete receivers earlier than media journal receiver
"AMQA000034".
Unable to get data for Queue Manager TESTQMGR
```

This output shows that four queue managers have been created on the system.

Queue managers *MQBLD* and *TESTQMGR* are both inactive and so have not produced any results, but queue managers *MP* and *INVOICING* are running and have reported which journal receivers are needed.

In this example we can see that queue manager *MP* no longer needs journal receivers older than *AMQA000002*, so you can run the **OS/400 WRKJRNRCV** command to see which journal receivers are older than *AMQA000002* in the queue manager library. For example:

```
WRKJRNRCV JRNRCV(QMMP/AMQA*)
```

produces the following list, showing that receivers *AMQA000000* and *AMQA000001* can be safely deleted:

```
Type options, press Enter.
  1=Create   4=Delete   5=Display attributes   13=Change description
     Journal
Opt  Receiver    Library     Text
     AMQA000000  QMMP        MQM local journal receiver
```

```
     AMQA000001  QMMP        MQM local journal receiver
     AMQA000002  QMMP        MQM local journal receiver
     AMQA000003  QMMP        MQM local journal receiver
     AMQA000004  QMMP        MQM local journal receiver
     AMQA000005  QMMP        MQM local journal receiver
     AMQA000006  QMMP        MQM local journal receiver
```

QRYMQJRN

```
Program:       QRYMQJRN                                              */
/*  Function:   Sample program to issue a RCDMQMIMG command         */
/*              against specified queue manager(s), then extract    */
/*              the information needed to report on the journal     */
/*              receivers that can be safely deleted.               */
/*  Parameters: &MQMNAME - Blank padded 48 character queue          */
/*                         manager name                             */
             PGM        PARM(&MQMNAME)
             DCL        VAR(&MQMNAME) TYPE(*CHAR) LEN(48)
             DCL        VAR(&DATFMT) TYPE(*CHAR) LEN(4)
             DCL        VAR(&ALLQMGRS) TYPE(*LGL) VALUE('0')
             DCLF       FILE(QTEMP/AMQQMGRS)
/*  Message handling variables                                      */
             DCL        VAR(&RSPMSG) TYPE(*CHAR) LEN(7)
             DCL        VAR(&DATA1) TYPE(*CHAR) LEN(60)
             DCL        VAR(&DATA2) TYPE(*CHAR) LEN(60)
             DCL        VAR(&DATALEN) TYPE(*DEC) LEN(5 0) VALUE(40)
             DCL        VAR(&MSGID1) TYPE(*CHAR) LEN(7)
             DCL        VAR(&MSGID2) TYPE(*CHAR) LEN(7)
             DCL        VAR(&DATE1) TYPE(*CHAR) LEN(17)
             DCL        VAR(&DATE2) TYPE(*CHAR) LEN(17)
             DCL        VAR(&NAME1) TYPE(*CHAR) LEN(10)
             DCL        VAR(&NAME2) TYPE(*CHAR) LEN(10)
             DCL        VAR(&RESPONSE) TYPE(*CHAR) LEN(300)
/*  Error handling variables                                        */
             DCL        VAR(&ERRORSW) TYPE(*LGL)
             DCL        VAR(&MSGID) TYPE(*CHAR) LEN(7)
             DCL        VAR(&MSGDTA) TYPE(*CHAR) LEN(100)
             DCL        VAR(&MSGF) TYPE(*CHAR) LEN(10)
             DCL        VAR(&MSGFLIB) TYPE(*CHAR) LEN(10)
             MONMSG     MSGID(CPF0000) EXEC(GOTO CMDLBL(STDERR1))
/* If processing all Qgmrs, populate file and set flag             */
             IF         COND(&MQMNAME *EQ '*ALL') THEN(DO)
             CALL       PGM(QMQM/AMQXLIST) PARM('-q')
             CHGVAR     VAR(&ALLQMGRS) VALUE('1')
             ENDDO
/* If processing all Qgmrs, read names from file                   */
 STARTLOOP:  IF         COND(&ALLQMGRS *EQ '1') THEN(DO)
             RCVF
             MONMSG     MSGID(CPF0864) EXEC(GOTO CMDLBL(FINISH))
             CHGVAR     VAR(&MQMNAME) VALUE(&AMQQMGRS)
             ENDDO
```

```
/* Save job date format and set to YY/MM/DD for easy comparison   */
             RTVJOBA    DATFMT(&DATFMT)
             CHGJOB     DATFMT(*YMD)
/* Record media image                                             */
             RCDMQMIMG  OBJ(&MQMNAME) OBJTYPE(*MQM) +
