31

MQ

update

January 2002

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/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 (temporary)

Harry Harris
E-mail: harrya.harris@virgin.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.

Improving TCP/IP channel reliability

MQSERIES CHANNELS AND TCP/IP SOCKETS – BACKGROUND

The sender and receiver message channel agents (MCAs) for a TCP/IP channel pair use a facility called TCP/IP sockets to communicate with each other. Once a socket connection is initialized (one end listens on a specific port number, the other connects to a specified IP address and port number), both ends can send and receive messages to and from each other.

It is important to understand the way in which the MQSeries channel protocol operates. When a channel sender/receiver pair is started but idle, the sender MCA is waiting for an MQGET from the transmission queue, and the receiver MCA is blocked, waiting for a socket receive call. The socket receive call is analogous to an MQGET call in that a wait period can be specified, and execution is blocked until either some data arrives or the wait period expires.

When a channel is idle (let's assume channel heartbeats are disabled – see later section for a discussion of this subject), the MCAs do not send or receive any data and each MCA will be unaware of the opposite MCA's status. In the case of a communications error, such as a network outage, the sender will be unaware that the receiver is unavailable until a message arrives on the transmission queue and it tries to send, over the socket, to the receiver MCA. The sender MCA will receive an error only when it tries to send data. Furthermore, the receiver MCA will not normally be aware that the sender is unavailable as it simply may not be sending any data.

This behaviour is advantageous for large installations where there may be thousands of channels and network traffic needs to be minimized, but may cause a lack of resilience where the availability of channels is a critical factor. Careful consideration of the following options will allow TCP/IP channels to be more resilient by detecting and correcting errors as soon as possible, with a minimum of manual operator intervention.

ADOPTNEWMCA PARAMETER

This parameter provides a solution in situations where a sender MCA has terminated because of an error and has closed its end of the TCP/IP socket, but the receiver MCA did not receive an error and still has its socket open waiting for data. The listener on the receive end will refuse the channel start request because the old orphaned instance of the channel is already running.

If the listener receives a channel start request but the receiver MCA is already running, then enabling this parameter will cause the listener to terminate the existing orphaned MCA and start a new MCA to service the start request.

The symptom of this problem is the receiver channel in a RUNNING state, but the corresponding sender is unable to start and has gone into a RETRY state. The channel will only start when the disconnect interval expires on the receiver MCA and the sender performs its next retry.

The *AdoptNewMCA* parameter was introduced with version 5.1 of MQSeries and is disabled by default. The parameter is enabled by specifying **AdoptNewMCA=ALL** in the *CHANNELS:* stanza within a queue manager's *QM.INI* file (in Windows NT/2000 all parameters are stored in the registry and can be changed from the queue manager properties dialog within the MQSeries services console). When enabled, the listener on the receiver end will give the orphaned MCA a stop request and will wait **AdoptNewMCATimeout** seconds for it to terminate. If it is still running, the listener will then kill the process.

For security purposes, it is also recommended to specify **AdoptNewMCACheck=ALL** to prevent channels from being inadvertently or maliciously closed down. This parameter instructs the listener to check (before killing the existing MCA) that the incoming channel start request is coming from the same IP address and queue manager.

KEEPALIVE PARAMETER

KeepAlive is an option available on TCP/IP sockets that causes each TCP/IP socket to periodically probe the other end of the socket and check it is still open and available. If, after retrying, the other end does

not respond, the socket is closed and an error is returned to the application. This probing is done by the operating system at the TCP/IP layer; the MCAs simply request *KeepAlive* when opening a socket.

A queue manager will request *KeepAlive* for all outgoing and incoming channel connections when *KeepAlive=YES* is specified in the *TCP:* stanza. This parameter is available for all recent versions of MQSeries.

On receiving a TCP/IP error, a sender MCA will go into a RETRY state and attempt to re-establish a new connection: a receiver MCA will go into the INACTIVE state and wait for a new incoming connection.

The *KeepAlive* timer determines how often the *KeepAlive* probe occurs; it is set in the operating system per TCP/IP stack and cannot be set on a per socket basis. The same *KeepAlive* timer value will be used for all TCP/IP sockets requesting that *KeepAlive* runs on that TCP/IP stack. The *KeepAlive* timer has a high default value of two hours: enabling *KeepAlive* will be of little use unless the timer is configured to a more suitable value.

- On Unix, the *KeepAlive* timer is a kernel parameter and instructions for changing its value will depend on the version of Unix running.
- On NT and 2000, the parameter is stored in the registry under the key *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters*: the value has the name *KeepAliveTime*, is of the type *REG_DWORD*, and is expressed in milliseconds. Look at the Microsoft Knowledge Base, article number Q120642, for further details.
- On AS/400, use the **CHGTCPA** command.

Some systems may also allow two other parameters to be altered: the number of *KeepAlive* probe retries and the time between each retry. Care must be taken to ensure that the chosen values are suitable for all applications that request *KeepAlive*.

CHANNEL HEARTBEATS

A heartbeat flow is sent by the sender MCA after the channel has been idle (ie transmission queue has been empty) for a period of time known as the effective heartbeat interval. This value is negotiated

when the channel is started and will be the larger of the *HBINT* parameters for the channel pair. The *HBINT* parameter must be supported by both queue managers otherwise no heartbeats will occur. This parameter is available with V5.0 of MQSeries and upwards, plus OS/390 V2.1 and AS/400 V4.2.1.

Do not confuse the heartbeat flow with a message flow: the heartbeat flow is sent only over the socket between the MCAs; it does not involve the sending of an MQ message. However, the heartbeat flows are reflected in the batches completed, buffers sent/received, and bytes sent/received statistics within the channel status.

The receiver MCA times-out from the socket receive call after twice the heartbeat interval, as it should have received either a message or a heartbeat flow within this interval. It can, therefore, assume that the sender MCA is no longer available and can terminate, returning the channel status to inactive. After sending the heartbeat flow the sender MCA expects the heartbeat flow to be echoed back, and if this doesn't happen it will assume that the receiver MCA is unavailable, causing the sender channel to terminate and go into the retry state. Channel heartbeats provide a similar function to *KeepAlive* in that they allow errors to be detected on idle and orphaned channels, but provide this facility at the program layer rather than at the TCP/IP layer.

Heartbeat flows also allow the receiver MCA to check whether the queue manager is quiescing. When a channel is started but idle, the heartbeat flow will allow the receiver MCA to return from the blocked socket receive call, check whether the queue manager is quiescing, and then continue with the next blocked receive call. Finally, the receiver MCA will also free any resources (close open queues and release buffers) when the heartbeat is received, as it can assume the channel is not being actively used.

DISCONNECT INTERVAL AND RETRY TIMERS

For queue managers that do not support channel heartbeats (and of course, *AdoptNewMCA*, which was introduced in the subsequent version) and have *KeepAlive* disabled, the value of the disconnect interval becomes more critical. In the orphaned MCA situation, where the sender MCA has failed and restarted but the receiver MCA is still

waiting on the old socket, the receiver MCA will only end when the disconnect interval has expired.

The default values for the *DISCINT* and *LONGTMR* parameters mean that a channel could be down for two hours before it recovers automatically. If this is unacceptable, a shorter *DISCINT* will have to be used. Consideration may also be given to the *SHORTTMR* and *SHORRTY* parameters to ensure that the sender channel is still in short-term retry when the receiver terminates, allowing the channel to restart as quickly as possible.

CONCLUSION

- When a sender MCA tries to send a message any error is detected immediately.
- *KeepAlive* and channel heartbeats allow sender and receiver MCAs to detect an error while the channel is idle. This may be important in order to detect errors on an infrequently-used channel with a long disconnect interval, for example.
- *AdoptNewMCA* allows a receiver channel to immediately recover from an orphaned receiver MCA.
- *KeepAlive* and channel heartbeats allow a receiver MCA to detect that it has been orphaned within a configured period of time. The channel will go back to the inactive state and wait for another incoming start request from the sender MCA.
- Channel heartbeats provide a similar error detection mechanism to *KeepAlive*, but are easier to implement, can be set on a per channel basis, and provide an additional benefit. There is no case for enabling *KeepAlive* on adjacent queue managers that support channel heartbeats.
- The default *HBINT* value of 300 seconds is adequate for most purposes. The value may be increased if there are a large number of started channels and heartbeat flows are causing network congestion. The value may be decreased to detect errors on idle channels more quickly, and to allow a queue manager with started receive channels to quiesce more quickly.

- The value chosen for the *KeepAlive* timer needs to be coordinated with those who are familiar with the network and other TCP/IP applications running on the systems hosting queue managers.
- For queue managers that do not support channel heartbeats (and *AdoptNewMCA*), enabling *KeepAlive* should be considered. Without *KeepAlive* or channel heartbeats, a short disconnect interval will terminate orphaned receiver MCAs more quickly.
- For receiver channels on queue managers that do not support *AdoptNewMCA*, ensure that the sender MCA is still in the short retry state when the receiver MCA terminates. The short retry time (**SHORTTMR x SHORTRTY**) should be a margin less than twice the heartbeat interval or the total time required for a *KeepAlive* to fail (**keepalive time + (number of retries x retry interval**).

Jonathan Rowe
Consultant, Workstar (UK)

Processing expired messages

MQSeries for OS/390 did not provide a dead letter queue handler until version 5.2. The problem with processing messages on the dead letter queue arises when some of the messages are expired messages.

Expired messages are counted on **display q(dead-letter-q) curdepth** commands but are ignored on **copy q(dead-letter-q)** commands. **Copy q** gives 'no message on the queue' whereas the queue depth is greater than zero.

This situation has been reported to IBM but the company doesn't regard it as an error, merely a processing difference between the **COPY** command and **DISPLAY CURDEPTH** command.

We have written a COBOL program – MQREMEXP – destined to remove expired messages from the dead letter queue and leaving only unexpired messages, which are then processed by the home-written dead letter handler program, RUNMQDLQ.

The logic in MQREMEXP is to MQGET a message with an unexpected *message-id* and *correlation-id*. MQGET gets MQSeries to search through the dead letter queue for this non-existent message, and when an expired message is met, MQSeries removes it from the dead letter queue.

The dead letter queue handler is similar to the one supplied on other platforms: it gets as input the *destination-q* name or the *destination-qmanager* name and the action type required. '*all queues*' can also be given as *destination-q* name.

Below, you will find the source code for MQREMEXP and RUNMQDLQ, the JCL source named RUNMQDLJ with two steps mqremexp and runmqdlq, and the input file MQDLQIN.

