30

MQ

December 2001

In this issue

3 Managing system resources on Unix

6 Message processing monitor

22 Enhancing MQSeries transaction coordination

39 MQSeries Integrator V2 performance

44 MQ news

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 (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.

Managing system resources on Unix

In general, MQSeries performance depends on how effectively its resources are applied to the requirements of different applications on the system. This includes operating system resources, which can sometimes prove to be a bottleneck and restrict optimum performance. From this perspective, CPU, semaphores, message queues, memory, disk I/O, and other device I/O can prove to be problems when troubleshooting.

The first step in eliminating these problems is determining what is causing the resource shortage. Many times, a queue manager shares an environment with resource 'hogs', eg database management systems. MQ will lose the battle to attain resources simply because the systems running will have already locked-up some of the required resources, such as memory. Other times, there is just ineffective control over how the resources are managed. In any event, when there is a shortage of system resources there is a limited number of options available to improve the situation: in short, you can add more, use less, or limit the amount allocated to each user.

Although managing the whole Unix system may be out of your control you can assist in troubleshooting and MQ resource management more effectively by executing proper shutdowns. If you are unable to execute a proper shutdown (and sometimes even when you are able) MQ may hold resources that need to be released in order to prevent a degradation in MQSeries and Unix performance and, eventually, cause you to perform a system reboot. Because of the numerous platforms that middleware administration requires you to know, many administrators may be unable to become sufficiently fluent in one operating system to perform the complex troubleshooting that performance tuning requires. The remainder of this article will trace the steps required to avoid system performance degradation by effectively cleaning up after a pre-emptive, manual, or otherwise abnormal stoppage of the queue manager on the Unix platform.

First, let's define an abnormal shutdown. For the purposes of this article an abnormal shutdown will be defined as any shutdown caused by issuing the **endmqm –p qmgrname** command or a manual

shutdown of the queue manager. While everyone who works with MQSeries is probably familiar with the various options of the **endmqm** command, not everyone has had the misfortune of shutting down a queue manager manually. This can be accomplished by issuing a **kill** command (or a **kill –9** command for a process that fails to terminate) for the queue manager process ids, in the following order:

1 *amqpcsea*: command server.
2 *amqhasmx*: logger.
3 *amqharmx*: log formatter (linear logs only).
4 *amqzllp0*: checkpoint processor.
5 *amqzlaa0*: queue manager agents.
6 *amqzxma0*: processing controller.
7 *amqrrmfa*: repository process (for clusters).

When the shutdown is complete you must assess what resources are still marked as 'in use' and what you must do to free them. We will begin this by using the **ipcs** command, which displays information about active interprocess communication facilities. In order to get the information that is pertinent just to MQSeries we must issue this command with a grep that searches for processes with an mqm *user-id*: **Ex. ipcs | grep mqm**.

The output of this command will look something like this:

```
IPC status from /dev/kmem as of Wed Sep  5 13:40:51 2001
T      ID     KEY          MODE         OWNER     GROUP
q       2     0x00000000   -rw----      mqm       mqm
q       3     0x00000000   -rw----      mqm       mqm
m      29     0x0c800054   -rw-rw--     mqm       mqm
m      30     0x0c800055   -rw-rw--     mqm       mqm
s      22     0x0cac0002   -ra-ra-ra-   mqm       mqm
s      23     0x0c800008   -ra-ra--     mqm       mqm
```

This output defines the facility type, id, key, mode, owner, and group. We have now effectively narrowed down the resources that are being held by MQSeries and the information required to release them. Now lets's take a look at the various facility types so you know what you are removing.

The symbol 'q' represents a message queue (unrelated to an MQSeries queue). A message queue is a linked list of messages, each of a fixed size. Messages are put at the end of the queue so that the sending order is preserved. Each message may have a type, allowing multiple message streams to be processed in the same queue.

The symbol ‘m’ represents a shared memory segment. Shared memory allows two or more processes to share a segment of memory so they can all examine and edit the contents. Prior to using a shared memory segment a process must obtain the queue identifier for it.

The symbol ‘s’ represents a semaphore. Semaphores are counters that are used to synchronize access to a shared object, such as a memory segment. They do not, however, exchange data between processes. By incrementing and decrementing the counter the semaphore can effectively synchronize the updating of shared resources.

Now that we have all the information we need from the system we must release the resources using this information with a command called **ipcrm**. The **ipcrm** command was made to remove one or more specified message queue, semaphore set, or shared memory identifiers.

```
Ex. ipcrm -s 22
Ex. ipcrm -m 29
Ex. ipcrm -q 2
```

The above commands remove a semaphore with an ID of 22, a shared memory segment with an ID of 29, and a message queue with an ID of 2. This command can also be used in a shell script to clean up these resources.

```
#!/bin/ksh
# Clear Unix system objects held by MQSeries by using IPCRM
setenv PATH /bin:/usr/bin
#Because the IPCS output may vary between platforms and releases, we are
going to use the awk command to separate the columns.
#This will clear all shared memory segments being held by the mqm id
for thing in ‘ipcs -m | grep mqm | awk -Fm ‘{print $2}’ | awk ‘{print
$1}’‘
  do
     ipcrm -m $thing
done
#This step will clear all semaphores being held by the mqm id
for thing in 'ipcs -s | grep mqm | awk -Fs '{print $2}' | awk '{print
$1}''
  do
ipcrm -s $thing
done
#This step will clear all message queues being held by the mqm id
for thing in 'ipcs -q | grep mqm | awk -Fq '{print $2}' | awk '{print
$1}''
  do
     ipcrm -q $thing
done
```

In summary, it is extremely important to monitor the system resources that MQSeries uses and, periodically, remove any resources that are being held in error. By failing to monitor this you run the risk of degrading the performance of MQSeries and other applications on the Unix box.

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

Message processing monitor

MQSeries is mainly used in asynchronous applications; however, problems can occur if a message is not processed for a long time. If the queue is processed sequentially it can also mean that other messages in the queue are not processed.

MQSeries Event Monitoring does not provide an event when a message stays in the queue for longer than the specified time interval. MQSeries Queue Service Interval Events are not generated for individual message processing problems, they are generated after a successful GET occurs for any message on the queue, and these events are not generated when the message is still on the queue.

The current depth of a queue does not indicate whether messages in the queue are being processed.

The only way to check whether or not there is a problem with message processing is to look at the queue content to see whether the first message is changing each time. For a site with hundreds of queues an automated solution is required to monitor all queues.

The solution we have developed runs on CICS/OS390. It may also be easily converted to non-CICS environments.

The COBOL program given here – QMONHOST – monitors selected queues and reports whether messages in each queue are processed properly or not. The information gathered is sent to a queue that everyone can process in their own way.

The program collects specific information about a set of queues. Queue names to be monitored are stored in a namelist. The output is

an MQ message for each queue. The output message includes status of the queue, put enabled, get enabled, trigger set, input use count, output use count, and queue depth. The status information of the queue indicates whether or not the first message in the queue is processed during a specified time interval. This program is invoked as a CICS transaction at CICS start-up and is triggered to start every three minutes.

The program gets the queue names from the namelist: for each queue it stores *the message-ids* of the first message to a temporary storage queue (TSQ) named QMON.

The next time it's started it compares the current *message-id* of the first message of the queue with the one stored previously in the TSQ. If they are identical, this means that the message has not been processed for three minutes and it sets the queue status flag as 'problem'. If *message-ids* are different it means that messages are being processed, so there is no problem. It then stores the current *message-id* of the first message in the TSQ for the next comparison three minutes later. Together with other information (put enabled, get enabled, trigger set, input use count, output use count, and queue depth), it builds the output as an MQ message and puts it to a queue.

