33

MQ

update

March 2002

In this issue

Published by

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

North American office

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

Contributions

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

MQ Update on-line

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

Commissioning Editor

Peter Toogood
E-mail: PeterT@xephon.net

Managing Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.75) each including postage.

Copying messages selectively

When testing, there is sometimes a requirement to select specific messages from a stream of messages on an IBM MQSeries queue manager and copy them to another queue in order to process only that type of message. You might want to make a copy of the third, tenth, and twenty-seventh messages, for example, or test a particular path through a program, or examine a specific message flow with MQSeries Integrator V2.

MessageCopy is a program which will allow you to copy a message from an MQSeries queue manager queue to the same or another queue, one or more times. It is invoked as follows:

```
MessageCopy <from_qname> <to_qname> <qmanager> #messages #times
skip_count.
```

Where:

- *<from_qname>* is the name of the MQSeries queue manager queue from which you want to copy messages. This is the source queue. (This parameter is compulsory.)
- *<to_qname>* is the name of the MQSeries queue manager queue to which you want to copy messages. This is the destination queue. It can be the same as the source queue. (This parameter is compulsory.)
- *<qmanager>* is the name of the MQSeries queue manager on which the source and destination queue are located. (This parameter is compulsory.)
- *#messages* is the number of messages to be copied from the starting point: the default is 50.
- *#times* is the number of times each selected message should be copied: the default is one.
- *Skip_count* is the number of messages to skip before copying: the default is zero.

The source for MessageCopy is given below. It was written for MQSeries 5.2 to run on Windows 2000 but should run just as easily

on Windows NT. Porting to other platforms should be straightforward, although I haven't tried it.

