32

MQ

February 2002

In this issue

update

Published by

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

North American office

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

Contributions

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

Printed in England.

MQSeries channel events: printing details

The RPG ILE program included in this article is designed to print out details of MQSeries channel events, of which there are seven in total.

The channel event program – EVENTCH – reads the system default queue *System.Admin.Channel.Event* and prints event details to the *EVTPRTF* printer file.

Included with the program are RPG ILE header constant files that will need to be declared for use with MQSeries. These constant values do not exist by default otherwise.

The *CONST* file contains the constant decimal values of each channel event name and all event field names.

The *MQCFH* file defines the event message header, which contains field data common to all event messages.

The *MQCFST* file defines the structure used to retrieve string data from event messages.

The *MQCFIN* file defines the structure used to retrieve integer data from event messages.

EVENTCH is the RPG ILE program; *EVTPRTF* is the printer output file.

EVENTCH

```
D**  File name:  CONST                                          **
D**  Description:   This file declares named constants for the   **
D**                 IBM Message Queue Interface (MQI) in the    **
D**                 OS/400 environment.                         **
D* MQ String Type
D MQ_STR          C                   CONST(4)
D* MQ Integer Type
D MQ_IN           C                   CONST(3)
D* Channel Started Event
D CHSTRT          C                   CONST(2282)
D* Channel Stopped Event
D CHSTOP          C                   CONST(2283)
D* Channel Activated
D CHACT           C                   CONST(2295)
```

```
D* Channel Not Activated
D CHNACT          C                   CONST(2296)
D* Channel Conversion Error
D CHCERR          C                   CONST(2284)
D* Channel Auto-Def OK
D CHADOK          C                   CONST(2233)
D* Channel Auto-Def Error
D CHADER          C                   CONST(2234)
D* Queue Manager Name
D QMNAME          C                   CONST(2Ø15)
D* Channel Name
D CHNAME          C                   CONST(35Ø1)
D* Transmission Queue Name
D XQNAME          C                   CONST(35Ø5)
D* Connection Name
D CNNAME          C                   CONST(35Ø6)
D* Aux Error Data String 1
D AEDS1           C                   CONST(3Ø26)
D* Aux Error Data String 2
D AEDS2           C                   CONST(3Ø27)
D* Aux Error Data String 3
D AEDS3           C                   CONST(3Ø28)
D* Reason Qualifier
D RSQUAL          C                   CONST(1Ø2Ø)
D* Error Identifier
D ERRIDR          C                   CONST(1Ø13)
D* Aux Error Data Int 1
D AEDI1           C                   CONST(1Ø7Ø)
D* Aux Error Data Int 2
D AEDI2           C                   CONST(1Ø71)
D* Conversion Reason Code
D CHCRC           C                   CONST(1Ø72)
D* Format Name
D FORNAM          C                   CONST(3533)
D* Channel Type
D CHTYPE          C                   CONST(1511)
D*****************************************************************
D**  Program name: MQCFH                                         **
D**  Description: MQSeries Header Structure                      **
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D* MQCFH Structure
D* Structure type
D  HTYPE                  1      4B Ø INZ(Ø)
D* Structure length
D  HSLEN                  5      8B Ø INZ(Ø)
D* Structure version number
D  HVNO                   9     12B Ø INZ(Ø)
D* Command identifier
D  HCOM                  13     16B Ø INZ(Ø)
D* Message sequence number
D  HMSNO                 17     2ØB Ø INZ(Ø)
```

```
D* Control
D  HCON                  21     24B Ø INZ(Ø)
D* CompCode
D  HCCODE                25     28B Ø INZ(Ø)
D* Reason
D  HREAS                 29     32B Ø INZ(Ø)
D* Parameter Count
D  HPCNT                 33     36B Ø INZ(Ø)
D*****************************************************************
D**  Program name:  MQCFST                                       **
D**  Description: MQSeries String Structure Definition           **
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D* MQCFST Structure
D* Structure type
D  STTYPE                 1      4B Ø
D* Structure length
D  STLENG                 5      8B Ø
D* Parameter
D  STPARM                 9     12B Ø
D* CodedCharSetID
D  STCCSID               13     16B Ø
D* String Length
D  STLEN                 17     20B Ø
D* String
D  STSTR                 21    284
D*****************************************************************
D**  Program name:  MQCFIN                                       **
D**  Description: MQSeries Integer Structure Definition          **
D*..1....:....2....:....3....:....4....:....5....:....6....:....7..
D* MQCFIN Structure
D*
D* Structure type
D  INTYPE                 1      4B Ø
D* Structure length
D  INSLEN                 5      8B Ø
D* Parameter identifier
D  INPARM                 9     12B Ø
D* Value
D  INVAL                 13     16B Ø
H
 ***************************************************************
 * Program name: EVENTCH                                       *
 * Description: Get a single event message from                *
 * System.Admin.Channel.Event Queue, parse the event message    *
 * details into the appropriate integer/string stucture and     *
 * print out the message details to a printer file to be read.  *
FEVTPRTF   O    E             PRINTER
 **  Declare MQI structures needed
D/COPY CONST
 * Message Header
D BUFFER          DS          1Ø24
```

```
D BUFCFH          DS
D/COPY MQCFH
 * MQCFST Parameter
D MQCFST          DS                  BASED(PS)
D/COPY MQCFST
D PS              S               *
 * MQCFIN Parameter
D MQCFIN          DS                  BASED(PI)
D/COPY MQCFIN
D PI              S               *
 * Char
D CHAR            DS
D                        1      1A
 * MQI named constants
D/COPY CMQR
 * Object Descriptor
D MQOD            DS
D/COPY CMQODR
 * Message Descriptor
D MQMD            DS
D/COPY CMQMDR
 * Get message options
D MQGMO           DS
I/COPY CMQGMOR
C                   EXSR      $INIT
C                   EXSR      $OPENQ
C                   EXSR      $GETMSG
C                   EXSR      $CLOSEQ
C                   SETON                                          LR
 *   Initialize values where required.                             *
C     $INIT         BEGSR
C                   MOVEL     *BLANKS       DATA             26
C                   EVAL      DATA = 'SYSTEM.ADMIN.CHANNEL.EVENT'
 *   Create object descriptor for source queue                     *
C                   MOVEL     DATA          ODON             48
 *   MQCONN is implicit on OS/400;                                 *
C                   Z-ADD     HCDEFH        HCONN
 *
C                   Z-ADD     0             CNT               2 0
C                   Z-ADD     0             AERDI1            4 0
C                   Z-ADD     0             AERDI2            4 0
C                   Z-ADD     0             ERIDR             8 0
C                   Z-ADD     0             RQUAL             4 0
C                   Z-ADD     0             CHCRSC            4 0
C                   Z-ADD     0             CHLTYP            4 0
C                   MOVEL     *BLANKS       RTQM             48
C                   MOVEL     *BLANKS       PDAT              8
C                   MOVEL     *BLANKS       PTIM              8
C                   MOVEL     *BLANKS       QMNAM            48
C                   MOVEL     *BLANKS       CHNAM            20
C                   MOVEL     *BLANKS       TQNAM            48
```

```
C                   MOVEL     *BLANKS       CNNAM           264
C                   MOVEL     *BLANKS       AERDS1           48
C                   MOVEL     *BLANKS       AERDS2           48
C                   MOVEL     *BLANKS       AERDS3           48
C                   MOVEL     *BLANKS       FORMNM            8
C                   ENDSR
 *   Open the System.Admin.Channel.Event                        *
C     $OPENQ        BEGSR
 * options are input-as-queue-def and fail-if-quiescing
C     OOINPQ        ADD       OOFIQ         OPTS
C                   Z-ADD     MQOPEN        CID
C                   CALL      'QMQM'
C                   PARM                    CID              9 0
C                   PARM                    HCONN            9 0
C                   PARM                    MQOD
C                   PARM                    OPTS             9 0
C                   PARM                    HOBJ             9 0
C                   PARM                    OCODE            9 0
C                   PARM                    REASON           9 0
 * report reason, if any; stop if failed
C     REASON        IFNE      RCNONE
 **  "MQOPEN ended with reason code ...."
C                   MOVEL     'MQOPEN'      CNAME             6
C                   MOVE      REASON        RCODE            10
C                   ENDIF
C                   ENDSR
 *   Get messages from message queue                            *
C     $GETMSG       BEGSR
 **  initial loop condition based on result of MQOPEN
C                   MOVE      OCODE         CCODE
 *   buffer length available ...
C                   EVAL      BUFLEN = %SIZE(BUFFER)
 * option is to wait up to 15 seconds for next message
C                   Z-ADD     GMWT          GMOPT
C                   ADD       GMCONV        GMOPT
 * Add ACCEPT TRUNCATED MSG to GetMessage Options
C                   ADD       GMATM         GMOPT
 * Wait upto 15 seconds
C                   Z-ADD     15000         GMWI
 **  MsgId and CorrelId are selectors that must be cleared
 **  to get messages in sequence, and they are set each MQGET
C                   MOVEL     MINONE        MDMID
C                   MOVEL     CINONE        MDCID
C     CCODE         IFNE      CCFAIL
C                   Z-ADD     MQGET         CID
C                   CALL      'QMQM'
C                   PARM                    CID              9 0
C                   PARM                    HCONN            9 0
C                   PARM                    HOBJ             9 0
C                   PARM                    MQMD
C                   PARM                    MQGMO
```

```
C                   PARM                    BUFLEN            9 Ø
C                   PARM                    BUFFER
C                   PARM                    MESLEN            9 Ø
C                   PARM                    CCODE             9 Ø
C                   PARM                    REASON            9 Ø
C     REASON        IFEQ      RCNONE
C                   WRITE     EVTPRTFH1
C                   EVAL      RTQM = MDRM
C                   EVAL      PDAT = MDPD
C                   EVAL      PTIM = MDPT
C                   WRITE     EVTPRTFD1
C                   EVAL      BUFCFH = BUFFER
C                   EVAL      PS = %ADDR(BUFFER)+%SIZE(BUFCFH)
C                   Z-ADD     Ø             CNT
 *   Loop to handle all fields of the event message
C     CNT           DOWLT     HPCNT
 *   Parse string values
C     STTYPE        IFEQ      MQ_STR
C                   SELECT
C                   WHEN      QMNAME = STPARM
C                   EVAL      QMNAM = %SUBST(STSTR:1:STLEN)
C                   WHEN      CHNAME = STPARM
C                   EVAL      CHNAM = %SUBST(STSTR:1:STLEN)
C                   WHEN      XQNAME = STPARM
C                   EVAL      TQNAM = %SUBST(STSTR:1:STLEN)
C                   WHEN      CNNAME = STPARM
C                   EVAL      CNNAM = %SUBST(STSTR:1:STLEN)
C                   WHEN      FORNAM = STPARM
C                   EVAL      FORMNM = %SUBST(STSTR:1:STLEN)
C                   OTHER
C                   EVAL      STPARM = STPARM
C                   ENDSL
C                   ENDIF
 *   Parse Integer values
C     STTYPE        IFEQ      MQ_IN
C                   EVAL      PI = PS
C                   SELECT
C                   WHEN      RSQUAL = INPARM
C                   EVAL      RQUAL = INVAL
C                   WHEN      ERRIDR = INPARM
C                   EVAL      ERIDR = INVAL
C                   WHEN      AEDI1 = INPARM
C                   EVAL      AERDI1 = INVAL
C                   WHEN      AEDI2 = INPARM
C                   EVAL      AERDI2 = INVAL
C                   WHEN      CHCRC = INPARM
C                   EVAL      CHCRSC = INVAL
C                   WHEN      CHTYPE = INPARM
C                   EVAL      CHLTYP = INVAL
C                   OTHER
C                   EVAL      INPARM = INPARM
```

```
C                   ENDSL
C                   ENDIF
C                   EVAL      PS = PS + STLENG
C                   EVAL      CNT = CNT + 1
C                   ENDDO
 *   For each channel event, print out the appropriate message
C     HREAS         IFEQ      CHSTRT
C                   WRITE     EVTPRTFH2
C                   WRITE     EVTPRTFD2
C                   ENDIF
C     HREAS         IFEQ      CHSTOP
C                   WRITE     EVTPRTFH3
C                   WRITE     EVTPRTFD2
C                   WRITE     EVTPRTFD3
C                   ENDIF
C     HREAS         IFEQ      CHACT
C                   WRITE     EVTPRTFH4
C                   WRITE     EVTPRTFD2
C                   ENDIF
C     HREAS         IFEQ      CHNACT
C                   WRITE     EVTPRTFH5
C                   WRITE     EVTPRTFD2
C                   ENDIF
C     HREAS         IFEQ      CHCERR
C                   WRITE     EVTPRTFH6
C                   WRITE     EVTPRTFD2
C                   WRITE     EVTPRTFD4
C                   ENDIF
C     HREAS         IFEQ      CHADOK
C                   WRITE     EVTPRTFH7
C                   WRITE     EVTPRTFD2
C                   WRITE     EVTPRTFD5
C                   ENDIF
C     HREAS         IFEQ      CHADER
C                   WRITE     EVTPRTFH8
C                   WRITE     EVTPRTFD2
C                   WRITE     EVTPRTFD6
C                   ENDIF
C                   ENDIF
C                   ENDIF
C                   ENDSR
 *   Close the System.Admin.Channel.Event queue (if open)       *
C     $CLOSEQ       BEGSR
 * test if queue had been opened
C     OCODE         IFNE      CCFAIL
 * ... close queue (with no options) if it is open
C                   Z-ADD     MQCLOS        CID
C                   Z-ADD     CONONE        OPTS
C                   CALL      'QMQM'
C                   PARM                    CID             9 Ø
C                   PARM                    HCONN           9 Ø
```

```
     C                   PARM                    HOBJ              9 0
     C                   PARM                    OPTS              9 0
     C                   PARM                    CCODE             9 0
     C                   PARM                    REASON            9 0
      *
     C                   ENDIF
     C                   ENDSR
A*****************************************************************
A**  Program name:  EVTPRTF (Event Printer File)                 **
A**  Description: Event message template                        **
A              R EVTPRTFH1
A                                       SKIPB(2)
A                                      2'MQSeries Event Monitor'
A                                      60'Page:'
A                                      67PAGNBR
A                                       EDTCDE(Z)
A                                       SPACEA(2)
A                                      1DATE
A                                       EDTCDE(Y)
A                                       SPACEA(4)
A              R EVTPRTFH2
A                                       SKIPB(2)
A                                      30'Event = Channel Started'
A                                       SPACEA(2)
A              R EVTPRTFH3
A                                       SKIPB(2)
A                                      30'Event = Channel Stopped'
A                                       SPACEA(2)
A              R EVTPRTFH4
A                                       SKIPB(2)
A                                      30'Event = Channel Activated'
A                                       SPACEA(2)
A              R EVTPRTFH5
A                                       SKIPB(2)
A                                      30'Event = Channel Not
Activated'
A                                       SPACEA(2)
A              R EVTPRTFH6
A                                       SKIPB(2)
A                                      30'Event = Channel Conversion
Error'
A                                       SPACEA(2)
A              R EVTPRTFH7
A                                       SKIPB(2)
A                                      30'Event = Channel Auto-Def
OK'
A                                       SPACEA(2)
A              R EVTPRTFH8
A                                       SKIPB(2)
A                                      30'Event = Channel Auto-Def
Error'
```

```
A                                      SPACEA(2)
A                R EVTPRTFD1
A                                     3Ø'*** Message Descriptor ***'
A                                      SPACEA(3)
A                                      1'Reply-To-QMgr:'
A                  RTQM          48A   17
A                                      SPACEA(2)
A                                      1'Put Date:'
A                  PDAT           8A   11
A                                      SPACEA(2)
A                                      1'Put Time:'
A                  PTIM           8A   11
A                                      SPACEA(3)
A                R EVTPRTFD2
A                                     3Ø'*** Message Data ***'
A                                      SPACEA(3)
A                                      1'Queue Manager:'
A                  QMNAM         48A   16
A                                      SPACEA(2)
A                                      1'Channel Name:'
A                  CHNAM         2ØA   15
A                                      SPACEA(2)
A                                      1'Transmission Queue:'
A                  TQNAM         48A   21
A                                      SPACEA(2)
A                                      1'Connection Name:'
A                  CNNAM        264A   18
A                                      SPACEA(2)
A                R EVTPRTFD3
A                                      SPACEA(3)
A                                      1'Reason Qualifier:'
A                  RQUAL          4S   19
A                                      SPACEA(2)
A                                      1'Error Identifier:'
A                  ERIDR          8S   19
A                                      SPACEA(2)
A                                      1'Aux Error Data Int1:'
A                  AERDI1         4S   22
A                                      SPACEA(2)
A                                      1'Aux Error Data Int2:'
A                  AERDI2         4S   22
A                                      SPACEA(2)
A                R EVTPRTFD4
A                                      SPACEA(3)
A                                      1'Conversion Reason Code:'
A                  CHCRSC         4S   25
A                                      SPACEA(2)
A                                      1'Format:'
A                  FORMNM         8A    9
A                                      SPACEA(2)
A                R EVTPRTFD5
```

```
A                                       SPACEA(3)
A                                      1'Channel Type:'
A                 CHLTYP         4S    14
A                                       SPACEA(2)
A               R EVTPRTFD6
A                                       SPACEA(3)
A                                      1'Channel Type:'
A                 CHLTYP         4S    14
A                                       SPACEA(2)
A                                      1'Error Identifier:'
A                 ERIDR          8S    19
A                                       SPACEA(2)
A                                      1'Aux Error Data Int1:'
A                 AERDI1         4S    22
A                                       SPACEA(2)
```

