35

MQ

May 2002

In this issue

update

MQ Update

Published by

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

North American office

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

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/nfc.

MQ Update on-line

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

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.

Dead letter queue browser: MQSeries for OS/390

The requirements for running the enclosed DLQ Browser are OS/390, MQSeries, and MQSeries SupportPac MA18 (available from *http://www-4.ibm.com/software/ts/mqseries/txppacs/ma18.html)*. The program has been tested with OS/390 V2.8 and MQSeries for OS/390 V2.1.

INSTALLATION

To install DLQ Browser follow the steps listed below.

- Send the REXX code to your mainframe (ASCII mode in FTP or ASCII and CR/LF in Personal Communications file transfer).
- Store it as a library member (RECFM=FB, LRECL=80) named MQGETDLQ.

Now you are ready to use it! To do so:

- Read this article.
- Use the supplied job (after tailoring to your site's requirements) to run DLQ Browser against your dead letter queue.

USE

DLQ Browser reads every message on the dead letter queue (in browse mode) and prints information about why these messages were not delivered to their destination queues. The most common reasons for non-delivery of messages were discussed in the April 2002 issue of *MQ Update*.

I have written DLQ Browser in the REXX programming language. Since the standard MQSeries API is not available for REXX I used a support pack (MA18) which implements just that. REXX is relatively simple and self-commenting. Feel free to change the code as you wish.

Messages that are put to the dead letter queue have a special format, namely an additional header called MQDLH that is appended in front of the application data. The format of a message sent to the dead letter queue is set to MQFMT_DEAD_LETTER_HEADER.

RUNNING DLQ BROWSER

DLQ Browser can be invoked either from the supplied job or run from the TSO or ISPF line command. The second option is not recommended since, depending on the number of messages on the dead letter queue, the output from DLQ Browser can be huge.

The job to run DLQ Browser is shown and explained below.

```
//MQGETDLQ JOB NOTIFY=&SYSUID
//* parameters in SYSTSIN:
//*  QMgrName
//*  DLQName
//GETDLQ    EXEC PGM=IKJEFT01,
//          PARM='%MQGETDLQ'
//STEPLIB   DD DSN=MA18.LOAD,DISP=SHR
//          DD DSN=MQM.SCSQAUTH,DISP=SHR
//SYSEXEC   DD DSN=YOUR.DSN,DISP=SHR
//SYSTSPRT  DD SYSOUT=*
//SYSTSIN   DD *
QmgrName
DLQName
/*
```

Tailoring the above code

- Line 1: supply a valid job card.
- Line 7: MA18 load library.
- Line 8: MQSeries hlq.SCSQAUTH library.
- Line 9: library containing the MQGETDLQ source.
- Line 10: a class for sysout (DLQ Browser reports will be printed there).
- Lines 11 and 12: parameters for DLQ Browser:
  - queue manager name (required)
  - dead letter queue name.
- The parameters for DLQ Browser must be given in the order shown; the dead letter queue name may be omitted (leave a blank line instead).
- The queue manager name is required – it has no default.

Default values for DLQ Browser parameters

- QMgr – none.
- DLQName – none. If it is blank, DLQ Browser will query the queue manager for it (it is stored in a queue manager attribute 'DEADQ'); if it cannot be obtained DLQ Browser will quit.

OUTPUT

DLQ Browser will produce a report for every message it finds on the dead letter queue. The following information will be given.

- The application data part of the message – the first 200 bytes. If you think this is too small an amount you can change it in the code: go to line three and change the value of the MaxMsgLen variable. Remember, however, that it must be at least 172 bytes since this is the MQDLH header length.
- The message descriptor MQMD. I will not describe here the full message descriptor (please refer to *MQSeries Application Programming Reference*) but several fields need a comment since they are subject to change when the message is put to the dead letter queue.
  - Format – this should always be set to MQFMT_DEAD_LETTER_HEADER. If the field has another value it is an error. It cannot be processed by a dead letter queue handler and may, therefore, stay in the dead letter queue indefinitely.
  - Encoding – this is the encoding used by the application that put the message to the dead letter queue.
  - CodedCharSetId – this is the character set identifier used by the application that put the message to the dead letter queue.
- Message descriptor context fields (eg PutApplType, PutApplName, PutDate, PutTime).
- The contents of the dead letter header MQDLH. The fields are listed below.
  - Reason – the reason why the message has been put to the dead letter queue.

- DestQName – destination queue the message was meant to go to.
- DestQMgrName – destination queue manager the message was meant to go to.
- Encoding – encoding of numeric data in the application data part of the message.
- CodedCharSetId – character set identifier of data in the application data part of the message.
- Format – format name of data in the application data part of the message.
- PutApplType – type of application that put the message on the dead letter queue.
- PutApplName – name of the application that put the message on the dead letter queue.
- PutDate – date when the message arrived on the dead letter queue.
- PutTime – time when the message arrived on the dead letter queue.

After browsing all messages on the dead letter queue DLQ Browser will print their number and quit. You should now be aware of the reasons why messages arrive on your dead letter queue and how to use a tool such as DLQ Handler to get rid of them.

DLQ BROWSER

```
MaxMsgLen = 200 + 172                  /* 172 is the MQDLH length */
PARSE EXTERNAL qmgr
PARSE EXTERNAL inqueue
/* Initialize the interface */
RXMQVTRACE = ''
rcc= RXMQV('INIT')
SAY 'RC = ' rcc
/* Connect to Queue Manager */
RXMQVTRACE = ''
rcc = RXMQV('CONN', qmgr)
SAY 'RC = ' rcc
IF WORD(rcc, 5) <> 'FAILED' THEN /* Connect OK */
DO
```

```
IF inqueue = '' THEN
DO /* Ask the queue manager for DLQ name */
  SAY 'DLQ name not supplied'
  SAY 'Getting DLQ name from the queue manager'
  RXMQVTRACE = ''
  od.OT = MQOT_Q_MGR
  oo = MQOO_INQUIRE
  rcc = RXMQV('OPEN', 'od.', oo , 'h1', 'ood.' )
  rcc = RXMQV('INQ', h1, MQCA_DEAD_LETTER_Q_NAME, 'InputQName')
  INQrcc = rcc
  rcc = RXMQV('CLOSE', h1, co)
  IF WORD(INQrcc, 1) <> Ø ! inqueue = '' THEN
  DO
    SAY 'DLQ name not known to the queue manager'
    SAY 'Quitting DLQ Browser'
    RXMQVTRACE = ''
    rcc = RXMQV('DISC')
    SAY 'RC = 'rcc
    RETURN
  END
END /* Ask the queue manager for DLQ name */
/* Open Queue for Input and Inquiry */
RXMQVTRACE = ''
oo = MQOO_BROWSE + MQOO_INQUIRE
rcc = RXMQV('OPEN', inqueue, oo , 'h2', 'ood.' )
SAY 'RC = ' rcc
IF WORD(rcc, 5) <> 'FAILED' THEN /* Open OK */
DO
  /* Get Current Queue Depth */
  RXMQVTRACE = ''
  rcc = RXMQV('INQ', h2, MQIA_CURRENT_Q_DEPTH, 'depth')
  SAY 'RC = ' rcc
  /* Browse messages                    */
  RXMQVTRACE = ''
  DO i=1 TO depth
     g.Ø      = MaxMsgLen
     g.1      = ''
     igmo.opt = MQGMO_ACCEPT_TRUNCATED_MSG + MQGMO_BROWSE_NEXT
     rcc = RXMQV('GET', h2,'g.','igmd.','ogmd.','igmo.','ogmo.')
     SAY 'RC = ' rcc
     IF ( WORD(rcc,1) = 2Ø33 ) THEN LEAVE
     SAY 'Data: <'x.1'>'
     SAY '   Message Descriptor: '
     SAY '     MsgId:            'ogmd.msgid
     SAY '     CorelId:          'ogmd.cid
     SAY '     Report:           'ogmd.rep
     SAY '     MsgType:          'ogmd.msg
     SAY '     Expiry:           'ogmd.exp
     SAY '     Feedback:         'ogmd.fbk
     SAY '     Encoding:         'ogmd.enc
     SAY '     CodedCharSetId:   'ogmd.ccsi
     SAY '     Format:           'ogmd.form
```

```
        SAY '     Priority:         'ogmd.pri
        SAY '     Persistence:      'ogmd.per
        SAY '     BackoutCount:     'ogmd.bc
        SAY '     ReplyToQ:         'ogmd.rtoq
        SAY '     ReplyToQMgr:      'ogmd.rtoqm
        SAY '     UserId:           'ogmd.uid
        SAY '     AccountingToken:  'ogmd.at
        SAY '     ApplIdentityData: 'ogmd.aid
        SAY '     PutApplType:      'ogmd.pat
        SAY '     PutApplName:      'ogmd.pan
        SAY '     PutDate:          'ogmd.pd
        SAY '     PutTime:          'ogmd.pt
        SAY '     ApplOriginData:   'ogmd.aod
        SAY
        /* Extract the dead letter header  */
        rcc = RXMQV('HXT', 'g.', 'x.')
        SAY 'RC = ' rcc
        IF WORD(rcc, 1) = 0 THEN /* Dead letter header ok */
        DO
          SAY 'Dead letter header:'
          SAY 'Type:        'x.TYPE
          SAY 'Reason:      'x.REA
          SAY 'DestQ:       'x.DQN
          SAY 'DestQMgr:    'x.DQM
          SAY 'Format:      'x.FORM
          SAY 'Encodig:     'x.ENC
          SAY 'CCSId:       'x.CCSI
          SAY 'PutApplType: 'x.PAT
          SAY 'PutApplName: 'x.PAN
          SAY 'PutDate:     'x.PD
          SAY 'PutTime:     'x.PT
      END /* Dead letter header OK */
    END /* DO i=1 TO depth*/
    SAY 'There are 'depth' messages on the dead letter queue'
    /* Stop access to a Queue */
    RXMQVTRACE = ''
    rcc = RXMQV('CLOSE', h2, mqco_none)
    SAY 'RC = ' rcc
  END /* Open OK */
  /* Disconnect from the QM */
  RXMQVTRACE = ''
  rcc = RXMQV('DISC', )
  SAY 'RC = ' rcc
END /* Connect OK */
/* Remove the Interface functions from the Rexx Workspace ... */
RXMQVTRACE = 'TERM'
rcc = RXMQV('TERM', )
SAY 'RC = ' rcc
RETURN
```

Global transactions with MQSeries and Oracle, part 1: specifications

In this first part of an article on using MQSeries and Oracle in global transactions we look at the X/Open DTP model and the XA and TX specifications for global transactions. We also show how MQSeries and Oracle implement these two specifications. The article concludes in next month's issue of *MQ Update*.