MQREMEXP

```
CBL NODYNAM,LIB,OBJECT,RENT,RES,APOST
       IDENTIFICATION DIVISION.
       PROGRAM-ID. MQREMEXP.
       AUTHOR.   SERKAN KOCAK.
       DATE-WRITTEN. JAN, 2000.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       I-O-CONTROL.
       DATA DIVISION.
       FILE SECTION.
       WORKING-STORAGE SECTION.
       01  TARGET-MSGID    PIC X(24) VALUE 'TISYS REMOVE EXPIRED MSG'.
       01  TARGET-CORRELID PIC X(24) VALUE 'TISYS REMOVE EXPIRED MSG'.
       01  W01-MQM-NAME                PIC X(48) VALUE SPACES.
       01  W01-QUEUE-NAME              PIC X(48) VALUE SPACES.
       01  W02-MQM                     PIC X(48) VALUE SPACES.
       01  W02-OBJECT                  PIC X(48) VALUE SPACES.
       01  W03-BUFFER-LENGTH           PIC S9(9) BINARY  VALUE 5200.
       01  W03-HCONN                   PIC S9(9) BINARY.
       01  W03-OPTIONS                 PIC S9(9) BINARY.
       01  W03-HOBJ                    PIC S9(9) BINARY.
       01  W03-DATA-LENGTH             PIC S9(9) BINARY.
       01  W03-COMPCODE                PIC S9(9) BINARY.
       01  W03-REASON                  PIC S9(9) BINARY.
       01  W03-MESSAGE-BUFFER          PIC X(5200).
       01  W05-MQM-OBJECT-DESCRIPTOR.
           COPY CMQODV.
       01  W05-MQM-MESSAGE-DESCRIPTOR.
           COPY CMQMDV.
       01  W05-MQM-GET-MESSAGE-OPTIONS.
```

```
     COPY CMQGMOV.
 Ø1  WØ5-MQM-CONSTANTS.
     COPY CMQV.
 Ø1  WØ6-CSQ4-OK             PIC S9(4) VALUE Ø.
 Ø1  WØ6-CSQ4-WARNING        PIC S9(4) VALUE 4.
 Ø1  WØ6-CSQ4-ERROR          PIC S9(4) VALUE 8.
 Ø1  WØ7-DISPLAY.
     Ø5 WØ7-QNAME            PIC X(48).
     Ø5 WØ7-ACTION           PIC X(1Ø).
 LINKAGE SECTION
 Ø1  PARMDATA.
     Ø5  PARM-LEN                PIC S9(Ø3) BINARY.
     Ø5  PARM-STRING             PIC X(1ØØ).
 PROCEDURE DIVISION USING PARMDATA.
 A-MAIN SECTION.
     PERFORM GET-PARAMETERS
        THRU GET-PARAMETERS-EXIT.
     PERFORM MAIN-WORK
        THRU MAIN-WORK-EXIT.
 GET-PARAMETERS.
* QMANAGER NAME AND THE DEAD_LETTER_Q NAME SHOULB BE
* BE GIVEN AS INPUT PARAMETERS FROM THE INVOKED JCL
     IF PARM-LEN = Ø THEN
        DISPLAY 'MQREMEXP: NO PARAMAETER SPECIFIED, GIVE QMGR'
        DISPLAY 'MQREMEXP: AND QUEUE NAME SEPARATED BY COMMA.'
        STOP RUN
     END-IF.
     UNSTRING PARM-STRING DELIMITED BY ALL ','
                             INTO WØ2-MQM
                                  WØ2-OBJECT.
     MOVE WØ2-MQM    TO WØ1-MQM-NAME.
     MOVE WØ2-OBJECT TO WØ1-QUEUE-NAME.
     IF WØ2-MQM = SPACES OR WØ2-MQM = LOW-VALUES THEN
        DISPLAY 'MQREMEXP: DEFAULT QMGR USED'
     END-IF.
     IF WØ2-OBJECT = SPACES OR WØ2-OBJECT = LOW-VALUES THEN
        DISPLAY 'MQREMEXP: SPECIFY DEAD_LETTER_QUEUE NAME'
        STOP RUN
     END-IF.
 GET-PARAMETERS-EXIT.
     EXIT.
 MAIN-WORK.
* CONNECT TO THE QMANAGER
     CALL 'MQCONN' USING WØ2-MQM
                         WØ3-HCONN
                         WØ3-COMPCODE
                         WØ3-REASON.
     IF (WØ3-COMPCODE NOT = MQCC-OK) THEN
        DISPLAY 'MQREMEXP: MQCONN ENDED WITH REASON ' WØ3-REASON
        STOP RUN
     END-IF.
```

```
* OPEN THE DEAD_LETTER_Q FOR INPUT
     MOVE MQOT-Q            TO MQOD-OBJECTTYPE.
     MOVE WØ2-OBJECT        TO MQOD-OBJECTNAME.
     COMPUTE WØ3-OPTIONS  = MQOO-BROWSE +
                            MQOO-INPUT-SHARED.
     CALL 'MQOPEN' USING WØ3-HCONN
                         MQOD
                         WØ3-OPTIONS
                         WØ3-HOBJ
                         WØ3-COMPCODE
                         WØ3-REASON.

     IF (WØ3-COMPCODE NOT = MQCC-OK) THEN
        DISPLAY 'MQREMEXP: MQOPEN ' WØ2-OBJECT 'ENDED WITH '
                                             WØ3-REASON
        PERFORM A-MAIN-DISCONNECT
     END-IF.
     COMPUTE MQGMO-OPTIONS = MQGMO-NO-WAIT +
                             MQGMO-SYNCPOINT.
* WE TRY TO MQGET A MESSAGE WITH A MESSAGE-ID AND CORREL-ID
* THAT WOULDN'T EXIST ON THE DEAD_LETTER_Q. SO PASSING OVER
* THE EXPIRED AND NON EXPIRED MESSAGES ON THE Q SEARCHING
* FOR A MESSAGE THAT DOES NOT EXIST, ALL EXPIRED MESSAGES
* WILL BE REMOVED FROM THE DEAD_LETTER_Q.
     MOVE TARGET-MSGID    TO MQMD-MSGID.
     MOVE TARGET-CORRELID TO MQMD-CORRELID.
     CALL 'MQGET' USING WØ3-HCONN
                        WØ3-HOBJ
                        MQMD
                        MQGMO
                        WØ3-BUFFER-LENGTH
                        WØ3-MESSAGE-BUFFER
                        WØ3-DATA-LENGTH
                        WØ3-COMPCODE
                        WØ3-REASON.
     IF WØ3-REASON = MQRC-NONE
         DISPLAY 'MQREMEXP: FIND TARGET MESSAGE WITH '
         DISPLAY 'MQREMEXP: MSGID    :' TARGET-MSGID
         DISPLAY 'MQREMEXP: CORRELID :' TARGET-CORRELID
         DISPLAY 'MQREMEXP: MESAJ    :' WØ3-MESSAGE-BUFFER
         CALL 'MQBACK' USING WØ3-HCONN
                             WØ3-COMPCODE
                             WØ3-REASON
         IF WØ3-REASON = MQRC-NONE
            DISPLAY 'MQREMEXP: MQBACK ' WØ2-OBJECT ' ENDED '
                                                 WØ3-REASON
         END-IF
     ELSE
        DISPLAY 'MQREMEXP: MQGET ' WØ2-OBJECT 'ENDED '
                                               WØ3-REASON
     END-IF.
```

```
           MOVE MQCO-NONE TO WØ3-OPTIONS.
           CALL 'MQCLOSE' USING WØ3-HCONN
                                WØ3-HOBJ
                                WØ3-OPTIONS
                                WØ3-COMPCODE
                                WØ3-REASON.
           IF (WØ3-COMPCODE NOT = MQCC-OK) THEN
              DISPLAY 'MQREMEXP: CLOSE ' WØ2-OBJECT 'ENDED ' WØ3-REASON
           END-IF.
       MAIN-WORK-EXIT.
           PERFORM A-MAIN-DISCONNECT.
       A-MAIN-DISCONNECT.
           CALL 'MQDISC' USING WØ3-HCONN
                               WØ3-COMPCODE
                               WØ3-REASON.
           IF (WØ3-COMPCODE NOT = MQCC-OK) THEN
              DISPLAY 'MQREMEXP: MQDISC ENDED WITH REASON' WØ3-REASON
           END-IF.
           STOP RUN.
```