Markian Jaworsky
Consultant (Australia)

The mechanics of MQSeries triggering

TRIGGERING OVERVIEW

Triggering is a very powerful yet often misunderstood and misused feature of MQSeries: it is a means of starting an application when a message arrives on a queue.

The aim of this article is to break down triggering into its component parts. It is by building this foundation of understanding that the reader will be in a better position to understand when and how triggering should be used in their environment.

MQSeries has a great deal of built-in functionality and sometimes we are tempted to use a specific function or feature of the product simply because it's there. So let's begin by taking a look at when using triggering does and doesn't make sense.

GOOD REASONS TO USE TRIGGERING

There are many good reasons to implement MQSeries application triggering. Many installations have hundreds of queues that need to be serviced by receiving applications. It would be impractical to have hundreds of receiving programs running all the time, just waiting for messages to process. This would be a terrific waste of system resources as well as making it very difficult to gauge the real workload on the system at any given time.

Applications that would be good candidates for triggering include the following:

- Applications where messages arrive sporadically, throughout the day. If the workload is not constant it doesn't make sense to have the server programs running all the time.
- Applications where there is no time-of-day dependence on when the server program runs.

GOOD REASONS FOR NOT USING TRIGGERING

Just as with any other function there is a time and a place to use triggering. It is not the answer for every situation.

Applications that might not be the best candidates for triggering include:

- Applications where the rate at which messages arrive on the server queues is fairly constant throughout the day. In this case it may be more efficient to leave the server program running instead of repeatedly triggering it.
- Applications where the server application has to be started by something other than just the arrival of a message on a queue. For example, an end-of-day application that needs to start at a specific time would need to use some type of operating system scheduling software, such as CRON on Unix or CA-7 on OS/390.

THE COMPONENTS OF TRIGGERING

The sending application

First, there is an application from which the messages will originate. This application will put messages on a triggered queue.

The target queue

This is the queue to which the sending application puts messages and from which the server application gets messages. The trigger attributes of this queue are set by the administrator.

The server application

This is the application program that we wish to trigger. The arrival of messages on the target queue will cause the server application to start.