QMONHOST

```
CBL XOPTS(ANSI85)
CBL NODYNAM,LIB,OBJECT,RENT,RES,APOST
       IDENTIFICATION DIVISION.
       PROGRAM-ID. QMONHOST.
      * MODULE NAME = QMONHOST                                        *
      * ENVIRONMENT = CICS, COBOL II                                  *
      * CICS TRANSACTION NAME = QMON                                  *
      * DESCRIPTIVE NAME = Queue Monitor (QMON)                       *
      * FUNCTION :                                                    *
      * DESCRIPTION :                                                 *
      *   The Program uses a number of parameters defined in          *
      *   the working-storage section of this program. These          *
      *   parameters are namelist name, MQ-NAMELIST-NAME and total    *
      *   number of queues in the namelist, TOTAL-QUEUE-NUMBER.       *
      *   Each queue is represented by a number, QUEUENO.             *
      *                      PROGRAM LOGIC                            *
      * START.                                                        *
      *   set transaction (QMON) to start 3 minutes later.            *
      *   get system time and write to time field of output message.   *
      *   check the QMON TSQ for existence.(READ-TSQ)                 *
      *   get name of the QManager.(GET-QMGR-NAME)                    *
```

```
*   get queue names to be monitored from the namelist.          *
*                                    (GET-QUEUE-NAMES)          *
*   for each queue in the namelist                              *
*      open queue(OPEN-APP-QUEUE)                               *
*      inquire queue(INQUIRE-APP-QUEUE)                         *
*      get "put enabled", "get enabled", "trigger set"          *
*                                         information           *
*      set put, get, trigger bit of output message              *
*          (1->OK, Ø->PROBLEM)                                  *
*      get "input use count", "output use count" and            *
*          "queue depth"                                        *
*      write "input use count", "output use count" and          *
*          "queue depth" to output message                      *
*      browse first message in queue(GET-BROWSE-APP-QUEUE)      *
*      close queue(CLOSE-APP-QUEUE)                             *
*      read stored message id in TSQ(READ-TSQ)                  *
*      compare browsed and stored message ids(COMPARE-MESSAGES) *
*      if browsed and stored message ids are different          *
*         messages in the queue are processed properly          *
*         set status bit of output message "1"(OK)              *
*         write browsed message id to TSQ(WRITE-TSQ)            *
*      else                                                     *
*         first messages is not processed                       *
*         set status bit of output message "Ø"(PROBLEM)         *
*         write stored message id again to TSQ(WRITE-TSQ)       *
*      end-if                                                   *
*      if all bits(status, put, get and trigger bit) are "1"(OK)*
*         set overall bit to "1" (OK)                           *
*      else                                                     *
*         set overall bit to "Ø" (PROBLEM)                      *
*      end-if                                                   *
*      open output queue(OPEN-OUTPUT-QUEUE)                     *
*      put output message to output queue(PUT-OUTPUT-QUEUE)     *
*      close output queue(CLOSE-OUTPUT-QUEUE)                   *
*   end-for.                                                    *
* END.                                                          *
 DATE-WRITTEN.  OCTOBER,     1998.
 DATE-COMPILED.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-3Ø9Ø.
 OBJECT-COMPUTER. IBM-3Ø9Ø.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.
 Ø1  QMGR-NAME                PICTURE X(48) VALUE SPACES.
 Ø1  MQ-OBJECTNAME            PICTURE X(48) VALUE SPACES.
 Ø1  MQ-HCONN                 PICTURE S9(9) BINARY VALUE ZERO.
 Ø1  MQ-OPTIONS               PICTURE S9(9) BINARY.
```

```
  Ø1  MQ-OUTPUT-Q-HOBJ         PICTURE S9(9) BINARY VALUE ZERO.
  Ø1  MQ-INPUT-Q-HOBJ          PICTURE S9(9) BINARY VALUE ZERO.
  Ø1  MQ-QUEUE-HOBJ            PICTURE S9(9) BINARY VALUE ZERO.
  Ø1  MQ-QMGR-HOBJ             PICTURE S9(9) BINARY VALUE ZERO.
  Ø1  MQ-COMPCODE              PICTURE S9(9) BINARY.
  Ø1  MQ-REASON                PICTURE S9(9) BINARY.
  Ø1  MQ-BUFFLEN               PICTURE S9(9) BINARY VALUE +3ØØØ.
  Ø1  MQ-DATALEN               PICTURE S9(9) BINARY.
  Ø1  MQ-GET-BUFFER            PICTURE X(3ØØØ) VALUE SPACES.
  Ø1  MQ-SELECTORCOUNT         PICTURE S9(9) BINARY VALUE 6.
  Ø1  MQ-SELECTORS-TABLE.
      Ø5  MQ-SELECTORS         PICTURE S9(9) BINARY OCCURS 6 TIMES.
  Ø1  MQ-INTATTRCOUNT          PICTURE S9(9) BINARY VALUE 6.
  Ø1  MQ-INTATTRS-TABLE.
      Ø5  MQ-INTATTRS          PICTURE S9(9) BINARY OCCURS 6 TIMES.
  Ø1  MQ-CHARATTRLENGTH-NLIST  PICTURE S9(9) BINARY VALUE 96ØØ.
  Ø1  MQ-CHARATTRS-TABLE-NLIST.
      Ø5  MQ-CHARATTRS-NLIST   PIC X(48) OCCURS 1ØØ TIMES.
  Ø1  MQ-CHARATTRLENGTH        PICTURE S9(9) BINARY VALUE 48.
  Ø1  MQ-CHARATTRS-TABLE.
      Ø5  MQ-CHARATTRS         PIC X(48) OCCURS 1   TIMES.
 * MQ-NAMELIST-NAME is the name of the namelist containing names *
 *                  of monitored queues.                         *
  Ø1  MQ-NAMELIST-NAME         PICTURE X(2Ø)  VALUE
                               'QMON.NAMELIST'.
  Ø1  MQ-HOBJ-NAMELIST         PICTURE S9(9) BINARY.
  Ø1  QNAME-TABLE-NLIST.
      Ø5  QNAME-NLIST          PIC X(48) OCCURS 1ØØ TIMES.
  Ø1  LEN-MQ-ERROR-MESSAGE    PICTURE S9(4) COMP VALUE 87.
* ERROR MESSAGE STRUCTURE                                       *
      Ø1  MQ-ERROR-MESSAGE.
      Ø5 YEAR-ERROR           PICTURE 9999.
      Ø5 FILLER               PICTURE X VALUE '.'.
      Ø5 MONTH-ERROR          PICTURE 99.
      Ø5 FILLER               PICTURE X VALUE '.'.
      Ø5 DAY-ERROR            PICTURE 99.
      Ø5 FILLER               PICTURE X VALUE ' '.
      Ø5 HOUR-ERROR           PICTURE 99.
      Ø5 FILLER               PICTURE X VALUE ':'.
      Ø5 MINUTE-ERROR         PICTURE 99.
      Ø5 FILLER               PICTURE X VALUE ':'.
      Ø5 SECOND-ERROR         PICTURE 99.
      Ø5 FILLER               PICTURE X(7) VALUE ' QMON: '.
      Ø5 MQ-ERROR-OPERATION  PICTURE X(1Ø) VALUE SPACES.
      Ø5 FILLER              PICTURE X(18) VALUE
                              ' ERROR: COMPCODE: '.
      Ø5 MQ-ERROR-COMPCODE   PICTURE Z9.
      Ø5 FILLER              PICTURE X(Ø4) VALUE SPACES.
      Ø5 FILLER              PICTURE X(Ø9) VALUE ' REASON: '.
      Ø5 MQ-ERROR-REASON     PICTURE Z(Ø8)9.
  Ø1  LEN-ABEND-MESSAGE       PICTURE S9(4) COMP VALUE 6Ø.
  Ø1  ABEND-MESSAGE.
```

```
     Ø5  FILLER           PICTURE X(14) VALUE 'TRANSACTION : '.
     Ø5  ABEND-TRANS-ID   PICTURE X(4)  VALUE 'QMON'.
     Ø5  FILLER           PICTURE X(44) VALUE
             ' ABNORMALLY ENDED                          '.
 Ø1  TSQ-MESSAGE-LEN     PIC S9(4)  COMP VALUE 72.
* TSQ contains items, which has a name same as Qname and        *
*     the message id of the first message in this Queue         *
*     for later comparison.                                     *
 Ø1  TSQ-MESSAGE.
     Ø5  Q-NAME           PICTURE X(48) VALUE ZEROS.
     Ø5  OLD-MSGID        PICTURE X(24) VALUE ZEROS.
 Ø1  GET-INT-BUFFER       PICTURE S9(9) COMP VALUE Ø.
 Ø1  PUT-INT-BUFFER       PICTURE S9(9) COMP VALUE Ø.
 Ø1  TRIGGER-INT-BUFFER   PICTURE S9(9) COMP VALUE Ø.
Ø1  MQ-PUT-BUFFER-LEN       PICTURE S9(9) COMP VALUE +52.
* MQ-PUT-BUFFER contains output message for each queue. This    *
*               structure is composed of queue number(QUEUENO), *
*               monitor time (YEAR-WORK, MONTH WORK, DAY-WORK,  *
*               HOUR-WORK, MINUTE-WORK, SECOND-WORK), overall   *
*               bit(OVERALL-BUFFER), status of the queue        *
*               (STATUS-BUFFER), put-enable bit(PUT-BUFFER),    *
*               get-enable bit(GET-BUFFER),trigger-set bit      *
*               (TRIGGER-BUFFER), input use count               *
*               (INPUT-USE-COUNT), output use count             *
*               (OUTPUT-USE-COUNT), queue depth(QDEPTH).        *
 Ø1  MQ-PUT-BUFFER.
     Ø5 QUEUENO              PICTURE 999.
     Ø5 YEAR-WORK            PICTURE 9999.
     Ø5 MONTH-WORK           PICTURE 99.
     Ø5 DAY-WORK             PICTURE 99.
     Ø5 HOUR-WORK            PICTURE 99.
     Ø5 MINUTE-WORK          PICTURE 99.
     Ø5 SECOND-WORK          PICTURE 99.
     Ø5 OVERALL-BUFFER       PIC  X(1) VALUE 'Ø'.
     Ø5 STATUS-BUFFER        PIC  X(1) VALUE 'Ø'.
     Ø5 PUT-BUFFER           PIC  X(1) VALUE 'Ø'.
     Ø5 GET-BUFFER           PIC  X(1) VALUE 'Ø'.
     Ø5 TRIGGER-BUFFER       PIC  X(1) VALUE 'Ø'.
     Ø5 INPUT-USE-COUNT      PIC  X(9) VALUE 'Ø'.
     Ø5 OUTPUT-USE-COUNT     PIC  X(9) VALUE 'Ø'.
     Ø5 QDEPTH               PIC  X(9) VALUE '1Ø'.
 Ø1  TIME-IN-MSEC        PICTURE S9(15)  COMP-3.
 Ø1  DATE-SYSTEM.
     Ø5  YEAR-SYSTEM     PICTURE 9999.
     Ø5  MONTH-SYSTEM    PICTURE 99.
     Ø5  DAY-SYSTEM      PICTURE 99.
 Ø1  TIME-SYSTEM.
     Ø5  HOUR-SYSTEM     PICTURE 99.
     Ø5  MINUTE-SYSTEM   PICTURE 99.
     Ø5  SECOND-SYSTEM   PICTURE 99.
     Ø5  FILLER          PICTURE 99.
* TOTAL-QUEUE-NUMBER is total queue number written in the       *
```

```
*                       namelist. This number is set           *
*                       automatically when namelist is read.   *
 Ø1  TOTAL-QUEUE-NUMBER   PICTURE 999   VALUE 1ØØ.
 Ø1  I                    PICTURE 999   COMP VALUE 1.
 Ø1  XXX                  PICTURE S9(8) COMP VALUE Ø.
 Ø1  YYY                  PICTURE S9(8) COMP VALUE Ø.
* MSGID-TABLE is filled with the message-ids of the first       *
*       in each monitored queue. Number of monitored queues is  *
*       assumed to be maximum 1ØØ.                              *
 Ø1  MSGID-TABLE.
     Ø5  MSGID             PICTURE X(24) OCCURS 1ØØ TIMES.
* ******* FLAG definitions                                      *
 Ø1  OPEN-QMGR-FLAG        PICTURE X     VALUE SPACE.
     88  OPEN-QMGR-OK                    VALUE 'Y'.
 Ø1  CLOSE-QMGR-FLAG       PICTURE X     VALUE SPACE.
     88  CLOSE-QMGR-OK                   VALUE 'Y'.
 Ø1  INQ-QMGR-FLAG         PICTURE X     VALUE SPACE.
     88  INQ-QMGR-OK                     VALUE 'Y'.