X/OPEN CAE XA AND TX SPECIFICATIONS

The importance of preserving the consistency of data in global transactions demanded an industry standard that would define the mechanisms which would ensure the atomicity, consistency, isolation, and durability (known as the ACID qualitities) of global transactions. The most widely accepted model today was introduced by the X/Open organization during the early 1990s. X/Open is an independent organization that specializes in selecting and adopting open system standards. It is supported by the world's largest companies and organizations operating in the software industry. The strategy of X/Open is to merge the adopted standards in what is called the CAE or Common Application Environment. The CAE is an open-system environment model that is intended to ease the processes of application integration and portability.

X/Open defines a model for describing global transaction environments. The model is called the DTP (Distributed Transaction Processing) model and it consists of the following components:

- User Application.
- Resource Manager (one or more).
- Transaction Monitor.

The User Application (AP) is an application which accesses data resources and defines boundaries for the global transactions. For example, an AP can be a program that connects to an MQSeries queue manager and Oracle database manager, reads a message from an MQSeries queue, and stores a record in an Oracle database table. The AP always specifies when a transaction starts and when and how it ends.

The Resource Manager (RM) is an application which provides access to a shared data resource. This can, for example, be a file system manager, a queue manager, or a database manager. RMs manage the local transactions on their resources.

The Transaction Manager (TM) is the central component of the DTP model. It is an application that manages global transactions. It communicates with other components in the model to enable them to participate in the global transactions. It assigns identifiers to the global transactions and makes sure that all components in the DTP model remain synchronized and aware of the progress of global transactions and their outcome. The TM ensures the atomicity of the global transactions and provides the mechanisms for data recovery in the case of failure of any component in the model: the AP, any of the RMs, or the TM itself.

X/Open adopted two specifications to describe two programming interfaces in the DTP model. These specifications declare the functions that need to be implemented by the RM and TM vendors. The specifications also describe how to use these functions to ensure the ACID properties of the global transactions.

The TX interface is a uni-directional interface between the AP and the TM. The functions in this interface are implemented by the TM and invoked by the AP. The major role of this interface is to enable the AP explicitly to start and end a global transaction in the DTP environment. The TX interface consists of eight functions. Their names and meanings are listed here.

- tx_open() – connects to the TM.
- tx_begin() – starts a new global transaction.
- tx_commit() – commits the global transaction.
- tx_rollback() – rolls back the global transaction.
- tx_close() – disconnects from the TM.
- tx_set_transaction_timeout() – sets the timeout value in seconds for a global transaction: this is the maximum time specification (in seconds) between the tx_begin() and tx_commit() or tx_rollback() during which the transaction is considered alive.

- tx_set_commit_return() – defines the moment when the tx_commit returns to the AP. This can be either after the transaction is fully committed or after the decision about the commit was made but not yet completed.
- tx_set_transaction_control() – specifies whether a new global transaction is always started explicitly with tx_begin() or explicitly only the first time and then implicitly after every tx_commit() and tx_rollback().

The XA interface is a bi-directional interface between the TM and an RM. The major role of this interface is to enable the TM to inform an RM how to act in a global transaction as well as to get the information about the internal RM status regarding a particular interrupted transaction during the TM recovery process.

The XA interface consists of two functions implemented by the TM and ten functions implemented by the RM. Their names and meanings are listed here.

Functions implemented by the TM and invoked by the RMs:

- ax_reg() – dynamically register RM for a global transaction
- ax_unreg() – dynamically deregister RM for a global transaction.

Functions implemented by the RMs and invoked by the TM:

- xa_open() – connect to the RM
- xa_start() – inform the RM of the start of a global transaction
- xa_end() – inform the RM that the global transaction is ending
- xa_prepare() – prepare to commit the data operations from a global transaction that belong to the RM
- xa_commit() – commit the data operations from a global transaction that belong to the RM
- xa_rollback() – rollback the data operations from a global transaction that belong to the RM
- xa_close() – disconnect from the RM
- xa_complete() – wait for the definite completion of a previously started asynchronous operation

- xa_forget() – instruct the RM to revert the heuristically completed operations on the RM. An RM can heuristically assume that the global transaction will eventually be committed and it can commit the data operations on its resource upfront, but it must provide the xa_forget() function to revert these operations.
- xa_recover() – obtains the list of global transactions for which the outcome was still not reported to the RM (for example, during the recovery process after the system crashes in the middle of a global transaction and restart).

The DTP model is displayed in Figure 1. Two RMs are displayed: a queue manager and a database manager. Both RMs participate in the same global transaction.

The AP calls tx_open() to connect to the TM [1]. In response to this the TM connects to the RMs that participate in the global transaction. The TM does this by sequentially calling xa_open() on all the RMs [2], [3]. The first argument of the xa_open() call is a string called XA open string. Every RM defines the format of its XA open string and the TM must be aware of this format prior to calling xa_open(). XA open string defines parameters of a global transaction that the RM needs in order to manage its data operations on behalf of the global transaction.

The AP then starts a new global transaction by invoking tx_begin() on the TM[4]. The TM initializes a new global transaction and assigns an identifier to it that is called the XID identifier. XID is actually a data structure that uniquely identifies the global transactions in the system. The TM then sequentially calls xa_start() on every RM to inform it that a new global transaction was started on behalf of the AP[5][6]. Importantly, xa_start() must be called from the same TOC (Thread Of Control) from which the AP previously called tx_begin(). A thread of control can be either a system process or a system thread. What is important is that the TM and the RMs must interpret the TOC in the same way and that the AP must be aware of what the agreed interpretation for the TOC is. The RM makes a note of the new global transaction and stores the assigned XID identifier and the TOC identifier (process ID or thread ID) in the internal transaction tables.

After the global transaction has been started the AP uses the native interfaces of the RMs to perform data operations on the corresponding

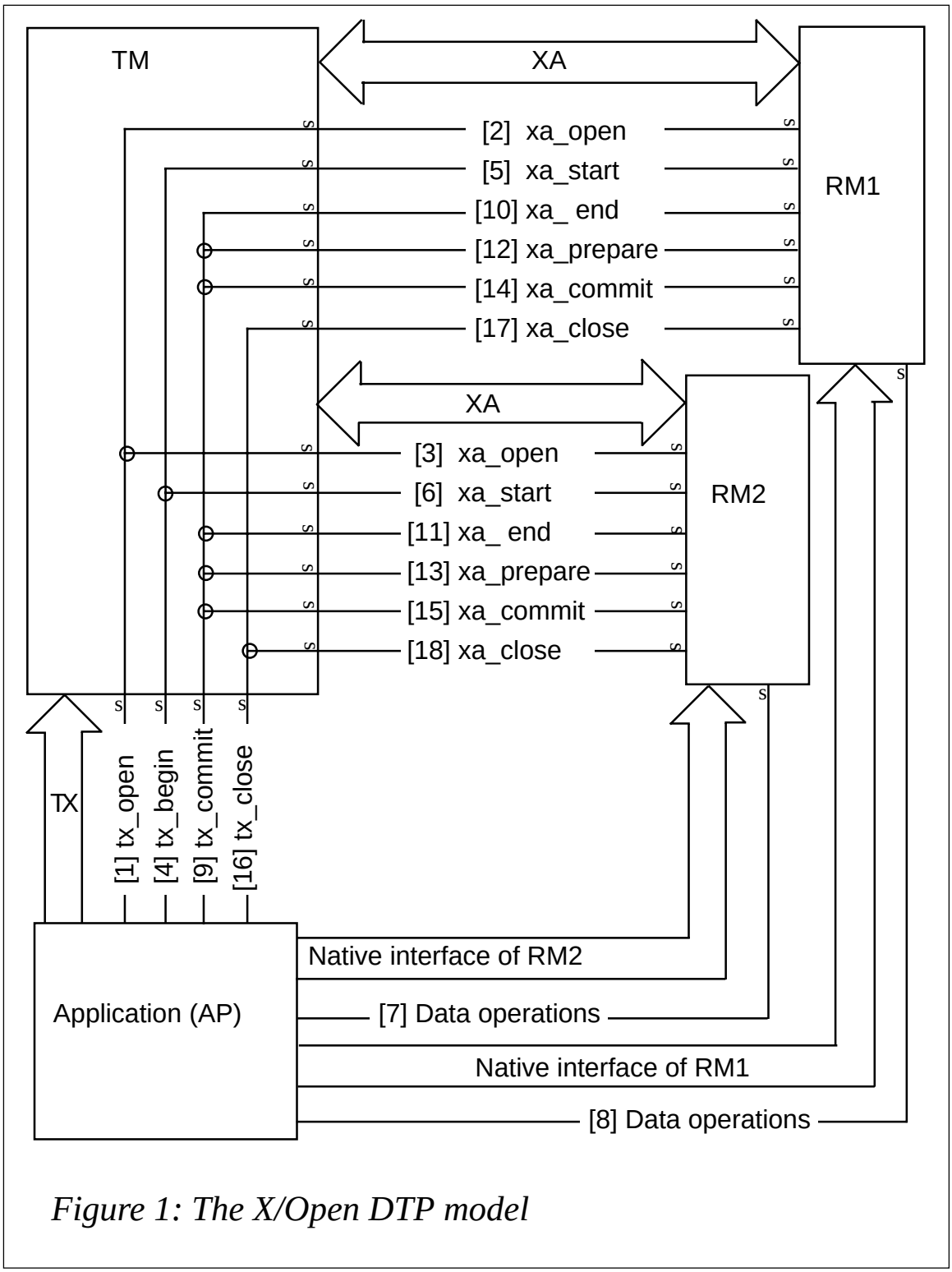

Figure 1: The X/Open DTP model

resources [7], [8]. These operations are for example: getting messages from and putting messages to the queues, and inserting, updating, and deleting data in the database tables. What is important is that the AP makes calls to the RMs from the same TOC from which it previously

called tx_begin(), that is from which the TM called xa_open() on the RMs. This enables the RMs to recognize that the operations are being done on behalf of the previously started global transaction.

When the AP calls an RM that RM checks the TOC from which the call was made and then looks in the internal transaction tables to see if a global transaction was started on behalf of the AP from that same TOC. When the RM recognizes that this has happened it performs the required operations and associates them with the corresponding global transaction.

When the AP decides to end the transaction it calls tx_commit() or tx_rollback() on the TM (depending on whether the transaction needs to be committed or rolled back[9]). The TM first informs the RMs that the transaction is ending by sequentially calling xa_end() on them [10], [11].

If the transaction has to be rolled back the TM logs this decision and then sequentially calls the RMs to inform them that they should roll back all the work that was performed on them on behalf of the global transaction.