The process definition

In order for the queue manager to start an application when a message arrives on a queue it needs to have some information about which application to start and any startup parameters that might be necessary. These values are stored in a process object in the queue manager. The

process object is associated with a particular queue via the queue's PROCESS attribute. There follows a sample process definition:

```
DEFINE    PROCESS (MyProcess) -
                  APPLICID ('/var/mqm/utilities/QueueReader') -
                  APPLTYPE (unix) -
                  ENVDATA (char 128) -
                  USERDATA (char 128)
```

- *PROCESS* is the name assigned to the process definition object.
- *APPLICID* is the name of the application program that you want to trigger. In this example we are triggering a program called *QueueReader*. For a CICS application this would be the CICS transaction ID and for IMS it would be the IMS transaction code.
- *APPLTYPE* is the type of application to be triggered. Valid application types are:
  - CICS: representing a CICS transaction.
  - DOS: representing a DOS application.
  - IMS: representing an IMS transaction.
  - MVS: representing an OS/390 application (batch or TSO).
  - NOTESAGENT: representing a Lotus Notes agent.
  - NSK: representing a Tandem NSK application.
  - OS2: representing an OS/2 Warp application.
  - OS400: representing an OS/400 application.
  - Unix: representing a Unix application.
  - VMS: representing a Digital OpenVMS application.
  - WINDOWS: representing a Windows application.
  - WINDOWSNT: representing a Windows NT application.
  - DEF: this causes the default application type for the platform on which the command is interpreted to be stored in the process definition. This default cannot be changed by the installation. If the platform supports clients, this is interpreted as the default application type of the server.

– ENVDATA is a character string that contains environment information pertaining to the application to be started. The maximum length is 128 characters.

  The meaning of ENVDATA is determined by the trigger monitor application. The trigger monitor provided by MQSeries appends ENVDATA to the parameter list passed to the started application. The parameter list consists of the MQTMC2 structure, followed by one blank, followed by ENVDATA with trailing blanks removed.

Notes

1 On OS/390, ENVDATA is not used by the trigger monitor applications provided by MQSeries.

2 On Unix systems, ENVDATA can be set to the ampersand character (&) to make the started application run in the background.

USERDATA is a character string that contains user information pertaining to the application defined in the *APPLICID* that is to be started. The maximum length is 128 characters.

The meaning of USERDATA is determined by the trigger monitor application. The trigger monitor provided by MQSeries simply passes USERDATA to the started application as part of the parameter list. The parameter list consists of the MQTMC2 structure (containing USERDATA), followed by one blank, followed by ENVRDATA with trailing blanks removed.

The initiation queue

The initiation queue is simply a local queue onto which the queue manager places trigger messages. The initiation queue is associated with a particular target queue via the target queue's INITQ attribute.

The trigger monitor

The trigger monitor program has just one role, which is to monitor the initiation queue for trigger messages and start the application identified in the trigger message. Generally, administrative procedures are put in place that will start the trigger monitor automatically when the queue manager is started. A single trigger monitor can easily handle

more than one application; however, it is common practice to have separate trigger monitors for different environments or applications. MQSeries provides a number of sample trigger monitors that you can use ‘as is’, or you can use them as a guide to write your own.

The trigger message

When the trigger conditions are met, the queue manager reads information from the queue definition and the process definition pointed to by the PROCESS attribute of the queue. This information is placed into a structure variable called the MQTM structure. This structure then becomes the application data portion of the trigger message when it is placed onto the initiation queue. The MQTM structure is defined in the standard header files and copybooks.

The MQTM structure that is written to the initiation queue contains the following information:

- Queue attributes
  - queue name
  - trigger data
  - process name.
- Process attributes
  - ApplType
  - ApplID
  - EnvData
  - UserData.

The MQTM structure that is read from the trigger message by the trigger monitor is converted to a single string (called the MQTMC2 structure). The command line that is put out is then of the following form:

> *ApplID "TMC 2QName* (padded with spaces to 48 characters) *ProcessName* (padded with spaces to 48 characters) *TriggerData* (padded with spaces to 64 characters) *ApplID* (padded with spaces to 256 characters) *EnvData* (padded with spaces to 128 characters) *UserData* (padded with spaces to 128 characters) *QMgrName* (padded with spaces to 48 characters)*" EnvData*

Note that the *QMgrName* is not contained in the MQTM structure.

This is taken from the queue manager to which the trigger monitor is connected.

PUTTING IT ALL TOGETHER

Now that we have an understanding of the basic parts of triggering let's refer to Figure 1 to see how all the pieces fit together.

To begin the process, application 1 puts a message on Q1. Note that application 1 does not need to know about triggering and can MQPUT the message onto Q1 in the normal way.

Q1 has been set up to be a triggered queue. This is done by the administrator, by setting the queue's TRIGGER and TRIGTYPE attributes. When the trigger conditions of the queue are met, the queue manager automatically generates a new message called a trigger message and puts that message on INITQ (called the initiation queue). It is very important to remember that the trigger message is an entirely new and separate message. The original message containing the business data remains on Q1.

The trigger monitor is a long-running background process. As I mentioned earlier, the trigger monitor program's only function is to monitor the initiation queue for incoming trigger messages and then

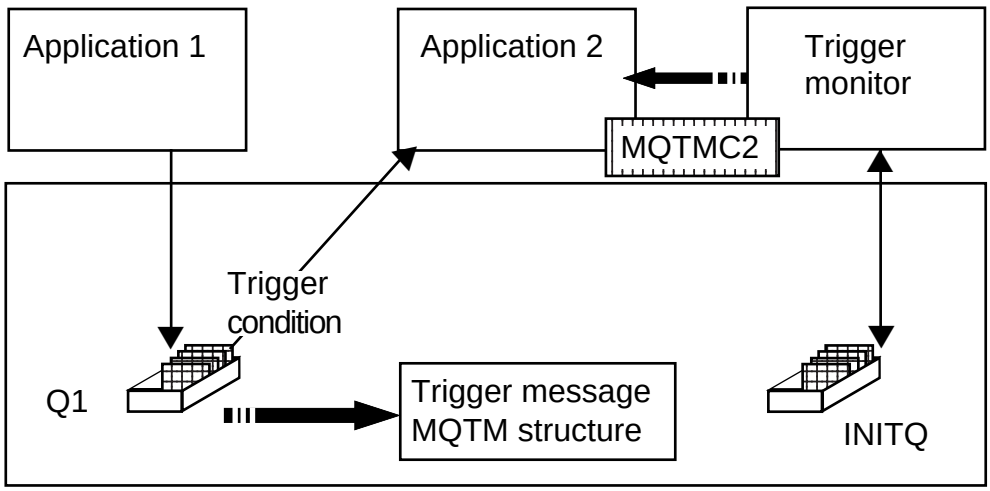

Figure 1: Trigger elements

start (or trigger) the intended application.

When the trigger monitor gets the trigger message it parses that message and puts out the command line needed to start the triggered application; in this case, application 2. In addition, the trigger monitor passes several other parameters contained in the trigger message to the started application.

Once the triggered application is started it can do whatever it wants, but this typically includes getting the original message from Q1.

TRIGGER CONDITIONS AND TRIGGER TYPES

When a triggered queue is first defined the trigger attributes of the queue are set. These attributes are:

- TRIGTYPE (first, depth, or every). This determines the trigger condition. If the TRIGTYPE is FIRST, the application will be started when the queue depth goes from 0 to 1. If TRIGTYPE (EVERY), a new instance of the application will be started for every new message that is put into the queue. If TRIGTYPE (DEPTH), the application will be started when the queue depth goes from X-1 to X. X is taken from the TRIGDPTH attribute.
- TRIGDPTH (X). This attribute is used only for TRIGTYPE (DEPTH). It indicates the depth at which triggering will occur.
- TRIGMPRI (0-9). Messages of this priority and higher count towards meeting the trigger conditions above.
- INITQ (Initiation Queue Name). This gives the name of the initiation queue onto which the queue manager will put the trigger message. It must be the same queue that is watched by the trigger monitor.
- PROCESS (Process Name). This is the name of the MQSeries process that defines how to start the application to be triggered. The queue manager reads this process definition and uses it to build the trigger message.

  The trigger message is a normal MQSeries message with a message descriptor and application data. The application data is a structure variable called the MQTM structure. The MQTM

structure is part of the MQSeries Trigger Monitor Interface (TMI), which is one of the MQSeries framework interfaces.

TRIGGER GENERATION

Providing the trigger conditions listed above are met, trigger messages will be generated as follows:

- When a message is put onto an application queue – this is normal operation.
- When an application queue is closed. By generating a trigger message when the application queue is closed the application does not have to worry about a message being put to the target queue between its last MQGET call and issuing an MQCLOSE call.
- When the initiation queue is opened for input. This ensures that any outstanding messages will be processed when the trigger monitor initially starts.
- When the application queue trigger attribute is changed. This allows an operator to temporarily suspend triggering of an application and subsequently restart it.

TRIGGERING AND LOGICAL UNITS OF WORK

By far the most common trigger type in use today is TRIGTYPE = FIRST. This will trigger on the current depth of a queue going from 0 to 1. Since uncommitted messages are counted in the MQSeries current depth attribute it follows that an application putting a message inside a unit of work will prevent any other application putting to the same queue from generating trigger messages. If the original application backs out its unit of work then there may still be other messages on the queue that need to be processed.

By always committing the trigger message we ensure that the worst that will happen is that a trigger message will be generated even when there are no messages on the queue. It is, therefore, necessary to code triggered applications so that they do not mind if they are started even when there are no messages for them to process.

TRIGGERING AND UNITS OF EXECUTION

An important consideration in triggering an application is to decide whether the application will run within or outside of the trigger monitor unit of execution. If the triggered application runs within the trigger monitor unit of execution then the MQCONN has already been carried out by the trigger monitor program. Issuing an MQCONN will result in the ALREADY CONNECTED return code and the current connection handle. Applications should be careful not to issue the MQDISC, which would result in the handle being terminated, causing the trigger monitor to abend.

Configuring a program to run outside of the trigger monitor unit of execution can be accomplished by the value in the *ApplicId* attribute of the associated process definition. For example, on Windows NT, the value ‘start program.exe’ could be used.

CODING TRIGGERED APPLICATIONS

The most important consideration in designing an application to be triggered is avoiding shutdown while there are still messages on the queue. If an application closes a queue while the depth is 1, for example, the depth will increase to 2 the next time that application is started. This will not meet the TRIGTYPE first condition. In fact, the depth can continue to increase forever and the application will never be started. In order to avoid this, the application should get messages from the queue until it receives reason code 2033 – MQRC_NO_MSG_AVAILABLE. In response to this reason code, it should close the queue and shut down so that it can be started again when a new message arrives.