 Ø1  INQ-NAMELIST-FLAG     PICTURE X     VALUE SPACE.
     88  INQ-NAMELIST-OK                 VALUE 'Y'.
 Ø1  OPEN-NAMELIST-FLAG    PICTURE X     VALUE SPACE.
     88  OPEN-NAMELIST-OK                VALUE 'Y'.
 Ø1  CLOSE-NAMELIST-FLAG   PICTURE X     VALUE SPACE.
     88  CLOSE-NAMELIST-OK               VALUE 'Y'.
 Ø1  OPEN-QUEUE-FLAG       PICTURE X     VALUE SPACE.
     88  OPEN-QUEUE-OK                   VALUE 'Y'.
 Ø1  INQ-QUEUE-FLAG        PICTURE X     VALUE SPACE.
     88  INQ-QUEUE-OK                    VALUE 'Y'.
 Ø1  GET-QUEUE-FLAG        PICTURE X     VALUE SPACE.
     88  GET-QUEUE-OK                    VALUE 'Y'.
 Ø1  CLOSE-QUEUE-FLAG      PICTURE X     VALUE SPACE.
     88  CLOSE-QUEUE-OK                  VALUE 'Y'.
 Ø1  OPEN-OUTPUT-FLAG      PICTURE X     VALUE SPACE.
     88  OPEN-OUTPUT-OK                   VALUE 'Y'.
 Ø1  PUT-OUTPUT-FLAG        PICTURE X     VALUE SPACE.
     88  PUT-OUTPUT-OK                    VALUE 'Y'.
 Ø1  CLOSE-OUTPUT-FLAG      PICTURE X     VALUE SPACE.
     88  CLOSE-OUTPUT-OK                  VALUE 'Y'.
 Ø1  READ-TSQ-FLAG          PICTURE X     VALUE SPACE.
     88  READ-TSQ-OK                      VALUE 'Y'.
 Ø1  WRITE-TSQ-FLAG         PICTURE X     VALUE SPACE.
     88  WRITE-TSQ-OK                     VALUE 'Y'.
 Ø1  COMPARE-MESSAGES-FLAG  PICTURE X     VALUE SPACE.
     88  COMPARE-MESSAGES-OK              VALUE 'Y'.
 Ø1  DELETE-TSQ-FLAG         PICTURE X     VALUE SPACE.
     88  DELETE-TSQ-OK                    VALUE 'Y'.
* ******* Data structures                                       *
 Ø1  MQM-OBJECT-DESCRIPTOR.
     COPY CMQODV.
 Ø1  MQM-MESSAGE-DESCRIPTOR.
     COPY CMQMDV.
 Ø1  MQM-GET-MESSAGE-OPTIONS.
```

```
     COPY CMQGMOV.
 Ø1  MQM-PUT-MESSAGE-OPTIONS.
     COPY CMQPMOV.
 Ø1  MQM-CONSTANTS.
     COPY CMQV SUPPRESS.
 LINKAGE SECTION.
 PROCEDURE DIVISION.
 START-PROG.
     PERFORM INIT-PROG
        THRU INIT-PROG-EXIT.
     PERFORM MAIN-PROG
        THRU MAIN-PROG-EXIT.
     PERFORM END-PROG
        THRU END-PROG-EXIT.
 INIT-PROG.
     EXEC CICS HANDLE ABEND
               LABEL    (ABEND-PROC)
     END-EXEC.
* Set transaction (QMON) start itself 3 minutes later.          *
     EXEC CICS START INTERVAL(ØØØ3ØØ) TRANSID('QMON') END-EXEC.
* Get absolute time and convert it to a readable format         *
     PERFORM DATE-CONVERSION
        THRU DATE-CONVERSION-EXIT.
     MOVE MQHC-DEF-HCONN TO MQ-HCONN.
 INIT-PROG-EXIT.
     EXIT.
 MAIN-PROG.
* Check TSQ whether it exists properly.                         *
     PERFORM READ-TSQ
        THRU READ-TSQ-EXIT.
* Get QManager name CICS is connected to.                       *
     PERFORM GET-QMGR-NAME
        THRU GET-QMGR-NAME-EXIT.
* Get Queue names from namelist                                 *
     IF QMGR-NAME NOT = SPACES
         PERFORM GET-QUEUE-NAMES
            THRU GET-QUEUE-NAMES-EXIT
     ELSE
         PERFORM END-PROG
            THRU END-PROG-EXIT.
* First queue is represented by number 1Ø in this program.      *
* You can give any number as desired.                           *
* (We used at least 2 digits number for compatibility with the  *
* GUI program presenting the results of this program.)          *
* So QUEUENO is initialized to 9 at start.                      *
     MOVE 9 TO QUEUENO.
* Procedure GET-ONE-MESSAGE gets information for each queue,    *
* compares the message id of the first message in the queue with*
* the message id of the first message 3 minutes before stored   *
* to the TSQ. If they are same, so the first message in the     *
* is not processed since 3 minutes.                             *
* Then the message for the status of the Queue is built and put *
```

```
* to the output queue.                                          *
     PERFORM GET-ONE-MESSAGE
         THRU GET-ONE-MESSAGE-EXIT
     VARYING I FROM 1 BY 1
         UNTIL I > TOTAL-QUEUE-NUMBER.
 MAIN-PROG-EXIT.
     EXIT.
 GET-QMGR-NAME.
     MOVE SPACES TO QMGR-NAME.
     PERFORM OPEN-QMGR
        THRU OPEN-QMGR-EXIT.
     IF OPEN-QMGR-OK
         PERFORM INQ-QMGR
            THRU INQ-QMGR-EXIT
         PERFORM CLOSE-QMGR
            THRU CLOSE-QMGR-EXIT.
 GET-QMGR-NAME-EXIT.
     EXIT.
 OPEN-QMGR.
     MOVE MQOT-Q-MGR               TO MQOD-OBJECTTYPE.
     MOVE SPACES                   TO MQOD-OBJECTNAME.
     MOVE SPACES                   TO MQOD-DYNAMICQNAME.
     MOVE ZERO                     TO MQ-QMGR-HOBJ.
     MOVE MQHC-DEF-HCONN           TO MQ-HCONN.
     MOVE MQOO-INQUIRE             TO MQ-OPTIONS.
     CALL 'MQOPEN' USING MQ-HCONN
                         MQOD
                         MQ-OPTIONS
                         MQ-QMGR-HOBJ
                         MQ-COMPCODE
                         MQ-REASON.
     IF (MQ-COMPCODE = MQCC-OK) THEN
         MOVE 'Y'  TO OPEN-QMGR-FLAG
     ELSE
         MOVE 'N'  TO OPEN-QMGR-FLAG
         MOVE 'MQOPEN  QM'    TO MQ-ERROR-OPERATION
         PERFORM HANDLE-ERROR
            THRU HANDLE-ERROR-EXIT
         PERFORM END-PROG
            THRU END-PROG-EXIT.
 OPEN-QMGR-EXIT.
     EXIT.
 INQ-QMGR.
     MOVE 1                    TO MQ-SELECTORCOUNT.
     MOVE Ø                    TO MQ-INTATTRCOUNT.
     MOVE MQCA-Q-MGR-NAME      TO MQ-SELECTORS(1).
     MOVE MQ-Q-MGR-NAME-LENGTH TO MQ-CHARATTRLENGTH.
    CALL 'MQINQ' USING MQ-HCONN
                       MQ-QMGR-HOBJ
                       MQ-SELECTORCOUNT
                       MQ-SELECTORS-TABLE
                       MQ-INTATTRCOUNT
```

```
                        MQ-INTATTRS-TABLE
                        MQ-CHARATTRLENGTH
                        MQ-CHARATTRS-TABLE
                        MQ-COMPCODE
                        MQ-REASON.
    IF MQ-COMPCODE = MQCC-OK
        MOVE 'Y'  TO INQ-QMGR-FLAG
        MOVE MQ-CHARATTRS(1) TO QMGR-NAME
    ELSE
        MOVE 'N'  TO INQ-QMGR-FLAG
        MOVE 'MQINQ   QM'    TO MQ-ERROR-OPERATION
        PERFORM HANDLE-ERROR
           THRU HANDLE-ERROR-EXIT
        PERFORM END-PROG
           THRU END-PROG-EXIT.
INQ-QMGR-EXIT.
    EXIT.
CLOSE-QMGR.
    CALL ‘MQCLOSE’ USING MQ-HCONN
                         MQ-QMGR-HOBJ
                         MQCO-NONE
                         MQ-COMPCODE
                         MQ-REASON.
    IF (MQ-COMPCODE = MQCC-OK) THEN
        MOVE 'Y'  TO CLOSE-QMGR-FLAG
    ELSE
        MOVE 'N'  TO CLOSE-QMGR-FLAG
        MOVE 'MQCLOSE QM'  TO MQ-ERROR-OPERATION
        PERFORM HANDLE-ERROR
           THRU HANDLE-ERROR-EXIT
        PERFORM END-PROG
           THRU END-PROG-EXIT.
CLOSE-QMGR-EXIT.
    EXIT.
GET-QUEUE-NAMES.
    PERFORM OPEN-NAMELIST
       THRU OPEN-NAMELIST-EXIT.
    IF MQ-COMPCODE = MQCC-OK
        PERFORM INQ-NAMELIST
           THRU INQ-NAMELIST-EXIT
        IF INQ-NAMELIST-OK
            MOVE MQ-INTATTRS(1) TO TOTAL-QUEUE-NUMBER.
        PERFORM CLOSE-NAMELIST
           THRU CLOSE-NAMELIST-EXIT.
GET-QUEUE-NAMES-EXIT.
    EXIT.
OPEN-NAMELIST.
    MOVE MQOT-NAMELIST    TO MQOD-OBJECTTYPE.
    MOVE MQ-NAMELIST-NAME TO MQOD-OBJECTNAME.
    MOVE QMGR-NAME        TO MQOD-OBJECTQMGRNAME.
    COMPUTE MQ-OPTIONS =  MQOO-INQUIRE.
    CALL 'MQOPEN' USING MQ-HCONN
```

```
                          MQOD
                          MQ-OPTIONS
                          MQ-HOBJ-NAMELIST
                          MQ-COMPCODE
                          MQ-REASON.
      IF (MQ-COMPCODE = MQCC-OK) THEN
          MOVE 'Y'  TO OPEN-NAMELIST-FLAG
      ELSE
          MOVE 'N'  TO OPEN-NAMELIST-FLAG
          MOVE 'MQOPEN  NL'  TO MQ-ERROR-OPERATION
          PERFORM HANDLE-ERROR
             THRU HANDLE-ERROR-EXIT
          PERFORM END-PROG
             THRU END-PROG-EXIT.
  OPEN-NAMELIST-EXIT.
      EXIT.
  INQ-NAMELIST.
      MOVE 2 TO MQ-SELECTORCOUNT.
      MOVE 1 TO MQ-INTATTRCOUNT.
      MOVE MQIA-NAME-COUNT  TO MQ-SELECTORS(1).
      MOVE MQCA-NAMES       TO MQ-SELECTORS(2).
      CALL 'MQINQ' USING MQ-HCONN
                         MQ-HOBJ-NAMELIST
                         MQ-SELECTORCOUNT
                         MQ-SELECTORS-TABLE
                         MQ-INTATTRCOUNT
                         MQ-INTATTRS-TABLE
                         MQ-CHARATTRLENGTH-NLIST
                         MQ-CHARATTRS-TABLE-NLIST
                         MQ-COMPCODE
                         MQ-REASON.
      IF (MQ-COMPCODE = MQCC-OK) THEN
          MOVE MQ-CHARATTRS-TABLE-NLIST TO QNAME-TABLE-NLIST
          MOVE 'Y'  TO INQ-NAMELIST-FLAG
      ELSE
          MOVE 'N'  TO INQ-NAMELIST-FLAG
          MOVE 'MQINQ   NL'     TO MQ-ERROR-OPERATION
          PERFORM HANDLE-ERROR
             THRU HANDLE-ERROR-EXIT
          PERFORM END-PROG
             THRU END-PROG-EXIT.
  INQ-NAMELIST-EXIT.
      EXIT.
  CLOSE-NAMELIST.
      CALL 'MQCLOSE' USING MQ-HCONN
                           MQ-HOBJ-NAMELIST
                           MQCO-NONE
                           MQ-COMPCODE
                           MQ-REASON.
       IF (MQ-COMPCODE = MQCC-OK) THEN
          MOVE 'Y'  TO INQ-NAMELIST-FLAG
      ELSE
```

```
           MOVE 'N'  TO INQ-NAMELIST-FLAG
           MOVE 'MQCLOSE  NL'     TO MQ-ERROR-OPERATION
           PERFORM HANDLE-ERROR
              THRU HANDLE-ERROR-EXIT
           PERFORM END-PROG
              THRU END-PROG-EXIT.
CLOSE-NAMELIST-EXIT.
    EXIT.
GET-ONE-MESSAGE.
    PERFORM OPEN-APP-QUEUE
       THRU OPEN-APP-QUEUE-EXIT.
    PERFORM INQUIRE-APP-QUEUE
       THRU INQUIRE-APP-QUEUE-EXIT.
    PERFORM GET-BROWSE-APP-QUEUE
       THRU GET-BROWSE-APP-QUEUE-EXIT.
    PERFORM CLOSE-APP-QUEUE
       THRU CLOSE-APP-QUEUE-EXIT.
    MOVE 'N'  TO COMPARE-MESSAGES-FLAG.
    COMPUTE QUEUENO = QUEUENO + 1.
    MOVE 'Ø'                    TO STATUS-BUFFER.
    MOVE 'Ø'                    TO OVERALL-BUFFER.
    IF READ-TSQ-OK THEN
        PERFORM COMPARE-MESSAGES
           THRU COMPARE-MESSAGES-EXIT.
    PERFORM WRITE-TSQ
       THRU WRITE-TSQ-EXIT.
     PERFORM WRITE-MESSAGES
        THRU WRITE-MESSAGES-EXIT.
GET-ONE-MESSAGE-EXIT.
    EXIT.
COMPARE-MESSAGES.
    PERFORM READ-TSQ
       THRU READ-TSQ-EXIT.
    MOVE 'N'  TO COMPARE-MESSAGES-FLAG.
    IF READ-TSQ-OK AND GET-QUEUE-OK
       IF ((OLD-MSGID = SPACES) AND (MSGID(I) = SPACES)) OR
           (OLD-MSGID NOT = MSGID(I)) THEN
         MOVE 'Y'  TO COMPARE-MESSAGES-FLAG.
COMPARE-MESSAGES-EXIT.
    EXIT.
WRITE-MESSAGES.
    IF (COMPARE-MESSAGES-OK AND GET-QUEUE-OK) THEN
        MOVE '1'                    TO STATUS-BUFFER
    ELSE
        MOVE 'Ø'                    TO STATUS-BUFFER.
    IF ((STATUS-BUFFER = '1') AND (PUT-BUFFER = '1') AND