If the AP requires that the transaction be committed the TM initiates the 2PC (two-phase commit protocol) to ensure that the data operations in the transaction are either committed in all data resources or that they are rolled back in all of them. The protocol begins with the TM sequentially calling xa_prepare() on all the RMs [12], [13]. An RM checks the system resources to ensure that its data operations can be committed. If an RM returns success when called on xa_prepare() it guarantees that it will be able to commit the transaction when later asked to do so. If any RM returns an error when called on xa_prepare() the TM decides to roll back the transaction. It stores this decision in the log file and then sequentially calls xa_rollback() on every RM to instruct it to roll back all the operations that were done on it on behalf of the global transaction.

If all the RMs return successfully from xa_prepare() the TM decides to commit the transaction and stores this decision in the log files. It then performs the second 2PC phase and asks every RM to commit the local data operations that were done on behalf of the global transaction by sequentially calling xa_commit() [14], [15].

When the tx_commit() call completes, the global transaction is also complete and all the results are made permanent.

After this the AP may initiate a new global transaction. The new transaction is started either implicitly or explicitly. This behaviour is controlled by the tx_set_transaction_control() function that sets the TM in chained or unchained mode of operation. If the TM is in the chained mode a new transaction automatically starts whenever the previous transaction completes. If the TM is working in the unchained mode a new transaction must be explicitly started every time with the tx_begin() call.

Before the AP completes and exits it calls tx_close() to disconnect from the TM[16]. The TM disconnects from the RMs by sequentially calling xa_close() on all of them [17], [18].

MQSERIES AND ORACLE IMPLEMENTATIONS OF THE XA AND TX INTERFACES

An MQSeries queue manager may be configured to work as a TM capable of managing global transactions. At the same time it acts as an RM that manages the message operations performed on its queues. The XA communication between the TX component and the RM component of the MQSeries queue manager is done internally in MQSeries and is not made obvious to the outside world. What happens behind the scenes is that the MQSeries queue manager still manages its local transactions except that the local transactions are in this case enhanced to include operations on other data resources, such as databases.

Figure 2 displays the function calls that are exchanged among the AP, the MQSeries TM, and the Oracle RM when an MQSeries queue manager serves as a TM that manages two RMs: the local queue manager and the Oracle database manager.

The MQSeries RM is not displayed as a separate component because the MQSeries TM manages MQSeries RM operations internally. The Oracle RM is an independent component that may reside on the same machine as MQSeries but can also run on a separate machine in the network.

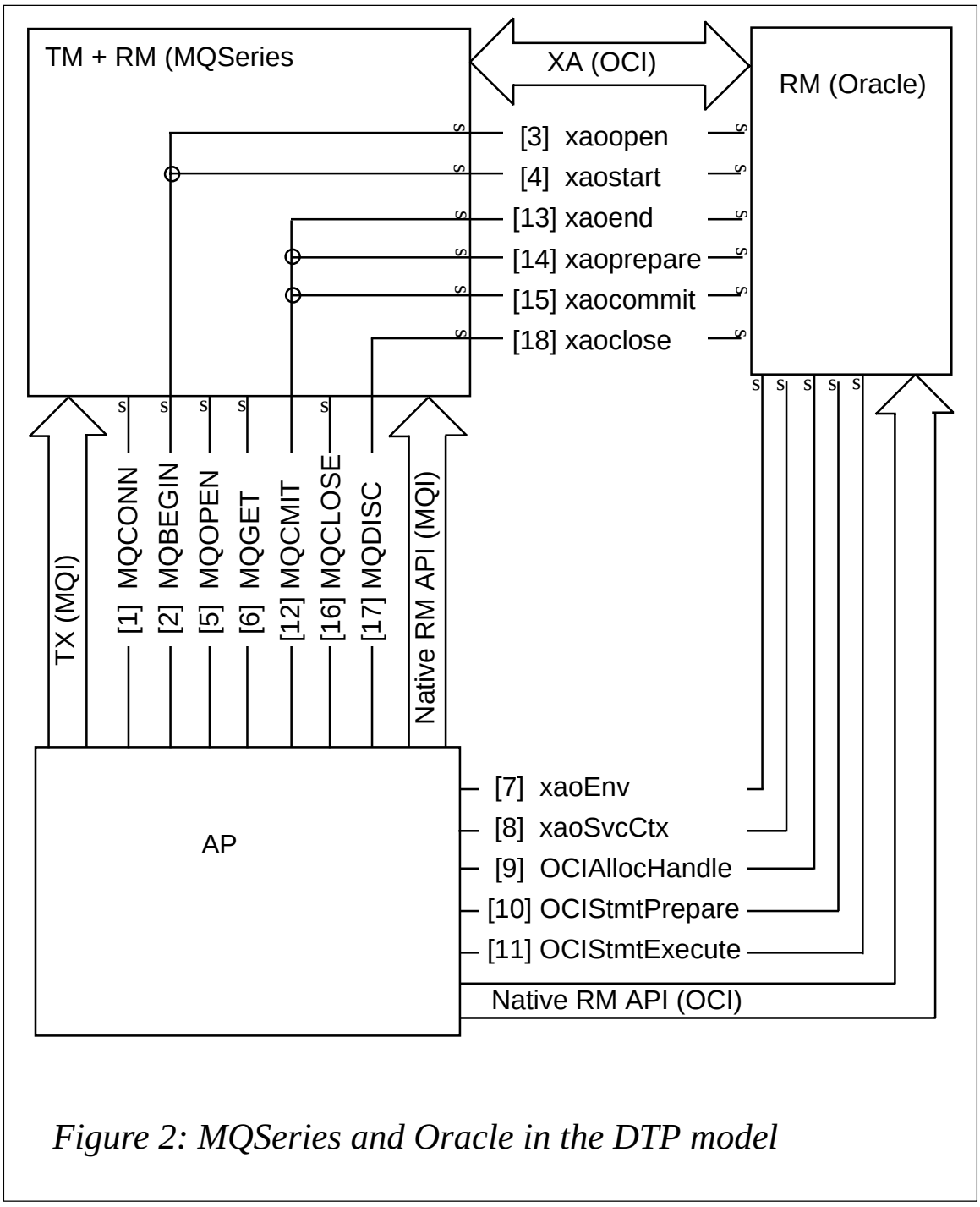

Figure 2: MQSeries and Oracle in the DTP model

The AP first connects to the MQSeries TM with the MQCONN() call [1]. When MQCONN() completes, the AP obtains a connection handle to the queue manager and a new local transaction automatically starts on that queue manager.

The AP then initiates a global transaction by invoking MQBEGIN() [2]. This call changes the previously started local transaction to a

global transaction. The MQBEGIN() function plays the role of the previously described tx_open() and tx_begin() TX functions. For the first MQBEGIN() call that the AP makes to the TM the TM checks whether there are any XAResourceManager stanzas specified on that queue manager. If not, the TM returns a warning message, MQRC_NO_EXTERNAL_PARTICIPANTS, which means that the global transaction that was started with MQBEGIN() remains a local transaction on that particular queue manager. If there are XAResourceManager stanzas specified, the TM loads the XA SLFs that provide the addresses of the XA functions that the TM calls on the RMs. In this example only one XAResourceManager stanza is specified for the Oracle RM. The TM loads the Oracle XA SLF and calls the xa_open() function to connect to the RM. Oracle implements the xa_open() as xaoopen() [3].

After the connection to the Oracle RM is established, the MQSeries TM informs the Oracle RM that a new transaction has been started. It does this by calling the xa_start() function on the Oracle RM that Oracle implements as xaostart() [4].

The Oracle RM acknowledges the presence of a new global transaction and remembers the TOC from which the xa_start() call was made. It is the same TOC from which the AP called MQBEGIN() because the xaostart() call is made from the MQBEGIN() function (xa_start() call is part of the MQBEGIN() implementation). The TOC can be either a process ID or a combination of a process ID and a thread ID. This depends on what was chosen for a TOC for the Oracle RM. This is described in more detail later, in the section on multi-threading.

The AP now opens a queue with MQOPEN() [5] and receives a message from the queue with MQGET() [6]. The AP uses the connection handle obtained in [1] when it opens the queue to receive the message. All MQSeries calls to the queue manager have to be made from the same thread from which the connection to that queue manager was established. The TM knows that the MQGET() operation belongs to the global transaction that was started in [2] because the queue manager handle is the same one that was used for MQBEGIN().

The AP can also call the Oracle RM. Before the AP can perform any operation on the Oracle database, it needs to obtain the environment handle and the service context handle for the XA connection. The

Oracle XA library provides two functions that are used to obtain these handles. These are xaoEnv() [7] and xaoSvcCtx() [8]. When the AP calls these functions, Oracle checks the identifier of the TOC from which these calls was made. It then searches its internal system tables to see if a global transaction was previously started from that same TOC. If it was, the Oracle RM provides the AP with the environment handle (from xaoEnv()) and the service context handle (from xaoSvcCtx()) of the XA connection – the connection that the MQSeries TM established to the Oracle RM. Later, when the AP invokes data operations on the Oracle RM using these two handles, the Oracle RM will know that those operations must be associated with the previously started global transaction and not with a local Oracle transaction. The AP uses these handles to allocate SQL statement handles, to prepare the statements, and finally to execute them [9][10][11]. For example, the AP can insert a record in the Oracle database as a part of the global transaction.

The AP commits the previously started global transaction by invoking MQCMIT() on the MQSeries TM [12]. MQCMIT() is the MQSeries TM implementation of the tx_commit() TX function. The TM now has to make sure that the atomicity of the global transaction is preserved. It does this by tracing all the steps in the global transaction completion process.

The TM first informs the Oracle RM that the transaction is ending by calling the xa_end() function that Oracle implements as xaoend() [13]. The xa_end() call is made from the same TOC from which the xa_start() was made. The Oracle RM knows which global transaction is ending and which local operations on the database were done up to that moment on behalf of that global transaction.

The MQSeries TM initiates the 2PC protocol with the Oracle RM. It first calls the xa_prepare() XA function that Oracle implements as xaoprepare() [14]. The Oracle RM makes sure that it has sufficient system resources to commit the data operations performed on behalf of the transaction. If it returns success to the MQSeries TM, it guarantees that it will commit the data operations. The MQSeries TM/RM also ensures internally that it can commit the operations on its queues. The TM then calls xa_commit() (which Oracle implements as xaocommit()) to inform the Oracle RM to commit the local data

operations that were performed on behalf of the global transaction [15]. The MQSeries TM/RM also makes sure internally that it commits the message operations performed on the local queues. When the MQCMIT() completes, the performed MQSeries and Oracle operations are made permanent.

The AP may choose to perform another transaction by issuing another MQBEGIN() call. Note that the MQSeries TM does not implement the tx_set_transaction_control() TX function that defines whether the transactions are chained or unchained. The MQSeries TM supports only the unchained mode of work, which means that every new global transaction must be explicitly started with a new MQBEGIN() call.

When the AP calls MQDISC [17] it disconnects from the TM. This is equivalent to the tx_close() function that is defined in the TX interface. From the tx_close() call the TM closes the Oracle RM by calling xa_close(). Oracle implements xa_close() as xaoclose() [18].