                          MQMNAME(&MQMNAME) DSPJRNDTA(*YES)
/* Restore job date format and pick up response messages          */
             CHGJOB     DATFMT(&DATFMT)
             RCVMSG     MSGID(&RSPMSG)
             RCVMSG     MSGDTA(&DATA1) MSGDTALEN(&DATALEN) +
                          MSGID(&MSGID1) /* Get AMQ746Ø    */
             RCVMSG     MSGDTA(&DATA2) MSGDTALEN(&DATALEN) +
                          MSGID(&MSGID2) /* Get AMQ7462    */
/* We succeeded if we got AMQ7486, AMQ746Ø and AMQ7462 messages   */
             IF         COND((&RSPMSG *NE 'AMQ7Ø86') *OR (&MSGID1 +
                          *NE 'AMQ746Ø') *OR (&MSGID2 *NE +
                          'AMQ7462')) THEN(DO)
             SNDPGMMSG  MSG('Unable to get data for Queue Manager' +
                          *BCAT &MQMNAME)
             GOTO       CMDLBL(ENDLOOP)
             ENDDO
/* Extract the relevant data from the messages                    */
          CHGVAR &DATE1 %SST(&DATA1 17 17) /*Extract date from AMQ746Ø*/
          CHGVAR &NAME1 %SST(&DATA1 36 1Ø) /*Extract name from AMQ746Ø*/
          CHGVAR &DATE2 %SST(&DATA2 17 17) /*Extract date from AMQ7462*/
          CHGVAR &NAME2 %SST(&DATA2 36 1Ø) /*Extract name from AMQ7462*/
/* Report the action to take                                      */
             CHGVAR     VAR(&RESPONSE) VALUE('For Queue manager:"' +
                          *TCAT &MQMNAME *TCAT '" the oldest +
                          startup journal entry is dated: "' *TCAT +
                          &DATE1 *TCAT '" in receiver: "' *TCAT +
                          &NAME1 *TCAT '". The oldest media +
                          recovery journal entry is dated: "' *TCAT +
                          &DATE2 *TCAT '" in receiver: ' *TCAT +
                          &NAME2 *TCAT '". ACTION: delete receivers +
                          earlier than')
             IF         COND(&DATE1 *LE &DATE2) THEN(SNDPGMMSG +
                          MSG(&RESPONSE *BCAT 'startup receiver "' +
                          *TCAT &NAME1 *TCAT '".'))
             ELSE       CMD(SNDPGMMSG MSG(&RESPONSE *BCAT 'media +
                          journal receiver "' *TCAT &NAME2 *TCAT '".'))
/* If processing all Qmgrs, loop to get next queue manager        */
ENDLOOP:     IF         COND(&ALLQMGRS *EQ '1') THEN(GOTO +
                          CMDLBL(STARTLOOP))
FINISH:     RETURN
/*  Handle unexpected errors                                      */
 STDERR1:
             IF         COND(&ERRORSW) THEN(SNDPGMMSG MSGID(CPF9999) +
                          MSGF(QCPFMSG) MSGTYPE(*ESCAPE))
             CHGVAR     VAR(&ERRORSW) VALUE('1')
 STDERR2:    RCVMSG     MSGTYPE(*DIAG) MSGDTA(&MSGDTA) MSGID(&MSGID) +
```

```
                    MSGF(&MSGF) MSGFLIB(&MSGFLIB)
            IF      COND(&MSGID *EQ '       ') THEN(GOTO +
                      CMDLBL(STDERR3))
            SNDPGMMSG MSGID(&MSGID) MSGF(&MSGFLIB/&MSGF) +
                      MSGDTA(&MSGDTA) MSGTYPE(*DIAG)
            GOTO      CMDLBL(STDERR2)
 STDERR3:   RCVMSG    MSGTYPE(*EXCP) MSGDTA(&MSGDTA) MSGID(&MSGID) +
                      MSGF(&MSGF) MSGFLIB(&MSGFLIB)
            SNDPGMMSG MSGID(&MSGID) MSGF(&MSGFLIB/&MSGF) +
                      MSGDTA(&MSGDTA) MSGTYPE(*ESCAPE)
            ENDPGM
```

QRYMQMJRN

```
/* Command name: QRYMQMJRN                                          */
/* Description: Query journal information for one or more queue     */
/*              managers                                            */
 QRYMQMJRN: CMD        PROMPT('Query MQSeries Journal Info')
            PARM       KWD(MQMNAME) TYPE(*CHAR) LEN(48) RSTD(*NO) +
                         DFT(*ALL) SPCVAL((*ALL *ALL)) MAX(1) +
                         PROMPT('Queue manager name')
```

Mark Phillips
MQSeries Development, IBM Hursley

HIS and its MSMQ to MQSeries Bridge (part 1)

HIS (Host Integration Server) 2000 provides bi-directional services for integrating and linking Microsoft Windows environments with legacy computing systems. HIS provides interoperability in the following areas:

- Application integration services.