Since starting an application creates considerable overhead, you want to avoid repeated startup and shutdown. Thus, the WaitInterval on the get should not be too short. Conversely, connected but idle applications still use system resources, so you don’t want the WaitInterval to be too long. Deciding on the proper balance point between these is an important decision for your application. The proper value will depend heavily on the purpose of the triggered application.

TRIGGERING MESSAGE CHANNELS

One of the first ways shops generally implement triggering is to use it to automatically start their message channels. This is a very good use of triggering indeed, especially in configurations where there is an inconsistent message flow throughout the day, rather sporadic bursts of messages, and periods of inactivity. Because of the special processing requirements of channels we use a special form of trigger monitor called a channel initiator. As far as the queue manager is concerned though, triggering a channel is no different from triggering an application. In the case of channel triggering, the triggered queue is the transmission queue.

As with all good triggered applications the channels allow the user to customize the time of inactivity before they end or disconnect. This is referred to as the Disconnect Interval. It allows you to customize the starting of channels based on the arrival rate of messages on the transmission queue.

Normally all channels are handled by a single channel initiator; however, on non-OS/390 platforms it is possible to have multiple channel initiators.

When a channel is manually stopped or is in RETRY mode it will set the transmission queue to NOTRIGGER and GET (DISABLED). This is done by MQSeries as a safety precaution to prevent unwanted starts.

SUMMING UP

There is a correct time and place for everything and MQSeries triggering is no exception. In this article I have tried to break down triggering into its basic parts. Once an understanding of the parts is gained then understanding the big picture of triggering is much easier. I hope this article has made you better equipped to tackle your next triggered MQSeries application.

Dale Eckert
Senior Consultant, MQSoftware (Canada)

Connecting tightly coupled legacy applications to loosely coupled applications: part 1

SCOPE

This is the first of two articles that explain how to connect two very different paradigms together: synchronous, representing tightly coupled environments, and asynchronous, representing loosely coupled environments. In the first article we look at the issues associated with using two-phase commit (2PC) protocols and soft locking, and in the second article we consider an overall end-to-end design. We will concentrate mainly on connecting a legacy application, as the initiator, to a new business application, as this is likely to be the most common scenario.

These articles will provide an overview; to address the subject in detail would require the production of a weighty design specification.

Any audit issues, whether relating to security or performance, are outside the scope of this article.

OVERVIEW

Within your own organizations many of you will have debated the relative merits of introducing loosely or closely coupled applications using 2PC protocols. In today's world, many enterprises are favouring loose coupling based on message queueing (MQ) protocols for their business applications and associated data repositories. This avoids such problems as:

- Time dependencies.
- System dependencies.
- Concurrent data locks.

However, there are many companies which still have their invaluable business data locked into tightly coupled legacy applications. Increasingly, companies are finding that there may be a need to make this information available to a newly-acquired message queueing-

enabled SAP system on an HP-UX platform, for example.

In this article we will consider the issues that have to be addressed and look at a solution to the problem of coupling together business applications that have been built to use the two different paradigms.

THE PROBLEM

Many people find they have to integrate back-end applications in order to support new business delivery channels and enhanced business processes, such as the supply chain.

What do you do when the IT budget is constrained and targeted mainly at new business opportunities, such as extending the delivery channels to provide Internet capability and better quality customer data (eg CRM)?

Having enhanced the IT architecture to encompass message queueing you will have also invested in an off-the-shelf integration broker to:

- Assist in linking the diverse business applications together.
- Reduce the cost of the development effort.
- Provide agility in responding to further business demands.
- Begin removing business functions, if appropriate.

You probably won't have the luxury of migrating all the legacy applications to the new architecture – and you may not want to. Still, they have to be integrated into the enterprise and in near real time, as current business models demand.

So, the reality is that you are wrestling with the problem of integrating legacy applications that currently use 2PC protocols with new business applications or packages that use MQ protocols. Also, existing functionality ported to the new message-based architecture may need to connect to the remaining legacy applications – applications continuing to use 2PC protocols.

Now you begin to appreciate that the legacy applications coordinate distributed updates between them using 2PC, and that MQ applications are not automatically part of this distributed unit of work.

It is at this point that you discover that the integration broker you purchased cannot assist you in this circumstance without the introduction of additional software.

So how do you synchronize the transactional integrity of 2PC legacy applications with 'new world' MQ applications when you can't integrate the legacy directly with the 'new world'? You need a gateway, but you need to be sure transactional integrity is maintained over both systems.

Inevitably there will be some change for the legacy applications, but you will strive to keep this to a minimum. No doubt there are maintenance teams in place only for the legacy applications and knowledge is probably very thin on how the application works, particularly with regard to communications.

TECHNOLOGY BACKGROUND

Before proceeding, it is worth taking time to look at the technologies that form the basis for the discussion within this article.

Peer-to-peer

Within the tightly coupled world, peer-to-peer communication is a programming model found in larger transaction programs where relatively large amounts of data or complicated interactions between separate transaction programs are involved. It follows an 'open, read/write, close' paradigm. There are two main models – SNA LU6.2 and OSI/TP – the former being by far the most common.

Connections are made to the remote peer using **ALLOCATE**, data is sent using **SEND**, and received using **RECEIVE**. As an alternative to **SEND** followed by **RECEIVE**, **CONVERSE** can be used. The application has to handle all communication errors.

The transaction manager/application will request either commitment of the local resources that it needs to update or a roll-back to their initial state. The transaction manager/application will also request, in parallel and via the peer-to-peer support, that the remote transaction manager/application updates or rolls back their own resources.

Under the covers, the peer-to-peer support will instigate a complex

protocol exchange (there are many weighty text books describing this in all its detail) to ensure that the commit or roll-back takes place in a controlled manner, thus ensuring the transaction manager/application is informed of the outcome.

Messaging queueing

For the purpose of this article we will assume that the *de facto* standard message queueing product, IBM's MQSeries, will provide this technology.

With message queueing technology the delivery is assured and time independent, and communications error handling is handled on behalf of the application. It provides a capability to send data using a minimum number of verbs: CONNECT, OPEN, PUT/GET, and CLOSE. The messages (ie the business data) will be written to queues residing on disk. Best practice is that the data is stored on a different disk – possibly using a different controller – from that used for the recovery logs.

It has been demonstrated (MQseries is now in widespread use in a number of industries, especially financial services) that the use of MQSeries guarantees that a message will not be lost once it has been placed on a queue by an application – this assumes that a persistent state has been selected.

Predicate locking

Predicate locking works by selecting a record from a database (eg at *timestampA*), changing it, but only updating the database with the changed value if the record has not subsequently been updated by another user since we took our copy at *timestampA*. The update operation is predicated on a data item in the originally-selected record. If the predicate is invalid (false) the update is discarded and the user is informed with a view to making them do the update again. It can be best thought of in terms of an artificial update count. The update is not made if the update count has been incremented in the interim between the selection and the update. At update time the update count is moved on.

With predicate locking, several users can simultaneously be in the process of updating a single record, but only one will be successful.

The other updates will be discarded and not serialized as in other database locking strategies. The use of predicate locking presumes that the likelihood of a lock conflict is small and it is, therefore, sometimes known as optimistic locking.

THE APPROACH

The direction of communications is important. In order for a tightly coupled application to communicate successfully with a loosely coupled application, a number of building blocks need to be in place:

- An intermediate gateway server.
- A transaction manager to act as a gateway and coordinate activities.
- An integration broker to provide transformation engine capabilities (the assumption is that it will be used for other integration needs, eg SAP access from the new business applications).
- Messaging technology that is XA-compliant.
- A communications protocol technology (eg LU6.2, OSI/TP) that supports syncpoint coordination.
- A standard message header.

The basic building blocks are shown in Figure 1.

The transaction manager gateway is key to enabling a successful connection of the two opposing worlds of synchronous and asynchronous. It provides management of the resources represented by the syncpoint protocol and messaging protocol.

The syncpoint protocol provides a conduit to the 'real' resource at the remote end of a communication link, whilst messaging is a resource in its own right that is directly manipulated.

The standard message header will be used to convey meta-data between the MQ applications, enabling them to ensure that they have a common view of the data being exchanged between the different systems – the content can be debated for a lifetime, so this article will assume it contains just what we need.

The transaction manager gateway (eg CICS, Tuxedo) will not be capable of managing the communication between the two systems