This article concludes in next month's issue of *MQ Update*.

Predrag Maksimovic
Software Engineer, 2d3D Incorporated (USA)

Using the MQAI to administer WebSphere MQ

There are various ways to administer MQSeries using utilities that either run in the background or are written for a human operator. Traditionally, the facilities of PCF (Programmable Command Formats) have been used and this is available on all platforms except OS/390 (z/OS).

An example of PCF can be found by downloading SupportPac MS02 from *http://www-4.ibm.com/software/ts/mqseries/txppacs/ms02.html.*

The basic concept is based upon creating the appropriate MQSeries message with the correct PCF header and sending it to the *SYSTEM.ADMIN.COMMAND.QUEUE*. This queue is read by the MQSeries command processor and any responses are written back to the 'reply-to-queue' specified on the original PCF message.

Additional SupportPacs using PCF are MS03 (Saveqmgr) and MS0B (Java classes for PCF).

If you have never used PCF then at first sight it can appear to be quite complicated. IBM has, therefore, created an interface that – in my opinion – simplifies the process. It is called the MQSeries Administration Interface (MQAI) and has been available since V5.1 of MQSeries. The following IBM manuals are a good source of information:

- *Administration Interface Programming Guide and Reference SC34-5390-01*.
- *Programmable System Management SC33-1482-08.*

My personal experience has shown that the best way to learn something is to actually do it. A good starting place on the Windows NT/2000 platform for the MQAI can be found in the directory *<install directory>\tools\c\samples*. There are three samples:

- **amqsaicq** – an example of how to create a local queue.
- **amqsaiem** – demonstrates a way of getting messages from an event queue.
- **amqsailq** – lists all local queues with their depth.

These samples – as well as the included program – can be easily compiled and linked using MQ Visual C++ V6: make sure you create a Win32 console application and include file *mqm.lib* (it's a server program) within the 'link' tab of the project settings. Once linked all programs are immediately available for testing.

MQAI STRUCTURE

Instead of: creating messages with the PCF header, allocating and deleting storage, dealing with the various PCF structures and arrays as well as directly putting messages to the command server, and dealing with its responses – the MQAI introduces the concepts of 'data bags'.

The application program calls the MQAI interface to create the data bags and populates the bags with 'data items'. The items define the object to be manipulated (inquiry, update, or deletion) as well as its

attributes. The MQAI changes the bags into PCF messages and passes them on to the command server. The response from the command server is changed from PCF into 'bag' format ready to be retrieved by the application program. Figure 1 provides a conceptual view of the way in which data is stored.

TYPES OF DATA BAG

A bag is created using the *mqCreateBag* program. At the end of processing all bags should be deleted using the *mqDeleteBag* program. The types of bag available are listed below.

- User – can be used to transfer user type data between applications:
  - field name: MQCBO_USER_BAG.
- Admin – used to administer MQSeries by sending administration messages in PCF format to the command server. Typically, an application program creates two bags, one for the request and one for the replies:

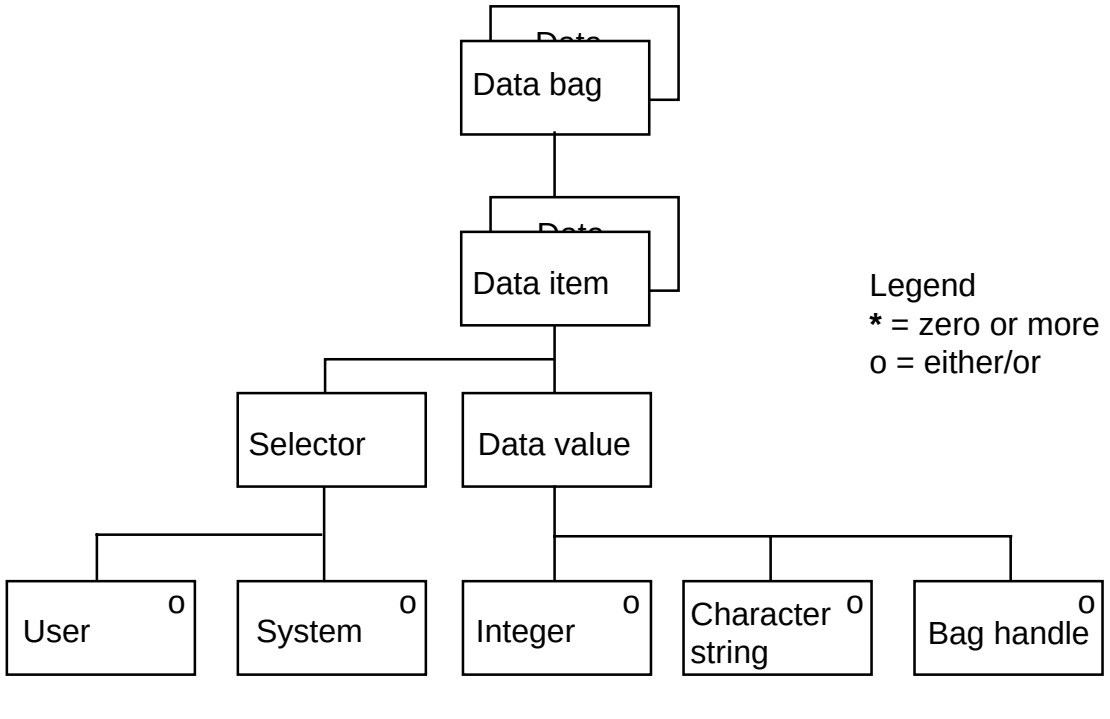

Figure 1: MQAI data bag structure

  - field name: MQCBO_ADMIN_BAG.

- Command – contains commands for administering MQSeries objects:
  - field name: MQCBO_COMMAND_BAG.
- System – automatically created for a reply message whose contents are placed into a user's output bag. The user cannot change a system bag:
  - field name: MQCBO_SYSTEM_BAG.

TYPES OF DATA ITEM

Items exist inside a data bag. They are created using the *mqAddInteger*, *mqAddString*, or *mqAddInquiry* programs. The trick is to know which one to use.

One of the parameters is the 'selector', which identifies the object. For example:

- To identify a queue name (a character string) use *MQCA_Q_NAME*.
- To identify the type of queue (of type integer) use *MQIA_Q_TYPE*.
- For a list of selectors see the header files *cmqc.h*, *cmqbc.h*, and *cmqcfc.h*.
- If the selector is of type 'character' use *mqAddString*; otherwise, use *mqAddInteger*.

Both *mqAddString* and *mqAddInteger* allow the programmer to further qualify (ie filter) the object(s) to be processed. For example, to process all queues specify:

```
mqAddString(adminBag, MQCA_Q_NAME, MQBL_NULL_TERMINATED,
"*", &compCode, &reason);
```

(Here the generic value '*' has been used. It is possible to specify an exact name or a partial name like 'AB*' for all queues starting with AB; you cannot specify *AB.)

To ensure that only locally defined remote queues are processed specify:

```
mqAddInteger(adminBag, MQIA_Q_TYPE, MQQT_REMOTE,
&compCode, &reason);
```

For those attributes which cannot be further qualified use the *mqAddInquiry* program.

For example, to pick up the queue descriptions specify:

```
mqAddInquiry(adminBag, MQCA_Q_DESC, &compCode, &reason);
```

Process the message

Instead of writing to the command queue using MQPUT and getting a response back using MQGET, use the **mqExecute** command. This command automatically waits for up to 30 seconds for a response. The response itself is placed in the 'response admin' bag. Make sure the correct command code is used – a list of command codes can be found in the *cmqcfc.h* header file. For example, to list queues, use **MQCMD_INQUIRE_Q**.

Retrieving the response

The response from the command server is automatically placed in a special system response bag. If there were no errors the results are copied to the user bags. First check the number of bags returned (which can be zero) from the original request, eg:

```
mqCountItems(responseBag, MQHA_BAG_HANDLE, &numberOfBags,
&compCode, &reason);
```

Now that the number of bags is known (numberOfBags) loop round and retrieve the objects and their attributes.

Programming example

The following program listing (**listchl.c**) contains numbered notes. Explanations for each note follow the code. The program was originally based on sample **amqsailq** (list local queues) but has been changed to display channel information with related fields; it is able to administer a remote queue manager via a local (default) queue manager.