RUNMQDLQ

```
CBL NODYNAM,LIB,OBJECT,RENT,RES,APOST
       IDENTIFICATION DIVISION.
       PROGRAM-ID. RUNMQDLQ.
       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT       INFILE
               ORGANIZATION IS       SEQUENTIAL
               FILE STATUS  IS       FS
               ASSIGN       TO       INFILE.
       I-O-CONTROL.
       DATA DIVISION.
       FILE SECTION.
       FD  INFILE
           LABEL     RECORDS    ARE     STANDARD
           RECORDING MODE       IS      F
           BLOCK     CONTAINS   Ø       CHARACTERS
           RECORD    CONTAINS   8Ø      CHARACTERS
           DATA      RECORD     IS      INFILE-REC.
       Ø1  INFILE-REC.
           Ø5  FILLER          PICTURE  X(8Ø).
       WORKING-STORAGE SECTION.
       Ø1  FS                     PICTURE XX.
       Ø1  INFILE-STATUS          PICTURE X     VALUE SPACE.
           88  INFILE-END                      VALUE 'E'.
       Ø1  INREC.
           Ø5  INREC-ARRAY OCCURS 5Ø TIMES.
              1Ø  INREC-VALUE-FLD.
```

```
           15 INREC-FIELD      PICTURE X(2Ø) VALUE SPACES.
           15 INREC-VALUE      PICTURE X(48) VALUE SPACES.
       1Ø  INREC-ACTION-FLD.
           15 INREC-ACTION     PICTURE X(1Ø) VALUE SPACES.
           15 INREC-FWDQUEUE   PICTURE X(48) VALUE SPACES.
           15 INREC-FWDQMGR    PICTURE X(48) VALUE SPACES.
* THE PROCESS CAN BE DONE FOR GIVEN DESTINATION QUEUE NAMES,
* FOR ALL DESTINATION QUEUES OR FOR GIVEN DESTINATION QMANAGER
* NAME
 Ø1  INREC-FIELDS.
     Ø5  DESTQNAME         PICTURE X(2Ø) VALUE 'DESTQNAME'.
     Ø5  ALLQUEUES         PICTURE X(2Ø) VALUE 'ALLQUEUES'.
     Ø5  DESTQMGRNAME      PICTURE X(2Ø) VALUE 'DESTQMGRNAME'.
* 3 TYPES OF ACTION : RETRY, DISCARD, FORWARD
 Ø1  INREC-ACTIONS.
     Ø5  RETRY             PICTURE X(1Ø) VALUE 'RETRY'.
     Ø5  DISCARD           PICTURE X(1Ø) VALUE 'DISCARD'.
     Ø5  FORWARD           PICTURE X(1Ø) VALUE 'FORWARD'.
 Ø1  COMPARE-VALUE         PICTURE X(48).
 Ø1  MATCH-STATUS          PICTURE X     VALUE SPACE.
     88  MATCH-FOUND                     VALUE 'E'.
 Ø1  INDEKS                PICTURE 99 VALUE Ø.
 Ø1  I                     PICTURE 99 VALUE Ø.
 Ø1  WØ1-MQM-NAME               PIC X(48) VALUE SPACES.
 Ø1  WØ1-QUEUE-NAME             PIC X(48) VALUE SPACES.
 Ø1  WØ2-MQM                    PIC X(48) VALUE SPACES.
 Ø1  WØ2-OBJECT                 PIC X(48) VALUE SPACES.
 Ø1  WØ3-BUFFER-LENGTH          PIC S9(9) BINARY  VALUE 5198.
 Ø1  WØ3-HCONN                  PIC S9(9) BINARY.
 Ø1  WØ3-OPTIONS                PIC S9(9) BINARY.
 Ø1  WØ3-HOBJ                   PIC S9(9) BINARY.
 Ø1  WØ3-HOBJ-OUT               PIC S9(9) BINARY.
 Ø1  WØ3-DATA-LENGTH            PIC S9(9) BINARY.
 Ø1  WØ3-COMPCODE               PIC S9(9) BINARY.
 Ø1  WØ3-PUTCODE                PIC S9(9) BINARY.
 Ø1  WØ3-REASON                 PIC S9(9) BINARY.
 Ø1  WØ3-MESSAGE-BUFFER.
     Ø5 WØ3-DLQ-HEADER.
        COPY CMQDLHL.
     Ø5 WØ3-MESSAGE-DATA        PIC X(5ØØØ).
 Ø1  WØ5-MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
 Ø1  WØ5-MQM-MESSAGE-DESCRIPTOR.
     COPY CMQMDV.
 Ø1  WØ5-MQM-GET-MESSAGE-OPTIONS.
     COPY CMQGMOV.
 Ø1  WØ5-MQM-PUT-MESSAGE-OPTIONS.
     COPY CMQPMOV.
 Ø1  WØ5-MQM-CONSTANTS.
     COPY CMQV.
```

```
 Ø1  WØ6-CSQ4-OK             PIC S9(4) VALUE Ø.
 Ø1  WØ6-CSQ4-WARNING        PIC S9(4) VALUE 4.
 Ø1  WØ6-CSQ4-ERROR          PIC S9(4) VALUE 8.
 LINKAGE SECTION.
 Ø1  PARMDATA.
     Ø5  PARM-LEN                PIC S9(Ø3) BINARY.
     Ø5  PARM-STRING             PIC X(1ØØ).
 PROCEDURE DIVISION USING PARMDATA.
 A-MAIN SECTION.
     PERFORM GET-PARAMETERS
        THRU GET-PARAMETERS-EXIT.
     PERFORM READ-INFILE
        THRU READ-INFILE-EXIT.
     PERFORM MAIN-WORK
        THRU MAIN-WORK-EXIT.
 GET-PARAMETERS.
* QMANAGER NAME AND THE DEAD_LETTER_Q NAME SHOULB BE
* BE GIVEN AS INPUT PARAMETERS FROM THE INVOKED JCL
     IF PARM-LEN = Ø THEN
        DISPLAY 'NO PARAMAETER SPECIFIED'
        DISPLAY 'GIVE QMGRNAME AND DLQNAME SEPARATED BY COMMA'
        STOP RUN
     END-IF.
     UNSTRING PARM-STRING DELIMITED BY ALL ','
                             INTO WØ2-MQM
                                  WØ2-OBJECT.
     MOVE WØ2-MQM    TO WØ1-MQM-NAME.
     MOVE WØ2-OBJECT TO WØ1-QUEUE-NAME.
     IF WØ2-MQM = SPACES OR WØ2-MQM = LOW-VALUES THEN
        DISPLAY 'DEFAULT QMGR USED'
     END-IF.
     IF WØ2-OBJECT = SPACES OR WØ2-OBJECT = LOW-VALUES THEN
        DISPLAY 'SPECIFY DLQ NAME'
        STOP RUN
     END-IF.
 GET-PARAMETERS-EXIT.
     EXIT.
 READ-INFILE.
* WE READ THE INPUT FILE OF COMMANDS FOR PROCESSING THE DLQ
     OPEN INPUT INFILE.
     MOVE SPACE TO INFILE-STATUS.
     PERFORM SCAN-FILE
        THRU SCAN-FILE-EXIT
       UNTIL INFILE-END.
     CLOSE INFILE.
 READ-INFILE-EXIT.
     EXIT.
 SCAN-FILE.
     ADD 1 TO INDEKS.
* WE READ THE INPUT FILE 2 LINES AT A TIME, FOR THE DESTINATION
```

```
* OBJECT NAME AND THE ACTION REQUESTED AND WE PUT EACH COMMAND
* ON AN ARRAY
           READ INFILE NEXT
          AT END MOVE 'E' TO INFILE-STATUS.
     UNSTRING INFILE-REC DELIMITED BY ALL ','
                           INTO INREC-FIELD (INDEKS)
                                INREC-VALUE (INDEKS).
     READ INFILE NEXT
          AT END MOVE 'E' TO INFILE-STATUS.
     UNSTRING INFILE-REC DELIMITED BY ALL ','
                           INTO INREC-ACTION   (INDEKS)
                                INREC-FWDQUEUE (INDEKS)
                                INREC-FWDQMGR  (INDEKS).
 SCAN-FILE-EXIT.
     EXIT.
 MAIN-WORK.
     CALL 'MQCONN' USING W02-MQM
                         W03-HCONN
                         W03-COMPCODE
                         W03-REASON.
     IF (W03-COMPCODE NOT = MQCC-OK) THEN
        DISPLAY 'MQCONN ENDED WITH REASON ' W03-REASON
        STOP RUN
     END-IF.
     MOVE MQOT-Q            TO MQOD-OBJECTTYPE.
     MOVE W02-OBJECT        TO MQOD-OBJECTNAME.
     COMPUTE W03-OPTIONS  = MQOO-BROWSE +
                            MQOO-INPUT-SHARED.
     CALL 'MQOPEN' USING W03-HCONN
                         MQOD
                         W03-OPTIONS
                         W03-HOBJ
                         W03-COMPCODE
                         W03-REASON.
     IF (W03-COMPCODE NOT = MQCC-OK) THEN
        DISPLAY 'MQOPEN ' W02-OBJECT 'ENDED WITH ' W03-REASON
        PERFORM A-MAIN-DISCONNECT
     END-IF.
* MQGET BROWSE FIRST MESSAGE
     MOVE MQGMO-NO-WAIT         TO MQGMO-OPTIONS.
     ADD  MQGMO-BROWSE-FIRST    TO MQGMO-OPTIONS.
     ADD  MQGMO-ACCEPT-TRUNCATED-MSG TO MQGMO-OPTIONS.
     CALL 'MQGET' USING W03-HCONN
                        W03-HOBJ
                        MQMD
                        MQGMO
                        W03-BUFFER-LENGTH
                        W03-MESSAGE-BUFFER
                        W03-DATA-LENGTH
                        W03-COMPCODE
                        W03-REASON.
```

```
* LOOP UNTIL END OF MESSAGES
     PERFORM WITH TEST BEFORE
             UNTIL ((WØ3-COMPCODE NOT = MQCC-OK) AND
             (WØ3-REASON NOT = MQRC-TRUNCATED-MSG-ACCEPTED))
        PERFORM PROCESS-MESSAGE
           THRU PROCESS-MESSAGE-EXIT
        MOVE MQMI-NONE TO MQMD-MSGID
        MOVE MQCI-NONE TO MQMD-CORRELID
        MOVE SPACES TO WØ3-MESSAGE-BUFFER
        MOVE MQGMO-NO-WAIT     TO MQGMO-OPTIONS
        ADD  MQGMO-BROWSE-NEXT TO MQGMO-OPTIONS
        ADD  MQGMO-ACCEPT-TRUNCATED-MSG TO MQGMO-OPTIONS
* MQGET BROWSE NEXT MESSAGE
        CALL 'MQGET' USING WØ3-HCONN
                           WØ3-HOBJ
                           MQMD
                           MQGMO
                           WØ3-BUFFER-LENGTH
                           WØ3-MESSAGE-BUFFER
                           WØ3-DATA-LENGTH
                           WØ3-COMPCODE
                           WØ3-REASON
* END LOOP
     END-PERFORM.
     IF ((WØ3-COMPCODE NOT = MQCC-OK) AND
        (WØ3-REASON NOT = MQRC-TRUNCATED-MSG-ACCEPTED)) THEN
        DISPLAY 'MQGET ' WØ2-OBJECT 'ENDED ' WØ3-REASON
     END-IF.
     MOVE MQCO-NONE TO WØ3-OPTIONS.
     CALL 'MQCLOSE' USING WØ3-HCONN
                          WØ3-HOBJ
                          WØ3-OPTIONS
                          WØ3-COMPCODE
                          WØ3-REASON.
     IF (WØ3-COMPCODE NOT = MQCC-OK) THEN
        DISPLAY 'CLOSE ' WØ2-OBJECT 'ENDED ' WØ3-REASON
     END-IF.
 MAIN-WORK-EXIT.
     PERFORM A-MAIN-DISCONNECT.
 PROCESS-MESSAGE.
     MOVE Ø TO I.
     MOVE 'H' TO MATCH-STATUS.
* FOR EACH MESSAGE BROWSED, LOOK IF IT IS SUBJECT TO THE
* COMMANDS IN THE ARRAY
     PERFORM FIND-MATCH
        THRU FIND-MATCH-EXIT
       UNTIL I NOT < INDEKS OR MATCH-FOUND.
     IF MATCH-FOUND
        PERFORM TAKE-ACTION
           THRU TAKE-ACTION-EXIT.
```

```
PROCESS-MESSAGE-EXIT.
    EXIT.
FIND-MATCH.
    ADD 1 TO I.
    EVALUATE INREC-FIELD (I)
       WHEN DESTQNAME
          MOVE MQDLH-DESTQNAME TO COMPARE-VALUE
          IF COMPARE-VALUE = INREC-VALUE (I)
             MOVE 'E' TO MATCH-STATUS
          END-IF
       WHEN ALLQUEUES
          MOVE 'E' TO MATCH-STATUS
       WHEN DESTQMGRNAME
          MOVE MQDLH-DESTQMGRNAME TO COMPARE-VALUE
          IF COMPARE-VALUE = INREC-VALUE (I)
             MOVE 'E' TO MATCH-STATUS
          END-IF
    END-EVALUATE.
FIND-MATCH-EXIT.
    EXIT.
TAKE-ACTION.
    EVALUATE INREC-ACTION (I)
       WHEN DISCARD
          PERFORM DISCARD-MESSAGE
             THRU DISCARD-MESSAGE-EXIT
       WHEN RETRY
          PERFORM RETRY-MESSAGE
             THRU RETRY-MESSAGE-EXIT
       WHEN FORWARD
          PERFORM FORWARD-MESSAGE
             THRU FORWARD-MESSAGE-EXIT
    END-EVALUATE.
TAKE-ACTION-EXIT.
    EXIT.
A-MAIN-DISCONNECT.
    CALL 'MQDISC' USING W03-HCONN
                        W03-COMPCODE
                        W03-REASON.
    IF (W03-COMPCODE NOT = MQCC-OK) THEN
       DISPLAY 'MQDISC ENDED WITH REASON' W03-REASON
    END-IF.
    STOP RUN.
DISCARD-MESSAGE.
    MOVE MQMI-NONE   TO MQMD-MSGID
    COMPUTE MQGMO-OPTIONS = MQGMO-NO-WAIT +
                            MQGMO-MSG-UNDER-CURSOR.
    ADD  MQGMO-ACCEPT-TRUNCATED-MSG TO MQGMO-OPTIONS.
    CALL 'MQGET' USING W03-HCONN
                       W03-HOBJ
                       MQMD
```

```
                        MQGMO
                        WØ3-BUFFER-LENGTH
                        WØ3-MESSAGE-BUFFER
                        WØ3-DATA-LENGTH
                        WØ3-COMPCODE
                        WØ3-REASON
    IF WØ3-COMPCODE = MQCC-OK THEN
       CALL 'MQCMIT' USING WØ3-HCONN
                           WØ3-COMPCODE
                           WØ3-REASON
       IF WØ3-COMPCODE NOT = MQCC-OK THEN
           DISPLAY 'CMIT' WØ2-OBJECT 'ENDED ' WØ3-REASON
       END-IF
    ELSE
       IF ((WØ3-COMPCODE NOT = MQCC-OK) AND
          (WØ3-REASON NOT = MQRC-TRUNCATED-MSG-ACCEPTED)) THEN
           DISPLAY 'GET ' WØ2-OBJECT 'ENDED ' WØ3-REASON
       END-IF
    END-IF.
DISCARD-MESSAGE-EXIT.
    EXIT.
RETRY-MESSAGE.
    MOVE MQCC-FAILED TO WØ3-PUTCODE.
    MOVE MQDLH-DESTQNAME       TO MQOD-OBJECTNAME.
    MOVE MQDLH-DESTQMGRNAME    TO MQOD-OBJECTQMGRNAME.
    PERFORM PUT-MESSAGE
       THRU PUT-MESSAGE-EXIT.
RETRY-MESSAGE-EXIT.
    EXIT.
FORWARD-MESSAGE.
    MOVE MQCC-FAILED TO WØ3-PUTCODE.
    MOVE INREC-FWDQUEUE (I)    TO MQOD-OBJECTNAME.
    MOVE INREC-FWDQMGR  (I)    TO MQOD-OBJECTQMGRNAME.
    PERFORM PUT-MESSAGE
       THRU PUT-MESSAGE-EXIT.
FORWARD-MESSAGE-EXIT.
    EXIT.
PUT-MESSAGE.
    MOVE MQOT-Q                TO MQOD-OBJECTTYPE.
    COMPUTE WØ3-OPTIONS = MQOO-OUTPUT.
    CALL 'MQOPEN' USING WØ3-HCONN
                        MQOD
                        WØ3-OPTIONS
                        WØ3-HOBJ-OUT
                        WØ3-COMPCODE
                        WØ3-REASON.
    IF WØ3-COMPCODE NOT = MQCC-OK
        DISPLAY 'OPEN ' MQOD-OBJECTNAME 'ENDED ' WØ3-REASON
    ELSE
        MOVE MQMT-DATAGRAM  TO MQMD-MSGTYPE
```

```
               MOVE MQDLH-FORMAT   TO MQMD-FORMAT
               SUBTRACT 172 FROM WØ3-DATA-LENGTH
               CALL 'MQPUT' USING  WØ3-HCONN
                                   WØ3-HOBJ-OUT
                                   MQMD
                                   MQPMO
                                   WØ3-DATA-LENGTH
                                   WØ3-MESSAGE-DATA
                                   WØ3-PUTCODE
                                   WØ3-REASON
               IF WØ3-PUTCODE = MQCC-OK
                   PERFORM DISCARD-MESSAGE
                      THRU DISCARD-MESSAGE-EXIT
               ELSE
                   DISPLAY 'PUT ' MQOD-OBJECTNAME 'ENDED ' WØ3-REASON
               END-IF
               CALL 'MQCLOSE' USING WØ3-HCONN
                                    WØ3-HOBJ-OUT
                                    MQCO-NONE
                                    WØ3-COMPCODE
                                    WØ3-REASON
               IF WØ3-COMPCODE NOT = MQCC-OK
                   DISPLAY 'CLOSE ' MQOD-OBJECTNAME 'ENDED' WØ3-REASON
               END-IF
           END-IF.
       PUT-MESSAGE-EXIT.
           EXIT.
```