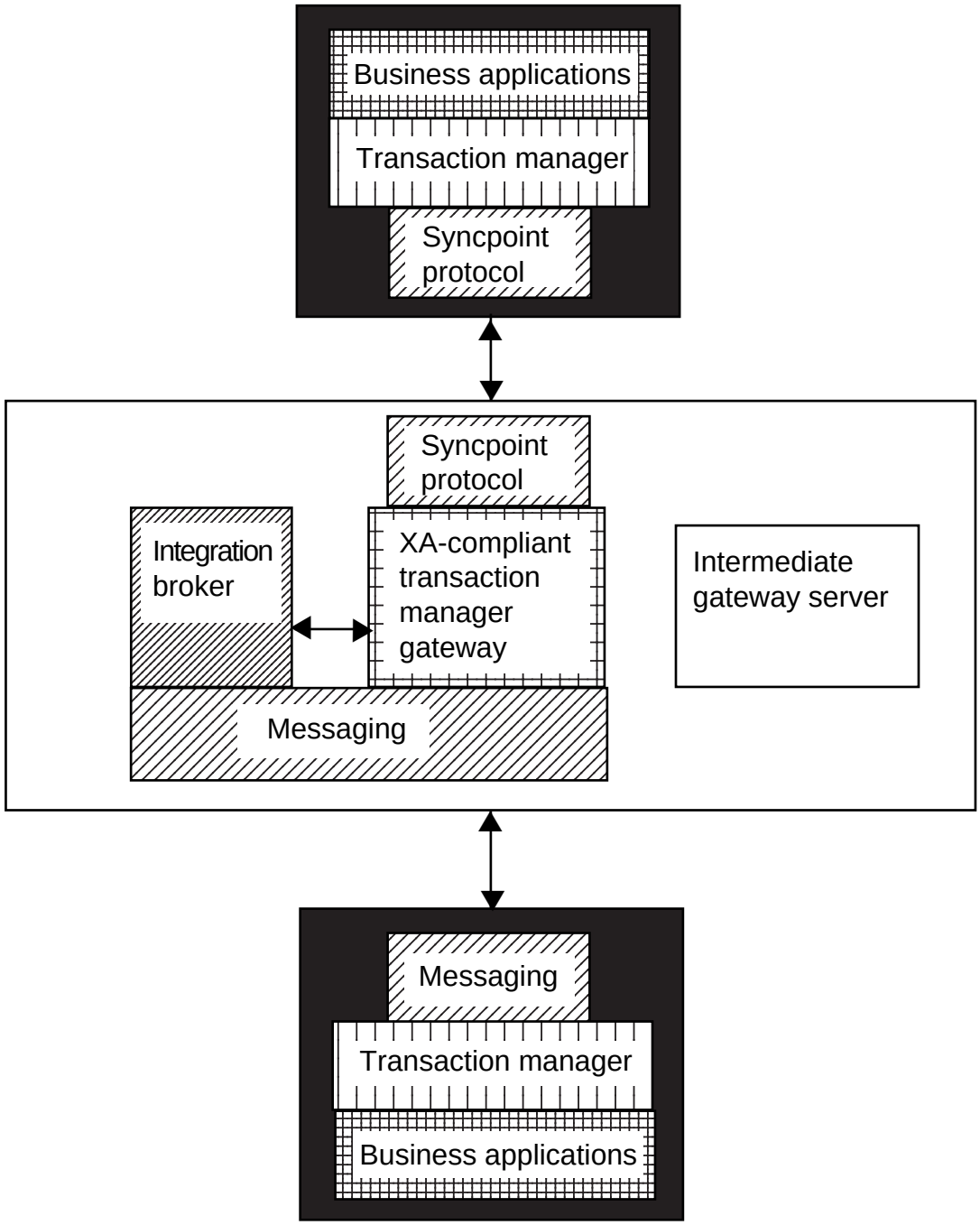

Figure 1: Basic building-blocks for integrating loosely and tightly coupled systems

without the development of some bespoke middleware infrastructure application, shown in Figure 2 as the gateway application.

Note

The term ‘transformation engine’ is used in preference to ‘integration broker’ (eg IBM’s WebSphere MQSI V2 Integration Broker), because this is the only feature of what is a multi-featured facility that is being used.

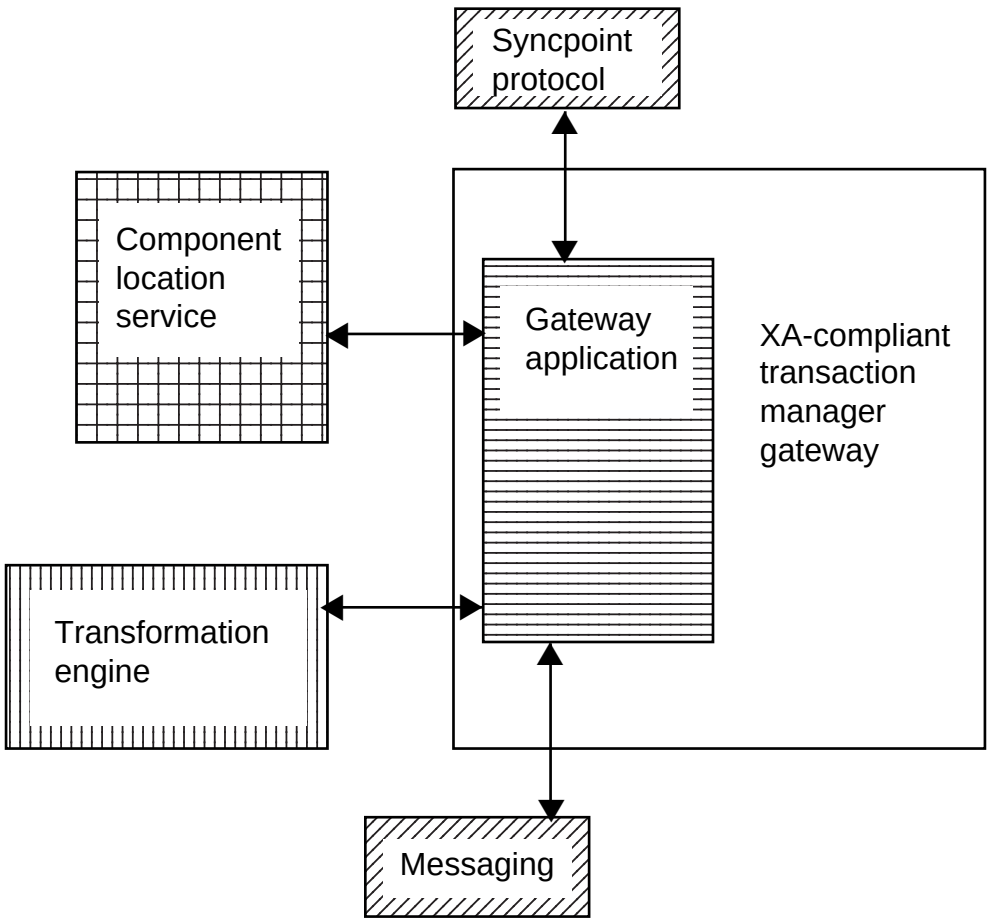

Figure 2: Bespoke middleware infrastructure gateway application

This application will respond to:

- Triggers emanating from messages arriving on a queue.
- A conversation allocation request.
- 'Create' or 'process standard message' headers.

It will process the message, requesting transformation (eg a minimum of EBCDIC to ASCII or *vice versa*).

A component location service is required, which enables mapping of the queue manager/queues to transactions/target systems.

The application will request services from the transaction manager to coordinate updates to resources.

SOFT LOCKING

Whilst 2PC and MQ can convey data between applications they cannot ensure the integrity of the data after it has been delivered. An additional mechanism is required to ensure that the data will not be overwritten by a local application, for example, when a remote conversational application is driving the update.

This section addresses soft locking from the viewpoint of the MQ-driven application, although some of the issues will apply to the 2PC-driven legacy application as well – for the latter they are assumed to have already been addressed.

Assumptions are that:

- Concurrent attempts to update the same data will not be a common occurrence.
- The locks are not going to be contested, which would otherwise impede overall performance – an analysis of data accesses will need to be undertaken to confirm this.

Based on predicate locking, the locking techniques discussed in this article need to ensure that no two applications will perform a concurrent update.

Basic soft locking is primarily used for enquiries to ensure a record is locked whilst a business 'decision' is enacted on the data, which may

then result in an update. The state of the soft lock may be reflected back to the legacy application or to the gateway application.

An extension of soft locking is ‘firm’ locking; it provides the means to implement a ‘hardened’ form of this technique, which prevents enquiries on the data as well as other attempts at updates.

A combination of soft and firm locking can be used if enquiries precede the update phase of a multi-conversation interaction.

The soft and firm locks are freestanding tables in the database that indicate whether a lock is set. The latter will consist of a number of attributes, such as *transaction-ID* (ie that has been passed on from the gateway application), *User-ID*, timestamp, etc.

The type of locking required will depend upon the business requirements.

Any built-in timeouts need to account for the delays associated with a network-related update.

When a conversational application requests multiple transactions pertaining to the same business operation to be driven in a messaging environment, there is a need to lock the data from the first message-based transaction until the final message-based transaction has been completed, as shown in Figure 3.

The key to enforcing this approach is to provide a shared library in the form of a soft lock manager that application developers will use to access the data rather than issue their own direct SQL. The option to use the database locking facilities is unlikely to provide the level of control required in these situations.

CONNECTING 2PC TO MQ

There is an initial phase, which we will term the ‘pseudo update’, where the first thing that happens is the the soft lock is updated, as it may need to be reflected back to the 2PC application (this may not be necessary in all situations).

Then a firm lock record is created in the firm lock table, which is kept for the duration of the entire business transaction. Once this lock is in place no other transaction will be permitted.