LISTCHL.C

```
/* Program name: listchl.C                                          */
/* History:       January 2002 - created by Ruud van Zundert        */
```

```
/* Description:  Example C program to inquire on the channel status   */
/*               attributes of a local queue manager using the        */
/*               MQSeries Administration Interface (MQAI):            */
/*               channel name, status, curmsgs, msgs, shortrts,       */
/*               indoubt, lstmsgti.                                   */
/*               Caters for both local and remote administration.     */
/*               Displays either all channel statuses, or only those  */
/*               with status 'running' or 'not running'.              */
/* Parameters:   Parm1: the local queue manager name (optional)       */
/*               Parm2: the remote queue manager name (could be same  */
/*                      as parm1)                                     */
/*               Parm3: channel status filter: r (only running),      */
*/                     n (not running  other (all)                   */
/* Includes                                */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <cmqc.h>                  /* MQI                             */
#include <cmqcfc.h>                /* PCF                             */
#include <cmqbc.h>                 /* MQAI                            */
/* Function prototypes                                                */
void CheckCallResult(MQCHAR *, MQLONG , MQLONG);
/* Function: main                                                     */
int main(int argc, char *argv[])
{
/* -------------- for remote administration only -----------------    */
MQOD     od = {MQOD_DEFAULT};       /* Object Descriptor               */
   MQHOBJ   Hobj;                   /* object handle                   */
   MQLONG   O_options;              /* MQOPEN options                  */
/* ----------------------- MQAI variables -------------------------  */
MQHCONN hConn;                     /* handle to MQ connection         */
MQCHAR qmName[MQ_Q_MGR_NAME_LENGTH+1]=""; /* default or supplied QMgr
name */
MQLONG reason;                     /* reason code                     */
MQLONG connReason;                 /* MQCONN reason code              */
MQLONG compCode;                   /* completion code                 */
MQHBAG adminBag = MQHB_UNUSABLE_HBAG;  /* admin bag for mqExecute     */
MQHBAG responseBag = MQHB_UNUSABLE_HBAG;/* response bag for mqExecute */
MQHBAG chlAttrsBag;                /* bag containing q attributes     */
MQHBAG errorBag;                   /* bag containing cmd server error*/
MQLONG mqExecuteCC;                /* mqExecute completion code       */
MQLONG mqExecuteRC;                /* mqExecute reason code           */
MQLONG actLength;                  /* Actual length of parameter      */
MQLONG i;                          /* loop counter                    */
MQLONG numberOfBags;               /* number of bags in response bag */
MQLONG qHandle;                    /* either MQHO_NONE or real handle*/
MQCHAR staQual[2];                 /* chl status  qualifier/filter    */
MQLONG ChlType,CurMsgs, Msgs, ShortRts, InDoubt, ChStat;
MQCHAR ChlName[MQ_CHANNEL_NAME_LENGTH+1]; /* channel name             */
MQCHAR LastTime[MQ_CHANNEL_TIME_LENGTH+1]; /* last message time       */
```

```
MQCHAR wChlStat[13], wChlType[9];   /* channel status and type        */
printf("listchl - C program which uses the MQAI to display channel
status\n\n");
printf("%-2Øs %12s %8s %8s %8s %8s %12s %8s\n\n", "Channel Name", "Chl
Type",
     "CurMsgs", "Msgs", "ShortRts", "InDoubt", "Chl Status", "LastMsg");
/* ------ Connect to the queue manager - Default or supplied ---- --- */
NOTE 1: if (argc > 1)               /* Use qmgr if supplied           */
      strncpy(qmName, argv[1], (size_t)MQ_Q_MGR_NAME_LENGTH);
   MQCONN(qmName, &hConn, &compCode, &connReason);
/* ----- Report the reason and stop if the connection failed. ---- -- */
   if (compCode == MQCC_FAILED)
   {
      CheckCallResult("Queue Manager connection", compCode, connReason);
      exit( (int)connReason);
   }
/* if a remote qmgr was supplied, set up the object descriptor, and open
it*/
NOTE 2: qHandle = MQHO_NONE;        /* Default: use local cmd queue   */
   if (argc > 2)                    /* if rmt qmgr given, store it in */
   {                                /* the MQMD for remote admin      */
       strncpy(od.ObjectQMgrName, argv[2],
(size_t)MQ_Q_MGR_NAME_LENGTH);
       strncpy(od.ObjectName, "SYSTEM.ADMIN.COMMAND.QUEUE",
(size_t)MQ_Q_NAME_LENGTH);
       O_options = MQOO_OUTPUT      /* open queue for output          */
        + MQOO_FAIL_IF_QUIESCING;   /* but not if MQM stopping        */
       MQOPEN(hConn,                /* connection handle              */
       &od,                         /* object descriptor for queue    */
               O_options,           /* open options                   */
               &Hobj,               /* object handle                  */
               &compCode,           /* MQOPEN completion code         */
               &reason);            /* reason code                    */
       qHandle = Hobj;              /* store for mqExecute use        */
if (reason != MQRC_NONE)
               printf ("MQOPEN ended with reason code %ld\n", reason);
       if (compCode == MQCC_FAILED)
               printf("unable to open queue for output\n");
   }
NOTE 3: memset (staQual, '\Ø', sizeof(staQual) ); /* Initialize status
qualifier  */
   if ( argc > 3 )                   /* if parm3 supplied, use it     */
       strncpy(staQual, argv[3] , sizeof(staQual) );
/* ------ Create an admin request bag for the mqExecute call -------  */
NOTE 4: mqCreateBag(MQCBO_ADMIN_BAG, &adminBag, &compCode, &reason);
   CheckCallResult("Create admin bag", compCode, reason);
/* -------Create an admin response bag for the mqExecute call ------- */
NOTE 5: mqCreateBag(MQCBO_ADMIN_BAG, &responseBag, &compCode, &reason);
   CheckCallResult("Create response bag", compCode, reason);
/* -------Put the generic channel name into the admin bag ----------- */
NOTE 6: mqAddString(adminBag, MQCACH_CHANNEL_NAME, MQBL_NULL_TERMINATED,
"*",
```

```
              &compCode, &reason);
   CheckCallResult("Add clusqmgr name", compCode, reason);
/* ------- Add an inquiry for various channel attributes ------------ */
NOTE 7: mqAddInquiry(adminBag, MQIACH_CHANNEL_TYPE, &compCode, &reason);
   CheckCallResult("Add chl type", compCode, reason);
mqAddInquiry(adminBag, MQIACH_CURRENT_MSGS, &compCode, &reason);
   CheckCallResult("Add chl curmsgs", compCode, reason);
mqAddInquiry(adminBag, MQIACH_MSGS, &compCode, &reason);
   CheckCallResult("Add chl msgs", compCode, reason);
mqAddInquiry(adminBag, MQIACH_SHORT_RETRIES_LEFT, &compCode, &reason);
   CheckCallResult("Add chl short retries", compCode, reason);
mqAddInquiry(adminBag, MQIACH_INDOUBT_STATUS, &compCode, &reason);
   CheckCallResult("Add chl indoubt", compCode, reason);
mqAddInquiry(adminBag, MQIACH_CHANNEL_STATUS, &compCode, &reason);
   CheckCallResult("Add chl status", compCode, reason);
mqAddInquiry(adminBag, MQCACH_LAST_MSG_TIME, &compCode, &reason);
   CheckCallResult("Add chl status", compCode, reason);
/* Send the command to find all local queue names and our attributes. */
/* The mqExecute call creates the PCF structure required, sends it to */
/* the command server, and receives the reply from the command server */
/* into the response bag. The attributes are contained in system bags */
/* embedded in the response bag, one set of attributes per bag.       */
/* By default, mqExecute waits for up to 30 seconds for a response.   */
NOTE 8: mqExecute(hConn,       /* MQ connection handle                */
          MQCMD_INQUIRE_CHANNEL_STATUS, /* Command to be executed     */
          MQHB_NONE,                    /* No options bag             */
          adminBag,          /* Handle to bag containing commands     */
          responseBag,       /* Handle to bag to receive the response */
          qHandle,           /* Put msg on SYSTEM.ADMIN.COMMAND.QUEUE */
          MQHO_NONE,         /* Create a dynamic q for the response   */
          &compCode,         /* Completion code from the mqexecute    */
          &reason);          /* Reason code from mqexecute call       */

NOTE 9: /* -- Check the command server is started.Only true for local
qmgr admin --*/
   if (reason == MQRC_CMD_SERVER_NOT_AVAILABLE)
   {
     printf("Please start the command server: <strmqcsv QMgrName>\n");
     MQDISC(&hConn, &compCode, &reason);
     CheckCallResult("Disconnect from Queue Manager", compCode, reason);
     exit(98);
   }
/* - Check the result from mqExecute call. If successful find all the */
/* - req'd attributes of all local queues. If failed find the error.  */
if ( compCode == MQCC_OK )                /* Successful mqExecute    */
   {
/* Count number of system bags embedded in the response bag from the  */
/* mqExecute call. Attributes for each queue are in a separate bag.   */
NOTE 10:  mqCountItems(responseBag, MQHA_BAG_HANDLE, &numberOfBags,
&compCode, &reason);
     CheckCallResult("Count number of bag handles", compCode, reason);
for ( i=0; i<numberOfBags; i++)
```

```
     {
/* Get the next system bag handle out of the mqExecute response bag.  */
/* This bag contains the queue attributes                              */
NOTE 11: mqInquireBag(responseBag, MQHA_BAG_HANDLE, i, &chlAttrsBag,
&compCode, &reason);
         CheckCallResult("Get the result bag handle", compCode, reason);
/* ------ Get the various channel attributes out of the bag ------    */
NOTE 12: mqInquireString(chlAttrsBag, MQCACH_CHANNEL_NAME, Ø,
MQ_CHANNEL_NAME_LENGTH,
   ChlName,  &actLength, NULL, &compCode, &reason);
CheckCallResult("Get channel name", compCode, reason);
mqInquireInteger(chlAttrsBag, MQIACH_CHANNEL_TYPE, MQIND_NONE, &ChlType,
   &compCode, &reason);
CheckCallResult("Get chl type", compCode, reason);
mqInquireInteger(chlAttrsBag, MQIACH_CURRENT_MSGS, MQIND_NONE, &CurMsgs,
   &compCode, &reason);
CheckCallResult("Get chl curmsgs", compCode, reason);
mqInquireInteger(chlAttrsBag, MQIACH_MSGS, MQIND_NONE, &Msgs,
   &compCode, &reason);
CheckCallResult("Get chl msgs", compCode, reason);
mqInquireInteger(chlAttrsBag, MQIACH_SHORT_RETRIES_LEFT, MQIND_NONE,
   &ShortRts, &compCode, &reason);
CheckCallResult("Get chl shorrts", compCode, reason);
mqInquireInteger(chlAttrsBag, MQIACH_INDOUBT_STATUS, MQIND_NONE,
   &InDoubt, &compCode, &reason);
CheckCallResult("Get chl indoubt", compCode, reason);
mqInquireInteger(chlAttrsBag, MQIACH_CHANNEL_STATUS, MQIND_NONE,
   &ChStat, &compCode, &reason);
CheckCallResult("Get chl status", compCode, reason);
mqInquireString(chlAttrsBag, MQCACH_LAST_MSG_TIME, Ø,
   MQ_CHANNEL_TIME_LENGTH, LastTime,  &actLength, NULL, &compCode,
   &reason);
CheckCallResult("Get last msg time", compCode, reason);
/* ----- Use mqTrim to prepare the queue name for printing. --------  */
/* ----- Print the result dependant upon the filtering required  ---- */
NOTE 13:    if ( ((strncmp (staQual, "r", 1) == Ø) && (ChStat == 3) ) ||
 ((strncmp (staQual, "n", 1) == Ø) && (ChStat != 3) ) ||
 ((strncmp (staQual, "r", 1) != Ø) && (strncmp (staQual, "n", 1) != Ø)))
        {
NOTE 14: switch ( ChlType )
        {
               case 1: { strncpy(wChlType,"Sender      ", 12); break; }
               case 2: { strncpy(wChlType,"Server      ", 12); break; }
               case 3: { strncpy(wChlType,"Receiver    ", 12); break; }
               case 4: { strncpy(wChlType,"Requestor   ", 12); break; }
               case 5: { strncpy(wChlType,"All         ", 12); break; }
               case 6: { strncpy(wChlType,"ClientConn  ", 12); break; }
               case 7: { strncpy(wChlType,"ServerConn  ", 12); break; }
               case 8: { strncpy(wChlType,"Cluster Rcvr", 12); break; }
               case 9: { strncpy(wChlType,"Cluster Sdr ", 12); break; }
               default: {strncpy(wChlType,"Unknown     ", 12); }
        };
```

```
        switch ( ChStat )
        {
               case Ø: { strncpy(wChlStat,"Inactive    ", 12); break; }
               case 1: { strncpy(wChlStat,"Binding     ", 12); break; }
               case 2: { strncpy(wChlStat,"Starting    ", 12); break; }
               case 3: { strncpy(wChlStat,"Running     ", 12); break; }
               case 4: { strncpy(wChlStat,"Stopping    ", 12); break; }
               case 5: { strncpy(wChlStat,"Retrying    ", 12); break; }
               case 6: { strncpy(wChlStat,"Stopped     ", 12); break; }
               case 7: { strncpy(wChlStat,"Requesting  ", 12); break; }
               case 8: { strncpy(wChlStat,"Paused      ", 12); break; }
               case 13: {strncpy(wChlStat,"Initialising", 12); break; }
               default: {strncpy(wChlStat,"Unknown     ", 12); }
         };
mqTrim(MQ_CHANNEL_NAME_LENGTH, ChlName, ChlName, &compCode, &reason);
mqTrim(12, wChlType, wChlType, &compCode, &reason);
mqTrim(MQ_CHANNEL_TIME_LENGTH, LastTime, LastTime, &compCode, &reason);
mqTrim(12, wChlStat, wChlStat, &compCode, &reason);
printf("%-2Øs %12s %8ld %8ld %8ld %8ld %12s %8s\n", ChlName, wChlType,
   CurMsgs, Msgs, ShortRts, InDoubt, wChlStat, LastTime);
        }
     }
   }
NOTE 15: else                                   /* Failed mqExecute     */
   {
     printf("Call to get queue attr failed: Completion Code = %ld :
Reason = %ld\n",
compCode, reason);
/* If the command fails to get the system bag handle out of the       */
/* mqexecute response bag. This bag contains the reason from the      */
/* command server why the command failed.                             */
     if (reason == MQRCCF_COMMAND_FAILED)
     {
mqInquireBag(responseBag, MQHA_BAG_HANDLE, Ø, &errorBag, &compCode,
   &reason);
CheckCallResult("Get the result bag handle", compCode, reason);
/* Get the completion code and reason code, returned by the command   */
/* server, from the embedded error bag.                               */
mqInquireInteger(errorBag, MQIASY_COMP_CODE, MQIND_NONE, &mqExecuteCC,
   &compCode, &reason );
CheckCallResult("Get the completion code from the result bag", compCode,
 reason);
mqInquireInteger(errorBag, MQIASY_REASON, MQIND_NONE, &mqExecuteRC,
   &compCode, &reason);
CheckCallResult("Get the reason code from the result bag", compCode,
   reason);
printf("Error from the cmd server: Completion Code = %ld : Reason =
%ld\n", mqExecuteCC, mqExecuteRC);
     }
   }
/* --------- Delete the admin bag if successfully created. ---------  */
NOTE 16: if (adminBag != MQHB_UNUSABLE_HBAG)
```

```
   {
      mqDeleteBag(&adminBag, &compCode, &reason);
      CheckCallResult("Delete the admin bag", compCode, reason);
   }
/* -------- Delete the response bag if successfully created. -------- */
NOTE 17: if (responseBag != MQHB_UNUSABLE_HBAG)
   {
      mqDeleteBag(&responseBag, &compCode, &reason);
      CheckCallResult("Delete the response bag", compCode, reason);
   }
/* --- Disconnect from the queue manager if not already connected --- */
NOTE 18: if (connReason != MQRC_ALREADY_CONNECTED)
   {
      MQDISC(&hConn, &compCode, &reason);
      CheckCallResult("Disconnect from Queue Manager", compCode,
reason);
   }
   return 0;
}
/* Logic: Display the description of the call, completion code and    */
/* reason code if the completion code is not successful               */
NOTE 19: void  CheckCallResult(char *callText, MQLONG cc, MQLONG rc)
{
   if (cc != MQCC_OK)
     printf("%s failed: Completion Code = %ld : Reason = %ld\n",
       callText, cc, rc);
}
```