RUNMQDLJ

```
//RUNMQDLQ JOB (ACCT÷),CLASS=1,MSGLEVEL=(1,1),
// NOTIFY=&SYSUID,MSGCLASS=X
//* THIS STEP WILL REMOVE ALL EXPIRED MESSAGES ON THE DLQ
//RUNMQEXP EXEC PGM=MQREMEXP,PARM=('PMQ1','PMQ1.DEAD.QUEUE')
//STEPLIB  DD   DSN=USER.TIS.LOADLIB,DISP=SHR
//         DD   DSN=SYS1.COB2LIB,DISP=SHR
//         DD   DSN=PMQM.SCSQANLE,DISP=SHR
//         DD   DSN=PMQM.SCSQAUTH,DISP=SHR
//         DD   DSN=PMQM.SCSQCOBC,DISP=SHR
//         DD   DSN=PMQM.SCSQLOAD,DISP=SHR
//SYSDBOUT DD   SYSOUT=*
//SYSABOUT DD   SYSOUT=*
//SYSPRINT DD   SYSOUT=*
//SYSOUT   DD   SYSOUT=*
//* THIS STEP WILL PROCESS UNEXPIRED MESSAGES
//RUNMQDLQ EXEC PGM=RUNMQDLQ,PARM=('PMQ1','PMQ1.DEAD.QUEUE')
//STEPLIB  DD   DSN=USER.TIS.LOADLIB,DISP=SHR
//         DD   DSN=SYS1.COB2LIB,DISP=SHR
//         DD   DSN=PMQM.SCSQANLE,DISP=SHR
//         DD   DSN=PMQM.SCSQAUTH,DISP=SHR
```

```
//         DD   DSN=PMQM.SCSQCOBC,DISP=SHR
//         DD   DSN=PMQM.SCSQLOAD,DISP=SHR
//SYSDBOUT DD   SYSOUT=*
//INFILE   DD   DSN=USER.TIS.JCL(MQDLQIN),DISP=SHR
//*    SAMPLE INPUT FILE CONTENT :
//*EACH COMMAND LIES ON TWO CONSECUTIVE LINES
//*1ST LINE : DESTQNAME,destination_Q_name
//*2ND LINE : action[,forwardq,forwardqmgr]
//* or
//*1ST LINE : DESTQMGRNAME,destination_qmgr_name
//*2ND LINE : action[,forwardq,forwardqmgr]
//* or
//*1ST LINE : ALLQUEUES
//*2ND LINE : action[,forwardq,forwardqmgr]
//*DESTQNAME,YSP2CICS.INITQ
//*DISCARD
//*ALLQUEUES
//*RETRY
//*DESTQMGRNAME,MQ888
//*RETRY
//*DESTQNAME,CMON
//*FORWARD,BACKQ,MQ888
//SYSABOUT DD   SYSOUT=*
//SYSPRINT DD   SYSOUT=*
//SYSOUT   DD   SYSOUT=*
```

MQDLQIN

```
DESTQNAME,YSPCICS.INITQ
DISCARD
DESTQMGRNAME,MQ887
DISCARD
DESTQNAME,CMON
FORWARD,BACKQ,MQ887
ALLQUEUES
RETRY
```

Serkan Kocak
Transaction and Messaging Systems Specialist (Turkey)

MQSeries Wrappers

INTRODUCTION

This article briefly describes why MQSeries Wrappers are commonly created. It then goes on to describe an example of an MQSeries Wrapper, detailing the specification and how it is used. Finally, some sample code is listed – including the key parts of the Wrapper itself and extracts from programs that call it. The article does not supply a 'ready-to-use' Wrapper, but provides the basic ideas and specification (backed by code) necessary to build one. The Wrapper was written to run on OS/390 mainframes (CICS or IMS and DB2), but can easily be ported to other platforms.

WHY CODE AN MQSERIES WRAPPER?

The IBM-supplied application programming interface to MQSeries (the MQI) is sometimes considered excessively complex because of the myriad of options that can be specified on the main calls. Most business applications do not want to bother with the specification of these options; however, these options are essential if MQSeries is to remain the flexible product that it is today.

There are many implementations of the MQI. Most programming languages (eg Java) have their own 'MQ Client' that allows the developer to make calls to MQSeries. These MQ Clients allow the developer access to MQSeries via their own set of calls (which do not usually match those calls supplied in the MQI). Also, most MQ Clients do not offer all of the available MQI options. These two features of MQ Client software are also usually features of MQSeries 'Wrappers'.