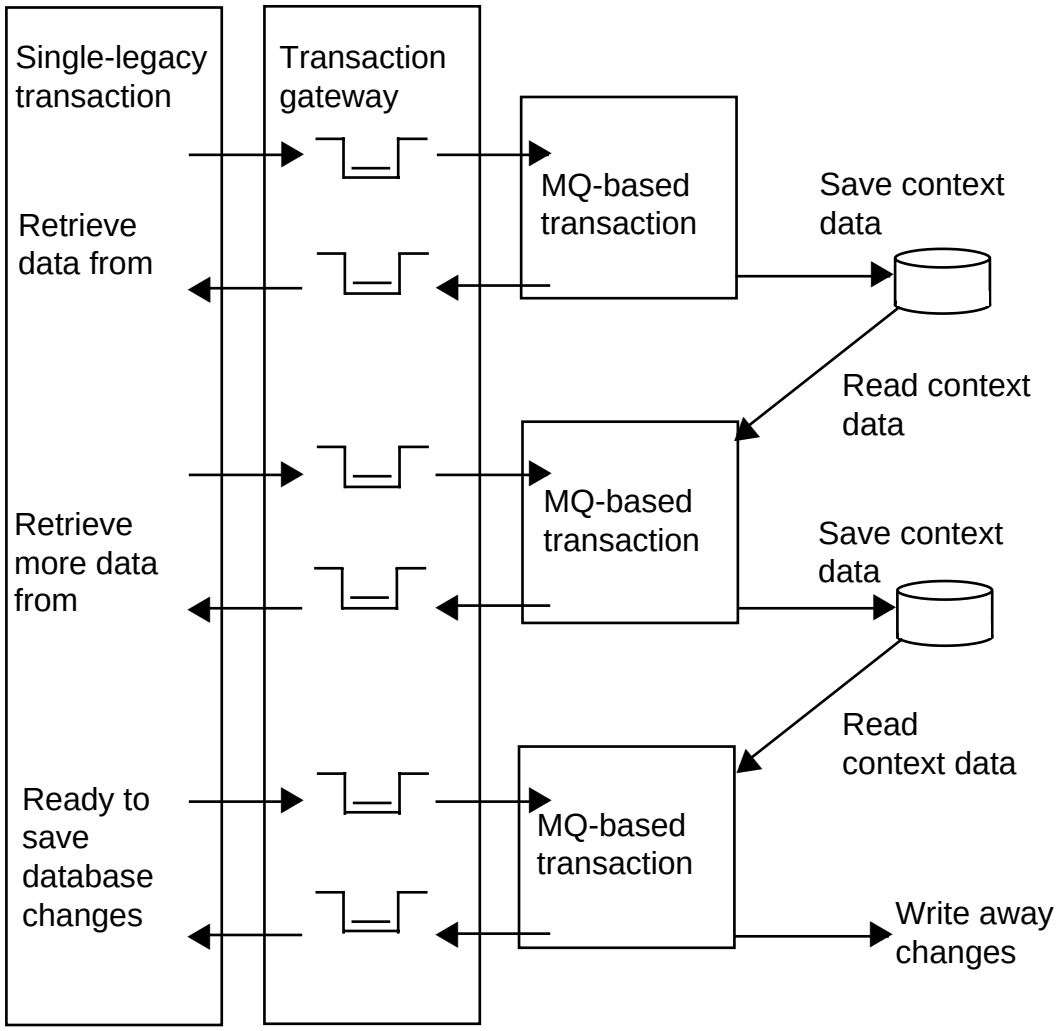

Figure 3: Soft locking

The MQ application issues a sync point (begin) and performs the update, then rolls back all the updates to the sync point. If any errors occur, the lock is deleted and the record freed. If things have gone well, the message is sent back to the 2PC legacy application and a two-phase commit takes place – updating the 2PC legacy resources and gateway application resources.

In the second phase, the gateway application indicates to the MQ application that the legacy two-phase commit was successful –

otherwise a failure is signalled to the MQ application, no update action takes place and the firm lock is removed. On receiving the success signal the MQ application again validates the firm lock table, and this time a row is expected to be retrieved from the database. If it is not (or if the lock *TRANID* does not match) an error is reported. Otherwise, the update takes place, data consistency is now directly controlled using the database management system's internal locking facilities, and, once the record is written successfully, the MQ application deletes the firm lock.

SUMMARY

We now have an overview of the two-phase commit process and the associated soft locking that must accompany it.

The scene has been set for a discussion in next month's *MQ Update* of an overall end-to-end design solution for connecting together synchronous and asynchronous worlds.

We will also look at an alternative failure recovery strategy, the role of a scavenger application, and exception handling. We will also consider connecting an MQ-based application, as the initiator, to a legacy application.

Mick Broderick
Solutions Architect (UK)

MQSeries Workflow V3.3 with MQSeries and MQSI – an introduction

MQSeries Workflow has supported a direct interface with MQSeries since version 3.2.1. The latest release – 3.3.0 – introduces new functionality for more advanced integration and automation. This new functionality makes the interface more robust and easier to use.

There are two main categories of MQSeries/XML messages understood by MQSeries Workflow:

- Starting/executing a new process instance from an XML message.

- Using XML messaging to implement an MQSeries Workflow process activity.

The first is a way for any application/program to start a new workflow process, for example, a call centre system may have the capability to start workflow processes to deal with calls that cannot be dealt with immediately.

This article concentrates on the second category, the interface between MQSeries Workflow and base MQSeries. This allows an MQSeries-enabled application (or MQSeries Integrator) to be invoked as part of a workflow process.

The straightforward interface to implement a process activity is explained first, and then the advanced functions are introduced. It is important to understand all of these for complete integration between MQSeries Workflow and MQSeries/MQSeries Integrator.

This article does not attempt to describe the background and business drivers behind Workflow, or what Workflow technology is all about.

It is also worth noting that MQSeries Workflow itself makes extensive use of MQSeries internally. This article concentrates on the standard MQSeries Workflow interface functionality available to the developer and should not be confused with the internal use of MQSeries by workflow. For the purposes of this article, consider the internal workings of MQSeries Workflow to be a 'black box'.

IMPLEMENTING A WORKFLOW ACTIVITY VIA MQ AND XML

First impressions indicate that there is nothing too complicated in making a process activity write an XML message onto an MQSeries queue. Indeed, there are few problems in the development environment, with super-user or administrator access rights, and the capability to define queues and deploy and run workflows, etc. However, it quickly becomes apparent that careful design is needed up-front before attempting release into a production environment.

Simplistically, a workflow process activity can be implemented as a User Program Execution Server (UPES), the properties of which define the XML message format and the destination MQ queue manager/queue. The MQ queue can be monitored by an application that may respond to a ReplyTo queue.

The following sections describe the workflow definitions for the activity and the UPES.

Activity definition

From the MQSeries Workflow Buildtime environment, when defining a process, the Execution tab in an activity properties window defines how the activity is to be implemented. Writing to an MQSeries queue is performed at the server level, so a UPES is used. The UPES is effectively the vehicle for placing the message onto a queue (see Figure 1).

Synchronous versus asynchronous

Note that the invocation mode of the UPES can be either synchronous or asynchronous:

- Simplistically, 'synchronous' places the message on the queue and workflow puts the activity into a 'running' state until a completion XML message is received.
- 'Asynchronous' places the message on the queue and puts the activity into a 'finished' state. No completion XML message is expected and if one is received it is ignored. Note that this mode does not allow any update of the container data.

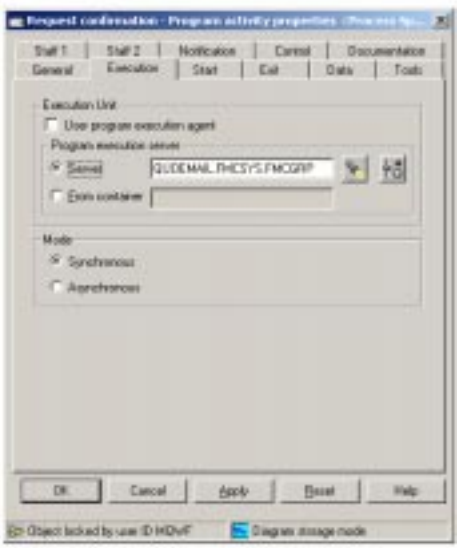

Figure 1: Activity definition

For a UPES, one would normally expect the start (as defined in the Start tab of the activity properties) to be 'Automatic'. Otherwise a user has to manually start the activity.

For the activity to be finished, the exit conditions must be met (as defined in the Exit tab). Again, for a UPES, this would normally be 'Automatic' with no special conditions. Also important in the activity properties are the General and Data tabs (see Figures 2 and 3).

In the General tab, a logical program name is defined that ultimately identifies a program associated with the activity. When used with XML messaging it can be used or ignored, depending on the interface implementation design. See below for more discussion on the program definition.

The Data tab defines the data structures that are passed to the activity and ones that are passed onwards from it. These subsequently form part of the XML message. See below for more discussion of the data definitions.

In the example above, we have used a UPES called

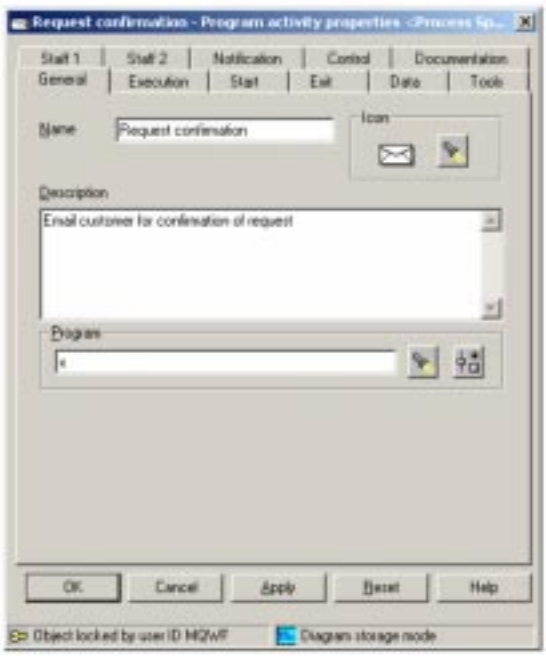

Figure 2: General tab

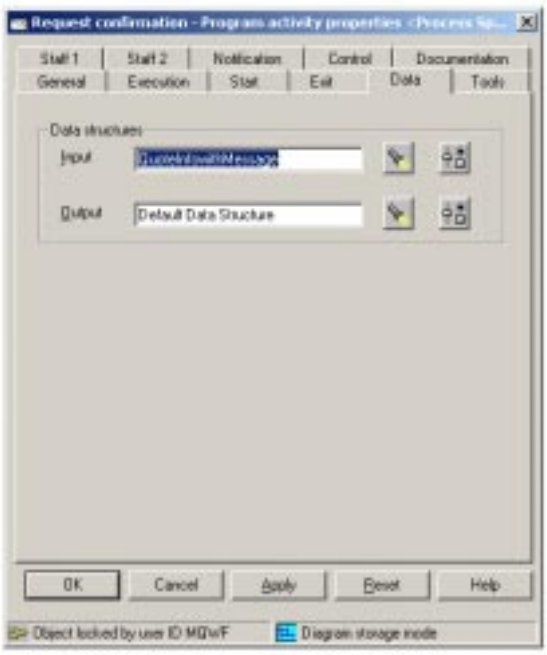

Figure 3: Data tab

QUOEMAIL.FMCSYS.FMCGRP. This has to be defined before we can use it.

UPES definition

A UPES is defined in the 'Network' area of Buildtime, for a specific Workflow system. The system effectively defines the server or 'node' where MQSeries Workflow will attempt to write the message (see Figure 4).

The name of the UPES is only a logical name, but it is limited to eight mixed-case alphanumeric characters; no special characters are allowed. The fully qualified name of the UPES is this logical name with a suffix of the Workflow System name and System Group name (FMCSYS and FMCGRP in this example).

Note the UPES version number – this will be discussed later.