Note 1

It is now agreed good practice to have no default queue manager so it is better to invoke the utility with this parameter.

Note 2

The original utility **amqsailq** did not cater for remote administration at all. By passing the remote queue manager name it is then a simple case of opening the remote command queue. For example, to display the local queues belonging to a remote queue manager type in **listchl QM1 JAV1** where QM1 is the local queue manager and JAV1 is the remote queue manager:

```
C:\>listchl QM1 JAV1
listchl - example C program which uses the MQAI to display channel
status
Channel    Chl Type  CurMsgs  Msgs ShortRts InDoubt  Chl       LastMsg
Name                                          Status
HUB2.JAV3  Sender      0      0      0       0       Retrying
HUB2.JAV2  Sender      0      0      9       0       Retrying
```

```
HUB2.HUB1  Sender      0      1      10       0       Running   15.35.06
HUB1.JAV1  Receiver    0      0      10       0       Running
```

Obviously, as with all remote administration, ensure that the relevant channels (two-way!), transmission queues, and the listener and command server are defined and active.

Note that the downside of using **runmqsc** for remote administration is that you are forced to make QM1 the default queue manager, eg **runmqsc –w 30 JAV1** (QM1 is not specified at all).

You can also set this parameter to be the same as the local queue manager, eg **listchl QM1 QM1**.

Note 3

An extra parameter has been introduced to qualify further the list of objects returned. For example, you cannot use the standard MQSeries facilities to list only those channels whose status is 'running' or 'not running'. This parameter actually allows you to do that:

- Values:
  - r – only display those channels with status 'running'
  - n – only show those channels with status other than 'running'
  - other – list all channel information.

```
C:\>listchl QM1 JAV1 r
listchl - example C program which uses the MQAI to display channel
status

Channel    Chl Type  CurMsgs  Msgs ShortRts  InDoubt  Chl       LastMsg
Name                                                   Status
HUB2.HUB1  Sender     0       5       10        0     Running   15.40.05
HUB1.JAV1  Receiver   0       1       10        0     Running   15.35.06
```

Note 4

This is how an administration bag is created. Note that after a call to the MQAI the return code is checked.

Note 5

The response also needs a bag.

Note 6

Add an item to the bag to list all channel information.

Note 7

Add an item to pick up specific attributes (channel status, current messages, messages sent so far, number of short retries, in doubt status, and time of last message).

Note 8

Once the bags have been populated with items, call **mqExecute** to change the data into PCF format and MQPUT it to the command queue. It waits for 30 seconds (the default period, which can be changed by setting the 'options' parameter) for a response.

The response is actually obtained from a temporary dynamic queue, eg *MQAI.REPLY.3C43F4D700002012*.

Note 9

To administer a local queue manager, the command server needs to be started. A nice feature is that the program can pick up this status. For remote administration the remote command server needs to be active; this cannot be reported upon although a time-out will occur if it's inactive.

Note 10

The original request can return zero or more bags (each channel is in a separate bag). Obtain this number and loop round all available response bags.

Note 11

Obtain the handle to the bag.

Note 12

Obtain the various attributes by using a combination of **mqInquireString** and **mqInquireInteger**.

Note 13

Display the result, but further qualified by parameter three.

Note 14

Convert the channel type and status into readable format.

Note 15

mqExecute failed. Pick up the failure code from the bag.

Below are two examples of how errors are reported.

Example one: channel SYSTEM.ADMIN.SVRCONN does not have the curmsgs, shorts, and in doubt attributes. You can change the program to cater for this if you wish.

```
C:\>listchl QM1 JAV1
listchl - example C program which uses the MQAI to display channel
status
Channel    Chl Type  CurMsgs  Msgs ShortRts InDoubt  Chl       LastMsg
Name                                                 Status
HUB2.JAV3  Sender      0         0       0     0     Retrying
HUB2.JAV2  Sender      0         0       0     0     Retrying
HUB2.HUB1  Sender      0        13      10     0     Running   16.52.21
HUB1.JAV1  Receiver    0         3      10     0     Running   16.52.04
Get chl curmsgs failed: Completion Code = 2 : Reason = 2309
Get chl shorrts failed: Completion Code = 2 : Reason = 2309
Get chl indoubt failed: Completion Code = 2 : Reason = 2309
SYSTEM.ADMIN.SVRCONN   0         18     10     0     Running   16.52.14
   ServerConn
```

Example two: run the program against a queue manager that has no active channels. The output is correct. RC3008 means MQRCCF_COMMAND_FAILED and RC3065 means MQRCCF_CHL_STATUS_NOT_FOUND.

```
C:\>listchl QM2
listchl - example C program which uses the MQAI to display channel
status
Channel    Chl Type  CurMsgs  Msgs ShortRts InDoubt  Chl       LastMsg
Name                                                 Status
Call to get queue attributes failed: Completion Code = 2 : Reason = 3008
Error returned by command server: Completion Code = 2 : Reason = 3065
```

Note 16

Tidy up by deleting the administration request bag (which also deletes the allocated storage).

Note 17

Tidy up by deleting the administration response bag.

Note 18

Disconnect from the local queue manager.

Note 19

Procedure called after each call to MQAI to display any errors.

ADDITIONAL USE FOR THE MQAI

The MQAI is not restricted to administration – it can also be used to pass user data between applications. It is a fairly simple exercise to convert the sample programs **amqsput0.c** and **amqsget0.c** to use the MQAI. Instead of MQPUT use the *mqPutBag* program and instead of MQGET use the *mqGetBag* program.

I did not want to reuse the system or user selectors and so defined my own.

Here is a selection of the code (the return code checking has been left out):

Changes required to amqsput0.c

```
#define MQCA_USER_1   5001L   /* my own user selector code for numeric*/
#define MQCA_USER_2   5002L   /* my own user selector code for string */
#define User_Data1    1L      /* my numeric data                      */
MQHBAG userBag = MQHB_UNUSABLE_HBAG;     /* user bag for mqPutBag     */
memcpy(md.Format, MQFMT_PCF, (size_t)MQ_FORMAT_LENGTH); /* must be PCF*/
/* Note that this is a USER bag - different to the listchl.c program */
mqCreateBag(MQCBO_USER_BAG, &userBag, &CompCode, &Reason);
/* Add my first bit of data - numeric data */
mqAddInteger(userBag, MQCA_USER_1, User_Data1, &CompCode, &Reason);
/* Add my second bit of data - string (from keyboard) */
mqAddString(userBag, MQCA_USER_2, MQBL_NULL_TERMINATED, buffer,
&CompCode, &Reason);
mqPutBag(Hcon, Hobj, &md, &pmo, userBag, &CompCode, &Reason);
mqDeleteBag(&userBag, &CompCode, &Reason);
```

Changes required to amqsget0.c

```
#define MQCA_USER_1   5001L   /* my own user selector code for numeric*/
#define MQCA_USER_2   5002L   /* my own user selector code for string */
#define User_Data1    1L      /* my numeric data                      */
MQHBAG userrespBag = MQHB_UNUSABLE_HBAG; /* user bag for mqGetBag     */
/* A user response bag needs to be created */
mqCreateBag(MQCBO_USER_BAG, &userrespBag, &CompCode, &Reason);
mqGetBag (Hcon, Hobj, &md, &gmo, userrespBag, &CompCode, &Reason);
```

```
mqCountItems(userrespBag, MQHA_BAG_HANDLE, &numberOfBags, &CompCode,
&Reason);
/* get my numeric data */
mqInquireInteger(userrespBag, MQCA_USER_1, MQIND_NONE, &User_Data1,
                        &CompCode, &Reason);
/* get my string data */
mqInquireString(userrespBag, MQCA_USER_2, 0, 100, buffer,
                       &buflen, NULL, &CompCode, &Reason);
mqDeleteBag(&userrespBag, &CompCode, &Reason);
```

Testing output

```
C:\>mqaiput FRED QM1
mqaiput start (using the MQAI to put messages)
target queue is FRED
This is my test data    ♦====== entered via keyboard as per amqsput
mqaiput end
C:\>mqaiget FRED QM1
mqaiget start (using the MQAI to get messages)
Data 1 (numeric) <1>
Data 2 (string) <This is my test data>
mqaiget end
```

So why would you want to use this method if the standard way to pass data works well? One reason is that MQSeries has a built-in method for converting PCF messages and, as the above example has shown, you can mix numeric and text data without having to worry about writing a conversion exit.