The two main aims of an MQSeries Wrapper are:

- To provide a simple interface for business application programs that wish to use MQSeries. This is best achieved by minimizing:
  - the number of calls that the business application must make
  - the amount of information that the business application needs to specify on each call.

- To maintain the flexibility of the MQSeries product, ie not to impose its own restrictions by assuming default values for MQSeries options.

It is very easy for an MQSeries Wrapper (or MQ Client) to achieve one of these objectives. It is more difficult to achieve both. For example, a Wrapper could supply calls of SEND and RECEIVE and translate these to the necessary MQCONN, MQOPEN, MQPUT, MQGET, MQCLOSE, and MQDISC calls, assuming its chosen default values for all the MQSeries options. This achieves the objective of simplicity, but not flexibility.

Alternatively, a more flexible Wrapper could supply the same SEND and RECEIVE calls, but allow the calling program to specify what the Wrapper considers to be the 'key' MQSeries options. However, as time passes, requirements usually arise for calling programs to specify an increasing number of different MQSeries options. The defaults assumed by the Wrapper are not sufficient. The Wrapper allows the specification of more and more 'key' options, but the further the Wrapper goes down this route, the more it loses sight of its first objective; the Wrapper becomes flexible, but not simple.

AN MQSERIES WRAPPER SPECIFICATION

The remainder of this article describes a very simple MQSeries Wrapper. It was developed to run on the mainframe (in batch or under any version of CICS or IMS), but could very easily be ported to other platforms. The Wrapper supplies a simple set of calls (SEND, RECEIVE, etc) and maintains the flexibility of the IBM-defined product. All MQI options are also available to users of the Wrapper.

It achieves the objectives described above by removing all the option-setting from program code – either the calling program or the Wrapper itself. All MQSeries options are defined in DB2 tables (these are very basic tables that could easily be ported to another database product).

Because all the MQSeries options are held in DB2 tables there is the additional benefit of being able to change any MQSeries option 'on-the-fly' – without the need for program changes. Equally, it allows MQSeries queue names to be changed dynamically.

New MQSeries functionality (eg additional options supplied by IBM) can also be exploited without the need for changes to code.

SPECIFYING THE MQSERIES OPTIONS

All main MQSeries options are specified within six IBM-defined structures.

- Object Descriptor (OD).
- Open Options (OO).
- Message Descriptor (MQMD).
- Put Message Options (PMO).
- Get Message Options (GMO).
- Close Options (CO).

Note that further structures can easily be added as and when required; for example, a Wrapper may be needed in order to format and add an MQRFH2 header. This can be achieved easily, using default values and a newly-created DB2 table to hold specific instances of MQRFH2 headers, should they need to be defined.

Each structure has its own DB2 table. For example, there is an OD_TABLE containing all the different Object Descriptors that are currently in use – each defined on a single row. Because the Object Descriptor contains the "Queue Name" field, the OD_TABLE has more rows than the other tables (there needs to be a row added to the OD_TABLE for each MQSeries queue that is accessed via the Wrapper).

Each row in the OD_Table, OO_Table, MD_Table, and CO_Table actually has room for two entries (ie two Object Descriptors). This is for use on the SEND RECEIVE function that will open and close two queues (the request queue and the reply queue).

Each row in each table is given a unique name (key) that is passed by the calling program to the Wrapper. The Wrapper is then able to retrieve the required structure(s) from DB2 and perform the necessary MQSeries calls.

The calling program need not specify names for all structures. The Wrapper contains default values for all MQSeries options – like many other Wrappers. However, the calling program can override any of the default values simply by specifying the name of a structure that has been defined in one of the DB2 tables. The Wrapper attempts to retrieve a row for each structure name that is not composed of spaces.

As mentioned previously, the most populous table will be the OD_TABLE. Entries in this table can be created automatically whenever a new MQSeries queue is defined. This can be achieved very easily in a mainframe environment by submitting a single batch job containing two steps, each of which executes an IBM-supplied utility:

- CSQUTIL – this defines the MQSeries queue.
- DSNTEP2 – this executes the SQL required to add a row to the OD_TABLE.

There are, of course, many ways to insert, update, or delete rows from DB2 tables. Submitting OS/390 batch jobs is simple, but effective. It would be relatively easy to develop a front-end to the DB2 tables. This would allow easy (and dynamic) control over all of the MQSeries options in use by any MQSeries program.

THE WRAPPER INTERFACE

As mentioned previously, the Wrapper being described is not restricted to OS/390 operation. However, for ease of explanation, the following describes an OS/390 MQSeries Wrapper.

The Wrapper exists as an individual load module. It may be dynamically called in the CICS, IMS, or batch environments. The calling program must specify three parameters to be passed to the Wrapper:

- Wrapper Control Data.
- MQSeries Structure Names.
- Message Data.

Each of these parameters is described below.

Wrapper Control Data

This is an input/output area that must be passed to the Wrapper (see Figure 1). Each field is described in more detail later in the article.

Field	Length	Format	Description
Function	30	Character	Input: must be specified by the calling program.
Queue Manager	48	Character	Input: must be specified by the calling program in IMS and batch. Unused in CICS.
Message Length	4	Binary	Input: must be specified by the calling program (if sending).
Buffer Length	4	Binary	Input: must be specified by the calling program (if receiving data).
Connection Handle	4	Binary	Output/input: returned by the Wrapper, but must be passed on subsequent calls if sending or receiving multiple messages.
Object Handle Out	4	Binary	Output/input: returned by the Wrapper, but must be passed on subsequent calls if sending or receiving multiple messages.
Object Handle In	4	Binary	Output/input: returned by the Wrapper, but must be passed on subsequent calls if sending or receiving multiple messages.
Return Code	4	Binary	Output: set by the Wrapper.
MQSeries Error Details Group consisting of	78	Group	Output: set by the Wrapper
..MQ Operation	12	Character	Output: describes the failing MQSeries call (if any).
..MQ Object Name	48	Character	Output: describes the object name (eg queue) relating to the failure (if any).
..Completion Code	9	Character	Output: IBM MQSeries Completion Code.
..Reason Code	9	Character	Output: IBM MQSeries Reason Code.

Figure 1: Wrapper Control Data

MQSeries Structure Names

This is an input/output area that must be passed to the Wrapper (see Figure 2). Structures not required must be specified as spaces. Further structures can easily be added if required.

Field	Length	Format	Description
Number of Structures	2	Binary	Input: specifies the number of structures that follow. Essential for allowing the easy addition of new structures, whilst retaining backwards integrity of the Wrapper code.
Object Descriptor (OD) Name	48	Character	Input: may be specified by the calling program. Unique name given to the Object Descriptor structure to be used by the calling program.
OD SQL Code	4	Binary	Output: returned by the Wrapper.
Open Options (OO) Name	48	Character	Input: may be specified by the calling program. Unique name given to the Open Options structure defined by the calling program.
OO SQL Code	4	Binary	Output: returned by the Wrapper.
Message Descriptor (MD) Name	48	Character	Input: may be specified by the calling program. Unique name given to the Message Descriptor structure defined by the calling program.
MD SQL Code	4	Binary	Output: returned by the Wrapper.
Put Message Options (PMO) Name	48	Character	Input: may be specified by the calling program. Unique name given to the Put Message Options structure defined by the calling program.
PMO SQL Code	4	Binary	Output: returned by the Wrapper.
Get Message Options (GMO) Name	48	Character	Input: may be specified by the calling program. Unique name given to the Get Message Options structure defined by the calling program.
GMO SQL Code	4	Binary	Output: returned by the Wrapper.
Close Options (CO) Name	48	Character	Input: may be specified by the calling program. Unique name given to the Close Options structure defined by the calling program.
CO SQL Code	4	Binary	Output: returned by the Wrapper.

Figure 2: MQSeries Structure Names

Message buffer

This is an input area when data is being sent and an output area when data is being received. The calling program must pass this area to the Wrapper. If the Wrapper does not wish to impose any limits of its own on MQSeries applications this buffer should be 4 megabytes in length (the IBM limit for an MQSeries message size). However, there may be performance reasons to make this buffer smaller.

Wrapper Control Data field descriptions

Function (input)

This field can take the following values:

- SEND – the Wrapper will open the appropriate MQSeries message queue, put a single message onto the queue, and then close it.
- RECEIVE – the Wrapper will open the appropriate MQSeries message queue, get a single message from the queue, and then close it.
- SEND FIRST – the Wrapper will open the appropriate MQSeries message queue, put a message onto it, but leave the queue open – expecting another message to be specified by the calling business application in a subsequent call (SEND AGAIN).
- SEND AGAIN – the Wrapper puts a message onto the already opened MQSeries message queue (using the Object Handle Out field returned to it by the calling business application). The Wrapper does not close the MQSeries message queue.
- SEND LAST – the Wrapper puts a message onto the already opened MQSeries message queue (using the Object Handle Out field, returned to it by the calling business application). The Wrapper then closes the MQSeries message queue.
- SEND RECEIVE – the Wrapper puts a message onto the appropriate MQSeries message queue and then gets another message (probably after waiting for a reply) from another MQSeries message queue.
- RECEIVE FIRST – the Wrapper will open the appropriate MQSeries message queue, get a message from it, but leave the

queue open – expecting a request to get another message from the queue in a subsequent call (RECEIVE AGAIN).

- RECEIVE AGAIN – the Wrapper gets a message from the already opened MQSeries message queue (using the Object Handle In field returned to it by the calling business application). The Wrapper does not close the MQSeries message queue.
- RECEIVE LAST – the Wrapper gets a message from the already opened MQSeries message queue (using the Object Handle In field returned to it by the calling business application). The Wrapper then closes the MQSeries message queue.
- COMMIT – the Wrapper should commit all puts/gets.

Queue Manager (input)

This field contains the name of the queue manager to which the Wrapper should connect. It must be specified in the IMS or batch environments. It is ignored in the CICS environment since, under CICS, there is an implicit connection to an MQSeries queue manager that is determined by the definition of the CICS region.

Message Length (input)

This field is only required for SEND-type functions. It determines the length of the MQSeries message that the Wrapper will put onto the specified MQSeries queue. In the interests of performance this field should be calculated accurately by the calling business application.