The message queueing tab in the UPES properties window is where the MQSeries queue and queue manager names are defined. This is also where the XML message format is set (see Figure 5).

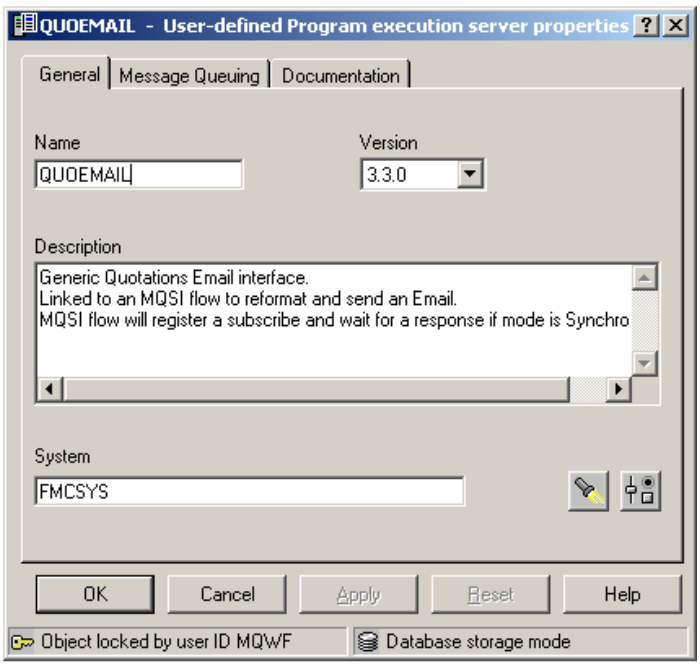

Figure 4: UPES definition

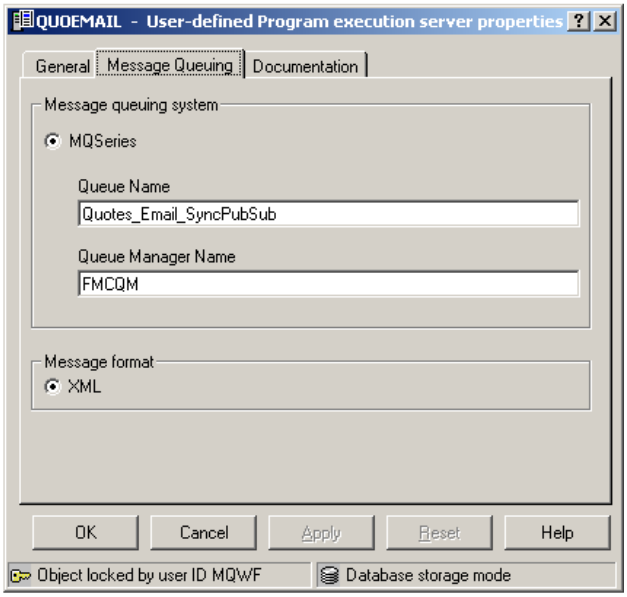

Figure 5: Message queueing tab

Remember that these are just definitions that form part of a process model. MQSeries Workflow does not create the queue/queue manager for you, and will not try to use the queue until an instance of the process is running and the activity is reached and started in the runtime environment. As with any MQSeries application, a careful release process is needed to ensure that the queue is present as and when expected.

Outgoing message

Previous versions of MQSeries Workflow (V3.2.1 and V3.2.2) supported a single activity message:

- 'ActivityImplInvoke'. This is the message that signifies that the activity has been started. A response is required if the mode is defined as synchronous.

The latest version – 3.3.0 – introduces the following new messages:

- 'ActivityExpired' if the activity has been started and the expiry time reached. No response is expected.
- 'TerminateProgram' if the activity has been terminated. No response is expected.

To maintain compatibility with MQSeries-enabled applications that have been coded for previous releases, a version number can be defined for each UPES:

- Setting the version number to 3.2.2 indicates that the application only understands the ActivityImplInvoke message. MQSeries Workflow will only send this message for an activity implementation.
- 3.3.0 as the version number indicates that the ActivityExpired and TerminateProgram messages are also understood. MQSeries Workflow will send these messages if necessary.

Once the message has been placed on the queue it is the responsibility of the application to get the message from the queue and respond if required. Standard MQSeries transactional features can be used and it is recommended that you do so.

A message placed onto a user-defined queue via a UPES has the following general sections:

- The MQSeries message descriptor (MQMD).
- An MQ Workflow XML message header.
- An MQ Workflow message name.
- Message specific parameters, including activity-specific data.

Certain fields in the MQSeries message descriptor (MQMD) are used. The following are key:

- CorrelID – XML requests sent by MQ Workflow contain a *correlation-ID* supplied by workflow. Responses sent back by an application should return the same *correlation-ID*.
- ReplyToQ/ReplyToQMgr – specifies the queue and queue manager the response should be sent to.
- UserIdentifier – the user that started/terminated the activity or, for the response message, the user that MQSeries Workflow should use to check authorization.
- Format – MQSTR for workflow XML messages.
- Persistence – XML requests sent by MQ Workflow are persistent

and responses sent by invoked activity implementations should also be persistent.

An example of the ActivityImplInvoke workflow message is:

```
<?xml version="1.Ø" encoding="UTF-8" standalone="yes"?>
<!-- This document is generated by a MQSeries Workflow Version 3.3.Ø
server -->
<WfMessage>
  <WfMessageHeader>
    <ResponseRequired>Yes</ResponseRequired>
  </WfMessageHeader>
  <ActivityImplInvoke>
    <ActImplCorrelID>RUEAAAABAC4AAAAAAAAAA … AAAAAAAABF</
ActImplCorrelID>
    <Starter>MQWF</Starter>
    <ProgramID>
      <ProcTemplID>AAAAA … AAAAA</ProcTemplID>
      <ProgramName>x</ProgramName>
    </ProgramID>
    <ImplementationData>
      <ImplementationPlatform>WindowsNT</ImplementationPlatform>
      <ProgramParameters></ProgramParameters>
      <ExeOptions>
        <PathAndFileName>x.exe</PathAndFileName>
        <InheritEnvironment>true</InheritEnvironment>
        <StartInForeGround>true</StartInForeGround>
        <WindowStyle>Visible</WindowStyle>
      </ExeOptions>
    </ImplementationData>
    <ProgramInputData>
      <_ACTIVITY>Request confirmation</_ACTIVITY>
      <_PROCESS>SpecialQuote</_PROCESS>
      <_PROCESS_MODEL>SpecialQuote</_PROCESS_MODEL>
      <QuoteInfowithMessage>
        <QuoteInfo>
          <QuoteKey>
            <CustomerID>123</CustomerID>
            <CustomerName>NickWhittle</CustomerName>
            <CustomerEmail>nick@mqsolutions.co.uk</CustomerEmail>
            <QuoteID>ABC</QuoteID>
          </QuoteKey>
          <Status>Confirm</Status>
        </QuoteInfo>
        <Message>Customised message text </Message>
      </QuoteInfowithMessage>
    </ProgramInputData>
    <ProgramOutputDataDefaults>
      <_ACTIVITY>Request confirmation</_ACTIVITY>
      <_PROCESS>SpecialQuote</_PROCESS>
```

```
      <_PROCESS_MODEL>SpecialQuote</_PROCESS_MODEL>
      <DefaultDataStructure>
      </DefaultDataStructure>
    </ProgramOutputDataDefaults>
  </ActivityImplInvoke>
</WfMessage>
```

Items of interest are:

- The ResponseRequired field, which shows that a response is expected – ie the execution mode is synchronous.
- The *Correlation-ID*, which is repeated from the MQMD, as is the activity starter *User-ID*.
- The ProgramID and ImplementationData. Traditionally, a UPES is expected to execute the program detailed. However, with XML messaging and workflow definitions pointing to a specific queue, this is not really needed. In fact there are many advantages to not having program-specific information in the workflow. Hence, the above example has dummy information for these properties. A more common scenario is for the queue to be monitored by MQSeries Integrator, and for that to perform format translations and workflow correlation.
- After the ACTIVITY, PROCESS, and PROCESS_MODEL information supplied by workflow, the ProgramInputData shows the workflow input container data structure.
- The ProgramOutputDataDefaults show the initial values set in the output container. The above example does not include any output container data fields.
- It is up to the application how to handle the ActivityExpired and TerminateProgram events. The Correlation ID is key information conveyed in these messages.

Response message

A response message is only needed if the synchronous mode has been selected. The response must be a well-formed workflow XML message, and must be placed into the workflow XML input queue, as specified in the MQMD.ReplyToQ and MQMD.ReplyToQMgr fields.

The response may report successful execution and pass a return code and output container data, or a failure, and pass an error code and reason code. Workflow will read and process this response and change the state of the activity accordingly.

The format of the standard response message will be something along these lines:

```
<WfMessage>
  <WfMessageHeader>
    <ResponseRequired>No</ResponseRequired>
  </WfMessageHeader>
  <ActivityImplInvokeResponse>
    <ActImplCorrelID>RUEAAAABAC4AAAAAAAAAA … AAAAAAAABF</
ActImplCorrelID>
    <ProgramRC>0</ProgramRC>
    <ProgramOutputData>
      <DefaultDataStructure>
      </DefaultDataStructure>
    </ProgramOutputData>
  </ActivityImplInvokeResponse>
</WfMessage>
```

If a synchronous activity is expired or terminated by workflow the response is no longer expected.

Input and output data

Customized workflow input and output data structures are defined in the process activity Data tab. Although MQSeries Workflow supports different data types for fields in data structures, the XML message is in Unicode (UTF-8) and all fields will be output/input as text. A Workflow data member name is represented as an XML element name, but the data member type does not appear.

Nested data structures and arrays are supported by MQSeries Workflow and will appear appropriately in XML: nested data structures as sub-structures rather than ‘do notation’; array elements sequentially, without any array index/number and as repeating fields.

For example:

```
<Customer>
  <Name>…</Name>
  <Address>
    <AddressLine>…</AddressLine>
```

```
    <AddressLine>…</AddressLine>
    <AddressLine>…</AddressLine>
    <PostalTown>…</PostalTown>
    <Postcode>…</Postcode>
  </Address>
</Customer>
```

Code page issues

MQSeries Workflow uses XML4C for parsing XML messages. This conforms to the XML 1.0 specification and uses ICU (International Components for Unicode) for code page conversion, supporting over 100 different code pages.