        (GET-BUFFER = '1') AND (TRIGGER-BUFFER = '1')) THEN
        MOVE '1'                    TO OVERALL-BUFFER.
    PERFORM PUT-MESSAGES
        THRU PUT-MESSAGES-EXIT.
WRITE-MESSAGES-EXIT.
    EXIT.
```

```
OPEN-APP-QUEUE.
     MOVE MQOT-Q                 TO MQOD-OBJECTTYPE.
     MOVE MQ-CHARATTRS-NLIST(I)  TO MQOD-OBJECTNAME.
     MOVE QMGR-NAME              TO MQOD-OBJECTQMGRNAME.
     MOVE ZEROS                  TO MQ-QUEUE-HOBJ.
     COMPUTE MQ-OPTIONS = MQOO-INQUIRE
                        + MQOO-BROWSE.
     CALL 'MQOPEN' USING MQ-HCONN
                         MQOD
                         MQ-OPTIONS
                         MQ-QUEUE-HOBJ
                         MQ-COMPCODE
                         MQ-REASON.
     IF MQ-COMPCODE = MQCC-OK
         MOVE 'Y'  TO OPEN-QUEUE-FLAG
     ELSE
         MOVE 'N'  TO OPEN-QUEUE-FLAG
         MOVE 'MQOPEN  Q '  TO MQ-ERROR-OPERATION
         PERFORM HANDLE-ERROR
            THRU HANDLE-ERROR-EXIT.
OPEN-APP-QUEUE-EXIT.
    EXIT.
INQUIRE-APP-QUEUE.
    MOVE 6 TO MQ-SELECTORCOUNT.
    MOVE 6 TO MQ-INTATTRCOUNT.
    MOVE MQIA-CURRENT-Q-DEPTH TO MQ-SELECTORS(4).
    MOVE MQIA-INHIBIT-GET     TO MQ-SELECTORS(1).
    MOVE MQIA-INHIBIT-PUT     TO MQ-SELECTORS(2).
    MOVE MQIA-TRIGGER-CONTROL TO MQ-SELECTORS(3).
    MOVE MQIA-OPEN-INPUT-COUNT TO MQ-SELECTORS(5).
    MOVE MQIA-OPEN-OUTPUT-COUNT TO MQ-SELECTORS(6).
    CALL 'MQINQ' USING MQ-HCONN
                       MQ-QUEUE-HOBJ
                       MQ-SELECTORCOUNT
                       MQ-SELECTORS-TABLE
                       MQ-INTATTRCOUNT
                       MQ-INTATTRS-TABLE
                       MQ-CHARATTRLENGTH
                       MQ-CHARATTRS-TABLE
                       MQ-COMPCODE
                       MQ-REASON.
    MOVE 'Ø'         TO GET-BUFFER.
    MOVE 'Ø'         TO PUT-BUFFER.
    MOVE 'Ø'         TO TRIGGER-BUFFER.
    MOVE 'Ø'         TO QDEPTH.
    MOVE '9'         TO INPUT-USE-COUNT.
    MOVE '9'         TO OUTPUT-USE-COUNT.
    IF MQ-COMPCODE = MQCC-OK
        MOVE 'Y'            TO INQ-QUEUE-FLAG
        MOVE MQ-INTATTRS(1) TO GET-INT-BUFFER
        MOVE MQ-INTATTRS(2) TO PUT-INT-BUFFER
        MOVE MQ-INTATTRS(3) TO TRIGGER-INT-BUFFER
```

```
           MOVE MQ-INTATTRS(4) TO QDEPTH
           MOVE MQ-INTATTRS(5) TO INPUT-USE-COUNT
           MOVE MQ-INTATTRS(6) TO OUTPUT-USE-COUNT
           IF PUT-INT-BUFFER = Ø THEN
              MOVE '1'         TO PUT-BUFFER
           END-IF
           IF TRIGGER-INT-BUFFER = 1 THEN
              MOVE '1'         TO TRIGGER-BUFFER
           END-IF
           IF GET-INT-BUFFER = Ø THEN
              MOVE '1'         TO GET-BUFFER
           END-IF
       ELSE
           MOVE 'N'           TO INQ-QUEUE-FLAG
           MOVE 'MQINQ    Q'  TO MQ-ERROR-OPERATION
           PERFORM HANDLE-ERROR
              THRU HANDLE-ERROR-EXIT.
   INQUIRE-APP-QUEUE-EXIT.
       EXIT.
   GET-BROWSE-APP-QUEUE.
       COMPUTE MQGMO-OPTIONS = MQGMO-ACCEPT-TRUNCATED-MSG +
                               MQGMO-BROWSE-FIRST.
       MOVE MQMI-NONE TO MQMD-MSGID.
       MOVE MQCI-NONE TO MQMD-CORRELID.
       CALL 'MQGET' USING MQ-HCONN
                          MQ-QUEUE-HOBJ
                          MQMD
                          MQGMO
                          MQ-BUFFLEN
                          MQ-GET-BUFFER
                          MQ-DATALEN
                          MQ-COMPCODE
                          MQ-REASON.
       IF (MQ-COMPCODE = MQCC-OK) OR
          (MQ-REASON = MQRC-TRUNCATED-MSG-ACCEPTED) THEN
           MOVE MQMD-MSGID             TO MSGID(I)
           MOVE 'Y'                    TO GET-QUEUE-FLAG
       ELSE
           IF (MQ-REASON = MQRC-NO-MSG-AVAILABLE)
               MOVE SPACES        TO MSGID(I)
               MOVE 'Y'           TO GET-QUEUE-FLAG
           ELSE
               MOVE 'N'           TO GET-QUEUE-FLAG
               MOVE '2222'        TO MSGID(I)
               MOVE 'MQGET    Q'  TO MQ-ERROR-OPERATION
               PERFORM HANDLE-ERROR
                  THRU HANDLE-ERROR-EXIT.
   GET-BROWSE-APP-QUEUE-EXIT.
       EXIT.
   CLOSE-APP-QUEUE.
       CALL 'MQCLOSE' USING MQ-HCONN
                            MQ-QUEUE-HOBJ
```

```
                          MQCO-NONE
                          MQ-COMPCODE
                          MQ-REASON.
    IF MQ-COMPCODE = MQCC-OK
        MOVE 'Y' TO CLOSE-QUEUE-FLAG
    ELSE
        MOVE 'N' TO CLOSE-QUEUE-FLAG
        MOVE 'MQCLOSE Q '  TO MQ-ERROR-OPERATION
        PERFORM HANDLE-ERROR
           THRU HANDLE-ERROR-EXIT.
CLOSE-APP-QUEUE-EXIT.
    EXIT.
READ-TSQ.
    EXEC CICS READQ TS QUEUE('QMON') INTO(TSQ-MESSAGE)
              ITEM(I) RESP(XXX)
    END-EXEC.
    IF ( XXX = DFHRESP(ITEMERR) ) OR (XXX = DFHRESP(QIDERR) )
         PERFORM DELETE-TSQ
            THRU DELETE-TSQ-EXIT
         MOVE 'N'  TO READ-TSQ-FLAG
         MOVE 'READTSQ  E'  TO MQ-ERROR-OPERATION
         PERFORM HANDLE-ERROR
            THRU HANDLE-ERROR-EXIT
    ELSE
         MOVE 'Y'  TO READ-TSQ-FLAG.
READ-TSQ-EXIT.
    EXIT.
DELETE-TSQ.
    EXEC CICS DELETEQ TS QUEUE('QMON') RESP(YYY)
    END-EXEC.
    IF (YYY = DFHRESP(QIDERR)) THEN
         MOVE 'N'  TO DELETE-TSQ-FLAG
         MOVE 'DELETETSQE'  TO MQ-ERROR-OPERATION
         PERFORM HANDLE-ERROR
            THRU HANDLE-ERROR-EXIT
    ELSE
         MOVE 'Y'  TO DELETE-TSQ-FLAG.
DELETE-TSQ-EXIT.
    EXIT.
WRITE-TSQ.
   MOVE QNAME-NLIST(I)         TO Q-NAME.
    MOVE MSGID(I)               TO OLD-MSGID.
    IF READ-TSQ-OK THEN
        EXEC CICS WRITEQ TS QUEUE('QMON') FROM(TSQ-MESSAGE)
                     ITEM(I) REWRITE RESP(XXX)
        END-EXEC
    ELSE
        EXEC CICS WRITEQ TS QUEUE('QMON') FROM(TSQ-MESSAGE)
                     ITEM(I) RESP(XXX)
        END-EXEC.
    IF XXX = DFHRESP(ITEMERR) THEN
         MOVE 'N'  TO WRITE-TSQ-FLAG
```

```
          MOVE 'WRITETSQE'  TO MQ-ERROR-OPERATION
          PERFORM HANDLE-ERROR
             THRU HANDLE-ERROR-EXIT
     ELSE
          MOVE 'Y'  TO WRITE-TSQ-FLAG.
WRITE-TSQ-EXIT.
     EXIT.
PUT-MESSAGES.
     PERFORM OPEN-OUTPUT-QUEUE
        THRU OPEN-OUTPUT-QUEUE-EXIT.
     PERFORM PUT-OUTPUT-QUEUE
        THRU PUT-OUTPUT-QUEUE-EXIT.
     PERFORM CLOSE-OUTPUT-QUEUE
        THRU CLOSE-OUTPUT-QUEUE-EXIT.
PUT-MESSAGES-EXIT.
     EXIT.
OPEN-OUTPUT-QUEUE.
      MOVE MQOT-Q                 TO MQOD-OBJECTTYPE.
      MOVE 'CMON'                 TO MQOD-OBJECTNAME.
      MOVE QMGR-NAME              TO MQOD-OBJECTQMGRNAME.
      MOVE ZEROS                  TO MQ-QUEUE-HOBJ.
      COMPUTE MQ-OPTIONS = MQOO-OUTPUT.
      CALL 'MQOPEN' USING MQ-HCONN
                          MQOD
                          MQ-OPTIONS
                          MQ-QUEUE-HOBJ
                          MQ-COMPCODE
                          MQ-REASON.
      IF MQ-COMPCODE = MQCC-OK
          MOVE 'Y'  TO OPEN-OUTPUT-FLAG
      ELSE
          MOVE 'N'  TO OPEN-OUTPUT-FLAG
          MOVE 'MQOPEN OQ '  TO MQ-ERROR-OPERATION
          PERFORM HANDLE-ERROR
             THRU HANDLE-ERROR-EXIT
          PERFORM END-PROG
             THRU END-PROG-EXIT.
OPEN-OUTPUT-QUEUE-EXIT.
     EXIT.
PUT-OUTPUT-QUEUE.
     MOVE MQMI-NONE TO MQMD-MSGID.
     MOVE MQCI-NONE TO MQMD-CORRELID.
     MOVE MQFMT-STRING TO MQMD-FORMAT.
     COMPUTE MQPMO-OPTIONS     = MQPMO-SYNCPOINT +
                                 MQPMO-FAIL-IF-QUIESCING.
     CALL 'MQPUT' USING  MQ-HCONN
                         MQ-QUEUE-HOBJ
                         MQMD
                         MQPMO
                         MQ-PUT-BUFFER-LEN
                         MQ-PUT-BUFFER
                         MQ-COMPCODE
```

```
                        MQ-REASON.
           IF MQ-COMPCODE = MQCC-OK
               MOVE 'Y'               TO PUT-OUTPUT-FLAG
           ELSE
               MOVE 'N'               TO PUT-OUTPUT-FLAG
               MOVE 'MQPUT  OQ '      TO MQ-ERROR-OPERATION
               PERFORM HANDLE-ERROR
                  THRU HANDLE-ERROR-EXIT
               EXEC CICS SYNCPOINT ROLLBACK END-EXEC
               PERFORM END-PROG
                  THRU END-PROG-EXIT.
       PUT-OUTPUT-QUEUE-EXIT.
           EXIT.
       CLOSE-OUTPUT-QUEUE.
           CALL 'MQCLOSE' USING MQ-HCONN
                                MQ-QUEUE-HOBJ
                                MQCO-NONE
                                MQ-COMPCODE
                                MQ-REASON.
           IF MQ-COMPCODE = MQCC-OK
               MOVE 'Y' TO CLOSE-OUTPUT-FLAG
           ELSE
               MOVE 'N' TO CLOSE-OUTPUT-FLAG
               MOVE 'MQCLOSEOQ '  TO MQ-ERROR-OPERATION
               PERFORM HANDLE-ERROR
                  THRU HANDLE-ERROR-EXIT
                  EXEC CICS SYNCPOINT ROLLBACK END-EXEC
               PERFORM END-PROG
                  THRU END-PROG-EXIT.
       CLOSE-OUTPUT-QUEUE-EXIT.
           EXIT.
       END-PROG.
           EXEC CICS RETURN END-EXEC.
           STOP RUN.
       END-PROG-EXIT.
           EXIT.
       DATE-CONVERSION.
           EXEC CICS ASKTIME ABSTIME (TIME-IN-MSEC) END-EXEC.
           EXEC CICS FORMATTIME ABSTIME (TIME-IN-MSEC)
                                YYYYMMDD (DATE-SYSTEM)
                                    TIME (TIME-SYSTEM)
           END-EXEC.
           MOVE DAY-SYSTEM     TO DAY-WORK.
           MOVE MONTH-SYSTEM   TO MONTH-WORK.
           MOVE YEAR-SYSTEM    TO YEAR-WORK.
           MOVE HOUR-SYSTEM    TO HOUR-WORK.
           MOVE MINUTE-SYSTEM  TO MINUTE-WORK.
           MOVE SECOND-SYSTEM  TO SECOND-WORK.
           MOVE DAY-SYSTEM     TO DAY-ERROR.
           MOVE MONTH-SYSTEM   TO MONTH-ERROR.
           MOVE YEAR-SYSTEM    TO YEAR-ERROR.
           MOVE HOUR-SYSTEM    TO HOUR-ERROR.
```

```
           MOVE MINUTE-SYSTEM  TO MINUTE-ERROR.
           MOVE SECOND-SYSTEM  TO SECOND-ERROR.
       DATE-CONVERSION-EXIT.
            EXIT.
       ABEND-PROC.
      * Error messages during the processing of this program    *
      * are written to a specific Cics TDQ TSSL.                 *
           EXEC CICS WRITEQ TD QUEUE('TSSL') FROM(ABEND-MESSAGE)
                            LENGTH(LEN-ABEND-MESSAGE) END-EXEC.
           PERFORM END-PROG
               THRU END-PROG-EXIT.
       ABEND-PROC-EXIT.
           EXIT.
       HANDLE-ERROR.
           MOVE MQ-COMPCODE   TO MQ-ERROR-COMPCODE.
           MOVE MQ-REASON     TO MQ-ERROR-REASON.
           EXEC CICS WRITEQ TD QUEUE('CKMQ') FROM(MQ-ERROR-MESSAGE)
                            LENGTH(LEN-MQ-ERROR-MESSAGE) END-EXEC.
       HANDLE-ERROR-EXIT.
           EXIT.
```