- Data integration services.
- Network integration services.

HIS 2000 offers a means to enable Internet, intranet, and client/server technologies while still allowing organizations to maximize existing and future investments in enterprise computing platforms such as SNA – a proprietary networking architecture developed by IBM. SNA defines a set of communication protocols and message formats for

managing network data; and methods for terminal access to mainframe computers, file transfer, printing, and peer-to-peer communications that allow applications to exchange data over a network. In this mainframe network HIS appears as a cluster controller PU 2 (physical unit type 2) device, which can be attached directly to the host via a high-speed data channel.

The MSMQ-MQSeries Bridge enables your applications to exchange messages between IBM MQSeries and Microsoft Message Queue Server (MSMQ). The Bridge provides connectionless store-and-forward messaging across messaging systems and computing platforms.

In this series of three articles, we will examine how system administrators can link MSMQ with MQSeries using the HIS integrated MSMQ-MQSeries Bridge. The first article will cover planning, the second will focus on deployment and configuration, and the third will focus on administration and performance.

MESSAGE QUEUEING AND BRIDGING CONCEPTS

A message queueing system is basically a store-and-forward system that allows applications running at different times to communicate across distributed networks. Programs can share data across a network without having a synchronized connection between the sending and receiving components of the distributed application – a sending application can put the data, or message, on a message queue, which is then retrieved by the receiving application.

MSMQ and MQSeries use the fundamental concepts of a message and message queue. A message is a set of data that needs to be transmitted from one application to another on the same or a different computer in a network. A message queue is the location where messages are stored and the messages can be written and read by applications.

MSMQ messages consist of fields known as message properties. Any number of properties is allowed – including zero – yielding a message with a dynamic data structure. This is in contrast to MQSeries, where the fields are in a fixed data structure. The process for sending a message is detailed below.

- A sending application specifies the message fields or properties, provides the field values, and then issues an API call to MSMQ or MQSeries to send a message.
- The MSMQ or MQSeries queue manager stores and forwards the message to the destination queue. If the destination queue is not available, the message is forwarded automatically when a connection is established.
- A receiving application issues an API call to receive the message from the queue.

In summary, MSMQ and MQSeries provide the following services:

- Connectionless, asynchronous messaging.
- Guaranteed delivery.
- Message prioritization, enabling the order in which the receiving application obtains the messages to be specified.
- A user-defined message structure, consisting of no message, a single byte, a text string, or a long and complex data structure – even involving encryption that only the communicating applications understand.
- Send-message or receive-message operations, which can participate in a transaction so that there can be coordination with other database operations, and the entire transaction can be cancelled and rolled back if any of the operations fail.
- Application programming interfaces (APIs) operating on the Application Layer of the ISO Reference Model for Open System Interconnection, thereby providing a simple interface between an application program and the network, freeing the application programmer from concern about network or communication details.

A bridge is required between MSMQ and MQSeries so that applications can exchange messages between the two systems. The MSMQ-MQSeries Bridge achieves this by mapping the messages, data fields, and values of the sending system to the fields and values of the receiving environment. After the mapping and conversion, the Bridge

completes the process by routing the message across the combined MSMQ and MQSeries networks.

The MSMQ-MQSeries Bridge was originally developed by Level 8 Systems and released as the FalconMQ Bridge. Since being sold and licensed to Microsoft in early 1998, the Bridge has become part of Microsoft's HIS 2000.

The bridge process is transparent, operating entirely in the background. An MSMQ application sends a message using a standard MSMQ **MQSendMessage()** API call or ActiveX control and the MQSeries application receives the message from the MQSeries queue using a standard MQSeries **MQGET()** API call. Since each application is dealing with its native environment, complexity is removed. Messages are routed to and from each messaging system even if the two systems are not connected to the network at the same time.

