15 MQ update

September 2000

In this issue

MQ Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: +44 1635 550955
e-mail: harryl@xephon.com

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: +1 303 410 9344
Fax: +1 303 438 0290

Contributions

Articles published in *MQ Update* are paid for at the rate of £170 ($250) per 1000 words and £90 ($140) per 100 lines of code. For more information about contributing an article, please check Xephon's Web site, where you can download *Notes for Contributors*.

MQ Update on-line

Code from *MQ Update* is available from Xephon's Web site at www.xephon.com/mqupdate.html (you'll need the user-id shown on your address label to access it). If you've a problem with your user-id or password call Xephon's subscription department on +44 1635 33886.

Editor

Harry Lewis

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 ($33.50) each including postage.

Backing up MQSeries messages

The need for a program to back up all messages on a queue manager arose when we migrated from MQSeries for Windows NT version 5.0 to 5.1. We decided to uninstall version 5.0 and then carry out a new installation of version 5.1, rather than perform a normal upgrade by simply running **Setup.exe**. There were several reasons for this:

- Our initial attempts at upgrading resulted in problems when we tried to start channels. We found similar problems documented in the PTFs for the product. However, a new installation also solved the problem.
- An upgrade did not give us a chance to look at some of the new features of version 5.1 such as 'Default Configuration' and 'API Exerciser'.
- The new installation directory (*Program Files\MQSeries*) was preferred to the old one (*Mqm*).

Because we were doing a new installation, we needed to redefine all our MQSeries objects and restore all persistent messages. Redefining queue definitions and other objects was simple. SupportPac MS03 provides a program (**Saveqmgr.exe**) that creates a file with all object definitions for a queue manager. This file can then be used as the input of RUNMQSC. Restoring messages was, however, not so simple. Candle's free PQEdit utility is useful for backing up and restoring a few different queues. However, each queue has to be selected and backed up manually. Another drawback of PQEdit is that it replaces the context information in the message descriptor with its own values.

So I wrote a program to find all messages on a queue manager and back them up to a single queue. The program does not check for non-persistent messages, though in our case this was not a problem, as all non-persistent messages are deleted by stopping and restarting the queue manager prior to running the program. The program saves each message descriptor and the queue name from which the message originated in a transmission queue header (XQH) that precedes the actual message. Adopting this procedure means there is no need to

write a corresponding restore program. If the messages are backed up to a queue on another queue manager, then restoring them is simply a matter of moving them to the transmission queue of the original queue manager. MQSeries then puts them back on their original queues with their original message descriptors intact. Moving the messages can be done with the *Forward* function of PQEdit or with the utility MQNQMTOQ that comes with SupportPac MA7A.

The program works fine if the queue manager from which the messages are copied and the one to which they are copied are on the same platform. However, if conversion is required between platforms, then the transmission queue header is problematic. My plan was to back up messages to a queue manager on OS/390. I thought that, if the channel definition specified *CONVERT(YES)*, then the transmission queue header would arrive on the OS/390 queue nicely converted, given that *MQFMT_XMIT_Q_HEADER* is a built-in format. Not so. The built-in conversion routines cater for the needs of message channel agents, which don't want the transmission queue header converted when they perform an *MQGET* with the *MQGMO_CONVERT* option set. This is explained in *Appendix D – Data-conversion* of the *MQSeries Application Programming Reference*. So my messages ended up on the dead letter queue with a reason code of either *MQRC_FORMAT_ERROR* or *MQFB_XMIT_Q_MSG_ERROR*.

The solution was to write a conversion exit called MQXMIT. This is called by the queue manager whenever the built-in conversion routine fails to convert the data. Instructions for writing data conversion exits can be found in the *MQSeries Application Programming Guide*. Once this exit is written and copied to the MQSeries *EXITS* directory, the *MsgBack* program could be used with an output queue on OS/390. I opted not to convert the remainder of the message following the transmission queue header as this part of the message would just need to be converted back when messages are restored. Also, the utility programs we have on MQSeries for OS/390 can convert messages when they retrieve them by using the *MQGMO_CONVERT* option, should we want to view the contents of these messages while they're in an OS/390 queue.

Below is the program that backs up all messages to a single queue. Note the use of the continuation character, '➤', in the code below to indicate a formatting line break that's not present in either the original source code or the code that can be downloaded from Xephon's Web site (*http://www.xephon.com/mqupdate.html*).