Barlas Solakoglu
Transaction and Messaging Systems Specialist (Turkey)

Enhancing MQSeries transaction coordination

INTRODUCTION

One of the functions introduced in MQSeries Version 5.0 for distributed platforms, released in 1997, was the ability to act as a transaction coordinator. This meant that updates to queues (messages being put and got) could be synchronized with updates made to other resource managers, such as SQL databases, without the need for additional products, such as CICS or Tuxedo. Although IBM is now delivering MQSeries Version 5.2, there have been no significant extensions to the transaction coordination functions since its original release.

This coordination facility has proven very useful to many customers and is also used by other IBM products such as WebSphere MQ Integrator (previously known as MQSeries Integrator). However, as with any successful feature, there have been requests for enhancements in order to make it easier to define and use the databases from within an application.

This article will begin by describing the technical details behind MQSeries transaction coordination. It will then go on to show how two of the suggested enhancements can be made without the need for any changes to MQSeries itself; specifically:

- The ability to vary the usernames used when connecting to databases.
- The ability to select the databases used by each individual application.

Using these options might improve the performance of your system and can extend your security configuration options. By the end of this article you should be able to understand and then customize for your own environment the example source code that implements these enhancements.

TRANSACTIONS AND XA

Transactions can be thought of as groups of operations, which either all succeed or all fail. They reduce the need for error recovery logic inside an application program. Databases and MQSeries can all be called 'Resource Managers', which handle transactions within their own scope of resources, such as tables or queues. When there is a requirement to update more than one Resource Manager (RM) as part of the same transaction, a Transaction Manager (TM) or Transaction Coordinator (TC) gets the job of driving all of the RMs so that they maintain a consistent state. Such a transaction, involving more than one RM, is often called a global unit of work.

While there are a number of ways in which this consistent state can be controlled, the most common protocol is called two-phase commit, or 2PC. One programming interface which has been defined to implement that protocol is called XA.

XA is an invention of the X/Open standards organization. The way a TM calls an RM is specified as a group of functions and variables in the C language. This does not mean that application programs, TMs, or RMs have themselves to be written in C, only that they will communicate using C calling conventions.

There are 12 function calls defined in the XA document, such as **xa_open**, **xa_close**, **xa_prepare**, and **xa_commit**. Ten of these are

used for calling from a TM to an RM; two are used for the RM to invoke functions in the TM. There is no standard for how an application program starts or specifies the outcomes of transactions; the syntax is dependent on the TM in use. For example, MQSeries provides the MQBEGIN, MQCMIT, and MQBACK verbs to manage global transactions. CICS programmers will be familiar with the EXEC CICS SYNCPT and EXEC CICS ROLLBACK statements.

The XA specification also defines state transitions and error codes. Other interfaces exist (for example, CICS on z/OS talks to MQSeries via a similar set of functions but using a different programming interface), but we will be concerned here only with XA.

Transaction management is often part of a much larger product. CICS and Tuxedo for example, have a lot of functions, such as the scheduling of transactions that are outside the basic TM operation. It was because these Transaction Processing Monitor products were often beyond what was needed by MQSeries customers that IBM developed the TM component of MQSeries.

APPLICATION PROGRAMMING WITH MQSERIES COORDINATION

An earlier article in *MQ Update* (March 2000) talks about using MQSeries as a transaction coordinator, and there are several sample programs shipped with MQSeries. I don't intend to repeat that information or go through it in detail. You can look at the IBM-supplied examples in the *samples/xatm* subdirectory after you have installed MQSeries. The basic structure of these programs, however, is much the same.

Leaving out all the parameter details, a program will often look something like this:

```
MQCONN()
MQOPEN()
Begin loop
  MQBEGIN()
  MQGET()
  EXEC SQL SELECT …
  EXEC SQL UPDATE …
  MQPUT1()
  MQCMIT()
End loop
MQCLOSE()
```

```
MQDISC()
```

There are a couple of things to note here. Firstly, we do not explicitly connect to the database. That would, in stand-alone programs, normally be done by the **EXEC SQL CONNECT** function: here, the queue manager does it on behalf of the application during the **MQBEGIN** call. Secondly, we do not use the **EXEC SQL COMMIT** statement. MQSeries handles that through the XA commitment protocol; if an application calls the database directly to resolve a transaction it is an error.

The significant verb in this program is the call to MQBEGIN. That tells the queue manager that the program is starting a global unit of work and that it needs to be ready to work with external resource managers. From there on, calls to MQCMIT and MQBACK will be able to drive the XA transaction processing functions inside those external resource managers as well as being able to control the MQSeries messages.

CONFIGURING MQSERIES FOR COORDINATION

MQSeries knows to coordinate an external resource from stanzas added to the queue manager's *qm.ini* file (or by updates to the corresponding part of the Windows NT registry). Each stanza tells the queue manager several things; the name of the database, the location of the switch file described below, and the parameters that must be passed to the database during **xa_open** and **xa_close** processing.

This example *qm.ini* file tells the queue manager that it will need to coordinate updates to two different DB2 databases. The XAOpenString field is what the DB2 client libraries use so that it can attach to the correct database and it may contain a number of additional parameters. Some of these attributes become important later, when we see how to extend the configuration possible with MQSeries.