Buffer Length (input)

This field is only required for RECEIVE-type functions. It informs the Wrapper of the length of the area in the business application program (Message Data) into which the Wrapper should put the MQSeries message. If the message is bigger than this buffer length the Wrapper will report an error. In this case, the message will either be retrieved truncated or the MQSeries get will fail, depending on the setting of Get Message Options.

Connection Handle (output/input)

This field is only required if the calling business application wishes to

send or receive multiple messages. It is returned by the Wrapper after the first successful SEND FIRST or RECEIVE FIRST request. The calling business application must pass back this field unchanged on all subsequent SEND AGAIN or RECEIVE AGAIN requests and also on the final SEND LAST or RECEIVE LAST request, when the Wrapper will disconnect from the MQSeries queue manager and so invalidate this field.

Object Handle Out (output/input)

This field is only required if the calling business application wishes to send multiple messages. It is returned by the Wrapper after the first successful SEND FIRST request. The calling business application must pass back this field unchanged on all subsequent SEND AGAIN requests and also on the final SEND LAST request, when the Wrapper will close the MQSeries output queue and so invalidate this field.

Object Handle In (output/input)

This field is only required if the calling business application wishes to receive multiple messages. It is returned by the Wrapper after the first successful RECEIVE FIRST request. The calling business application must pass back this field unchanged on all subsequent RECEIVE AGAIN requests and also on the final RECEIVE LAST request, when the Wrapper will close the MQSeries input queue and so invalidate this field.

Return Code (output)

This field is set by the Wrapper on all requests. It can contain the following values:

- 0 – successful.
- +8 – MQSeries error (see *Error Details*, below).
- +12 – DB2 error.
- +16 – invalid function specified.

Error Details

The Wrapper should report all errors and also return the error details to the calling program.

SUMMARY

This Wrapper provides a simple, easy-to-maintain, easy-to-use, powerful interface to MQSeries. A business application that simply wishes to put messages onto a queue need specify only:

- SEND.
- Queue manager name (even this is not necessary if the queue is on the default queue manager or if the calling program is executing in CICS).
- A chosen name to identify the queue.
- The message length.
- The message data.

Defaults can be taken for most MQSeries options, remembering that any can be overridden. Maintaining the MQSeries options in DB2 tables allows dynamic flexibility and also an element of control. If access to the tables was restricted to MQSeries experts then developers would need to consult with these experts to agree on the options most suitable for their particular business application, rather than simply adopt the options used in some sample code that happen to work.

It makes sense to define the IBM structures in DB2 (not structures invented by the Wrapper) because this makes the Wrapper easy to maintain/enhance as new releases of MQSeries arrive.

Each of the user-defined structure names provides the unique-key index into the DB2 table. In usual DB2 terms, the tables are very small, so there is minimal performance overhead.

SAMPLE CODE (COBOL)

The Wrapper (CICS, batch, or IMS)

I stress that this is not a complete 'ready-to-use' Wrapper. The pieces of code that follow are extracts from a Wrapper and are intended to illustrate the basic ideas. Wrappers can be easily built by extending these extracts and by developing parts to suit individual needs (in particular, error handling). To save space, I have only included the DB2 declarations and the necessary fetch SQL for one of the six

MQSeries structures (the Put Message Options structure). The code was originally written for MQSeries for OS/390 Version 1.2.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. WRAPPER.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
     EXEC SQL DECLARE PMO_TAB TABLE
     ( UNIQUEKEY                         CHAR(3Ø),
       STRUCID                           CHAR(4),
       VERSION                           INTEGER,
       OPTIONS                           INTEGER,
       TIMEOUT                           INTEGER,
       CONTEXT                           INTEGER,
       KNOWNDESTCOUNT                    INTEGER,
       UNKNOWNDESTCOUNT                  INTEGER,
       INVALIDDESTCOUNT                  INTEGER,
       RESOLVEDQNAME                     CHAR(48),
       RESOLVEDQMGRNAME                  CHAR(48)
     ) END-EXEC.
* Example MQSeries Structure (MQPMO)   *
 Ø1  MQPMO-ROW.
     Ø5  MQPMO-UNIQUE-KEY                PIC X(48).
     Ø5  PUT-MESSAGE-OPTIONS.
     EXEC SQL
         INCLUDE CMQPMOV
     END-EXEC.
 Ø1  INITIALIZED-MQOD.
     COPY CMQODV.
 Ø1  INITIALIZED-MQMD.
     COPY CMQMDV.
 Ø1  MQSERIES-CONSTANTS.
     COPY CMQV.
 LINKAGE SECTION.
 Ø1  WRAPPER-CONTROL-DATA.
     Ø5  WRAPPER-FUNCTION                PIC X(3Ø).
     Ø5  WRAPPER-QUEUE-MANAGER           PIC X(48).
     Ø5  WRAPPER-MESSAGE-LENGTH          PIC S9(9) BINARY.
     Ø5  WRAPPER-BUFFER-LENGTH           PIC S9(9) BINARY.
     Ø5  WRAPPER-CONNECTION-HANDLE       PIC S9(9) BINARY.
     Ø5  WRAPPER-OBJECT-HANDLE-OUT       PIC S9(9) BINARY.
     Ø5  WRAPPER-OBJECT-HANDLE-IN        PIC S9(9) BINARY.
     Ø5  WRAPPER-RETURN-CODE             PIC S9(9) BINARY.
     Ø5  WRAPPER-MQSERIES-ERROR-DETAILS.
         1Ø  WRAPPER-MQ-OPERATION        PIC X(12).
         1Ø  WRAPPER-MQ-OBJECTNAME       PIC X(48).
         1Ø  WRAPPER-MQ-COMPLETION-CODE  PIC S9(9) BINARY.
         1Ø  WRAPPER-MQ-REASON-CODE      PIC S9(9) BINARY.
 Ø1  WRAPPER-MQSERIES-STRUCTURES.
     Ø5  NUMBER-OF-STRUCTURES  VALUE 6   PIC 9(4) BINARY.
```

```
     Ø5  MQOD-NAME                        PIC X(48).
     Ø5  MQOD-SQL-CODE                    PIC S9(9) BINARY.
     Ø5  MQOO-NAME                        PIC X(48).
     Ø5  MQOO-SQL-CODE                    PIC S9(9) BINARY.
     Ø5  MQMD-NAME                        PIC X(48).
     Ø5  MQMD-SQL-CODE                    PIC S9(9) BINARY.
     Ø5  MQPMO-NAME                       PIC X(48).
     Ø5  MQPMO-SQL-CODE                   PIC S9(9) BINARY.
     Ø5  MQGMO-NAME                       PIC X(48).
     Ø5  MQGMO-SQL-CODE                   PIC S9(9) BINARY.
     Ø5  MQCO-NAME                        PIC X(48).
     Ø5  MQCO-SQL-CODE                    PIC S9(9) BINARY.
 Ø1  WRAPPER-MESSAGE-BUFFER               PIC X(1Ø48576).
 PROCEDURE DIVISION USING WRAPPER-CONTROL-DATA
                          WRAPPER-MQSERIES-STRUCTURES
                          WRAPPER-MESSAGE-BUFFER.
 MAIN SECTION.
     PERFORM INITIALIZATION.
     PERFORM GET-MQSERIES-STRUCTURES.
     PERFORM PROCESS-REQUEST.
     GOBACK.
 INITIALIZATION SECTION.
* Set default values in MQSeries Structures. *
     INITIALIZE WRAPPER-MQSERIES-ERROR-DETAILS.
     MOVE INITIALIZED-MQOD TO MQOD-OUT
                              MQOD-IN.
     MOVE INITIALIZED-MQMD TO MQMD-OUT
                              MQMD-IN.
     MOVE MQOO-OUTPUT TO         OPEN-OPTIONS-OUT.
     MOVE MQOO-INPUT-AS-Q-DEF TO OPEN-OPTIONS-IN.
     MOVE MQCO-NONE TO CLOSE-OPTIONS-OUT
                       CLOSE-OPTIONS-IN.
 GET-MQSERIES-STRUCTURES SECTION.
* Retrieve named MQSeries Structures   *
     IF MQOD-NAME NOT = SPACES
         PERFORM RETRIEVE-MQOD
     END-IF.
     IF MQOO-NAME NOT = SPACES
         PERFORM RETRIEVE-MQOO
     END-IF.
     IF MQMD-NAME NOT = SPACES
         PERFORM RETRIEVE-MQMD
     END-IF.
     IF MQPMO-NAME NOT = SPACES
         PERFORM RETRIEVE-MQPMO
     END-IF.
     IF MQGMO-NAME NOT = SPACES
         PERFORM RETRIEVE-MQGMO
     END-IF.
     IF MQCO-NAME NOT = SPACES
         PERFORM RETRIEVE-MQCO
```

```
     END-IF.
 RETRIEVE-MQPMO SECTION.
* Retrieve the named Put Message Options. *
     EXEC SQL
         SELECT * INTO :MQPMO-UNIQUE-KEY,
                       :MQPMO-STRUCID,
                       :MQPMO-VERSION,
                       :MQPMO-OPTIONS,
                       :MQPMO-TIMEOUT,
                       :MQPMO-CONTEXT,
                       :MQPMO-KNOWNDESTCOUNT,
                       :MQPMO-UNKNOWNDESTCOUNT,
                       :MQPMO-INVALIDDESTCOUNT,
                       :MQPMO-RESOLVEDQNAME,
                       :MQPMO-RESOLVEDQMGRNAME
             FROM MQPMO_TAB D
                 WHERE D.UNIQUE-KEY = :MQPMO-NAME
     END-EXEC.
     MOVE SQLCODE TO MQPMO-SQL-CODE.
 PROCESS-REQUEST SECTION.
* Main logic                          *
     IF WRAPPER-FUNCTION = 'SEND'
     OR WRAPPER-FUNCTION = 'SEND FIRST'
     OR WRAPPER-FUNCTION = 'SEND RECEIVE'
         PERFORM CONNECT-QMGR
         PERFORM OPEN-QUEUE-OUT
         PERFORM PUT-MESSAGE
     END-IF.
     IF WRAPPER-FUNCTION = 'SEND AGAIN'
     OR WRAPPER-FUNCTION = 'SEND LAST'
         PERFORM PUT-MESSAGE
     END-IF.
     IF WRAPPER-FUNCTION = 'RECEIVE'
     OR WRAPPER-FUNCTION = 'RECEIVE FIRST'
         PERFORM CONNECT-QMGR
         PERFORM OPEN-QUEUE-IN
         PERFORM GET-MESSAGE
     END-IF.
     IF WRAPPER-FUNCTION = 'RECEIVE AGAIN'
     OR WRAPPER-FUNCTION = 'RECEIVE LAST'
         PERFORM GET-MESSAGE
     END-IF.
     IF WRAPPER-FUNCTION = 'SEND RECEIVE'
         PERFORM OPEN-QUEUE-IN
         PERFORM GET-MESSAGE
     END-IF.
     IF WRAPPER-FUNCTION = 'COMMIT'
         PERFORM COMMIT-MESSAGE
     END-IF.
     IF WRAPPER-FUNCTION = 'SEND'
     OR WRAPPER-FUNCTION = 'SEND LAST'
```