The MSMQ-MQSeries Bridge system contains two principal components:

- The Bridge, which converts and transmits messages between the two environments.
- The Bridge Explorer/Manager, which enables configuration, monitoring, and control of messaging traffic through the Bridge.

In sending a message from MQSeries to MSMQ, the bridge maps the fields of the MQSeries message to its MSMQ counterpart and, if a non-existent additional field is required at the destination, then the bridge provides the field. The support documentation states, "suppose, for example, that an MSMQ message includes the *PROPID_M_TIME_TO_BE_RECEIVED* property with a specific value. The Bridge maps this property to the MQSeries *MQMD.Expiry* property and multiplies the value by ten to change the units from seconds to tenths of seconds".

The advantage in using the Bridge comes from the fact that there is no restriction on message content. The sending and receiving applications can impose their own internal structure or encryption schemes, which are left unaltered by the Bridge.

The MSMQ-MQSeries Bridge is an MSMQ Connector Server application. The connector allows MSMQ applications to communicate

with foreign computers that use other messaging systems. As part of HIS 2000, the Bridge is installed on a Windows 2000 system that serves as a connection point between the networks. So the Bridge itself consists of Windows 2000, HIS 2000, and its integrated bridging capabilities. The computer must be connected by a TCP/IP or SNA link to an MQSeries Queue Manager.

You can connect any number of MSMQ or MQSeries systems or networks using MSMQ-MQSeries Bridge. For example, you can connect one or more MSMQ-MQSeries Bridge(s) to one or more MQSeries Queue Manager(s).

MQSERIES AND MSMQ MESSAGE FIELDS OR PROPERTIES

A message can contain one or more fields. Examples include the message buffer or body, label, priority, or sender-ID.

- In MQSeries, the fields are members of a fixed data structure.
- In MSMQ, the fields are known as properties and there can be any number of properties, even zero. An application assembles a dynamic (rather than fixed) data structure from one or more properties.

SENDING AND RECEIVING MESSAGES

When creating a message, an application specifies the message fields or properties plus field values and then issues an MSMQ or MQSeries API call to send the message. The MSMQ or MQSeries queue manager (server) transmits the message to the destination message queue. If the destination location is not currently connected to the network the message queueing system stores the message at an interim location and then forwards the message automatically when a connection is established. An application on the receiving side issues an API call that reads the message from the queue.

SENDING MESSAGES FROM MSMQ TO MQSERIES

You first define an MSMQ ‘foreign’ computer, representing the MQSeries Queue Manager. This assumes that the MQSeries destination

queue already exists. As stated earlier, the MSMQ-MQSeries Bridge uses the MSMQ Connector, which allows MSMQ applications to communicate with 'foreign' computers that use other messaging systems, connected networks, and queues. Once the 'foreign' connected network is created you then create a 'foreign' computer, which would be the MQSeries Queue Manager – essentially a node hosting queues or functioning like a 'computer' under MSMQ. Next, you need to activate the MSMQ Open Connector security permissions. Once these steps have been completed, the messaging process works as follows:

- An MSMQ application issues an MSMQ **MQCreateQueue()** API call to create a 'foreign' queue on the 'foreign' computer representing the MQSeries destination queue. This 'foreign' queue can also be created using MSMQ. In Windows 2000 the 'foreign' queue is a part of the Active Directory Users and Computers.
- The application calls **MQOpenQueue()** to open the 'foreign' queue.

A message being sent from MSMQ to MQSeries follows this process:

- The MSMQ application sends a message to the MQSeries queue.
- MSMQ stores the message on a connector queue.
- The Bridge takes the message from the connector queue and then converts and forwards it to MQSeries.
- MQSeries delivers the message to the destination queue.
- The MQSeries application receives the message from the MQSeries queue.
- The MSMQ application calls the **API MQSendMessage()** to send a message to the foreign queue. MSMQ routes the message and stores it temporarily on an MSMQ connector queue. MSMQ processes the message from the initial **MQSendMessage()** call until it is placed on the connector queue.
- The Bridge takes the message from the connector queue and converts the message properties to the MQSeries message data

structure. The Bridge routes the message to the MQSeries destination queue. The Bridge converts and transmits the message to MQSeries, which handles the transmission from that point on.