QM.INI

```
------------------- /var/mqm/qmgrs/QMNAME/qm.ini ------------------
#* Module Name: qm.ini                                              *#
#* Type       : MQSeries queue manager configuration file           *#
#  Function   : Define the configuration of a single queue manager *#
#* Notes      :                                                     *#
#* 1) This file defines the configuration of the queue manager     *#
```

```
ExitPath:
   ExitsDefaultPath=/var/mqm/exits/
Service:
   Name=AuthorizationService
   EntryPoints=9
ServiceComponent:
   Service=AuthorizationService
   Name=MQSeries.Unix.auth.service
   Module=/usr/mqm/lib/amqzfu
   ComponentDataSize=0
Log:
   LogPrimaryFiles=3
   LogSecondaryFiles=2
   LogFilePages=64
   LogType=CIRCULAR
   LogBufferPages=17
   LogPath=/var/mqm/log/TWOPC/
XAResourceManager:
     Name=DB2 MQBankDB
     SwitchFile=/var/mqm/exits/db2swit
     XAOpenString=db=MQBankDB
XAResourceManager:
     Name=DB2 SAMPLE
     SwitchFile=/var/mqm/exits/db2swit
     XAOpenString=db=SAMPLE
------------------------- End qm.ini file -----------------------------
```

THE SWITCH FILE

The XA standard says that the fundamental part of the interface between a TM and an RM is a variable of a particular datatype. It is actually called the *xa_switch_t structure*, whose fields are defined further in the XA specification. The documentation for each RM defines the name of that variable, which has to be used by the TM to get access to the functions in the RM that do the real work.

How can we use that information inside a product like MQSeries? When C programs are compiled and linked you normally want or need to have all function symbols resolved during the link process. When IBM compiles MQSeries it does not know about symbols in other RMs, so there has to be a way to get and use that information at run-time.

The switch file is a way of deferring symbol resolution until run-time. The switch file is compiled as a dynamically-loadable module (or DLL), whose single entry-point returns the address of the *xa_switch* variable to the queue manager. Just like MQSeries channel exits, the

queue manager knows how to load these modules and execute the code at its entry-point. Because the switch file is itself linked with the database run-time libraries, all the functions that the application or queue manager needs to work with the database are now available directly inside the process and can be called or used.

There is normally one switch file for each database product such as DB2 or Oracle installed on a machine. Depending on the amount of compatibility between different versions of the RDBMS, there may need to be more than one switch file. For example, one vendor modified the XA capabilities when a new release of the product was shipped, and had a new name for the *xa_switch* variable to access this new function. This meant that separate switch files were needed; one referring to the old name and one referring to the new name.

Most switch files contain only one real line of C code. The difficulties that some people experience arise from finding the XA information in the first place, and then compiling and linking the module.

Here, as an example, is the absolute minimum source code for the DB2 switch file.

```
#include "xa.h"
extern struct xa_switch_t db2xa_switch;
struct xa_switch_t * MQStart(void){
  return(&db2xa_switch);
}
```

The important line here is the ‘return’. This tells the queue manager the address of DB2’s XA interface structure. To write this small piece of code we had to look in the DB2 manuals where the *db2xa_switch* variable is documented. MQSeries ships a file called *xaswit.mak*, which can be used to compile and link this module.

STATIC AND DYNAMIC XA

One of the flags returned in the *xa_switch* structure indicates whether or not the RM supports dynamic registration for transactions. The alternative static interface is slightly easier to implement and understand, but may not perform as well in an environment where many RMs might be known to a TM, but only a small number are actually used by any given transaction.

In the static XA interface, the TM tells every RM that it knows about every time a transaction is started . The TM does not get involved any further with the work carried out by the application program. Once the application has finished its work it tells the TM to do the commit processing and the TM blindly drives the two-phase protocol for all of these RMs. If the application has not done any work that requires a recoverable transaction on some of these RMs, the TM has just wasted some time asking it to commit non-existent work.

In dynamic XA, the job of enrolling in a transaction is left to the RM. As in static XA, the TM issues an **xa_open** to all RMs to create the connection, but this time the TM does not explicitly start any transaction. Instead, the application issues a recoverable request to an RM and the RM tells the TM (via the **ax_reg** call-back function) that it has been asked to be part of a transaction and requests the global transaction ID that it can share with the TM and any other RMs that are used. Now, when it is time to commit the work, the TM knows that it only need call the RMs that have actually participated in the transaction.

Dynamic XA is to be preferred wherever possible. In the case of a single RM being used with a queue manager, the performance of dynamic XA is about the same as static XA. Once more, RMs are defined to the TM, then the performance benefit becomes more noticeable.

Not all TMs and RMs support the dynamic XA option. However, when MQSeries is running as a coordinator, it does support RMs that wish to use dynamic XA.

WHEN DOES MQSERIES CALL THE DATABASE?

As most people probably know, the MQSeries design splits work between two processes – the application program and the agent acting on its behalf. When the application issues an MQGET call, only a small amount of code is executed in that process. The real work is done by the separate agent process that has access to all the important and protected resources, such as files, locks, and shared memory. This split of responsibility is continued in the XA implementation, where some of the XA verbs are executed directly by the application, some by the queue manager agent, and some by both processes. Of course, most databases also split function calls into a small stub inside the

application process and the real work is done inside their own agent processes, so there might be a number of interconnected processes in total. We do not need to worry about the distinction here; as far as MQSeries is concerned, all XA operations on an RM are executed in-line.

The XA functions can be grouped into several sets; connection management, transaction association, and transaction resolution. Both the application and the agent need to connect to the database, so the first MQBEGIN causes both processes to dynamically load the switch file and, in the case of static XA, to then issue the **xa_open** call to the database. In dynamic XA, the **xa_open** from the agent process is delayed until it is actually needed. The connection from the application process to the database is done immediately during the MQBEGIN call for both static and dynamic XA.

Transaction association (starting, suspending, and moving transactions) is all done within the application program. The queue manager does not need to issue any XA verbs from its own processes, although it does need to keep track internally of the state of each transaction.

The queue manager does all transaction resolution – when the application issues **MQCMIT** its agent drives the **xa_prepare** and **xa_commit** interfaces. Figure 1 illustrates this for a dynamic XA session. Static XA is similar, except that the **ax_reg** calls are replaced by **xa_start** in the application process.

In both static and dynamic XA the application program issues the **xa_open** call to all databases defined in the *qm.ini* file. This means that the security configuration of the database must be set up to allow any application that is using the queue manager to connect to the database. Often, the security controls are set so that the username associated with the running process is automatically used by the database (based on the assumption that the operating system has already authenticated the user), but this is not always the case. Sometimes, the database will be configured to require a *user-id* and password to be provided during the connection flows or, in this case, during the **xa_open** flows. If *user-id* and password parameters are needed they will be set as part of the *XAOpenString* parameter in the *qm.ini* file.

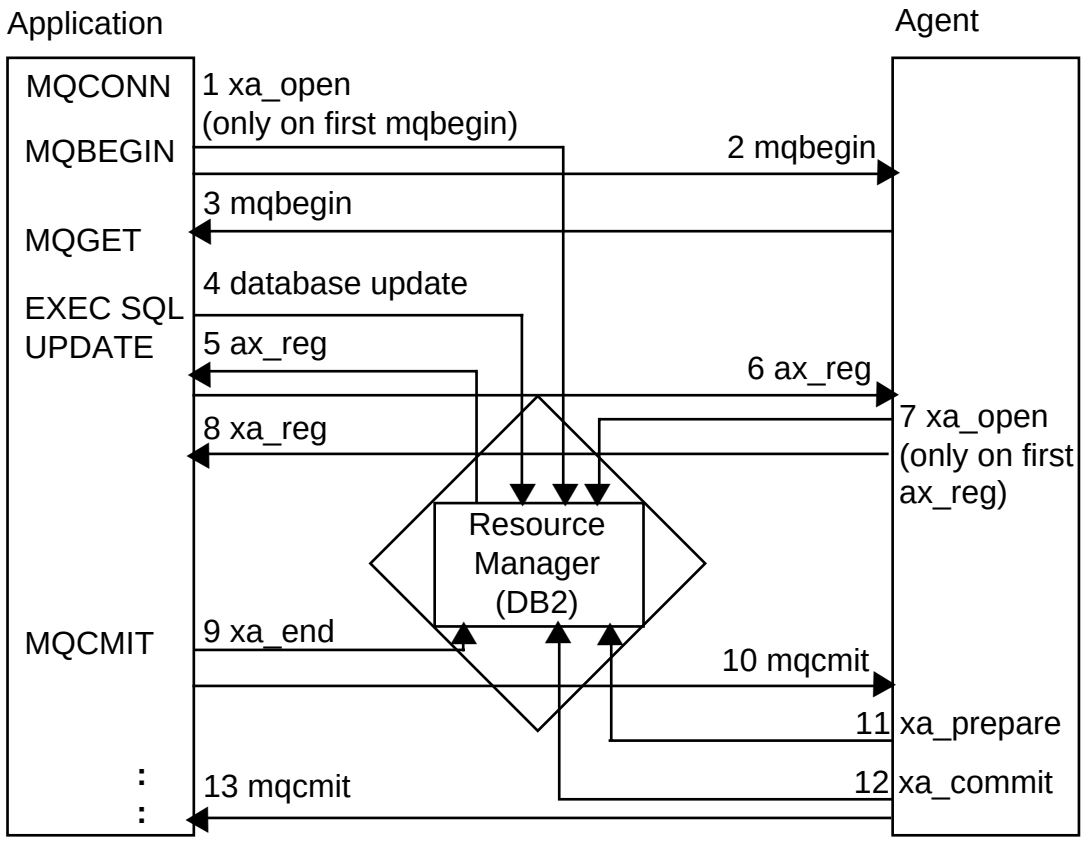

Figure 1: Interaction of MQI and XA verbs for dynamic XA resources

USING THE RIGHT DATABASE CONNECTION

In the *qm.ini* file above, two databases are configured. When the application calls MQBEGIN the queue manager does not know which of the databases is going to be used by the application – it could be one or both – so the MQSeries code in both the application and the agent sides of the picture will call **xa_open** for both databases. A call to **xa_open** is approximately equivalent to an **EXEC SQL CONNECT** and sets up all of the internal structures needed by the database for the application to work with.

The application should not (and probably must not) call **EXEC SQL CONNECT** as that would create a separate session, which could not be part of a global transaction. So the application starts doing its **SQL UPDATE** and **SQL INSERT** work. But which database is it actually working with? Experiments with DB2 show that, if the first thing you

do after **MQBEGIN** is to simply call **EXEC SQL UPDATE**, then that operation is sent to the last database listed in the *qm.ini* file – the last one that has been opened by the **xa_open** processing in the application side of the queue manager.