```
     OR WRAPPER-FUNCTION = 'SEND RECEIVE'
         PERFORM CLOSE-QUEUE-OUT
     END-IF.
     IF WRAPPER-FUNCTION = 'RECEIVE'
     OR WRAPPER-FUNCTION = 'RECEIVE LAST'
     OR WRAPPER-FUNCTION = 'SEND RECEIVE'
         PERFORM CLOSE-QUEUE-IN
     END-IF.
     IF WRAPPER-FUNCTION = 'SEND'
     OR WRAPPER-FUNCTION = 'SEND LAST'
     OR WRAPPER-FUNCTION = 'SEND RECEIVE'
     OR WRAPPER-FUNCTION = 'RECEIVE'
     OR WRAPPER-FUNCTION = 'RECEIVE LAST'
         PERFORM DISCONNECT-QMGR
     END-IF.
 CONNECT-QMGR SECTION.
* Connect to Queue Manager             *
     CALL 'MQCONN' USING WRAPPER-QUEUE-MANAGER
                         WRAPPER-CONNECTION-HANDLE
                         WRAPPER-MQ-COMPLETION-CODE
                         WRAPPER-MQ-REASON-CODE
     END-CALL.
     IF WRAPPER-MQ-COMPLETION-CODE NOT = MQCC-OK
         MOVE +8 TO                    WRAPPER-RETURN-CODE
         MOVE 'CONNECT-QMGR' TO        WRAPPER-MQ-OPERATION
         MOVE WRAPPER-QUEUE-MANAGER TO WRAPPER-MQ-OBJECTNAME
         GOBACK
     END-IF.
 OPEN-QUEUE-OUT SECTION.
* Open output queue                   *
     CALL 'MQOPEN' USING WRAPPER-CONNECTION-HANDLE
                         MQOD-OUT
                         OPEN-OPTIONS-OUT
                         WRAPPER-OBJECT-HANDLE-OUT
                         WRAPPER-MQ-COMPLETION-CODE
                         WRAPPER-MQ-REASON-CODE
     END-CALL.
     IF WRAPPER-MQ-COMPLETION-CODE NOT = MQCC-OK
         MOVE +8 TO              WRAPPER-RETURN-CODE
         MOVE 'OPEN-Q-OUT' TO    WRAPPER-MQ-OPERATION
         MOVE MQOD-OUT.MQOD-OBJECTNAME TO
                                 WRAPPER-MQ-OBJECTNAME
         GOBACK
     END-IF.
 PUT-MESSAGE SECTION.
* Put message onto queue               *
     CALL 'MQPUT' USING WRAPPER-CONNECTION-HANDLE
                        WRAPPER-OBJECT-HANDLE-OUT
                        MQMD-OUT
                        PUT-MESSAGE-OPTIONS
                        WRAPPER-MESSAGE-LENGTH
```

```
                        WRAPPER-MESSAGE-BUFFER
                        WRAPPER-MQ-COMPLETION-CODE
                        WRAPPER-MQ-REASON-CODE
     END-CALL.
     IF WRAPPER-MQ-COMPLETION-CODE NOT = MQCC-OK
         MOVE +8 TO              WRAPPER-RETURN-CODE
         MOVE 'PUT-MESSAGE' TO   WRAPPER-MQ-OPERATION
         MOVE MQOD-OUT.MQOD-OBJECTNAME TO
                                 WRAPPER-MQ-OBJECTNAME
         GOBACK
     END-IF.
 OPEN-QUEUE-IN SECTION.
* Open input queue           *
     CALL 'MQOPEN' USING WRAPPER-CONNECTION-HANDLE
                         MQOD-IN
                         OPEN-OPTIONS-IN
                         WRAPPER-OBJECT-HANDLE-IN
                         WRAPPER-MQ-COMPLETION-CODE
                         WRAPPER-MQ-REASON-CODE
     END-CALL.
     IF WRAPPER-MQ-COMPLETION-CODE NOT = MQCC-OK
         MOVE +8 TO              WRAPPER-RETURN-CODE
         MOVE 'OPEN-Q-IN' TO     WRAPPER-MQ-OPERATION
         MOVE MQOD-IN.MQOD-OBJECTNAME TO
                                 WRAPPER-MQ-OBJECTNAME
         GOBACK
     END-IF.
 GET-MESSAGE SECTION.
* Get message                  *
     CALL 'MQGET' USING WRAPPER-CONNECTION-HANDLE
                        WRAPPER-OBJECT-HANDLE-IN
                        MQMD-IN
                        GET-MESSAGE-OPTIONS
                        WRAPPER-BUFFER-LENGTH
                        WRAPPER-MESSAGE-BUFFER
                        WRAPPER-MESSAGE-LENGTH
                        WRAPPER-MQ-COMPLETION-CODE
                        WRAPPER-MQ-REASON-CODE
     END-CALL.
     IF WRAPPER-MQ-COMPLETION-CODE NOT = MQCC-OK
         MOVE +8 TO              WRAPPER-RETURN-CODE
         MOVE 'GET-MESSAGE' TO   WRAPPER-MQ-OPERATION
         MOVE MQOD-IN.MQOD-OBJECTNAME TO
                                 WRAPPER-MQ-OBJECTNAME
         GOBACK
     END-IF.
 COMMIT-MESSAGE SECTION.
* Commit the put / got message          *
     CALL 'MQCMIT' USING WRAPPER-CONNECTION-HANDLE
                         WRAPPER-MQ-COMPLETION-CODE
                         WRAPPER-MQ-REASON-CODE
```