CONCLUSION

The MQAI provides a simpler way than PCF to administer WebSphere MQ.

The concept of data bags is easy to understand and utilities written to use the MQAI are relatively easy to program.

Although the easiest way to transfer data between MQ-connected systems is as 'text' data, some installations that decide to pass mixed numeric and character data – and which do not want to write a conversion exit – could use the MQAI to pass user data between their programs as PCF messages.

Ruud van Zundert
Independent MQSeries Consultant (UK)
ruudvz@btclick.com

Configuring MQSeries with Microsoft Cluster Server, part 1: planning

The first part of this article covers the groundwork involved in configuring MQSeries to work with Microsoft Cluster Server (MSCS), and includes an introduction to the two products and an overview of the planning stages. In next month's *MQ Update* we look at installation and configuration.

INTRODUCTION

Before embarking on a project to integrate MQSeries and (MSCS) clustering it is important to understand the following:

- How to enable a queue manager to be run under MSCS control.
- The relationship between MQSeries clusters and MSCS clusters, where the word 'cluster' has two very different meanings.

This article explains these concepts and describes how the two different styles of cluster can be used separately or in combination to achieve different degrees of availability for particular message types. It will then guide the reader through the planning process and present step-by-step instructions on how to install and configure the necessary support.

MQSeries

The assured once-only delivery of messages provided by MQSeries makes it possible to configure a messaging network such that messages are not lost even when systems fail. The fact that a message is recoverable following a failure is in some cases sufficient but in others there may be a need to limit the time between the occurrence of the failure and the message being accessible once again.

MQSeries includes a capability known as MQSeries clustering or queue manager clustering. An MQSeries cluster is a collection of queue managers that use a common set of definitions (for queues and channels) stored within one or more repositories that are managed by queue managers within the collection. The advantages of MQSeries

clusters and their relationship to MSCS clusters are explained in detail later in this article.

In Windows NT and Windows 2000, MQSeries contains a component known as the IBM MQSeries Service, which is responsible for starting and restarting queue managers that are configured to run on a computer. This service can detect the failure of a queue manager and restart it on the same computer provided that the computer remains running throughout. It can also automatically start a queue manager upon restart of the computer following a failure or shutdown. The service therefore provides automation and a degree of fault detection and availability improvement compared to a system that has no monitoring or restart capabilities. However, the service is not able to perform remote monitoring and is not able to move a queue manager from one computer to another in the event of a system failure.

Microsoft Cluster Server

Microsoft Cluster Server is a high-availability cluster environment which provides a cluster framework for control and monitoring of business-critical applications. MSCS is supplied as part of Windows NT 4.00 Enterprise Edition and as part of Windows 2000 Advanced Server or Windows 2000 Datacentre. MSCS can be configured and controlled through a graphical user interface known as the Cluster Administrator.

MSCS monitors and controls resources. Resources can be anything from disk drives or network addresses to application processes. MSCS is able to start and stop resources on the computers in the cluster and can move groups of resources from one computer to another. The movement of a group of resources in response to a failure is called a failover. Each resource has a resource type that defines how instances of that type should be started and stopped and how to monitor them. There are a number of built-in resource types which come as standard with MSCS and which cater for a number of commonly occurring resources. It is also possible to add further resource types to manage resources that are not covered by the set of built-in types. These are referred to as custom resource types and are created by writing a custom resource DLL and an extension to the Cluster Administrator graphical interface.

IBM MQSeries SupportPac MC74 contains a custom resource type that can be used with MQSeries. The MC74 SupportPac can be found at *http://www.ibm.com/software/ts/mqseries/txppacs/mc74.html*.

MQSeries and MSCS together

By using MQSeries and MSCS together it is possible to greatly extend the availability that can be achieved by using MQSeries alone. It is possible to configure MSCS to detect a broader range of failure conditions and to respond by either restarting a queue manager on the same computer that it was previously running on or by failing it over to a different computer.

Relationship between MSCS and MQSeries clusters

MQSeries clusters reduce administration and provide load balancing of messages across instances of cluster queues. They also offer higher availability than a single queue manager because, following a failure of a queue manager, messaging applications can still access surviving instances of a cluster queue. However, unlike MSCS, MQSeries clusters will not provide automatic detection of queue manager failure and automatic triggering of queue manager restart or failover. The two types of cluster can be used together to good effect and this is described below.

PLANNING

When to use MSCS with MQSeries

It is common to hear people say that they want '24x7'. But what are they actually referring to? To achieve 24x7 is difficult – it means zero downtime. One hopes that most of these people would actually be satisfied with something slightly less than 24x7; this is commonly measured as a percentage, ie 99.9%. A previous article published in *MQ Update* in June 2000 describes this in more detail.

As important as the quantification of availability is the qualification of what is to be available. Is it the ability to access messages that are already on queues or is it the ability to put messages onto queues? If access needs to be rapidly restored for messages that are on queues then a high-availability cluster solution, such as MSCS, is essential.

MSCS can restart a queue manager on a different computer if necessary and hence much more quickly than would be the case without MSCS.

Alternatively, if the ability to put new messages onto queues is important then it may be acceptable to leave existing messages on the queues of a queue manager that cannot be restarted until the computer it is on has been fixed. New messages to be put onto queues could be handled by additional queue managers with equivalent queues. This could be achieved using MQSeries clustering and creating multiple instances of a cluster queue.

Another way of looking at this is that, by having alternative instances of cluster queues, an MQSeries cluster can provide continuous availability for new messages but requires manual intervention to allow access to existing messages already in queue managers affected by a failure. Contrasted with that, MSCS can provide high availability for either type of message because a queue manager that holds existing messages or to which new messages need to be put will be restarted without manual intervention by a local restart or failover within as little as 30 seconds. This time does not include any allowance for log replay on restart, which will depend on the state of the queue manager at the time of the failure and whether there are any long-running units of work.

MSCS Clustering basics

Almost all MSCS clusters consist of two computers called nodes, both of which have access to disks on a shared storage bus. Many people refer to such a cluster as a shared-disk cluster but the disks are never simultaneously accessed by more than one node at a time so the term ‘shared’ is a little misleading: MSCS is actually an example of a shared-nothing cluster.

The unit of failover in MSCS is a resource group, which is a collection of resources that are always kept together. When a resource group is moved from one node to another all the resources it contains are moved to that node. In order to make failovers as efficient as possible each resource group should ideally contain only the resources that are necessary for a particular service. This maximizes the independence of each resource group, providing flexibility and minimizing disruption during a failure or planned maintenance.

Mapping queue managers to MSCS groups

In order to put an MQSeries queue manager under MSCS control you need to create a resource of the custom resource type 'IBM MQSeries MSCS' and put it onto a resource group. You don't have to do anything weird like creating a queue manager with the same name on the other node or anything like that – each highly available queue manager just needs to be created once on one node and then prepared for clustering.

The smallest unit of failover of MQSeries is a queue manager since you cannot move part of a queue manager without moving the whole thing. It follows from the comments above about minimizing the sizes of resource groups that the optimal configuration is to place each queue manager in a separate resource group, which should contain the shared drives used by the queue manager, the IP address used to connect to the queue manager, and the queue manager resource itself.

You could put multiple queue managers into a resource group, but if you did so they would all have to failover to another node together, even if the problem causing the failover were confined to one queue manager. This would cause unnecessary disruption to the other queue managers so this is not recommended.

Possible configurations

MSCS defines two styles of cluster configuration, which are referred to as active/passive (see Figure 1) and active/active (see Figure 2). In an active/passive configuration one node is running production workload and the other stands by in case of a problem with the active node. An active/active configuration is one in which both cluster nodes are running production workload at the same time.

A queue manager cannot span more than one node so the construction of an active/active cluster configuration in which both nodes are running MQSeries workload requires at least one queue manager per cluster node. If the queue managers in an MSCS cluster are members of an MQSeries cluster then they can each run instances of clustered queues and can thus present a symmetrical (location transparent) messaging architecture to applications or clients.

Such a configuration can support workload balancing using the mechanisms provided in MQSeries clustering. If the queue managers