This might not be the right database for your work and so there are a couple of ways of forcing the right session to be used. The easiest method is to use **EXEC SQL SET CONNECTION**, where the parameter to this operation is the name of the database. For example, when using the two databases configured in the *qm.ini* file, I added the line **EXEC SQL SET CONNECTION MQBANKDB** to my test program immediately after the **MQBEGIN** verb. An alternative syntax supported by some products is to name the database on the actual operation **EXEC SQL AT MQBANKDB SELECT …**

This second method is most useful when you are only switching temporarily to a second database: if an application program only ever works with one database the first method is preferable.

THE SAMPLE XA SWITCH FILE

Earlier, I said that a typical XA switch file contains only a single functional line of C code, but there's nothing that enforces that. As the switch file tells the queue manager the address of the XA functions within the RM it's possible to modify those addresses and insert user-written code that gets called before (or even possibly instead of) the real functions inside the RM. And that is exactly what the sample code does. In particular, what it is designed for is to modify the **xa_open** step, so that the *XAOpenString* parameter can be modified, and in some cases, an RM can be completely bypassed.

The code extends the queue manager configuration in two ways. Firstly, it allows us to have different ways to authenticate to the database. Instead of requiring operating system authentication or that the *qm.ini* file should contain a *user-id* and password for all users of that database to connect with, we can now modify the authentication programmatically.

Secondly, the new switch file can decide to not even attempt to connect to a database that the application is not going to use. This is a performance improvement beyond that provided by the dynamic

XA capability, as setting up a connection is often a comparatively expensive operation.

How does it work?

The new switch file obtains the addresses of the DB2 XA functions in the usual way, by getting the address of the **xa_switch** variable. Now, however, instead of returning that address to the queue manager, a modified structure is filled in, containing the address of a new **xa_open** function implemented within the switch file itself. That new function needs to know the original address because after doing its work it will pass through to the original function.

Note that the code supplied here is written for DB2 and AIX, but it should be clear how it could be extended for other databases and other operating systems. There are a number of comments in the code to explain how it works and how it might be modified. The switch file also traces its behaviour to *stdout* if it is running in an interactive application, or writing to */tmp/db2.out* if it is loaded into a queue manager process.

DB2SWIT.C

```
/* Module name: db2swit.c
/* Description: MQSeries XA switch program for DB2 & AIX
#include <stdio.h>
#include <pwd.h>
#include <cmqc.h>                              /* MQ header             */
#include "xa.h"                                /* MQ supplied XA header */
/* This variable is supplied by DB2 and contains XA function pointers */
extern struct xa_switch_t db2xa_switch;
/* Local variables
static FILE *fp = NULL;            /* For debug printing
static struct xa_switch_t myxa;    /* The modified switch vector
static struct xa_switch_t dbxa;/* The original switch vector from DB2 */
static int (*reg_func)(int,char *,const char *) = NULL;  /* Appl
callback */
/* The max number of XA Resource Managers that might be defined to    */
/* MQSeries at this site. There is no 'real' maximum; choosing a      */
*/ moderate number lets me have a fixed size array for ease of use    */
*/ later. */ MQSeries allocates rmid values sequentially from 1; one  */
*/ for each stanza in the qm.ini file.                                 */
#define MAXRMS (50)
/* Is this rmid being used by this application? Assume 'yes'.         */
int rmlist[MAXRMS] = {TRUE};
int my_xa_open(char *xa_info,int rmid, long flags)
{
```

```
  int rc;
  struct passwd *pw;
  char local_xa_info[MAXINFOSIZE+1]={Ø}; /* length is defined in xa.h */
  char *p = xa_info;
  pw = getpwuid(geteuid());                          /* Who are we? */
  fprintf(fp,"!! Running as user %d (\"%s\")\n",pw->pw_uid,pw->pw_name);
  if (pw && (strcmp(pw->pw_name,"mqm")==Ø))
  {
/* We are running as 'mqm' ... This probably means we are running     */
/* inside a queue manager process, which will need full access to all */
/* resource managers so it can manage all transactions.               */
/* We will modify the xa_info, to add 'standard' username/password    */
/* information, perhaps read from a file to which only mqm has read   */
/* access. In this case, I'm going to add some hardcoded info.        */
/* (Bad idea for real code, good idea for samples!) Remember that the */
/* username must have authorities on the database to connect and then */
/* manage other people's transactions.                                */
    strcpy(local_xa_info,xa_info);
    strcat(local_xa_info,",uid=mqm,pwd=mqm");
    p = local_xa_info;
    rc = dbxa.xa_open_entry(p,rmid,flags);
  }
  else
  {
    p = local_xa_info;
    if (rmid >= MAXRMS)
    {
      fprintf(fp,"!! Too many Resource Managers defined\n");
      rc = XAER_RMERR;
    }
    else if (reg_func &&
(reg_func(sizeof(local_xa_info),local_xa_info,xa_info)==TRUE))
    {
      fprintf(fp,"!! Leaving resource turned on\n");
      rc = dbxa.xa_open_entry(strlen(p)?p:xa_info,rmid,flags);
    }
    else
    {
/* RMs which only support static registration cannot be disabled    */
/* through this interface. They must be opened always, as the qmgr  */
/* will also always open them and attempt to commit/rollback later  */
/* for this transaction branch.                                     */
/* Which would, of course, be an error if the rmid were not known   */
/* to the RM being driven.                                          */
      if (dbxa.flags & TMREGISTER)
      {
     fprintf(fp,"!! Bypassing resource\n");
     rmlist[rmid] = FALSE;
     rc = XA_OK;
      }
      else
      {
```

```
/* The app's registration function has said it doesn't want         */
/* to call this RM. But we're going to have to anyway. If the       */
/* local_xa_info has been modified, use that. Otherwise             */
/* use the original xa_info string.                                 */
     fprintf(fp,"!! Cannot bypass RM which uses static registration\n");
     rc = dbxa.xa_open_entry(strlen(p)?p:xa_info,rmid,flags);
      }
    }
  }
  fprintf(fp,"!! xa_open:\"%s\" rmid=%Ø8X rc=%d\n",p,rmid,rc);
  return rc;
}
/* A fairly simple xa_close function. We don't want to close a        */
*/ database that we've never really opened as that might be an error. */
int my_xa_close(char *xa_info,int rmid, long flags)
{
  int rc;
  if (rmlist[rmid])
    rc = dbxa.xa_close_entry(xa_info,rmid,flags);
  else
    rc = XA_OK;
  fprintf(fp,"!! xa_close:\"%s\" rmid=%Ø8X
rc=%d\n",xa_info?xa_info:"NULL",rmid,rc);
  return rc;
}
/* This is the entrypoint to the module. It's called once per process */
/* that loads it, no matter how many resource managers are defined    */
*/ that use the same switch file.                                     */
/* The name of the function has to be MQStart.
struct xa_switch_t * MQENTRY MQStart(void)
{
/* Take two copies of the switch .. one that we'll change and return, */
/* and one we'll leave alone, containing the original pointers        */
/* for this database.                                                 */
   memcpy(&myxa,&db2xa_switch,sizeof(struct xa_switch_t));
   memcpy(&dbxa,&db2xa_switch,sizeof(struct xa_switch_t));
/* These are the 2 functions that need to be overridden if we're      */
*/ going to bypass default processing.                                */
   myxa.xa_open_entry  = my_xa_open;
   myxa.xa_close_entry = my_xa_close;
/* Set up some debug print paths ... user apps will print to stdout,  */
/* while queue manager processes will be sent to a file which you can */
/* follow with 'tail -f'. I'll assume the fopen succeeds.             */
   if (isatty(fileno(stdout)))
     fp = stdout;
   else
     fp = fopen("/tmp/db2.out","a");
   setbuf(fp,Ø);
   fprintf(fp,"\n\n!! In the switch file %s: compiled at %s
%s\n",__FILE__,__DATE__,__TIME__);
   fprintf(fp,"!! XA flags %Ø8X\n",dbxa.flags);
   find_registration_function();
```

```
   /* And return the modified XA function pointers
return(&myxa);
}
#include <dlfcn.h>
typedef int (_PFN)(int,char *,const char*);
typedef _PFN *PFN;
find_registration_function()
{
  char *p=0;
  char *s=0;
  char *e=0;
  p = dlopen(NULL,RTLD_NOW);
  fprintf(fp,"dlopen p=%08X error=%s\n",p,e=dlerror()?e:"NULL");
  s = dlsym(p,"application_xa_registration");
  fprintf(fp,"dlsym  s=%08X error=%s\n",s,e=dlerror()?e:"NULL");
  dlclose(p);
  if (s)
    reg_func = (PFN)s;
}
```

Authentication modifications

Authentication is often be done by putting additional parameters on the *XAOpenString*. In the case of DB2, the format of the XAOpenString is **db=dbname,uid=xxx,pwd=yyy**.

If the username and password are not in the string then operating system authentication is assumed and the connection is made with the authority of the process issuing the **xa_open**. The Oracle syntax is different, but has similar capabilities.

The sample switch file is expecting that the *XAOpenString* passed to it does not contain a username or password. All authorizations inside the database will have been set for the application program, but transaction control functions will be issued by the *user-id* of the queue manager, the mqm account. There are several reasons why you might not want that account to be authorized in the same way as the application program; it is, in effect, an administrative account in the database as it is going to carry out management of other people's transactions. You might also want to have different *user-id*s associated with different queue managers, which is not possible unless you put that *user-id* in the *XAOpenString*.

The sample switch file is coded so that, if the *user-id* loading and running the switch is *mqm*, then a special *user-id* and password is added to the *XAOpenString*; otherwise the string is untouched and

passed directly to the database. The sample program has got the special *user-id* and password written in the source code; a more secure version would probably use an external configuration file that only the mqm user could read.

Clearly, this can be extended in a number of ways. For example, you could have the application program use different *user-id*s and passwords that might have to be read from privately-protected configuration files. Or you could pop-up a window asking for a *user-id* and password at run-time. You might like to design your database systems so that the *XAOpenString* stored in the *qm.ini* file can never be used directly for connection to a database: a separate *user-id* and password must always be used.

Dynamic selection of databases

The queue manager must always be able to connect to every database that it might need to coordinate. But application programs may only be using one or two of these databases and the user might not even be authorized to connect to the other databases.

When the queue manager attempts to open a connection to a database by calling the **xa_open** function, the switch file now has the opportunity to ask the application if the connection is really needed. The way I implemented this was to add a call-back from the switch file to the application code; because there is no way to add parameters to the load phase of the switch file, this seemed like a reasonably easy way to do the job.

An alternative approach that might be suitable for some environments is to have the list of RMs used by an application in an external configuration file (or even directly coded in the switch itself).

If the application program has a function called *application_xa_registration*, whenever the switch file has its **xa_open** function called it will ask the application code whether or not this RM (and the only readily available identification is from the *XAOpenString*) is going to be used by this program during the lifetime of this session from MQCONN to MQDISC. If the answer is 'yes' the **xa_open** is passed on to the real function inside the database. If the answer is 'no' we return directly to the queue manager and let it think that the database was opened. The code does remember that the connection was bypassed, so that it can also bypass the **xa_close** function later on,

but that is the only extra piece of processing needed. The call-back function is also allowed to modify the XAOpenString. My test program needed to do that in order to add a *user-id* and password that is not defined in the *qm.ini* file.