MSGBACK

```
// Msgback: This program finds all messages on a queue manager
//          and backs them up to a single queue. Dynamic queues
//          and queue names starting with "SYSTEM." are ignored.
//
//          Each backed-up message is preceded by an MQSeries
//          transmission queue header that contains enough
//          information to restore the message and its message
//          descriptor to its original state.
//
//          If the messages are backed up to a queue on a different
//          queue manager, they may be restored by simply moving
//          them to the transmission queue of the original queue
//          manager.
//
//          This program is run from the command line with three
//          positional parameters:
//          - The queue manager that owns the queue for the messages.
//            This can be the local queue manager, but is usually a
//            remote queue manager.
//          - The queue to which the messages are copied.
//          - The queue manager to which to connect (optional).
//
/* Include standard libraries */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Include MQSeries headers */
#include <cmqc.h>
#include <cmqcfc.h>
#include <cmqxc.h>

void ProcessReplyMsgs ( MQHCONN hConn,
                        MQHOBJ hReplyQ,
                        PMQCHAR48 pReplyToQ,
                        PMQCHAR48 pLocalQMgr,
                        PMQCHAR48 pRemoteQMgr,
                        PMQCHAR48 pRemoteQName );

void ProcessApplMsgs ( MQHCONN hConn,
```

```
                        MQHOBJ hBackupQ,
                        PMQOD pObjDescBack,
                        MQHOBJ hApplQ,
                        PMQOD pObjDescAppl);

void MQError ( MQCHAR8 Function,
               PMQCHAR48 pObject,
               MQLONG CompCode,
               MQLONG Reason );

PMQCHAR48 Trim ( PMQCHAR48 pObject );

int main( int argc, char *argv[] )
{
   MQCHAR48     LocalQMgr;
   MQCHAR48     ReplyToQ;

   MQHCONN      hConn;
   MQLONG       CompCode;
   MQLONG       Reason;

   MQHOBJ       hAdminQ;
   MQHOBJ       hReplyQ;
   MQOD         ObjDescAdmin = {MQOD_DEFAULT};
   MQOD         ObjDescReply = {MQOD_DEFAULT};
   MQLONG       OpenOpts;
   MQMD         MsgDesc = {MQMD_DEFAULT};
   MQPMO        PutMsgOpts = {MQPMO_DEFAULT};

   MQLONG       AdminBufLen;
   PMQBYTE      pAdminMsg;
   PMQCFH       pPCFHeader;
   PMQCFST      pPCFString;
   PMQCFIN      pPCFInteger;
   PMQCFIL      pPCFIntegerList;
   PMQLONG      pValue;

   MQCHAR48     RemoteQMgr;
   MQCHAR48     RemoteQName;

   if (argc < 3)
   {
      printf ( "Required parameter(s) missing - remote qmgr and queue
      ➤  name\n" );
      printf ( "USAGE: MsgBack RemoteQMgr RemoteQName
      ➤  [LocalQMgrName]\n" );
      exit ( -1 );
   }

   strncpy( RemoteQMgr, argv[1], MQ_Q_MGR_NAME_LENGTH );
```

```
strncpy( RemoteQName, argv[2], MQ_Q_NAME_LENGTH );

memset( LocalQMgr, '\0', MQ_Q_MGR_NAME_LENGTH );

if ( argc > 3 )
{
   memcpy( LocalQMgr, argv[3], MQ_Q_MGR_NAME_LENGTH );
}

MQCONN( LocalQMgr,
        &hConn,
        &CompCode,
        &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQCONN",
             &LocalQMgr,
             CompCode,
             Reason );
}

memcpy( ObjDescAdmin.ObjectName, "SYSTEM.ADMIN.COMMAND.QUEUE\0",
➤  MQ_Q_NAME_LENGTH );
memcpy( ObjDescAdmin.ObjectQMgrName, LocalQMgr,
➤  MQ_Q_MGR_NAME_LENGTH );
OpenOpts = MQOO_OUTPUT;

MQOPEN( hConn,
        &ObjDescAdmin,
        OpenOpts,
        &hAdminQ,
        &CompCode,
        &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQOPEN",
             &ObjDescAdmin.ObjectName,
             CompCode,
             Reason );
}

memcpy( ObjDescReply.ObjectName, "SYSTEM.DEFAULT.MODEL.QUEUE\0",
➤  MQ_Q_NAME_LENGTH );
memcpy( ObjDescReply.ObjectQMgrName, LocalQMgr,
➤  MQ_Q_MGR_NAME_LENGTH );
memcpy( ObjDescReply.DynamicQName, "MSGBACKUP.*\0",
➤  MQ_Q_NAME_LENGTH );
OpenOpts = MQOO_INPUT_EXCLUSIVE;
```

```
MQOPEN( hConn,
        &ObjDescReply,
        OpenOpts,
        &hReplyQ,
        &CompCode,
        &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQOPEN",
             &ObjDescReply.ObjectName,
             CompCode,
             Reason );
}

memcpy ( ReplyToQ, ObjDescReply.ObjectName, MQ_Q_NAME_LENGTH );
memcpy ( LocalQMgr, ObjDescReply.ObjectQMgrName,
➤  MQ_Q_MGR_NAME_LENGTH );

printf ( "MsgBack: Obtaining local queue names on qmgr %s\n",
➤  Trim ( &LocalQMgr ) );

/* Set the length for the message buffer */
AdminBufLen = MQCFH_STRUC_LENGTH
              + MQCFST_STRUC_LENGTH_FIXED
              + MQCFIN_STRUC_LENGTH
              + MQCFIL_STRUC_LENGTH_FIXED
              + 16;

/* Allocate storage for the message buffer and set a pointer to */
/* its start address.                                           */
pAdminMsg   = (MQBYTE *)malloc( AdminBufLen );
memset( pAdminMsg, '\0', AdminBufLen );

/* pPCFHeader is set equal to pAdminMsg in order to provide a */
/* structure to the newly allocated block of storage. We can  */
/* then create a request header of the correct format in the  */
/* message buffer.                                            */
pPCFHeader = (MQCFH *)pAdminMsg;

/* pPCFString is set to point into the message buffer immediately */
/* after the request header. This allows us to specify a context  */
/* for the request in the required format.                        */
pPCFString = (MQCFST *)((MQBYTE *)pPCFHeader
             + MQCFH_STRUC_LENGTH );

pPCFInteger = (MQCFIN *)((MQBYTE *)pPCFString
              + MQCFST_STRUC_LENGTH_FIXED
              + 4 );
```

```
pPCFIntegerList = (MQCFIL *)((MQBYTE *)pPCFInteger
                  + MQCFIN_STRUC_LENGTH );

/* Set up request header */
pPCFHeader->Type           = MQCFT_COMMAND;
pPCFHeader->StrucLength    = MQCFH_STRUC_LENGTH;
pPCFHeader->Version        = MQCFH_VERSION_1;
pPCFHeader->Command        = MQCMD_INQUIRE_Q;
pPCFHeader->MsgSeqNumber   = MQCFC_LAST;
pPCFHeader->Control        = MQCFC_LAST;
pPCFHeader->ParameterCount = 3;

/* Set up parameter block */
pPCFString->Type           = MQCFT_STRING;
pPCFString->StrucLength    = MQCFST_STRUC_LENGTH_FIXED + 4;
pPCFString->Parameter      = MQCA_Q_NAME;
pPCFString->CodedCharSetId = MQCCSI_DEFAULT;
pPCFString->StringLength   = 1;
memcpy( pPCFString->String, "*   ", 4 );

/* Set up parameter block */
pPCFInteger->Type           = MQCFT_INTEGER;
pPCFInteger->StrucLength    = MQCFIN_STRUC_LENGTH;
pPCFInteger->Parameter      = MQIA_Q_TYPE;
pPCFInteger->Value          = MQQT_LOCAL;

/* Set up parameter block */
pPCFIntegerList->Type           = MQCFT_INTEGER_LIST;
pPCFIntegerList->StrucLength    = MQCFIL_STRUC_LENGTH_FIXED + 12;
pPCFIntegerList->Parameter      = MQIACF_Q_ATTRS;
pPCFIntegerList->Count          = 3;
pValue = (MQLONG *)((MQBYTE *)pPCFIntegerList +
➤  MQCFIL_STRUC_LENGTH_FIXED);
*pValue = MQCA_Q_NAME;
pValue = pValue + 1;
*pValue = MQIA_CURRENT_Q_DEPTH;
pValue = pValue + 1;
*pValue = MQIA_DEFINITION_TYPE;

MsgDesc.Persistence = MQPER_NOT_PERSISTENT;
MsgDesc.MsgType = MQMT_REQUEST;
memcpy( MsgDesc.ReplyToQ, ReplyToQ, MQ_Q_NAME_LENGTH );
memcpy( MsgDesc.Format, MQFMT_ADMIN, MQ_FORMAT_LENGTH );

MsgDesc.Expiry = 1800;  /* 3 minutes */

PutMsgOpts.Options = MQPMO_NO_SYNCPOINT;

MQPUT( hConn,
       hAdminQ,
```

```
        &MsgDesc,
        &PutMsgOpts,
        AdminBufLen,
        pAdminMsg,
        &CompCode,
        &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQPUT",
             &ObjDescAdmin.ObjectName,
             CompCode,
             Reason );
}

/* Free the storage allocated to the message buffer */
free( pAdminMsg );

MQCLOSE( hConn,
         &hAdminQ,
         MQCO_NONE,
         &CompCode,
         &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQCLOSE",
             &ObjDescAdmin.ObjectName,
             CompCode,
             Reason );
}

ProcessReplyMsgs( hConn,
                  hReplyQ,
                  &ReplyToQ,
                  &LocalQMgr,
                  &RemoteQMgr,
                  &RemoteQName );

MQCLOSE( hConn,
         &hReplyQ,
         MQCO_NONE,
         &CompCode,
         &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQCLOSE",
             &ReplyToQ,
             CompCode,
```

```
                Reason );
   }

   MQDISC ( &hConn,
            &CompCode,
            &Reason );

   if ( CompCode != MQCC_OK )
   {
      MQError ( "MQDISC",
                &LocalQMgr,
                CompCode,
                Reason );
   }

   printf ( "MsgBack: successful completion\n" );

   return(0);

}  /* end main */

void ProcessReplyMsgs( MQHCONN hConn,
                       MQHOBJ hReplyQ,
                       PMQCHAR48 pReplyToQ,
                       PMQCHAR48 pLocalQMgr,
                       PMQCHAR48 pRemoteQMgr,
                       PMQCHAR48 pRemoteQName )
{
   MQOD        ObjDescBack = {MQOD_DEFAULT};
   MQOD        ObjDescAppl = {MQOD_DEFAULT};
   MQMD        MsgDesc = {MQMD_DEFAULT};
   MQGMO       GetMsgOpts = {MQGMO_DEFAULT};
   MQLONG      CompCode;
   MQLONG      Reason;
   MQHOBJ      hBackupQ;
   MQHOBJ      hApplQ;
   MQLONG      OpenOpts;
   MQLONG      QueueDepth;
   MQLONG      DefinitionType;
   char        QueueName[MQ_Q_NAME_LENGTH];

   MQLONG      AdminMsgLen;
   MQLONG      AdminBufLen;
   PMQBYTE     pAdminMsg;
   PMQCFH      pPCFHeader;
   PMQCFST     pPCFString;
   PMQCFIN     pPCFInteger;
   PMQLONG     pPCFType;
   long        Index;
```

```
memcpy( ObjDescBack.ObjectName, pRemoteQName, MQ_Q_NAME_LENGTH );
memcpy( ObjDescBack.ObjectQMgrName, pRemoteQMgr,
➤  MQ_Q_MGR_NAME_LENGTH );
OpenOpts = MQOO_OUTPUT;

MQOPEN( hConn,
        &ObjDescBack,
        OpenOpts,
        &hBackupQ,
        &CompCode,
        &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQOPEN",
             &ObjDescBack.ObjectName,
             CompCode,
             Reason );
}
else
{
   printf ( "    Opened backup queue %s on qmgr %s\n",
          Trim ( &ObjDescBack.ObjectName ),
          Trim ( &ObjDescBack.ObjectQMgrName ) );