- An MQSeries application issues an MQSeries **MQGET()** API call to receive the message from the MQSeries queue.

SENDING MESSAGES FROM MQSERIES TO MSMQ

A message being sent from MQSeries to MSMQ goes through the following stages.

- The MQSeries application sends a message to the MQSeries queue.
- MQSeries stores the message on the transmission queue.
- The message is transmitted from the MQSeries transmission queue to the Bridge.
- The Bridge takes the message and converts and then forwards it to MSMQ.
- The message resides in the MSMQ destination queue.
- The MSMQ application receives a message from the MQSeries queue and MQSeries application.

Excepting a few minor differences, the process for sending a message from MQSeries to MSMQ is essentially the inverse of MSMQ to MQSeries. You define MQSeries aliases, transmission queues, and channels for the MSMQ destination queue and/or the MSMQ Server – assuming the MSMQ destination queue already exists.

The messaging process will be as follows:

- An MQSeries application issues an **MQOPEN()** API call for a remote queue representing the MSMQ destination queue.
- The MQSeries application issues an **MQPUT()** API call to send a message to the remote queue. MQSeries transmits the message and stores it temporarily on an MQSeries transmission queue located at the MQSeries Queue Manager.

- The Bridge takes the message from the transmission queue and converts the message structure to MSMQ message properties. The Bridge transmits the message to the MSMQ destination queue.
- An MSMQ application issues an **MSMQ MQReceiveMessage()** API call to receive the message from the MSMQ queue.

Normal and high service

Messages are sent by normal service or high service.

Normal service entails the 'deliver once' feature of MSMQ and MQSeries that ensures that each message is delivered exactly once to the receiving application. However, there is an overhead. High service can improve performance, but a message may be delivered more than once in the event of a system failure during transmission.

The MSMQ-MQSeries Bridge sends transacted messages by normal service and untransacted messages by high service when sending from MSMQ to MQSeries. When sending from MQSeries to MSMQ, you specify a particular alias for the remote queue manager address to send a message by normal or high service.

TRANSACTIONAL AND NON-TRANSACTIONAL MESSAGE PIPES

With the Bridge, you can send messages through transactional or non-transactional message pipes. The transactional message pipe supports the 'deliver once' feature, but, as with normal service, there is a performance overhead.

The non-transactional message pipe improves performance, although a message may be delivered more than once. MSMQ messages in the foreign transactional queue will go through transactional message pipes, and MSMQ messages in the foreign non-transactional queue will go through the non-transactional message pipes.

From MSMQ to MQSeries the Bridge sends transacted messages by normal service and untransacted messages by high service.

From MQSeries to MSMQ you can send a message by transactional or non-transactional message pipe by specifying the appropriate

remote queue manager alias, such as *BRIDGEQMNAME* for the transactional message pipe and *BRIDGEQMNAME%* for the non-transactional message pipe.

MSMQ OPERATION

As previously stated, MSMQ comprises several objects and components, of which the two fundamental parts are the message and the queue.

A message can contain text or binary data using any format, as long as both the sender and receiver agree on the format.

Queues hold the messages and there are two types of queue:

- Application queues, which are used by applications to send and receive data.
- System queues, which are created when MSMQ is installed, and include the dead letter queue, transactional dead letter queue, and journal queues.

The Message Queue Information Store (MQIS) is an SQL database that contains the definitions of the queues, and MSMQ topology. Active Directory Services replaces the MQIS in Windows 2000. The messages themselves exist in memory or in the file system.

Using a GUI tool, modifications can be carried out on the following:

- The MSMQ Enterprise.
- Servers.
- Connected networks.
- Queues.
- Messages.

The MSMQ Enterprise

The Enterprise is the top layer in the MSMQ hierarchy and all other objects are affiliated with an Enterprise. A 'right-click' on the Enterprise icon brings up the Enterprise Properties page with the tabs: General, MQIS Defaults, and Security. The General tab contains the Enterprise

name, Enterprise server, and Default lifetime of a message on the network. The MQIS Defaults tab contains the Replication interval – seconds between updates (external – inter-site; internal – intra-site). The Security tab provides buttons for setting permissions, auditing, and ownership.

Servers

The PEC (Primary Enterprise Controller) provides the infrastructure foundation for the MSMQ by holding the master copy of the MQIS. It can also serve as a PSC (Primary Site Controller), BSC (Backup Site Controller), or Routing Server.