If a process running with the mqm *user-id* loads the switch file, the switch will always try to connect to the database. The main queue manager processes will need to connect to all databases in order to manage all transactions and perhaps recover them at start-up time. There is no simple way in the switch file to tell whether the process loading it is a queue manager internal process or an application program; basing the behaviour on the current *user-id* is a reasonable test.

Bypassing the connections is only suitable for RMs that support dynamic XA. If the RM only supports static XA we must not fake the connection processing as the queue manager will always attempt to commit transactions on all RMs. If it tries to do that for an RM that has not seen a started transaction, the RM will report an error that the queue manager will pass on.

My test program

To test the switch file and demonstrate the functions, I used the sample program *amqsxas0.sqc* provided with MQSeries. That sample requires you to define a database called *MQBankDB* with a single table called *MQBankT*. Full instructions on how to configure and run the sample are included in the *MQSeries Application Programming Guide*. I also created the DB2 sample database used by DB2 installation verification so that the queue manager could be defined with two XA resources, as shown in the *qm.ini* file.

Security was set on the two databases so that my personal *user-id* could not connect to *MQBankDB*. Instead, a new *user-id* was defined with a password in DB2. The mqm *user-id* was also defined to both databases with appropriate privileges to allow it to manage transactions for other *user-id*s. Operating system authentication was not allowed for the mqm *user-id* when connecting to the database and a password was defined for it in DB2.

There were only two modifications made to the IBM-supplied

amqsxas0.sqc. As mentioned earlier, immediately after the call to **MQBEGIN** I added the line:

```
EXEC SQL SET CONNECTION MQBankDB.
```

The second modification was to add a function that knows this application will be using only one of the defined databases. This code is the dynamic selection call-back invoked from the switch file. On AIX, in order that this function be accessible from the call-back, add "–bexpall" to the link step when compiling the program. Exporting the symbol may not be necessary on other operating systems.

```
int application_xa_registration(
    int len,
    char *outbuf,
    const char *inbuf)
{
  int rc;
  if (strstr(inbuf,"MQBankDB"))
  {
    strcpy(outbuf,inbuf);
    strcat(outbuf,",uid=dbu1,pwd=dbu1pwd"); /* use this id */
    rc = TRUE;
  }
  else
    rc = FALSE;
  printf("app_xa_reg: rc=%d inbuf \"%s\"\n",rc,inbuf);
  return rc;
}
```

CONCLUSION

Using MQSeries as a coordinator can be a cost-effective way of building business transactions that reliably update other resources. Getting the best out of MQSeries in such configurations can only be done with an understanding of how MQSeries works. While the sample switch file does not attempt to implement a production-ready solution for all systems, I hope it has given some ideas to system administrators and programmers on how to extend MQSeries capabilities further into their enterprises.

Mark Taylor
Technical Strategist, IBM Hursley (UK)

MQSeries Integrator V2 performance

This article outlines how we first tested the performance of MQSI V2 and explains some of our conclusions. IBM does publish some benchmark performance figures as a support pack but the information provided – although useful – is not a substitute for testing with the actual flows, data, and platforms expected in a given environment.

OBJECTIVES OF OUR STUDY

- Measure MQSI throughput performance using flows similar to our intended asynchronous application, which updates a DB/2 database.
- Measure MQSI performance with a typical client/server request/reply scenario (synchronous; with two flows used, one for the request, one for the reply).
- Suggest the production hardware capacity for MQSI in terms of TPC-C units.

HARDWARE AND SOFTWARE USED

MQSI V2.0.1 was installed under Windows NT 4 on an IBM Netfinity 6000R with four Intel Pentium III 700 MHz processors and 2 GB of memory. Three internal SCSI disks (10,000 RPM) were fitted and configured as separate disks (non-RAID). This machine is rated at approximately 34,000 TPC-C. MQSeries 5.1 was installed.

We tested the effect of using MQSeries 5.2 as this is meant to increase the queue manager performance. However, we could not detect any difference and the reason would appear to be that 5.2 optimizes the MQ log locking algorithm, and this does not improve performance until a larger number of broker applications are running.

TESTING METHODS

Two testing techniques were used; the first was a Java program, which had been developed as a stress-testing tool (a similar program has been published in *MQ Update* previously). Running on several PCs,

this was able to send multiple MQ requests to the MQSI broker, receive MQ replies, and time the responses.

The second method involved loading an MQSeries queue with several thousand sample messages and then running them through the MQSI broker. With both methods, the Windows NT performance monitor (perfmon) was able to measure the output queue message enqueue rate, CPU usage, and disk activity levels.

Limitations of the testing

- Performance will depend on the MQSI flow complexity and, since this can vary almost infinitely, we made broad assumptions with our sample flows.
- With the request/reply flow we did not include any application server processing time, but assumed a broker turnaround target of 1.5 seconds. (One flow was used to route the inbound request to a queue and another flow was used to route the reply from this queue back to the requestor.)
- Our database accesses were being made to a very small DB2 UDB database (this being the sample database that comes with UDB).
- We did not have access to real test data so we used messages of a length estimated to represent the average expected.
- The disks tested were 10,000 RPM disks configured on a single controller (non-RAID). The performance of other disks could vary considerably.

OUR REQUIREMENTS FOR THROUGHPUT

- We estimated a requirement throughput of 20 transactions per second for the asynchronous application. These test messages were 2K in length.
- We had no specific requirements for request/reply transactions. Most of these messages would be non-persistent. Our test messages were 8K in length.

TESTING RESULTS OBTAINED

Various factors affect the throughput, which will ultimately be limited either by the CPU or disk performance. With persistent messages that are logged to disk (the MQ logs), the disk performance will normally be the limiting factor. If database updates are XA coordinated with MQ transactions, then extra database logging takes place.

XA coordination links the database transaction to the MQseries transactions in the same unit of work, so that both are either committed or rolled-back. Using XA coordination had a dramatic effect on performance, reducing the throughput (in our tests) by about 30% because of disk logging.

Node complexity (ESQL statements) requires more CPU power as the complexity increases. Adding more CPUs is an easy way to increase capacity in this case, assuming the application can be scaled up without a bottleneck occurring. We collected a large number of test figures, but publishing them all would be rather confusing. I have summarized them here (using four MQSI execution groups):

- The asynchronous flow (XA on) sustained a rate of 49 messages per second. (CPUs 75% used, disks 65% busy.)
- The request/reply flows sustained a rate of 20 messages per second. (CPUs 40% used, disks 99% busy.)
- A combination of the above sustained a rate of 30 (async) and 14 (r/r). (CPUs 92% used, disks 83% busy.)

ACCEPTABILITY OF THE RESULTS

- The 6000R (with a 34,000 TPC-C rating) processed our asynchronous throughput comfortably, even with a synchronous workload overlaid.
- Routing real-time request/replies through a broker will add a delay overhead to the direct response times, in the order of 700 milliseconds.

RECOMMENDATIONS

- The results suggest a target platform of around 40,000 TPC-C rating with room for extra CPUs, ideally with fast-write disk caches (or ESS).

- Capacity would need to be re-checked if additional workloads are added or the complexity of the flows grows beyond those assumed.
- Tests should be conducted on other platforms if Intel/NT is not used in production. The performance under Unix might be quite different.
- The good performance under Windows NT suggests that one approach to minimize hardware costs would be to use clustered Intel platforms.

CONCLUSIONS AND AFTERTHOUGHTS

Since we conducted this study we have found that our application complexity in terms of lines of ESQL and the number of MQSI nodes in the flows has grown far more than anticipated. Our application is now more CPU- than disk-constrained, given our use of high-performance disk sub-systems. Therefore, the ability to scale to more CPUs is critical and any hardware should be chosen with this in mind.

Version 2.0.2 of MQSI has become available with significant improvements in performance. We have seen our application perform 50% faster on the new release, and would recommend its use for this reason alone. There are also ways to optimize the coding of flows and I will expand on this subject in a future article. MQSI v2 performance is manageable with careful testing and planning – providing that the applications are scalable (that is several flows can run well in parallel).

APPENDIX A: DETAILS OF TESTED MQSI FLOWS

Asynchronous functional flow

This simulated a typical asynchronous flow in terms of processing complexity, ie number and type of nodes, amount of SQL and database activity. XA co-ordination was switched on, and commit count was found to be most effective with a value of one.

- Message size in/out: 2K persistent messages.
- SQL: 110 lines.
- Database: 20 selects: one update, which consisted of an insert immediately followed by a delete.

The flow sequence comprises the following nodes:

- Message in.
- Exception subflow – a generic trycatch flow.
- Filter.
- Conversion to XML (compute).
- Generic translation subflow:
  - database lookup
  - compute processing
  - database update.
- Message out.

Synchronous flow (request/reply)

This simulates a typical synchronous request/reply process and consists of two flows: one for request, one for reply. XA co-ordination was switched off.

- Message size in/out: 8K non-persistent messages.
- SQL: 150 lines.
- Database: one select.

The flow sequence comprises the following nodes:

- Request:
  - message in
  - XML to XML transformation (compute)
  - filter
  - database lookup
  - conversion of XML to MRM COBOL – use of the MQSI Message Repository
  - filter x2
  - message out.
- Reply:
  - message in
  - reset content to use new message set
  - conversion of MRM to XML COBOL – use of the MQSI Message Repository
  - filter
  - message out.

MQ news

IBM has announced Version 1 Release 1 of its Tivoli Manager for MQSeries Integrator application integration software, the systems management software for MQSI.

It monitors and manages the MQSI infrastructure and helps optimize availability and response time. It operates as a management tool that manages MQSI as a component of overall business systems, linking to databases, applications, and other middleware products.

Specific features include availability management, discovery of MQSeries Integrator components, configuration management of MQSeries Integrator and Tivoli components, performance management with an MQSeries Integrator node, and operations management.

New features include support for MQSI V2.0.1 and V2.0.2, tracking changes to broker topology, performance monitoring of message flows with a Tivoli monitoring node, and availability monitoring of MQSI components, administration, event filtering, and automation.

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

* * *

Xephon's annual *MQ Update 2001* event runs 12-13 December at the Radisson SAS Portman Hotel in London. This two-day Conference provides a thorough analysis of recent product developments in the MQ environment and provides essential pointers on how to maximize performance within the enterprise.

For further information contact:
Xephon, 27-35 London Road, Newbury, Berks, RG14 1JL, UK.
Tel: + 44 (0) 1635 33823
Fax: + 44 (0) 1635 38345
Web:http://www.xephon.com/events

* * *

Sonic Software has recently introduced SonicMQ 4.0, the next generation of its messaging software, which is based on the Java Message Service (JMS) specification.

Enhancements include secure HTTP support, multi-part message streaming with guaranteed delivery for any message size, per-message encryption to ensure maximum security, and client-side persistence that guarantees client-to-broker message delivery and extends client resilience to network outages.

For further information contact:
Progress Software, 14 Oak Park, Bedford, MA 01730, USA
Tel: +1 781 280 4000
Fax: +1 781 280 4095
Web: http://www.progress.com

Progress Software, 210 Bath Road, Slough, Berkshire SL1 3XE, UK
Tel: +44 1753 216300
Fax: +44 1753 216301

* * *