} /* endif */

/* AdminBufLen is to be set to the length of the expected reply */
/* message.                                                     */
AdminBufLen = MQCFH_STRUC_LENGTH
              + MQCFST_STRUC_LENGTH_FIXED
              + (MQCFIN_STRUC_LENGTH * 2)
              + MQ_Q_NAME_LENGTH;

/* Set pointers to message data buffers */
pAdminMsg = (MQBYTE *)malloc( AdminBufLen );

do
{
   GetMsgOpts.Options       = MQGMO_WAIT;
   GetMsgOpts.WaitInterval  = 10000;  /* 10 seconds */
   memcpy ( MsgDesc.MsgId, MQMI_NONE, MQ_MSG_ID_LENGTH );
   memcpy ( MsgDesc.CorrelId, MQCI_NONE, MQ_CORREL_ID_LENGTH );

   MQGET( hConn,
          hReplyQ,
          &MsgDesc,
          &GetMsgOpts,
          AdminBufLen,
          pAdminMsg,
          &AdminMsgLen,
```

```
        &CompCode,
        &Reason );

if ( CompCode != MQCC_OK )
{
   MQError ( "MQGET",
             pReplyToQ,
             CompCode,
             Reason );
}

QueueDepth = 0;
DefinitionType = 0;
memset( QueueName, '\0', MQ_Q_NAME_LENGTH );

/* Examine Header */
pPCFHeader = (MQCFH *)pAdminMsg;

if ( pPCFHeader->CompCode != MQCC_OK )
{
   printf( "Error returned from command server. CC=%d RC=%d\n",
           pPCFHeader->CompCode, pPCFHeader->Reason
         );
   exit( -1 );
} /* endif */

/* Examine first parameter */
pPCFType = (MQLONG *)(pAdminMsg + MQCFH_STRUC_LENGTH);

Index = 1;

while ( Index <= pPCFHeader->ParameterCount )
{
   /* Establish the type of each parameter and allocate  */
   /* a pointer of the correct type to reference it.     */
   switch ( *pPCFType )
   {
      case MQCFT_INTEGER:
         pPCFInteger = (MQCFIN *)pPCFType;
         switch (pPCFInteger->Parameter)
         {
         case MQIA_CURRENT_Q_DEPTH:
            QueueDepth = pPCFInteger->Value;
            break;
         case MQIA_DEFINITION_TYPE:
            DefinitionType = pPCFInteger->Value;
            break;
         default:
            break;
         }
```

```
            Index++;
            /* Increment the pointer to the next parm */
            /* by the length of the current parm.     */
            pPCFType = (MQLONG *)( (MQBYTE *)pPCFType
                        + pPCFInteger -> StrucLength );
            break;
         case MQCFT_STRING:
            pPCFString = (MQCFST *)pPCFType;
                switch (pPCFString->Parameter)
            {
            case MQCA_Q_NAME:
               memcpy( QueueName, pPCFString->String, pPCFString->
               ➤  StringLength );
               break;
            default:
               break;
            }
            Index++;
            /* Increment the pointer to the next parm */
            /* by the length of the current parm.     */
            pPCFType = (MQLONG *)( (MQBYTE *)pPCFType
                        + pPCFString -> StrucLength );
            break;
         default:
            printf( "Error: Received parameter block is of an
            ➤  unrecognized type.\n" );
            printf( "       Program ending.\n" );
            exit( -1 );
            break;
      } /* endswitch */
   } /* endwhile */

   if ( QueueDepth > 0 )
   {
      if ( !strncmp(QueueName, "SYSTEM.", 7) )
      {
         printf ( "    Skipping system queue %s.\n", Trim (
         ➤  &QueueName ) );
      }
      else if ( DefinitionType != MQQDT_PREDEFINED )
      {
         printf ( "    Skipping dynamic queue %s.\n",
         ➤  Trim ( &QueueName ) );
      }
      else
      {
         /* browse queue, write MQXQH and message to the */
         /* back-up queue                                */

         memcpy ( ObjDescAppl.ObjectName, QueueName,
```

```
      ➤  MQ_Q_NAME_LENGTH );
      memcpy ( ObjDescAppl.ObjectQMgrName, pLocalQMgr,
      ➤  MQ_Q_MGR_NAME_LENGTH );
      OpenOpts = MQOO_INPUT_AS_Q_DEF + MQOO_BROWSE;

      MQOPEN( hConn,
              &ObjDescAppl,
              OpenOpts,
              &hApplQ,
              &CompCode,
              &Reason );

      if ( CompCode != MQCC_OK )
      {
         MQError ( "MQOPEN",
                   &ObjDescAppl.ObjectName,
                   CompCode,
                   Reason );
      }

      printf ( "    Browsing %d message(s) from queue %s\n",
             QueueDepth,
             Trim ( &QueueName ) );

      ProcessApplMsgs ( hConn,
                        hBackupQ,
                        &ObjDescBack,
                        hApplQ,
                        &ObjDescAppl);

      MQCLOSE( hConn,
               &hApplQ,
               MQCO_NONE,
               &CompCode,
               &Reason );

      if ( CompCode != MQCC_OK )
      {
         MQError ( "MQCLOSE",
                   &ObjDescAppl.ObjectName,
                   CompCode,
                   Reason );
      }
   }
}   /* endif QueueDepth > 0 */

/*************************************************************/
/* Finished the current message, so process the next one. */
/*************************************************************/
```

```
   } while ( pPCFHeader->Control == MQCFC_NOT_LAST ); /* end do */

   free( pAdminMsg );

   MQCLOSE( hConn,
            &hBackupQ,
            MQCO_NONE,
            &CompCode,
            &Reason );

   if ( CompCode != MQCC_OK )
   {
      MQError ( "MQCLOSE",
                &ObjDescBack.ObjectName,
                CompCode,
                Reason );
   }
   return;
}

void ProcessApplMsgs ( MQHCONN hConn,
                       MQHOBJ hBackupQ,
                       PMQOD pObjDescBack,
                       MQHOBJ hApplQ,
                       PMQOD pObjDescAppl )
{
   MQMD          MsgDescBack = {MQMD_DEFAULT};
   MQMD          MsgDescAppl = {MQMD_DEFAULT};
   MQGMO         GetMsgOpts = {MQGMO_DEFAULT};
   MQPMO         PutMsgOpts = {MQPMO_DEFAULT};
   MQXQH         XmitQHdr = {MQXQH_DEFAULT};
   MQLONG        CompCode;
   MQLONG        Reason;
   MQLONG        UserMsgLen;
   MQLONG        UserBufLen;
   PMQBYTE       pUserMsg;
   MQLONG        UserBufLen2;
   PMQBYTE       pUserMsg2;
   PMQXQH        pXmitQHdr;
   long          RecordLen;
   long          PutCount = 0;

   UserBufLen = 8192;
   /* Set pointer to message data buffer */
   pUserMsg = (MQBYTE *)malloc( UserBufLen );

   UserBufLen2 = UserBufLen + sizeof ( XmitQHdr );
   /* Set pointer to message data buffer */
   pXmitQHdr = (PMQXQH)malloc( UserBufLen2 );
```

```
do
{
   GetMsgOpts.Options = MQGMO_NO_WAIT
                      + MQGMO_BROWSE_NEXT
                      + MQGMO_ACCEPT_TRUNCATED_MSG;
   memcpy ( MsgDescAppl.MsgId, MQMI_NONE, MQ_MSG_ID_LENGTH );
   memcpy ( MsgDescAppl.CorrelId, MQCI_NONE, MQ_CORREL_ID_LENGTH );

   MQGET( hConn,
          hApplQ,
          &MsgDescAppl,
          &GetMsgOpts,
          UserBufLen,
          pUserMsg,
          &UserMsgLen,
          &CompCode,
          &Reason );

   switch ( CompCode )
   {
      case MQCC_OK :
         RecordLen = UserMsgLen + sizeof ( XmitQHdr );
         memcpy ( pXmitQHdr, &XmitQHdr, sizeof ( XmitQHdr ) );
         memcpy ( pXmitQHdr -> RemoteQName, pObjDescAppl ->
         ➤  ObjectName, MQ_Q_NAME_LENGTH );
         memcpy ( pXmitQHdr -> RemoteQMgrName, pObjDescAppl ->
         ➤  ObjectQMgrName, MQ_Q_MGR_NAME_LENGTH );
         memcpy ( & ( pXmitQHdr -> MsgDesc ), &MsgDescAppl,
         ➤  sizeof ( MsgDescAppl ) );
         pUserMsg2 = ( MQBYTE* ) ( pXmitQHdr + 1 );
         memcpy ( pUserMsg2, pUserMsg, UserMsgLen );

         MsgDescBack.MsgType = MQMT_DATAGRAM;
         MsgDescBack.Expiry = pXmitQHdr -> MsgDesc.Expiry;
         MsgDescBack.Persistence = MQPER_PERSISTENT;
         memcpy ( MsgDescBack.Format, MQFMT_XMIT_Q_HEADER,
         ➤  MQ_FORMAT_LENGTH );

         PutMsgOpts.Options  = MQPMO_SYNCPOINT;

         MQPUT( hConn,
                hBackupQ,
                &MsgDescBack,
                &PutMsgOpts,
                RecordLen,
                pXmitQHdr,
                &CompCode,
                &Reason );

         if ( CompCode != MQCC_OK )
```

```
            {
               MQError ( "MQPUT",
                         &pObjDescBack -> ObjectName,
                         CompCode,
                         Reason );
            }

            PutCount ++;

            break;
         case MQCC_WARNING :
            if ( Reason == MQRC_TRUNCATED_MSG_ACCEPTED )
            {
               free( pUserMsg );
               free( pXmitQHdr );
               UserBufLen = UserMsgLen;
               /* Set pointer to message data buffer */
               pUserMsg = (MQBYTE *)malloc( UserBufLen );
               UserBufLen2 = UserBufLen + sizeof(XmitQHdr);
               /* Set pointer to message data buffer */
               pXmitQHdr = (PMQXQH)malloc( UserBufLen2 );
            }
            else
            {
               MQError ( "MQGET",
                         &pObjDescAppl -> ObjectName,
                         CompCode,
                         Reason );
            }

            break;
         case MQCC_FAILED :
            if ( Reason != MQRC_NO_MSG_AVAILABLE )
            {
               MQError ( "MQGET",
                         &pObjDescAppl -> ObjectName,
                         CompCode,
                         Reason );
            }

            break;
      }  /* end switch */
   } while ( CompCode != MQCC_FAILED ) ;  /* enddo */

   free( pUserMsg );
   free( pXmitQHdr );

   printf ( "       Saved %d message(s) for queue  %s\n",
          PutCount,
          Trim ( &pObjDescAppl -> ObjectName ) );
```

```
   return;
}

void MQError ( MQCHAR8 Function,
               PMQCHAR48 pObject,
               MQLONG CompCode,
               MQLONG Reason )
{
   MQCHAR48 Object;

   memcpy ( Object, pObject, MQ_Q_NAME_LENGTH );

   printf( "%s failed for object %s. CC=%d RC=%d\n",
           Function,
           Trim ( pObject ),
           CompCode,
           Reason );
   exit( -1 );
}

PMQCHAR48 Trim ( PMQCHAR48 pObject )
{
   char *pBlank;

   pBlank = memchr ( pObject, ' ', MQ_Q_NAME_LENGTH );
   if ( pBlank != NULL )
      *pBlank = 0;

   return ( pObject );
}
```