```
    END-CALL.
    IF WRAPPER-MQ-COMPLETION-CODE NOT = MQCC-OK
        MOVE +8 TO                    WRAPPER-RETURN-CODE
        MOVE 'CMIT-MESSAGE' TO        WRAPPER-MQ-OPERATION
        MOVE WRAPPER-QUEUE-MANAGER TO WRAPPER-MQ-OBJECTNAME
        GOBACK
    END-IF.
CLOSE-QUEUE-OUT SECTION.
```

End-to-end error handling

INTRODUCTION

Over the years, it has become accepted that MQSeries provides the resilience and throughput capacity that is required of a business-critical communication backbone system. The management tools provided with the product have also been continuously improved to provide the functionality that enables problems, when they arise, to be identified and rectified within an acceptable timeframe. However, this does not mean that everything in the garden is rosy and that we have attained Nirvana.

It is common in many situations to find that the environment implementation was developed from a single-application pilot, and that it has inherited many of the limitations that were initially acceptable but which can cause bottlenecks when in full production.

Much has been written about MQSeries naming, channel configurations, logging recovery, etc so I do not propose to go through a 'best practices' guide, but I will focus on aspects of the end-to-end error handling, its design, and the place compensatory transactions have in that design.

PROCESS FLOW

First and foremost, we must define what is meant by 'end-to-end' (E2E). In this article, E2E will refer to an end user generating a request, and that request being fulfilled. Notice that I used the term 'fulfilled', not 'responded to'; this is because we are considering an asynchronous environment. To take full advantage of the functionality provided, the user can and should be responded to before the process has completed, thereby taking advantage of overlapping operations. This means that the result of the process is not known when the user receives the response and the user may not be available to handle errors. So E2E in this context is the completion of the whole business process.

When this aspect is considered, many necessary functions take on a different aspect because they have to span many different processes, and transactions and must be capable of being correlated with the different requests and responses. In this case, the handling of errors is particularly relevant because an error is indicated in different ways on different platforms and is complicated by the possibility that the user is not available to handle the error.

Just look at the example of updating several databases in a single process. This is typically the case in a change-of-address application, where, for example, one database has been successfully updated but a subsequent update to another database fails. After several re-tries the failure persists.

What is the error recovery action? Does the failing process attempt to back-out the change to the earlier database, does it notify an operator, or does it just continue? These are really business issues and the recovery process used will involve extra cost and a process delay.

This update is just an example of the relationship between message queueing and transaction processing, which has been explored in earlier issues of *MQ Update*. What we are going to explore in this article is a possible approach to handling that error. This is where compensatory transactions come into the overall process design.

We will consider an asynchronous process flow where none of the processes has direct communications. The output may be reported back to the user, but this is not essential. Figure 1 shows the type of flow under discussion. It is important to note that this is an update flow. The way errors are dealt with is very different in an inquiry process, which is much simpler because there are no permanent changes to handle. The only point that needs to be borne in mind is that, if several applications are involved, do you want to re-run the whole query again or restart from the failing process?

ERROR HANDLING AND THE USE OF COMPENSATORY TRANSACTIONS

What we are looking at here is the handling of errors from an asynchronous system. This is very different from handling errors in an RPC environment. In this case, the error is generally reported as an

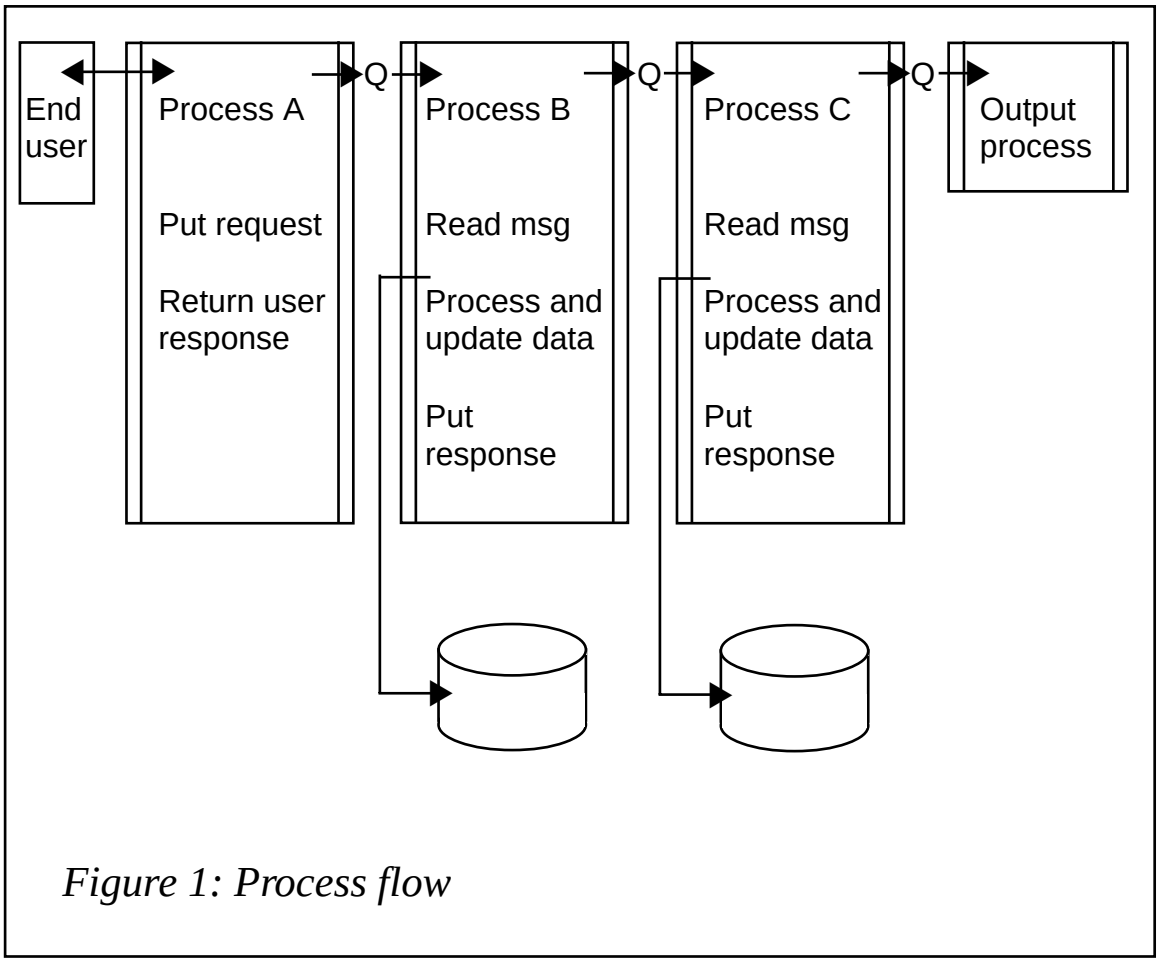

Figure 1: Process flow

internal error to the end user and left at that. But with asynchronous calls, errors need to handled with far more thought, as the end user is generally not available when the error is encountered.

So if we can't take the soft option of allowing the user to handle the error, what do we do? The key here is to attempt to understand the error and then try to use the facilities that the asynchronous process gives us in order to handle it. These facilities can be built into the interface layer that many enterprises have already developed to enforce standards within their organizations. This then gives a controlled error-handling mechanism.

Error sources

There are several sources of error.

System errors

System errors are related to the facilities that the applications are

using. Examples include: disks full, records locked, printer not available, etc. The key here is not to lose the message while the operators are being notified that corrective action is required.

There are several vendor tools available that enable system resources to be monitored and operators alerted. The message must be retained in addition to being correlated to the relevant resource, and operators also need to know when the error condition has been rectified. This can be done by the interface layer generating an error message and putting it to the appropriate queue. This queue can then be processed when the system error is rectified.

It should be noted here that there is no need to stop the server while the error exists; however, it is important to monitor the message depth of the error queue. This will then allow the error to be escalated to 'serious' if the message depth is exceeded and the server shuts down.

These system errors will not have a major effect on the overall process flow. They will delay the completion but there are no logic problems and so the end result will be consistent. However, we must also consider application errors. In this case, an error may be in the process step, for example where a message that contains corrupt data is received or where the execution of a process may in itself cause an error.

When this happens we need to produce a compensating transaction. This requires the error handler to generate a message that contains sufficient information to enable the compensatory transaction to process the error. This error message is then put to the error queue where it is processed by the compensatory transaction. This is a very interesting approach to handling errors because the logic required in the compensatory transaction would in any case have had to have been provided in the application. So the result of using a compensatory transaction actually simplifies applications by providing a common external error handling process. It also gives a convenient location to handle application or process errors from several process steps.

In the worst case, a compensatory transaction will need to route the error message to a skilled operator who knows the application and process structure that is required to handle the recovery actions. Once again, the message must contain the necessary information to enable

recovery to be achieved. In many cases, handling manual recovery will require additional information that can be built up by the compensatory transaction. So what we see is that error information held in the message being built up from the application experiencing the error and the compensatory transaction.

This then raises an interesting question concerning compensatory transactions. Because they simplify the application code by concentrating the error handling in one place, they can be used by multiple applications. But this then makes the compensatory transaction much more complex, as each application produces a slightly different error situation and so it needs to be able to identify the source of the error. Therefore, some kind of grouping of compensatory transactions should be considered.

There are several methods that can be used but here we consider only two:

- Grouping by development teams.
- Grouping by data structures or database use.

Grouping by development teams will probably be the easiest to implement because the teams are likely to agree amongst themselves on how the errors should be handled and the interfaces that should be used. However, the transaction that results may not be as generic as required. By opting for coupling the compensatory transaction to the data structure a general approach will be adopted. But now we need to define the interfaces and action more clearly.

The logic used in these two types of compensatory transaction is very different, so a standard needs to be established and adhered to.

ERROR MESSAGE FORMAT AND RECOVERY PROCEDURES

Compensatory transaction actions can be regarded as part of the application, when you consider that the content of a message, the application, and the process all determine the data that it carries. This means that you may have to add data to the message content that is handled by the compensatory transaction.

Below is a suggested message format that includes some of the information you may wish to include in the message.

```
<Flow_Message>
<Header>
<Target_process/>
</Header>
<DataSection>
<Application_Data/>
</DataSection>
<errorSection>
<errorService>ErrorServer1</errorService>
<error>Application or interface error </error>
<errorText>Meaningful Error Text</errorText>
<timeStamp>2001-10-25T10:01:32.333Z</timeStamp>
<Host>DetectingHostName</Host>
<QueueMgrName>FailingQueueManagerName</QueueMgrName>
<Queue>FailingQueueName</Queue>
<Process>FailingProcessName</Process>
<ProcessData>Input Data</ProcessData>
<Requester>Originator of the Request</Requester>
<BusinessServiceRequestId>Original RequestorID</
BusinessServiceRequestId>
<Owner>Original Owner Id</Owner>
<OriginalData>Original Message Data</OriginalData>
</errorSection>
<BackoutDetails>
<BackoutData>Delete * from AAA Where XXX=????</BackoutData>
<BeforeImage>Data Content before Process step execution</BeforeImage>
</BackoutDetails>
</Flow_Message>
```

The content of this typical error message will vary according to the nature of the process flow, but this shows some of the information that will be required.

Some of the points considered in determining the content of the message were:

- The processes and serious error handling routines might be held on different systems, hence the need to store the queue manager name.
- There may be a need to inform the originator of the request that there was an error – hence the originator information. In many cases, the originator may not be available and so alternative routes to inform them of the error, eg e-mail, may be required. The

originator may not need to made aware of any errors if corrections can be achieved.

- The back-out information is interesting: this is intended to help operators clean the data. After due thought the automatic recovery of data was considered to be too complex and so we reverted to human intervention.

One option that was also considered was having each process build up its own back-out sections, including the information in the message. This was discarded though, because forward recovery was deemed to be the best recovery process, requiring each of the subsequent processing steps to be executed independently. This meant that the initial request was fulfilled even though the process took considerably longer. It was also much simpler.

CONCLUSIONS

Error handling in an asynchronous environment is very different from error handling in a synchronous environment. The whole process must be thoroughly assessed and cannot be passed to the application. However, there are many functions provided with MQ that help with error recovery while maintaining overall availability of the system. One of these is the use of compensatory transactions driven by a serious error queue.

Compensatory transactions should be considered as shared resources that hold common error recovery. This means they need to be designed for general use, but, for simplicity, you should consider grouping them.

Also, information required for error recovery must be held within the message and will need to be inserted by the application in error.

Finally, not all errors are recoverable. In the most complex cases, recovery must be passed to the most powerful recovery mechanism we have, the human operator.

Harry A Harris
Senior Consultant (UK)

MQ news

IBM has announced WebSphere MQIntegrator for z/OS V2.1, extending MQ message broker availability to z/OS and adding new capabilities to those previously provided by the MQSeries Integrator for OS/390 and DB2.

Previously known as MQSeries Integrator, it helps create, deploy, and control message-based business applications, integrates existing applications within enterprises and across the Internet, and connects applications between enterprises and their suppliers.

The company has also announced WebSphere MQ Integrator V2.1 with improved message, import, and plug-in node support for all existing platforms and databases.

The product comes with New Era of Networks Rules and Formatter V5.6 from Sybase, visual administration and debugging tools, national language support, improved Message Repository Manager support for XML messages, and support for plug-in input nodes.

Other features include the ability to author plug-in nodes written in Java, improved performance of the Message Repository Manager, extensions to ESQL capability, Oracle and Sybase XA support, MRM tagged message support, DTD import capability, MRM multi-part tagged message support for formats such as EDI and SWIFT, message aggregation capability, and a revised table of tiered charges.

For further information, contact your local IBM representative.
Web: http://www.software.ibm.com

* * *

CommerceQuest has announced general availability of its Business Process Integrator (BPI) version 3.1, said to exploit IBM WebSphere Studio Workbench based on the Eclipse Open Source Project to deliver a comprehensive integrated development environment and integrated software version control, also based on an Open Source project known as Concurrent Versioning System (CVS).

BPI is described as a process-centric, component-based development and integration framework, with XML support, exploitation of MQSeries, and a range of cross-platform integration capabilities for applications, databases, and files.

For further information contact:
CommerceQuest, 2202 N Westshore Blvd, Tampa, FL, 33607, USA
Tel: +1 813 639 6300
Fax: +1 813 639 6900
Web: http://www.CommerceQuest.com

CommerceQuest (UK), Doncastle House, Doncastle Road, Bracknell, Berkshire, RG12 8PE, UK
Tel: +44 (0) 1344 861010
Fax: +44 (0) 1344 861011

* * *