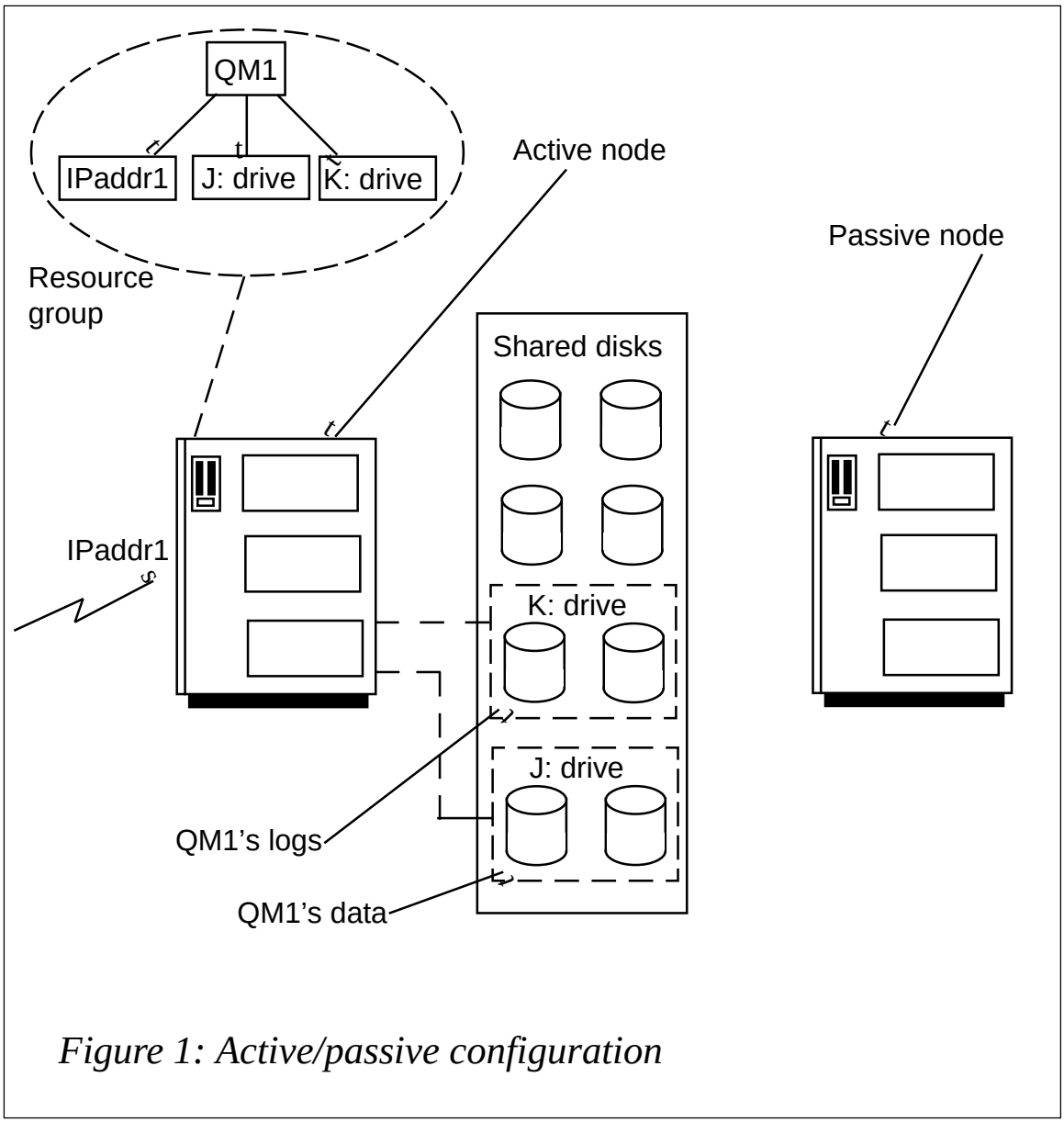

Figure 1: Active/passive configuration

are not members of an MQSeries cluster they appear to applications or clients as separate entities and the active/active configuration is equivalent to a pair of active/passive configurations superimposed one upon the other.

Applications

Applications running on the same machine as a queue manager to which they are locally bound can be started and stopped under cluster control if they are represented by MSCS resources. Such resources should be made dependent on the queue manager resource so that they are started after the queue manager and stopped before it. MSCS

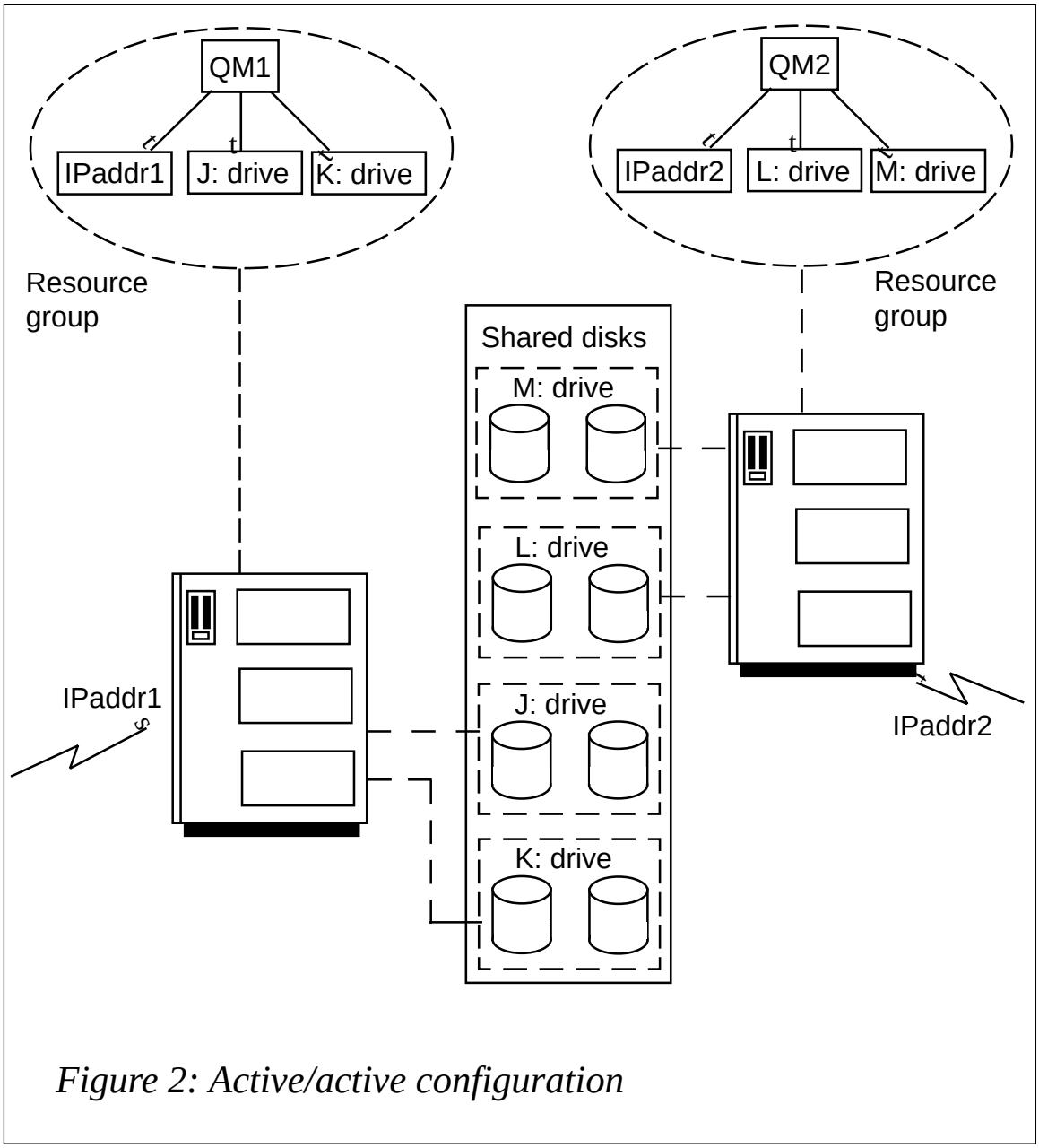

Figure 2: Active/active configuration

provides built-in resource types that can be used for this, or you could introduce a custom resource type.

Ports and listeners

If the MSCS cluster contains multiple queue managers, two or more queue managers may need to run on the same node following a failover even if they are normally on different nodes. To provide correct routing of MQ channel traffic to the queue managers, you must use a different TCP/IP port number for each queue manager. The

standard MQSeries port is 1414. It is common practice to use a range of port numbers immediately above 1414 for additional queue managers. Note that, whatever port number you assign to a queue manager, that port needs to be available on all cluster nodes that may host the queue manager and all channels to that queue manager need to refer to the port.

The MQ listener can be started and stopped automatically when the queue manager is started and stopped. You don't need to configure MSCS to control the listener.

Channels

Remote queue managers or MQSeries clients which are exchanging messages with a queue manager that is under MSCS control need their channels configured to use an IP address (and port) which is under MSCS control in the same resource group as the MSCS-controlled queue manager. This IP address will be moved between nodes by MSCS. Channels must not use an IP address or hostname that is dedicated to either of the cluster nodes.

During a failover of an MSCS-controlled queue manager, MQSeries clients or channels from remote queue managers will see a broken TCP connection. Channels from remote queue managers will automatically retry the connection. Such retries will fail until the IP address has been failed over and brought online and the queue manager has also been failed over and brought online; they will then succeed. The time required for reconnection of the channels will depend on the number of channels involved. Clients should be written so that a broken connection is retried repeatedly.

It is a good idea to trigger channels.

The MQ channel initiator can be started and stopped automatically when the queue manager is started and stopped. You don't need to configure MSCS to control the initiator.

Choosing hardware

Microsoft publishes a list of hardware configurations that have been tested with MSCS and certified. It is a good idea to make sure that your hardware is on the list – or that you understand any differences

between your hardware configuration and the closest match from the list. The hardware compatibility list can be found at *http://www.microsoft.com/hcl*. Search using the term 'cluster' and optionally, include the name of your hardware manufacturer.

Choosing software

Make sure your configuration complies with the prerequisites listed in the MC74 SupportPac and keep your software versions and configuration symmetric across the MSCS cluster.

Choosing networks

MSCS supports the use of IP addresses and these can be configured to run over various physical transports, including Ethernet or Token Ring.

Capacity planning

Make sure that you consider the peak load that might be placed on any one machine in the cluster as a result of possible failovers of resources from other machines.

User exits

It is possible to define optional user exits that are invoked just after a queue manager is brought online (started) and just before a queue manager is taken offline (ended). These allow you to start and stop additional processes or provide notification of these cluster actions to other systems. This feature can be used to integrate an MSCS-controlled queue manager into existing system management framework or operational procedures.

This article concludes in next month's *MQ Update*.

Graham Wallis
Software Engineer
IBM Hursley (UK)

iWay Software, an Information Builders company, has recently announced the release of iWay Security Exchange, which, so the company claims, is a complete software and services solution that enables government, law enforcement, and private industry to effectively exchange and share information easily in real time.

The Security Exchange provides the components necessary to develop and deploy collaborative environments while protecting the resources owned by individual participants.

Included with the Security Exchange is a process flow engine, assured delivery, a J2EE compliant Java application servers, built-in auditing and logging, and iWay intelligent adapters that enable customers to interconnect widely disparate information resources with little or no programming.
The iWay Security Exchange is based upon the WebSphere software platform (which includes WebSphere MQ Integrator, WebSphere MQ, and WebSphere Application Server) the iWay Enterprise Integration Suite, and Information Builder's WebFOCUS.

For further information contact:
Information Builders, Two Penn Plaza, New York, NY 10121-2898, USA
Tel: +1 212 736 4433
Fax: +1 212 967 6406
URL: http://www.ibi.com

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

* * *

IBM has announced enhancements to its WebSphere Application Server Enterprise Edition (AS EE) Version 4.1 with a host of enterprise services, TXSeries support for AIX, Solaris, and Windows NT/2000, MQSeries support, and a business process beans technology preview.

The enterprise services include business rule beans, message beans and JMS listener, internationalization, shared work areas, bidirectional CORBA connectivity (AIX, Solaris, and Windows NT/2000), a C++ CORBA software development kit, and an ActiveX to Enterprise Java Beans bridge for AIX, Solaris, and Windows NT/2000.

The new release runs on AIX, HP-UX, Solaris, Windows NT, and Linux (Red Hat and SuSE). It consists of the WebSphere Application Server Advanced Edition (AS AE), Enterprise Services, TXSeries, and MQSeries.

For further infomation contact your local IBM representative.
URL: http://www.ibm.com/software/webservers