Below is the source code for the MQXMIT conversion exit program.

MQXMIT

```
// MQXMIT: This is a data conversion exit for converting messages
//         with MQMD.Format MQFMT_XMIT_Q_HEADER. The built-in
//         conversion routines fail to convert messages with this
//         format as the routines are really written for message
//         channel agents. MCAs do not require the transmission
//         queue header to be converted, only the data after the
//         header. According to Appendix D - Data Conversion of the
//         MQSeries Application Programming Reference, a user-supplied
//         exit with the same name as a built-in format will be
//         invoked if the built-in conversion routine fails to
//         convert the data.
//
//         This exit program caters for the needs of program MsgBack
//         when messages are backed up from an MQSeries for Windows
```

```
//         NT queue manager to an OS/390 queue manager. MsgBack
//         puts its own transmission queue header at the beginning
//         of each message. This needs to be converted to simplify
//         restoring the messages to their original location. The
//         message data following the transmission queue header is
//         left unconverted.
//
//         This program is based on the sample data conversion exit
//         AMQSVFC0. The function ConvertMQXMIT is generated by the
//         utility crtmqcvx.

 #include <cmqc.h>              /* For MQI datatypes                 */
 #include <cmqxc.h>             /* For MQI exit-related definitions  */
 #include <amqsvmha.h>          /* For sample macro definitions      */

 /*********************************************************************/
 /* Insert the function prototypes for the functions produced by      */
 /* the data conversion utility program.                              */
 /*********************************************************************/
MQLONG ConvertMQXMIT( PMQBYTE *in_cursor,
                      PMQBYTE *out_cursor,
                      PMQBYTE in_lastbyte,
                      PMQBYTE out_lastbyte,
                      MQHCONN hConn,
                      MQLONG  opts,
                      MQLONG  MsgEncoding,
                      MQLONG  ReqEncoding,
                      MQLONG  MsgCCSID,
                      MQLONG  ReqCCSID,
                      MQLONG  CompCode,
                      MQLONG  Reason );

 MQDATACONVEXIT MQStart;

 void MQENTRY MQStart(
   PMQDXP   pDataConvExitParms, /* Data-conversion exit parameter    */
                                /* block                             */
   PMQMD    pMsgDesc,           /* Message descriptor                */
   MQLONG   InBufferLength,     /* Length in bytes of InBuffer       */
   PMQVOID  pInBuffer,          /* Buffer containing the unconverted */
                                /* message                           */
   MQLONG   OutBufferLength,    /* Length in bytes of OutBuffer      */
   PMQVOID  pOutBuffer)         /* Buffer containing the converted   */
                                /* message                           */
 {
   MQLONG   ReturnCode   = MQRC_NONE;
   MQHCONN  hConn        = pDataConvExitParms->Hconn;
   MQLONG   opts         = pDataConvExitParms->AppOptions;
   PMQBYTE  in_cursor    = (PMQBYTE)pInBuffer;
   PMQBYTE  out_cursor   = (PMQBYTE)pOutBuffer;
```

```
PMQBYTE  in_lastbyte  = (PMQBYTE)pInBuffer + InBufferLength - 1;
PMQBYTE  out_lastbyte = (PMQBYTE)pOutBuffer + OutBufferLength - 1;
MQLONG   MsgEncoding  = pMsgDesc->Encoding;
MQLONG   ReqEncoding  = pDataConvExitParms->Encoding;
MQLONG   MsgCCSID     = pMsgDesc->CodedCharSetId;
MQLONG   ReqCCSID     = pDataConvExitParms->CodedCharSetId;
MQLONG   CompCode     = pDataConvExitParms->CompCode;
MQLONG   Reason       = pDataConvExitParms->Reason;
int      count;

/*****************************************************************/
/* Insert calls to the code fragments to convert the format's    */
/* structure(s) here.                                            */
/*****************************************************************/
ReturnCode = ConvertMQXMIT( &in_cursor,
                            &out_cursor,
                            in_lastbyte,
                            out_lastbyte,
                            hConn,
                            opts,
                            MsgEncoding,
                            ReqEncoding,
                            MsgCCSID,
                            ReqCCSID,
                            CompCode,
                            Reason);

/*****************************************************************/
/* Check whether the conversion succeeded                        */
/*****************************************************************/
if ( (ReturnCode == MQRC_NONE) ||
     (ReturnCode == MQRC_TRUNCATED_MSG_ACCEPTED) )
{
   count = out_lastbyte - out_cursor + 1;
   if (count > 0)
   {
      /* Copy the rest of the message without conversion */
      memcpy(out_cursor, in_cursor, count);
   }
  pDataConvExitParms->ExitResponse = MQXDR_OK;
  /*****************************************************************/
  /* The sample exit suggests the statement below to set the       */
  /* length to be returned on exit of this program. However, if    */
  /* the output buffer is longer than the message, then an         */
  /* incorrect length is returned. Because this conversion         */
  /* routine does not change the length of the data, DataLength    */
  /* may be left as is.                                            */
  /*****************************************************************/
  /* pDataConvExitParms->DataLength = out_cursor                   */
  /*                                - (PMQBYTE)pOutBuffer;         */
```

```
    /*****************************************************************/
  }
  /*****************************************************************/
  /* Otherwise indicate that conversion of the message data failed. */
  /*****************************************************************/
  else
  {
    pDataConvExitParms->ExitResponse = MQXDR_CONVERSION_FAILED;
  }
  /*****************************************************************/
  /* It the message is not already truncated update comp code, and  */
  /* reason.                                                        */
  /*****************************************************************/
  if (Reason != MQRC_TRUNCATED_MSG_ACCEPTED)
  {
     pDataConvExitParms->Reason = ReturnCode;

    if (ReturnCode == MQRC_NONE)
    {
      pDataConvExitParms->CompCode   = MQCC_OK;
    }
    else
    {
      pDataConvExitParms->CompCode = MQCC_WARNING;
    }
  }
  return;
}

/*****************************************************************/
/* Insert the functions produced by the data conversion exit      */
/* utility program.                                               */
/*****************************************************************/

MQLONG ConvertMQXMIT( PMQBYTE *in_cursor,
                      PMQBYTE *out_cursor,
                      PMQBYTE in_lastbyte,
                      PMQBYTE out_lastbyte,
                      MQHCONN hConn,
                      MQLONG  opts,
                      MQLONG  MsgEncoding,
                      MQLONG  ReqEncoding,
                      MQLONG  MsgCCSID,
                      MQLONG  ReqCCSID,
                      MQLONG  CompCode,
                      MQLONG  Reason )
{
    MQLONG ReturnCode = MQRC_NONE;

    ConvertChar(4); /* StrucId */
```

```
AlignLong();
ConvertLong(1); /* Version */

ConvertChar(48); /* RemoteQName */

ConvertChar(48); /* RemoteQMgrName */

ConvertChar(4); /* StrucId */

AlignLong();
ConvertLong(1); /* Version */

AlignLong();
ConvertLong(1); /* Report */

AlignLong();
ConvertLong(1); /* MsgType */

AlignLong();
ConvertLong(1); /* Expiry */

AlignLong();
ConvertLong(1); /* Feedback */

AlignLong();
ConvertLong(1); /* Encoding */

AlignLong();
ConvertLong(1); /* CodedCharSetId */

ConvertChar(8); /* Format */

AlignLong();
ConvertLong(1); /* Priority */

AlignLong();
ConvertLong(1); /* Persistence */

ConvertByte(24); /* MsgId */

ConvertByte(24); /* CorrelId */

AlignLong();
ConvertLong(1); /* BackoutCount */

ConvertChar(48); /* ReplyToQ */

ConvertChar(48); /* ReplyToQMgr */

ConvertChar(12); /* UserIdentifier */
```

```
    ConvertByte(32); /* AccountingToken */

    ConvertChar(32); /* ApplIdentityData */

    AlignLong();
    ConvertLong(1); /* PutApplType */

    ConvertChar(28); /* PutApplName */

    ConvertChar(8); /* PutDate */

    ConvertChar(8); /* PutTime */

    ConvertChar(4); /* ApplOriginData */

Fail:
    return(ReturnCode);
}
```

Eric Judd
Technical Consultant
Metropolitan Life (RSA)

MQSeries and Microsoft Transaction Server

Component-based development is becoming a widely used method of building new applications. Developers are able to use this approach to separate application logic from the implementation of the user interface, thus making development much easier. This approach allows different systems to share the same components, also allowing the same components to be accessed by different clients. As business logic shifts from traditional fat clients to servers, scalable applications servers that manage server components are needed. Microsoft Transaction Server (MTS) is a good example of an application server and more. It acts as a middle tier for running components. So, how does MQSeries fit into this picture? IBM recently released an MQSeries SupportPac that enables business objects running in Microsoft Transaction Server to access MQSeries functions and services. This can be used to exploit many of MTS's capabilities, such as thread

pooling, process isolation, and transaction monitoring. In this article, I discuss what capabilities MTS brings to the development and deployment of applications, and also describe how these capabilities can be integrated with MQSeries. Though I deal only with the basic aspects of MTS in this article, this coverage should be sufficient to allow those unfamiliar with this software to understand its basic terms and concepts.