A 'right-click' on the PEC computer icon allows you to bring up the Computer Properties page with the tabs: General, Network, Events, Status, IS Status, Dependent Clients, Tracking, and Security.

The General tab specifies the following:

- Pathname: the machine name prefix on the queue's physical location pathname.
- The Original Site: the site with which the machine was originally affiliated.
- The Site: permit machine relocation to another site.
- ID: GU-ID (Globally Unique Identifier) value automatically assigned to each object.
- Service: function of the computer.
- Limit Message Storage in Kilobytes and Limit Journal Storage in Kilobytes: this limits the bytes allocated for all messages in all queues but doesn't limit the size of each message. In NT, a message is limited to 4MB and in Win98 to 400K.

The Network tab contains buttons to Add, Edit, or Remove connected networks (CNs). A CN is a logical grouping of computers where any two can have a direct session with each other.

The Events tab provides a filtered listing of MSMQ-related events from the Event Viewer and includes a button to launch the full Event Viewer. To further isolate a problem, set up auditing using the security tab.

The Status tab provides MSMQ-related MQIS resource statistics from the Performance Monitor, such as sessions (IP and IPX), incoming messages/sec, MSMQ incoming messages, outgoing messages/sec, MSMQ outgoing messages, total messages in all queues, and total bytes in all queues.

A button is provided to launch the full Performance Monitor. The IS Status tab provides statistics on database replication. The Dependent Clients tab shows all dependent clients connected to the server – these clients have synchronous MSMQ sessions with the server from RPC applications.

With the Tracking tab, you can enable a queue and specify the types of message to track as they move through the MSMQ network. This is useful for checking that messages are reaching their destinations using the best route. The Security tab is the same throughout and was discussed earlier.

Connected Networks (CNs)

CNs map the protocol types running in your network and represent a logical group of computers where any two can establish a session with each other. For example, clients using IP would be in their own CN. You have the option to create an IP, IPX, or foreign CN. For example, you would create a foreign CN to connect to an MQSeries queueing system. 'Right-clicking' on a CN brings up a Connected Network Properties page with two tabs: General and Security. The General tab displays the CN name, protocol such as IP, and ID (GU-ID).

Queues

The queue is a logical representation of the physical data storage area for messages. 'Right-clicking' on a queue brings up the tabs: General, Advanced, Status, and Security. The General tab displays the following:

- A GU-ID for the queue object.
- An application-defined LABEL that can be used in combination with the Type-ID to aid in queue selection.
- A user-defined Type-ID, typically a GU-ID generated using **guidgen.exe** and used in combination with a LABEL to aid in queue selection.

- Queue creation date/time and queue last modified date/time.

The Advanced tab provides the ability to:

- Limit message storage in kilobytes for total message accumulation.
- Specify that the queue only accepts authenticated messages.
- Specify that the queue can be used to hold transactional messages.
- Store a copy of an outgoing message via the 'Journal-enabled' option.
- Limit journal storage in kilobytes to limit the accumulation of journal messages.
- Specify the privacy level, so that the queue accepts private (encrypted messages), non-private, or both.
- Specify a base priority of –32768 to +32767 for messages sent to a public queue. Queue priority takes precedence over message priority.

Messages

Message size is limited to 4 MB in NT and to 400 KB in Win98. 'Right-clicking' on a message brings up a Message Properties page with the tabs: General, Queues, Body, and Sender.

The General tab displays the following:

- The application-defined message label, which can be used to denote the purpose of the message, for example.
- The ID or message GU-ID created automatically when the message is placed into the queue.
- The application-defined priority level of between 0 to 7, with 0 as the default (with no priority, messages are processed first in, first out – FIFO).
- A tracked flag – indicating that the message is tracked.
- The MSMQ-set class or message type, such as normal, positive or negative acknowledgement (arrived and read), or report.

- Sent date and time for outgoing message.
- Arrived date and time for incoming message.

The Queues tab displays the destination queue name, the response queue name for reply messages, and the administration queue name for administration-type messages such as acknowledgements.

The Body tab displays the body content of the message and provides a byte count.

The Sender tab provides information on the sending computer and application, including the sending computer GU-ID, Pathname or name of the computer, the User ID and Security Identifier (SID) of the sending application, and an indication as to whether or not the message is authenticated and/or encrypted (and the algorithm(s) used).