XML messages sent to MQSeries Workflow are converted from their format (as specified in the MQMD.CodedCharSetId and MQMD.Encoding fields) to the appropriate code page associated with the workflow platform.

Programs

Even though a program is not necessarily used by whatever is monitoring the MQ queue, it still has to be defined to Workflow.

If the target is not going to use the program information, then only the 'Program can run unattended' setting and data structures are important. The data structures are already defined in the activity properties so it is often easiest to specify that the program can handle any data structures.

MAKING SENSE OF THE MESSAGE

As can be seen from the ActivityImplInvoke message, there is a lot of information available for an application to use. Some of this is needed for synchronous replies back to Workflow, but most is unlikely to be understood by the application (unless it is specifically coded for MQSeries Workflow). Often an application will only be concerned with the input and output data, but possibly not in the same form as is passed from/to MQSeries Workflow.

Hence, MQSeries Integrator (MQSI) is an ideal product to monitor activity interface queues and pre-process messages before passing them on to an application. Sometimes, the functionality within MQSI

is sufficient to implement the activity (eg simple database transactions), but more often a line-of-business (LoB) application is the target and must be used.

When MQSI is used to integrate LoB applications the context data needed for replying to workflow has to be stored. If the application doesn't support context-specific data being passed in the transaction (most don't), then either an adapter and synchronous calling method have to be used, or MQSI has to store the context information in a temporary database table. Part of the interface design is then tying up the reply application-specific message with the stored context data. Usually, an application is using some of the context data as a key, so this could be used. For example, a customer information lookup function may pass and return a customer ID as the key.

USING MQSI PUB/SUB

Processes in MQSeries Workflow are often long-running, perhaps taking days, weeks, or even months to complete. So far, we have talked about fairly straightforward interfaces and MQSI message flows – the customer database lookup is likely to take a relatively short time.

The same MQ/XML interface method can, however, be used to support a synchronous call that takes a long time to return. An example might be a function that sends an e-mail and waits for a reply. Rather than hard-code the e-mail send and reply function you'd expect several steps to both construct/send the e-mail and identify the reply. This would often involve using common e-mail composition and sending modules and a generic incoming e-mail handler. A good solution is to use MQSI publish and subscribe facilities.

It makes sense for an incoming mail handler to publish indexing information for applications that may be interested. If so, then after sending the initial e-mail the MQSI message flow simply has to subscribe to wait for an appropriate reply, in addition to storing the workflow context-specific data (see Figure 6).

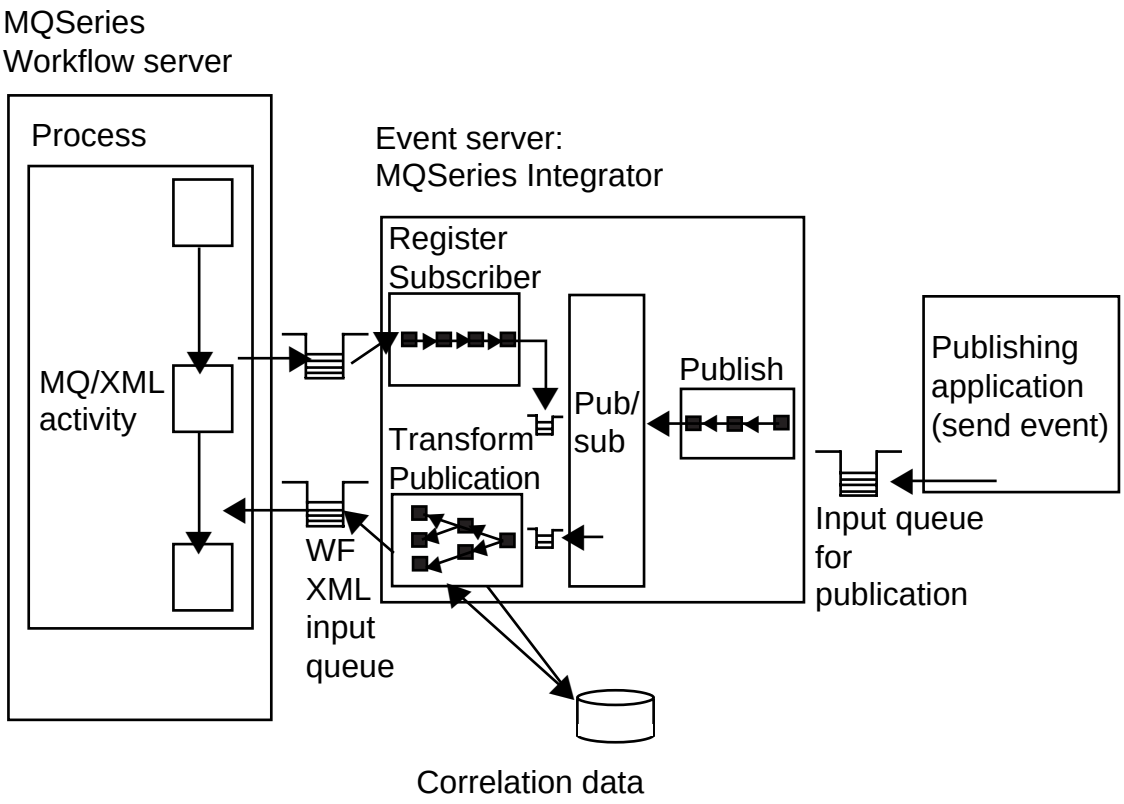

Figure 6: Workflow context-specific data

ADVANCED WORKFLOW, MQSI, PUB/SUB, AND APPLICATION CONSIDERATIONS

The ActivityExpired and TerminateProgram events clearly need special processing from an application or MQSI perspective. If publish/subscribe is being used, the subscribe registration needs to be removed, as does any workflow context-specific data from any database. Furthermore, if any data is being used by an application to make changes, then the design has to consider how these can be reversed. Remember that 'changes' may be more than just database updates – what if some external correspondence has been dispatched? Without getting into a business and technical architecture discussion, the answer will probably involve catering for the situation from both application and process design aspects. However, there are different approaches and strategies available depending upon specific business requirements.

Expiry times

Although this article is not intended to address process design issues there are some aspects that are relevant. Any workflow process activity that is implemented using XML messages in a synchronous execution mode has to protect itself from the possibility of not receiving a response in a timely manner.

Good process design should consider specifying notification times for processes and certain activities. Notifications, however, are really aimed at highlighting to a senior person that something is stuck. This is not always an appropriate mechanism for dealing with 'business as usual' situations.

Following the e-mail example, a realistic business requirement would be to send a chasing e-mail if no response is received within a certain time. Instead of using notifications or writing custom code that examines the audit log to detect activities that need chasing, or designing an over-complicated process to deal with chases, a new feature in Workflow V3.3 can be used.

An 'expiry time' can now be specified at the activity level. This does not cause a notification; it is simply a different finishing state for activities. This state information is available as part of the activity output container data so it can be used in a transition condition, ie the workflow process can take appropriate action if an activity has expired. I would strongly recommend that expiry times are placed on all synchronous activities implemented by the MQ interface.

Brokers and the MQ interface

Instead of approaching the interface from an application functional perspective, using an individual queue for each application/function required by the process design, one could consider the interface from the workflow perspective.

A workflow process often uses the same or similar data containers throughout. These have meaning to the process and usually have a few key fields. The key fields may be understood and acted on by a common module monitoring a single MQ queue that is used for all activities implemented by the MQ interface in that process. This module may then act as a broker and pass on a distilled message to

specific applications.

This, hopefully, gives an insight into the capability of the MQSeries Workflow XML/MQ interface and some of the options available in the design, together with a few of the issues to consider. Perhaps the most important point to remember is that we are talking about an interface between a business process and information systems. Therefore, the people considering interface design solutions should understand both business and technical aspects. Ideally, they will also have solid experience of workflow design and implementation, so have met the issues before.

CONCLUSION

The integration between MQSeries Workflow and MQSeries/MQSI for activity implementations can be described as being 'loose'. The MQSeries Workflow activity does not know anything about queue specifics (other than a queue and queue manager name), so the two are not closely linked or dependent on one another. This allows the full capabilities of MQSeries to be utilized without constraint. The use of MQSeries and MQSI gives great flexibility and support for applications on many platforms.

The transactional support and additional messages in V3.3 enable a robust interface to be implemented with a minimum of fuss, whilst ensuring business process and application data integrity. This is important when considering workflow as a critical business system.

For more information and a quick-start, examine the IBM SupportPacs and Redbooks for MQSeries Workflow. The following SupportPac contains a simple integration example:

- *WA06 – Event Server sample using MQSeries Integrator*.

Always refer to the *Programmers' Guide* for a definitive reference specific to the level of the product.

Nick Whittle
MQSolutions (UK)

iWay Software, an Information Builders company, has announced an expanded Plug-in Suite for WebSphere MQ Integrator that eliminates custom programming while enabling the simplified integration of: EDI e-business exchanges such as CommerceOne and Covisint; HIPAA-compliant application systems; packaged application systems such as SAP, PeopleSoft, Siebel Systems, and Oracle Applications; existing transaction systems such as CICS, IMS, and Tuxedo; programs written in languages such as COBOL and RPG; and diverse legacy information assets.

The company claims that the elimination of custom programming over such a wide range of information assets will dramatically reduce the cost, time, and effort of implementing integration solutions with IBM application integration middleware products.

The iWay v5.1 Plug-in Suite for WebSphere MQ Integrator supplies a set of flow design tools directly within the WebSphere MQ Integrator Control Center. These tools allow WebSphere MQ Integrator users to incorporate widely diverse information resources into message flows, hiding the proprietary interfaces and message formats they use.

The iWay Plug-in Suite is compatible with existing WebSphere MQ Integrator processing nodes, but also allows WebSphere MQ Integrator to access iWay Intelligent Adapters. It is availabile on Windows NT/2000, AIX, Sun Solaris, and HP/UX.

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

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

* * *