MTS BASICS

So what exactly is MTS? Microsoft Transaction Server is a component-based transaction processing system for developing, deploying, and managing server applications. It defines an application model that determines how distributed, component-based applications are developed.

MTS provides three major services – it acts as an:

- Object Request Broker (ORB)
- Resource manager

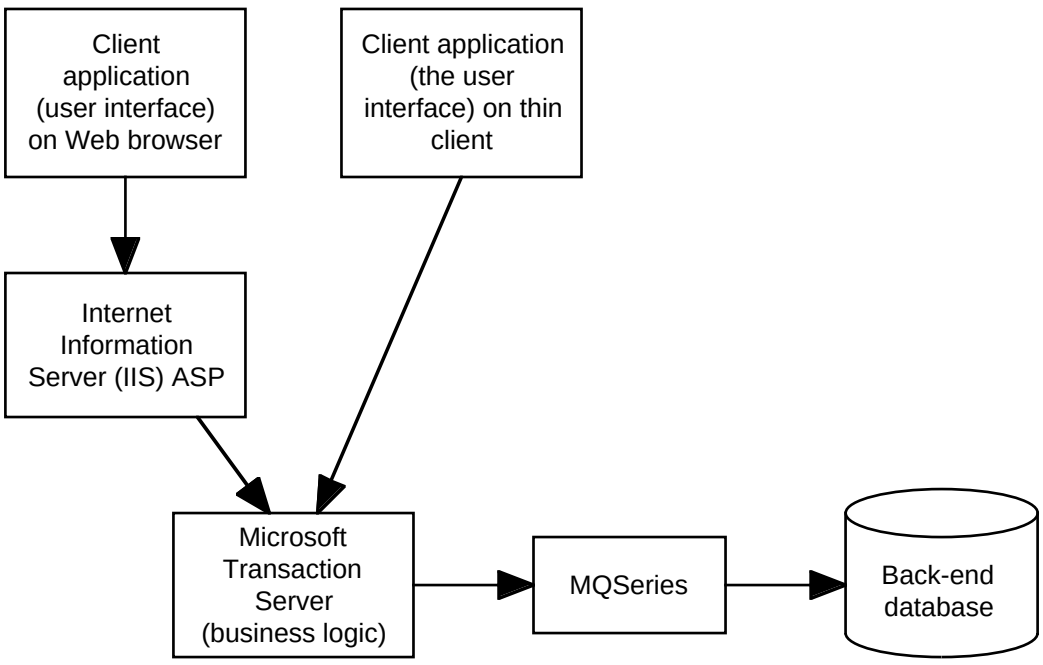

Figure 1: MQSeries integration with MTS

- Transaction monitor.

MTS has capabilities that are essential for scaling server-based components to support a large number of simultaneous client requests. It performs thread pooling, process isolation, transaction coordination, and object instance management and provides role-based security, performing all these functions automatically with no need for special code or scripts. Typically, MTS sits between client applications, which are normally no more than a user interface (in other words, a thin client) and database servers or other back-end servers. COM objects are simply installed on an MTS server and an instance of the object is created when a client application requests it.

Let's now discuss how to integrate this architecture with MQSeries and how MQSeries can take advantage of many of MTS's features.

THREADING

One aspect of MTS that the MQSeries MTS SupportPac uses intensively is its threading capability. Threading is very important if you want your application to support a large number of user requests. Different types of threading model are available, and you should be familiar with each of them. The list below briefly explains each threading model, and you may consider checking other sources for a more detailed discussion.

- *Single threading*

  In this threading model, all objects are executed on the main thread and methods calls are serialized across all objects in the component.

- *Apartment threading*

  In this threading model, each object runs in its own apartment. The model allows multiple single-threaded apartments, but not multi-threaded apartments.

- *Free threading*

  This threading model allows objects to share threads. It allows multi-threaded apartments, but not single-threaded apartments.

- *Mixed threading*

  This threading model comprises a multi-threaded apartment plus one or more single-threaded apartments.

We should all be aware of the fact that MQSeries can use an object only if it was created using the same thread that called it. For example, if we get a connection handle to a queue manager, the handle can be used only by objects within the thread that initiated the connection. However, when using MQSeries MTS Support, the MQSeries COM object uses a free threading model to allow method calls to be serviced by any thread. MTS Support creates a connection handle with a corresponding context that is managed by MTS and it can create new threads for routing calls to MQSeries, depending on its needs. MTS will automatically switch threads to provide a more efficient way of servicing requests. MTS manages these threads, which is why components running under MTS neither need nor should create their own threads.

CONNECTION POOLING

If a client frequently connects to and disconnects from a queue manager, this often creates a considerable overhead on the server as a result of the associated processes. Resources are continually acquired and released, though some may not be freed if an erroneous application terminates abnormally, which can severely affect performance. MTS provides 'resource dispensers' that manage a pool of resources. This pool can provide simple and efficient sharing of its resources among various objects. The resources, such as database, network, and socket connections, are dynamically created and do not persist.

MQSeries MTS Support provides pooling of connection handles to the queue manager. Instead of passing the request to disconnect from a queue manager (*MQDISC*) to MQSeries, the SupportPac uses Microsoft Dispenser Manager to pool the connection and issue a commit, so it doesn't interfere with other MQSeries transactions. Queues that are left opened by applications to which the connection belongs are closed before the connection is put back into the pool. When another client application tries to connect to the queue manager, the resource dispenser forwards the request to a 'holder' (an object

that maintains the actual resource inventory for each dispenser), which, in turn, tries to examine the pool and decide whether to create a new resource or get one from the pool.

The SupportPac has no control over the number of connections to be pooled – this is regulated by the holder. If the holder decides that there are not enough resources in the inventory, it calls the resource dispenser to create new ones, so as to increase the number of simultaneous requests that it can service. The dispenser manager periodically checks the holder's inventory to clean up the resource pool. If the connection resource is not in use by business objects and is not taking part in a transaction, it is put in the holder's 'unenlisted inventory'. The holder checks for expired resources that are inactive for more than a pre-defined timeout value in this inventory. The connections are then removed from the inventory and destroyed.

When applications terminate without first disconnecting all their active connections, all transactions associated with those connections are automatically backed out. This prevents unfinished transactions from completing.

In MTS, there is no cross-process pooling of resources – each process has its own separate inventory. Dispensers manage only transient data, such as connections that are relevant only to the process that created them. So, when connection handles from the pool are to be reused, their queue manager names must be identical. For example, suppose that our default queue manager is named *DEFT.QUEUE.MANAGER* and that an application tries to connect to it using the default *MQCONN("", …).* If this resource is placed in the pool after it's used, then the previous connection handle will not be used by another application if tries to connect to a default queue manager by name: *MQCONN("DEFT.QUEUE.MANAGER", …)*; instead it will create its own resource pool.

Note that the 'fastpath' binding option for connections (*MQCNO_FASTPATH_BINDING*) is not available in this SupportPac. The reason for this is that this option allows no more than one Windows NT thread to be connected to a queue manager at any time, and this is not consistent with the multi-threading capability of the SupportPac.

SECURITY IN A POOLED CONNECTION

You may ask whether any security issues are associated with the use of a resource pool. We are used to MQSeries applications that are validated every time they try to connect to a queue manager, but this may not be the case in a pooled connection. MQSeries MTS Support does not re-check the security credentials of objects that are reused from the pool. If an application uses a security method called 'impersonation', which allows a thread to be executed in a security context that is different from that of the process that owns the thread, MQSeries will only validate the impersonator the first time the connection is created, which is referred to as the 'real connection'. The connection that was created can then be used by another application using impersonation. MTS does not promote the use of impersonation, for obvious reasons, which is why MQSeries for MTS is not supported by this product. MTS's role-based security is recommended instead.

TRANSACTIONS

MQSeries is able to act both as a resource manager and a transaction coordinator, which means that it can participate in a coordinated transaction with other resources. However, when using MQSeries with MTS, it's the Microsoft Distributed Transaction Coordinator (MS DTC) that coordinates MQSeries queue manager resources along with other resource managers, such as database servers (DB2 UDB, Oracle, etc), that participate in a transaction.