MSMQ capabilities include:

- Integration with Windows, taking advantage of its security features, event/performance monitoring and logging, and transaction and clustering support.
- Easy administration through the GUI-based MSMQ Explorer.
- ActiveX components primarily for use with Visual Basic and in Active Server Pages (C++ and J++ are supported), and a C language API.
- Message and queue prioritization. Messages can be assigned a priority (0 to 7 – 7 is the highest) by the programmer and the receiving application will receive the messages in priority order. Queues can also be assigned a priority resulting in messages being routed based upon queue priority first, followed by message priority.
- Dynamic message routing. This is set by the administrator and is based upon site link costs allowing load balancing.
- Retention of a copy of the message until it's delivered successfully by the Microsoft Transaction Server (MTS). An application can manage a unit of work (UOW) to prevent data loss in the event of a failure. The MTS or DTC (Distributed Transaction Coordinator) coordinates the transaction.

Under syncpoint control, MTS can ensure safe delivery by retrieving a message, and writing the message at the destination. A syncpoint is the point where outstanding updates are made available and it works like a flag on a process thread. A UOW usually controls the transaction and MTS manages all the threads. When all the UOW threads are successful they are committed. If only one fails, then all the threads are stopped and the resources are rolled back to the state before the UOW was initiated.

MSMQ ARCHITECTURE

Four principal servers comprise the MSMQ infrastructure. These are described below.

- The Primary Enterprise Controller (PEC). This is the first element to be installed when building an MSMQ infrastructure. In the MQIS SQL database the PEC holds the enterprise configuration data and the certification keys used in authenticating messages.

  The MQIS contains the master definitions for the enterprise, site, site links, connected networks, user settings, and the master copy of each site's computers and queue definitions. In addition, the PEC contains a read-only copy of the MQIS data from other sites obtained by replication.

  The PEC can perform the jobs of the other three servers though this isn't advisable for obvious performance reasons.

- The Primary Site Controller (PSC). A site is typically a geographical location and a PSC should be installed at each site. For example, you could place a PEC at the company headquarters and one at each branch office location.

  The PSC MQIS database holds the master data about the computers and queues at the site. In addition, the PSC contains a read-only copy of the MQIS data from other sites obtained by replication. The PSC can perform the job of a Routing Server.

- The Backup Site Controller (BSC). A BSC should be installed for each site containing a PSC or PEC to provide load balancing and redundancy support in the event of failure.

The BSC holds a read-only copy of the MQIS database replicated from the PSC or PEC so no queues can be created. The BSC can perform the job of a Routing Server. A BSC requires a previously installed PSC or PEC.

- Routing Server (RS). You install RSs to provide more than one path for messages to reach their destination queues. As a result, RSs can distribute messages through different servers and prevent session concentration where you have too high a volume accessing a resource.

  RSs provide load balancing, intermediate store-and-forward message queueing, and dynamic routing. An RS doesn't hold the MQIS database and, therefore, requires a previously installed PEC.

When changes occur to the infrastructure, such as queue or computer changes, the MQIS is replicated. By default, intra-site replication for changes to be replicated within a site occur every two seconds while inter-site replication between sites occur every ten seconds.

There are two types of client supported: an independent client (IC), and a dependent client (DC). The programming interface is identical whether it's a client or server, but clients require fewer resources. The client software can be installed in Win98/95/ME and NT/2000 computers. IC software requires NT4 or better.

- DCs require a highly available network where the session is maintained with the MSMQ server. If the link is broken then the services of MSMQ are not available to the application.
- ICs can store messages in private queues, usually used as reply queues, which are not registered in the MQIS. These queues can only be addressed by their direct format name. A client application creates a private queue, sends the format name in the message so that the responding application can reply to the named queue. Unlike DCs, an IC's application will continue to run when the link is broken with the network. Any sent messages are just stored locally. (*Editor's note: this article will be continued next month.*)

Stephen Ibaraki, ISP
Chief Architect, iGen Knowledge Solutions (Canada)

The su facility

This article describes how to use the Unix platform's **su** facility to provide security and accountability of the MQ Administrator account.

All Unix systems provide a facility to log all attempts to become root. The good news is that you can also use this facility for other administrative accounts. This information can be invaluable when:

- Your business has one generic administrative account with more than one person using it. This can be useful when there was a mysterious system change at 5.00 am, for example, and you need to see who was logged into the administrative account at that time.
- You have developers who need administrative access to use *MQCONNX* or other trusted applications and you need to log their usage.
- You can track negative login attempts by hackers from inside or outside the company.

THE SULOG DISPLAY

Typically, messages logged in the sulog are written out along the following lines:

```
SU Ø6/17 12:37 - ttyp1 gabled-mqm
SU Ø6/17 12:38 + ttyp1 gabled-mqm
SU Ø6/25 1Ø:44 + ttyp2 oracle-gabled
SU Ø6/25 11:Ø5 + ttyp2 gabled-mqm
```

The log lists all uses of the **su** command, not those just used to su to an id. This can be seen when the Oracle account first su'd to *gabled* and then to *mqm*. As you may have guessed, the log uses positive signs for a successful login and negative signs for failed logins.

Table 1 lists the directories of the **su** log on some of the different flavours of Unix.

There follows a simple shell script that will e-mail you the id of anyone who tried unsuccessfully to login to the *mqm* account:

HP-UX 9	/usr/adm/sulog
HP-UX 10, 11,	/var/adm/sulog
Solaris	Listed in SULOG setting in /etc/default/su
AIX and IRIX	/var/adm/sulog
Digital Unix	/var/adm/sialog

Table 1: Directories of the su log on different Unix variations

```
#!/usr/bin/ksh
#Tested on HP-UX 11.0
SERVER=‘hostname‘
grep mqm /var/adm/sulog > su.test
grep  " - " su.test > su.test1
awk -F " " '{print $6}' su.test1 > su.test2
awk -F "-" '{print $1}' su.test2 > su.test3
elm -s "People who unsuccessfully tried to login to the mqm account on
   $SERVER"
mqm@yourcompany.com < su.test3
rm su.test
rm su.test1
rm su.test2
rm su.test3
```

INHIBITING DIRECT LOGINS

Now that you know where to look and what to look for, you'll have to block everyone from logging directly into the MQSeries administrative account and force them to use the **su** facility. Inserting this piece of code in the */etc/profile* (a system-wide login initialization file) is an easy way to do this.

```
LOG1='logname'
if [ $LOG1 = "mqm" ]
      then
      echo "You must use the su command from your own ID to access mqm."
        exit 1
fi
```

This entry in the */etc/profile* file will not only force usage of the **su** facility, but will send the person logging in an error message describing what action they should take next.

Paul Siracusa, Middleware Architect
Blue Cross Blue Shield of Missouri (USA)

MQ news

MQSoftware has announced the release of Version 2.3.1 of Q Pasa!, a management tool providing control of the MQSeries middleware environment including: configuration, performance, problem, operations, and analysis management.

New features in Version 2.3.1 include MQSeries object security, monitoring of OS/390 buffer manager statistics, expanded MQSeries Integrator Version 2 support, expanded message management support, and improved history reporting.

Separately, MQSoftware has also announced the release of Q Liner Version 1.6.5, which provides centralized administration and management of all enterprise file transfers from one location.

For further information contact:
MQSoftware, 7575 Golden Valley Road, Suite 140, Minneapolis, MN55427, USA
Tel: +1 763 546 9080
Fax: +1 763 546 9082
Web: http://www.mqsoftware.com

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

* * *

Candle Corporation has announced the release of the CandleMonitor message processing plug-in node. The new release is part of CandleNet Command Center for MQSeries Integrator Version 2.

CandleMonitor provides message flow performance and event monitoring for MQSI plug-in nodes, which, Candle claims, enables more effective management of applications.

Each node is deployed in message flows using standard MQSI control centre facilities. Performance statistics are gathered by placing the node at various points in a message flow.

Message flow events are produced automatically when the node detects a message flow exception, and events can be specifically configured by the user by assigning an event message attribute to a node placed in a failure path in a flow.

The CandleMonitor node is available for MQSeries Integrator Version 2 on Windows NT, Sun Solaris, and AIX.

For further information contact:
Candle Corporation, 201 N Douglas St, El Segundo, CA 90245, USA
Tel: +1 310 535 3600
Fax: +1 310 727 4287

Candle Service, 1 Archipelago, Lyon Way, Frimley, Camberley, Surrey, GU16 7ER, UK
Tel: +44 276 414 777
Fax: +44 1276 414 856

* * *