Here are some pointers to bear in mind when you want MQSeries to be part of a transaction that is coordinated by DTC:

1 Always use the syncpoint option when issuing an *MQGET* or *MQPUT*, so that your request is included in a unit of work. You have to use the *MQPMO_SYNCPOINT* put message option when putting message to a queue and either the option *MQGMO_SYNCPOINT* or *MQGMO_SYNCPOINT_IF_PERSISTENT* when you're getting a message from a queue.

2 MTS/DTC will be the coordinator for transactions under MTS, so don't use an *MQBEGIN* in your application.

3 The MTS/DTC will determine whether transactions are successful, so don't use either *MQCMIT* or *MQBACK*.

As some of MQSeries' transaction management features are off-loaded to MTS, MTS has a substantial impact on the behaviour of MQSeries objects. An example of this is in the use of connection handles. Although MQSeries for MTS supports free threading, a connection handle can be used only by one transaction at a time, though it can be re-used later, once the transaction is complete. MQSeries MTS Support determines the transactional context of a connection handle, so when you issue the first syncpoint call (for instance, an *MQPUT* with *MQPMO_SYNCPOINT*), the handle can be used only in that transaction from that point onwards. A this point, you might wonder what's the use of free threading in this scenario, if other objects can't use the same handle. The answer is that you may pass the handle to other objects in the same process, provided that the handle is used within the scope of the same transaction, so it executes in the same transactional context.

Another characteristic of MTS is DTC time-outs. You have to be aware of this as it can result in unexpected errors. For instance, suppose an application creates an object that does an *MQGET* with 'wait' time of, say, three minutes. If DTC is configured to time out in two minutes, after two minutes DTC will communicate with MQSeries for MTS, which will then close MQSeries objects and terminate the transaction. When the application's wait time ends, the original object is still there, and the application may try another operation on the object. This will fail with an error stating that it's not part of a transaction and an *MQRC_UOW_ENLISTMENT_ERROR* reason code will be sent to the application.

It is also advisable not to use resources that are created by objects that are already destroyed, as the Dispenser Manager may destroy resources that belong to defunct objects.

COORDINATION

With the use of free threading, which comes with MQSeries MTS Support, different types of transaction can be realized. One example is to use a single object to open two connections to two different queue managers, where one is used for putting messages on a queue and the other for getting messages from a queue. Another possibility is to have

two objects opening a single queue manager, which means that you have to coordinate only one object. Queue managers can be spread across the network but still participate in the transaction using DCOM. However, as MQSeries transactions have to use the server's MQI, objects must reside on the same system as the queue manager being used.

NEW REASON CODES AND TRANSACTIONAL IMPLICATIONS

As a result of changes in the way MQSeries handles transactions, there are some new errors that we may encounter during the course of developing applications under MTS. Below are the usual error codes we encounter, which may have new meanings when running under MTS, plus some new reason codes that will enable you to correctly debug your application.

- *MQRC_HCONN_ERROR*

  Under MTS, this error may be caused by an attempt to use a connection handle in a context where it is not valid. This could arise if a connection handle is passed between processes or packages. Remember that you can use connection handles only within the same process.

- *MQRC_ENVIRONMENT_ERROR*

  If you try to use the MQSeries transaction coordination features by issuing an *MQBEGIN*, *MQCMIT*, or *MQBACK* while in a DTC transaction, you are likely to encounter this error. Remember that DTC is the component that coordinates transactions in this environment.

- *MQRC_ALREADY_CONNECTED* and *MQRC_ANOTHER_Q_MGR_CONNECTED*

  These errors are normally encountered when you try to connect to a queue manager that is already connected or is in the process of connecting to another queue manager. However, these errors may not be relevant if your application uses the MQSeries MTS Support, as additional connections are allowed. Remember that you can connect other objects provided they are part of the same transaction.

- *MQRC_GLOBAL_UOW_CONFLICT*

  This means that a conflict was detected in a global unit of work resulting from the use of connection handles. This may happen when an application tries to pass a connection handle between objects that are performing different DTC transactions. Remember that we can use a connection handle in only one transaction at a time (you are, however, free to pass connection handles in non-transactional operations). There are two ways to handle this problem: either check that the MQSeries MTS attribute value of the object is the right one for the application or correct the application itself.

- *MQRC_LOCAL_UOW_CONFLICT*

  This reason code tells you that an application is trying to pass a connection handle between objects, where one object is participating in a DTC transaction (usually referred to as a global unit of work), while the other is under a queue manager-coordinated transaction (usually referred to as a local unit of work). You could still use a connection handle in this scenario, as long as the handle is not participating in a transaction in both the GUW and the LUW. To correct this error, try to check the MQSeries MTS attribute value to ensure it's correct. Alternatively, correct the application by preventing it from sharing a connection handle between different types of unit of work.

- *MQRC_HANDLE_IN_USE_FOR_UOW*

  This error may arise if an application tries to pass a connection handle between objects, where one object is still participating in a DTC transaction while the other isn't. Note that, under DTC, transactions are completed asynchronously, which happens when applications complete transactions or objects are finalized. This error is also encountered when DTC has already timed out or shut down. In this case, the object that was created inside the transaction loses the association with the transaction, causing the error. Again, check your application so objects don't use connection handles when running in different units of work.

- *MQRC_UOW_ENLISTMENT_ERROR*

  As discussed under the heading *Transactions* (see above), this error may be caused by DTC time-outs. When the time-out for DTC is reached, MQSeries objects are closed along with the transaction. The application may not realize this and try to use the object again, thus causing the error. Another possible reason for this error is that you failed to reinstall NT Service Pack SP6a after installing MTS from the NT Option Pack.

- *MQRC_UOW_MIX_NOT_SUPPORTED*

  Performing a mix of transactional and non-transactional calls may cause this error. Currently, it is not possible to combine DTC-coordinated transactions with MQSeries-coordinated transactions in the same process. There are no restrictions if either an MQSeries transactional call comes from a transactional object or a non-transactional call to MQSeries comes from a non-transactional object. If a mixed call is attempted, only the first transactional or non-transactional call will succeed and subsequent calls of the other type will fail, causing the error.

Below are new constant definitions you can use in your application. Use the definitions by placing them in header or module files.

```
MQRC_GLOBAL_UOW_CONFLICT= 2351
MQRC_LOCAL_UOW_CONFLICT = 2352
MQRC_HANDLE_IN_USE_FOR_UOW = 2353
MQRC_UOW_ENLISTMENT_ERROR = 2354
MQRC_UOW_MIX_NOT_SUPPORTED = 2355
```

You may encounter some of the problems detailed above if you are new to programming MQSeries with MTS. It’s a good idea to check your environment every time you try to run your component. Check whether the NT Service Pack has been installed and whether the resource managers are running. You can also use the installation verification test application that comes with MQSeries MTS Support to test your environment.

There are some threading issues you have to be familiar with that can substantially affect performance. Every request for an *MQCONN* results in the creation of a new thread. MTS sets a limit of 100 MTS

threads, and only one object can run on any thread at any time. Thus, successive requests for connections from an object result in the creation of more threads, and, as IBM states, if the thread count climbs to 600, performance degradation can be observed. Try to code your components wisely to avoid this situation.

OTHER CONSIDERATIONS

There are programming changes you have to deal with in order to exploit the capabilities of MTS. One of them regards the creation of objects. If your component uses other components, use the *CreateInstance* method of an object's context object to create an instance of an object rather than *CoCreateInstance*. The former creates an object that inherits from the creating object's context, including transactions, if any are present, and all security identities. This is important if you want all components to run in a coordinated state and help MTS to determine the outcome of a transaction. Below is an example of creating an object using *CreateInstance.*

```
Dim ObjCtx As ObjectContext
Dim AccountObj As BankAcct.Request

' Try to get the objects context.
Set ObjCtx = GetObjectContext()

' Use the object's context to instantiate another object.
Set AccountObj = ObjCtx.CreateInstance("BankAcct.Request ")
```

Another consideration concerns the use of the *SetComplete* and *SetAbort* methods, which are used to indicate whether components have completed their work. *SetComplete* can be used to indicate that an object has successfully completed its work and is ready to be deactivated. Similarly, *SetAbort* indicates an unsuccessful task that also results in the state of the object being released.

If your component doesn't use these methods, automatic transactions will complete only when the client releases the object. While MTS will attempt to commit the transaction, the client won't be able to determine whether the transaction was committed or aborted. This is a very difficult situation for both your application and MTS. Your component should inform the client application that a particular task

either completed or failed. This method also helps MTS to scale server components by allowing server resources to be used more efficiently. Below is an example of *SetComplete* and *SetAbort*.

```
Dim ObjCtx As ObjectContext
Set ObjCtx = GetObjectContext()
On Error GoTo AbortWork

' Perform MQSeries work and database transaction here. Call
' SetComplete if the work was successful.
ObjCtx.SetComplete
Set ObjCtx = Nothing
SendAccount = TRUE
Exit Function

' Call SetAbort in the error handler if errors occurred.

AbortWork:
        ObjCtx.SetAbort
        Set ObjCtx = Nothing
        SendAccount = FALSE
        Exit Function
```

Rommel K Abdon
Senior System Engineer
Client Server Technologies Inc (The Philippines)

MQSeries clusters: a hands-on view (part 1)

If you think you're ready for MQSeries clustering, be careful! Before implementing clustering in your production environment, make sure you understand its set-up and operations, as there are some surprises in store for you.

This article is based on my experience of using MQSeries clusters, and it attempts to inform you of the advantages and pitfalls of clustering. Although most of my experience was gained on OS/390 and Windows NT, the concepts hold true for all environments. I firmly believe that there is no substitute for trying things out when it comes

to understanding what clusters are about. Furthermore, in spite of all the work I've done with clusters, I think there is still more for me to learn!

DEFINITION

A cluster is a set or network of queue managers that are logically associated in some way. They may, and in most cases will, reside either on different platforms or across an OS/390 Sysplex. For Version 5.1 platforms, clustering is already available, though only OS/390 users with Version 2.1 installed can benefit from it.

MQSeries clustering aims to address three important issues:

- Reducing the number of definitions needed for a network of connected queue managers.
- Enabling applications to continue to send messages even if the target queue manager becomes unavailable.
- Enabling workload to be evenly distributed across a number of available queue managers in the cluster.

Don't confuse MQSeries clustering with hardware clustering, such as HACMP using shared disks. An important point to remember is that clustering is about extending the ability to add messages to queues (via MQPUT) – it does not enable remote MQGETs.

A very good source of information on clustering can be found in the IBM manual *MQSeries Queue Manager Clusters* (SC34-5349-00), which can also be found on the Web (the continuation character, '➤', below indicates a formatting line break – the two lines comprise one URL):

```
http://www-4.ibm.com/software/ts/mqseries/library/manualsa/
➤  csqzah/csqzah.htm
```

COMPONENTS

In order to satisfy the above aims, several new components have been introduced:

- *The repository*

  This is an inventory of ‘clustered’ items that resides in a new system queue (*SYSTEM.CLUSTER.REPOSITORY.QUEUE*). A new repository manager is used to manage it. The repository manager is started when the channel initiator (*CHIN* on OS/390) address space starts. Note that this necessarily means that the queue depth of the repository increases when clustered items are added. Thus, after a queue manager restart, the queue depth may decrease – this is perfectly normal, and is just the result of messages being combined into one or more larger records.

- *SYSTEM.CLUSTER.COMMAND.QUEUE*

  The cluster command queue is used to send commands between cluster queue managers.

- *SYSTEM.CLUSTER.TRANSMIT.QUEUE*

  The cluster transmission queue acts as the single transmission queue for sending messages between cluster queue managers.

- *Cluster receiver and cluster sender channels*

  These are two new types of channel. Models of them are provided with the product as *SYSTEM.DEF.CLUSRCVR* and *SYSTEM.DEF.CLUSSDR* respectively.

- *Cluster workload exit*

  This determines how messages are distributed across queue managers in the cluster. The default is a ‘round-robin’ approach, where messages are equally distributed.

- Several new attributes are available for queue managers, queues, and channels:

  – *Cluster Name*, which is just the name of the cluster.

  – *Cluster NameList*, which is a list of available clusters – this allows an MQSeries resource to be part of more than one cluster.

  – *Bind Option*. Once MQSeries has selected an ‘available’

queue manager, the bind option determines whether all messages are sent to that queue manager alone or distributed evenly among other queue managers.

The available values of *Bind Option* are:

O (*Open*), meaning 'stay' with this queue manager.

N (*Not fixed*), meaning that MQSeries will distribute the messages evenly across available queue managers. (I think option *O* should have been *F* for 'Fixed'.)

– *Cluster workload exit.*

The platforms on which clustering is supported are:

- MQSeries for AIX Version 5.1
- MQSeries for HP-UX Version 5.1
- MQSeries for OS/2 Warp Version 5.1
- MQSeries for OS/390 Version 2.1
- MQSeries for Sun Solaris Version 5.1
- MQSeries for Windows NT Version 5.1
- MQSeries for AS400 Version 5.1.

SUPPORTED PROTOCOLS

TCP/IP or LU 6.2 on any platform, NetBIOS or SPX on OS/2 or Windows NT, and UDP/IP on AIX

HOW TO SET UP A CLUSTERING ENVIRONMENT

We are now in a position to set up a cluster of queue managers (see Figure 1). Choose a stable platform for the queue manager that is to host a full inventory of clustered resources (the so-called *full repository*). It is a good idea to choose another queue manager to host a full repository that will act as a back-up to the first one.

Simply follow the steps below to build the first cluster (called *TEST_REPOS* in this example). This contains two OS/390 queue

managers called *QM1* and *QM2*. Commands can submitted on OS/390 via the MQSeries TSO ISPF admin SupportPac, either from MVS via the console or from other platforms using the RUNMQSC utility.

- Ensure that the cluster components are available. On OS/390, this requires the *MSTR* address space to be started with the sample cluster definitions in *CSQ4INSX*. On Version 5.1 on other platforms, the cluster components are automatically included. Note that, on OS/390, the *INDXTYPE* of the cluster's transmission queue is set to *CORRELID*, while that of the repository is set to *MSGID*.
- Alter both queue managers by adding the cluster attribute:

```
ALTER QMGR REPOS(TEST_REPOS)
```

- On *QM1*, define a cluster receiver channel that points to itself:

```
DEFINE CHANNEL(TO.QM1) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(xx.xx.xx.xx(1415)) CLUSTER(TEST_REPOS)
```

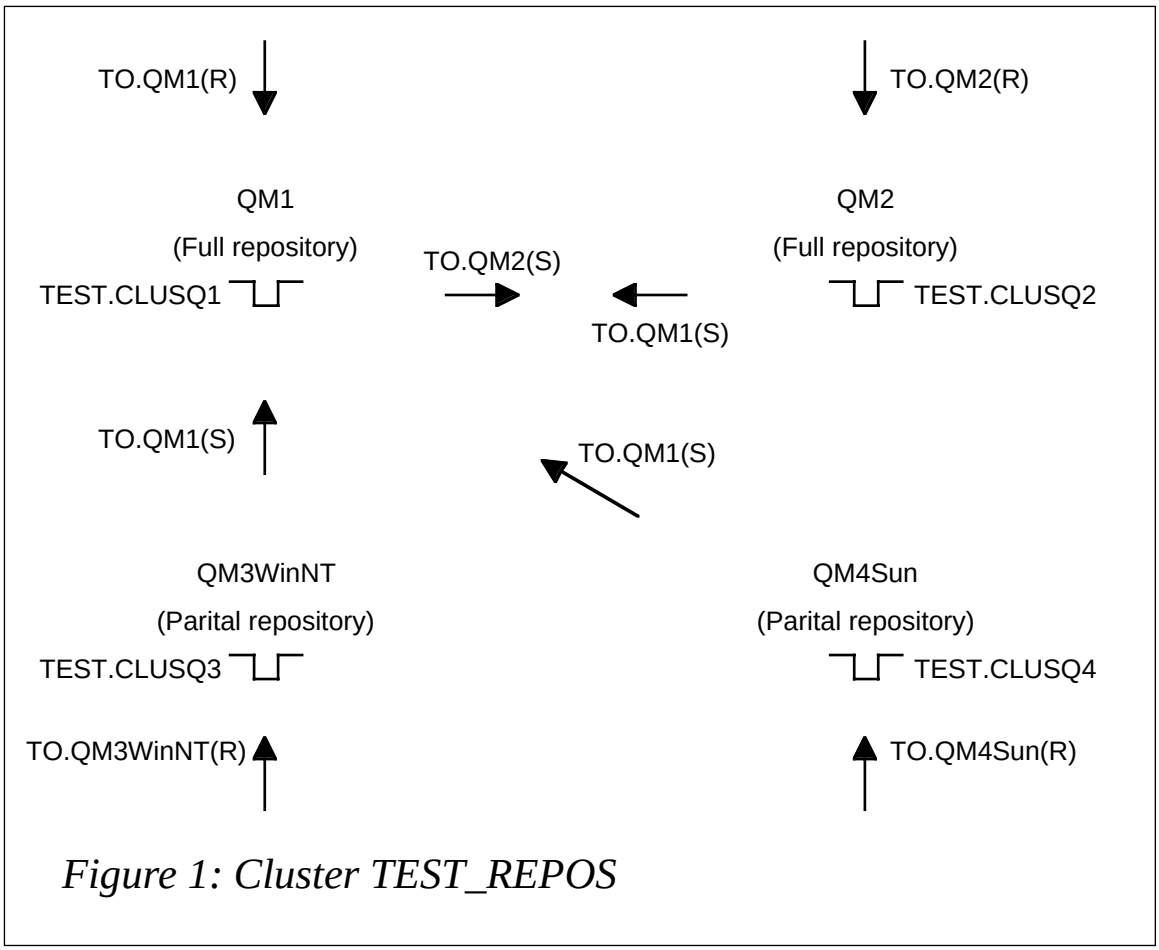

Figure 1: Cluster TEST_REPOS

xx.xx.xx.xx is the IP address and *1415* is the TCP/IP port number on which MQSeries listens. If you choose to use the default port number (*1414*), then it doesn't need to be specified, though I choose to do so to make documentation easier. You can also use DNS, though make sure it's active on all platforms where clustering is used.

- On *QM2* define a cluster receiver channel that points to itself:

```
DEFINE CHANNEL(TO.QM2) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(yy.yy.yy.yy(1416)) CLUSTER(TEST_REPOS)
```

- On *QM1* define a cluster sender channel that points to queue manager *QM2*:

```
DEFINE CHANNEL(TO.QM2) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(yy.yy.yy.yy(1416)) CLUSTER(TEST_REPOS)
```

- On *QM2* define a cluster sender channel that points to queue manager *QM1*:

```
DEFINE CHANNEL(TO.QM1) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(xx.xx.xx.xx(1415)) CLUSTER(TEST_REPOS)
```

Note that it is very important that the above definitions are made as this will ensure that both 'full repositories' remain in step. You can, of course use LU6.2/APPC instead of TCP/IP.

To test this initial set-up, define a cluster queue on *QM1*:

```
DEFINE QLOCAL(TEST.CLUSQ1) CLUSTER(TEST_REPOS)
```

and a cluster queue on *QM2*:

```
DEFINE QLOCAL(TEST.CLUSQ2) CLUSTER(TEST_REPOS)
```

Check that the queues are properly defined by issuing the following command to both queue managers:

```
DISPLAY QCLUSTER(*) ALL
```

Both queues should be visible.

From *QM2* write a message to cluster queue *TEST.CLUSQ1* and browse it from *QM1*. Again, notice that you cannot browse the cluster queue from *QM2*, even though you can both write to it and display it as a cluster queue.

To make the cluster more realistic, a third queue manager called *QM3WinNT* that resides on another platform (I'm sure you won't be surprised to hear the platform is Windows NT) should now be added using the following steps:

- Define a cluster receiver channel on *QM3WinNT* pointing to itself:

  ```
  DEFINE CHANNEL(TO.QM3WinNT) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
  CONNAME(zz.zz.zz.zz(1414)) CLUSTER(TEST_REPOS)
  ```

- Define a cluster sender channel on *QM3WinNT* that points to queue manager *QM1*:

  ```
  DEFINE CHANNEL(TO.QM1) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
  CONNAME(xx.xx.xx.xx(1415)) CLUSTER(TEST_REPOS)
  ```

  (You could have chosen to connect to *QM2* instead.)

  These are all the definitions you require. At this point the repository on *QM3WinNT* is 'partial' – it contains enough information to find a full repository and, thereby, the route to any cluster queue. Note that you have to issue the *DIS QMGR* command and check the *REPOS* and *REPOSNL* fields to establish whether your queue manager has a full or partial repository; if the fields are blank, the repository is partial, otherwise it's full.

- Check the set-up by issuing the following commands from *QM3WinNT*:

  ```
  DIS QCLUSTER(*) ALL
  ```

  The output of the above command should show *TEST.CLUSQ1* and *TEST.CLUSQ* 2.

  ```
  DIS CLUSQMGR(*) ALL
  ```

  The output should show *QM1* and *QM3WinNT* (it displays all cluster queue managers when issued from a full repository manager).

  Write a message using one of the sample programs:

  ```
  AMQSPUT TEST.CLUSQ2
  ```

  (if you want to be really adventurous, use the 'client' version *AMQSPUTC*) and browse it from *QM2*.

Note that, while the only connection we've built is between *QM3WinNT* and *QM1*, the cluster queue resides on *QM2*. MQSeries dynamically defines the cluster channels between *QM3WinNT* and *QM2* 'under the covers'. Definitions for these dynamic channels were obtained from the cluster receiver channel, whose *CONNAME* and *CONVERT* were used as a model. If the cluster receiver uses a DNS name, then the cluster sender that's defined automatically uses the same DNS name. This means that you need to ensure that DNS is active on that platform. To check whether your channel was manually or automatically defined, check the type of the channel: *CLUSSDRA* means it was defined automatically and *CLUSSDRB* means it was defined manually.

To test the 'return' route, define a cluster queue on *QM3WinNT* called *TEST.CLUSQ3*, and write a message to it from, say, *QM1*.

Finally, we add a queue manager called *QM4Sun* (on Sun Solaris) , with its cluster queue *TEST.CLUSQ4*, using the same method as for *QM3WinNT*.

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

Ruud van Zundert
Independent Consultant (UK)

Production Workflow Concepts and Techniques

During a recent visit to IBM Hursley, UK, a copy of *Production Workflow Concepts and Techniques* by Frank Leymann and Dieter Roller (Prentice Hall PTR, 2000, ISBN 0-13-021753-0) happened to fall into my briefcase. Leymann and Roller are two IBMers who were instrumental in the design and architecture of MQSeries Workflow (Leymann is the product's chief architect); both are based in Germany, and Leymann is also a professor at Stuttgart University, while Roller is a senior staff member at IBM Böblingen.

Despite being written by two IBMers, *Production Workflow* isn't about MQSeries Workflow (this is stated categorically in the *Preface*, and the first two screenshots in the book are, indeed, of products from rival companies); instead it's a hefty-ish tome (about 450 pages) on the topic of workflow – a kind of 'Everything you wanted to know about workflow but were afraid to ask in case you got an over-long and excessively technical answer'. The book is aimed at just about anyone who has a technical interest in workflow, from managers to programmers and system architects, and covers the subject pretty thoroughly and in substantial technical detail.

To some companies, the exercise of mapping business processes in preparation for the introduction of workflow software is a revelation. It's a surprise to many managers to discover that no record exists anywhere in the company of many of their key business processes, which are simply an 'oral tradition' in the department concerned. However, as *Production Workflow* demonstrates, workflow is more than just mapping computer systems to business processes. As with any other business activity, competitive advantage lies with those who best understand the methods involved and can apply them most appropriately to their own enterprise. *Production Workflow* can certainly help those implementing workflow software – or even just mapping the company's business processes – to understand what's involved and how workflow software could best benefit the company. The book discusses such topics as business process discovery, analysis, modelling, and re-engineering, as well as aspects of such topics as transactions and software engineering that are specifically relevant to building workflow systems.

Just occasionally the book gets a bit 'mathematically intense'. It's all very well saying that "the map $\kappa : N \cup (\wp(N) - \varnothing) \rightarrow \varepsilon \cup \{NOP\}$ is called a *compensation map*", but was it really necessary to raid so many alphabets just to construct that sentence? However, if you can wade through (or skip) the fifty or so pages that contain a scattering of $\exists$s, $\forall$s, and *iff*s, this is a very worthwhile read and an excellent text for all those seeking a deeper understanding of workflow.

System Engineer (UK)

CNT has added MQSeries support to its Enterprise/Access EAI product set. This is designed to speed up the integration of non-MQ-enabled mainframe and mid-range systems into the Enterprise/Access framework, freeing organizations from having to recreate business logic or adding code to legacy systems.

For further information contact:
Computer Network Technology Corp, 605 N Highway 169, Suite 800, Minneapolis, MN 55441, USA
Tel: +1 612 797 6000
Fax: +1 612 797 6800
Web: http://www.cnt.com

CNT International, c/o Network Solutions, 2 Langley Quay, Waterside Drive, Langley Slough, SL3 6EX, UK
Tel: +44 1753 792400
Fax: +44 1753 792499

* * *

New Era of Networks and IBM are working on ways to improve migration from MQSeries Integrator Version 1 to Version 2. Version 2 includes additional features, such as content-based publish/subscribe, transactions, and GUI-based management.

Separately, NEON announced its e-Business Adapter Development Kit (e-ADK) and Adapters are shipping for MQSeries Integrator V2. The set of tools and libraries are designed to accelerate and simplify the development of adapters. Pre-built Adapters and Accelerators include ones for XML, Oracle, and JD Edwards.

For further information contact:
NEON, 7400 East Orchard Road, Englewood, CO 80111, USA
Tel: +1 303 694 3933
Fax: +1 303 694 3885
Web: http://www.neonsoft.com

New Era of Networks Ltd, Aldermary House, 15 Queen Street, London EC4N 1TX, UK
Tel: + 44 171 329 4669
Fax:+ 44 171 329 4567

* * *

Willow Technology has announced versions of its MQSeries products for SCO OpenServer and UnixWare and for SGI's IRIX. Each product is an MQSeries V2-compatible server and interoperates with all MQSeries V1, V2, and V5 client and server products available from IBM and Willow.

Out now, MQSeries for SCO OpenServer and UnixWare starts at US$4,820 and US$9,440 for the SGI IRIX version.

For further information contact:
Willow Technologies Inc, PO Box 320005, Los Gatos, CA 95032, USA
Tel: +1 408 377 7292
Fax: +1 408 377 7293
Web: http://www.willowtech.com

* * *

IBM is to include Landmark Systems' monitoring products in its SystemPac for OS/390, including The Monitor for MQSeries.