MESSAGECOPY

```
 /*
*(c) Copyright IBM Corp. 2001
 *   MessageCopy is a C program to copy messages from one
 *   queue to the same or another queue
 *   Stops if there is an MQI completion code or MQCC_FAILED
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
 #include <cmqc.h>
 #include <time.h>
 #include <windows.h>
 void reportCounts();
 void leave(int reason);
/*   Some declarations.  Things are Global and so easier to share   */
 MQOD objDescInput = {MQOD_DEFAULT};   /* Object Descriptor for the
'from' queue */
 MQMD  msgDescInput = {MQMD_DEFAULT};  /* Message Descriptor for the
'from' queue */
 MQOD  objDescOuput = {MQOD_DEFAULT};  /* Object Descriptor for the 'to'
queue */
 MQMD  msgDescOutput = {MQMD_DEFAULT}; /* Message Descriptor for the
'to' queue */
 MQGMO  getMsgOptsOutput = {MQGMO_DEFAULT}; /* get message options  */
 MQPMO  putMsgOptsOutput = {MQPMO_DEFAULT};  * put message options  */
 MQHCONN  handleConnection;     /* connection handle               */
 MQHOBJ  handleObjectInput;     /* input object handle             */
 MQLONG  openOptionsInput;      /* input MQOPEN options            */
 MQHOBJ  Hobjo;                 /* output object handle            */
 MQLONG  openOptionsOutput;     /* output MQOPEN options           */
 MQLONG  closeOptions;          /* MQCLOSE options                 */
 MQLONG  compCode;              /* completion code                 */
 MQLONG  reason;                /* reason code                     */
 MQLONG  reasonCode;            /* reason code for MQCONN          */
 MQCHAR  *buffer;               /* message buffer                  */
 MQLONG  buffLen;               /* buffer length                   */
 MQLONG  msgLen;                /* message length received         */
 MQCHAR  QMName[40];            /* queue manager name              */
 MQLONG  i;                     /* loop counter                    */
 MQLONG  j;                     /* loop counter                    */
 MQLONG  totalMsgsPut =0 ;      /* Number of messages read         */
 MQLONG  copyCount;    /* Number of times message is to be repeated */
 MQLONG  skipCount=0;  /* Number of messages to skip before copying  */
 MQLONG  messagesToCopy=50;  /* Number of messages to be copied from the
'from' queue */
```

```
 MQLONG totalMsgsRead;       /* Number of messages read in total on the
'from' queue */
 MQCHAR8  firstPutTime;      /* Time of the first put                */
 MQCHAR8  lastPutTime;       /* Time of the last put                 */
 MQLONG   connected=0;       /* Used to indicate the MQCONN was OK   */
 MQLONG    openInput=0;      /* Used to indicate the MQOPEN for the
'source' Q was OK */
 MQLONG    openOutput=0;     /* Used to indicate the MQOPEN for the
'destination' Q was OK */
 int main(int argc, char **argv)
 {
  printf("Start of MessageCopy\n\n");
  if (argc < 4)
  {
    printf("Format is: MessageCopy from_queue_name to_queue_name
queue_manager #messages #times skip_count\n");
    printf("from_queue_name - name of the source queue *\n");
    printf("to_queue_name   - name of the target queue *\n");
    printf("queue_manager   - name of the queue manager containing the
queues *\n");
    printf("#messages       - the number of messages to be copied from
the starting point(default 50)\n");
    printf("#times          - number of times each selected message
should be copied(default 1)\n");
    printf("skip_count      - number of messages to skip before copying
(default 0)\n");
    printf("* marks a compulsory parameter\n");
    leave(99);
  }
  /*   Create object descriptor for 'from' queue     */
  strcpy(objDescInput.ObjectName, argv[1]);
  /*   Create object descriptor for 'to' queue       */
  strcpy(objDescOuput.ObjectName, argv[2]);
  /* Set Queue Manager name                          */
  if (argc > 3)
    strcpy(QMName, argv[3]);
  /* Determine the number of messages to copy from the source Q      */
  if (argc > 4)
    messagesToCopy = atoi(argv[4]);
  /*   Determine the number of times each message is to be copied    */
  if (argc > 5)
    copyCount = atoi(argv[5]);
  /*   Determine the number of messages to be skipped before copying */
  if (argc > 6)
    skipCount = atoi(argv[6]);
  printf("Run parameters are as follows:\n");
  printf("Copy %d messages\n",messagesToCopy);
  printf("Repeat each message %d times\n",copyCount);
  printf("Skip %d messages before starting to copy\n",skipCount);
      /*   Time to connect */
  MQCONN(QMName,
```

```
    &handleConnection,
    &compCode,
    &reasonCode);
  /* Check the outcome */
  if (compCode == MQCC_FAILED)
  {
    printf("MQCONN ended with reason code %ld\n", reasonCode);
    leave( reasonCode );
  }
  else
    connected = 1;
  /*   Open the source queue for input and in shared mode  */
  openOptionsInput = MQOO_INPUT_SHARED
              + MQOO_FAIL_IF_QUIESCING
              + MQOO_BROWSE;
   MQOPEN(handleConnection,
    &objDescInput,
    openOptionsInput,
    &handleObjectInput,
    &compCode,
    &reason);
  /* Check the outcome */
  if (compCode == MQCC_FAILED)
  {
    printf("MQOPEN for %s ended with reason code %ld\n",
objDescInput.ObjectName, reason);
     leave( reason );
  }
  else
     openInput = 1;
  /*   Open the destination queue for output */
  openOptionsOutput = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING;
  MQOPEN(handleConnection,
    &objDescOuput,
    openOptionsOutput,
    &Hobjo,
    &compCode,
    &reason);
  /* Check the outcome */
  if (compCode == MQCC_FAILED)
  {
    printf("MQOPEN for %s ended with reason code %ld\n",
objDescOuput.ObjectName, reason);
    leave( reason );
  }
  else
    openOutput = 1;
  /*   Prepare to read a message from the source queue   */
  getMsgOptsOutput.Version = MQGMO_VERSION_2;
  getMsgOptsOutput.MatchOptions = MQMO_NONE;
  getMsgOptsOutput.Options = MQGMO_WAIT;
```

```
  getMsgOptsOutput.WaitInterval = 100;
  getMsgOptsOutput.Options += MQGMO_BROWSE_NEXT ;
  /* Alternatively we could wait for ever by using
      * getMsgOptsOutput.WaitInterval = MQWI_UNLIMITED;
   */
  msgDescInput.Encoding       = MQENC_NATIVE;
  msgDescInput.CodedCharSetId = MQCCSI_Q_MGR;
  /* Allocate a buffer to hold the message */
  buffLen = 419404;
  buffer = malloc(419404);
  if (buffer == 0 )
  {
    printf("Failure in malloc for message buffer\n");
    leave(99);
  }
  /*   Skip over the first 'n' messages if required  */
  for (i=0; i <skipCount ; i++)
  {
    MQGET(handleConnection,
        handleObjectInput,
        &msgDescInput,
         &getMsgOptsOutput,
        buffLen,
         buffer,
         &msgLen,
         &compCode,
         &reason);
    /* Check the outcome */
    if (compCode == MQCC_FAILED)
    {
      printf("MQGET for %s ended with reason code %ld\n",
objDescInput.ObjectName, reason);
      leave( reason );
    }
    totalMsgsRead += 1;
    /*
     * Set the MessageID and CorrelID to 0 in order
     * to ensure that we see all the messages
    memcpy(msgDescInput.MsgId, MQMI_NONE, sizeof(msgDescInput.MsgId));
    memcpy(msgDescInput.CorrelId, MQCI_NONE,
sizeof(msgDescInput.CorrelId));
  }
  i=0;
  while ( i < messagesToCopy )
  {
    /* Get a message to copy */
       MQGET(handleConnection,
      handleObjectInput,
      &msgDescInput,
      &getMsgOptsOutput,
      buffLen,
```

```
      buffer,
      &msgLen,
      &compCode,
      &reason);
    /* Check the outcome */
    if (reason == MQRC_NO_MSG_AVAILABLE)
    {
       printf("End of source queue reached, terminating...\n");
       leave(Ø);
    }
    else if (compCode == MQCC_FAILED)
      {
        printf("MQGET for %s ended with reason code %ld\n",
objDescInput.ObjectName, reason);
        leave( reason );
      }
    totalMsgsRead += 1;
    /* Write it the required number of times */
    for (j = Ø; j < copyCount; j++)
    {
      MQPUT(handleConnection,
        Hobjo,
        &msgDescInput,
        &putMsgOptsOutput,
        msgLen,
        buffer,
        &compCode,
        &reason);
      /* Check the outcome */
      if (compCode == MQCC_FAILED)
      {
        printf("MQPUT for %s ended with reason code %ld\n",
objDescOuput.ObjectName, reason);
        leave( reason );
      }
       /* Record the time of the first PUT if no messages put so far
        * otherwise save it as the last time to date.
        */
      if (totalMsgsPut == Ø)
      {
        memcpy(firstPutTime, msgDescInput.PutTime,
sizeof(msgDescInput.PutTime));
        memcpy(lastPutTime, msgDescInput.PutTime,
sizeof(msgDescInput.PutTime));
      }
      else
        memcpy(lastPutTime, msgDescInput.PutTime,
sizeof(msgDescInput.PutTime));
      totalMsgsPut++;   /* Increment the count */
    }
    i++;
```

```
  }
  /* Time to leave */
  leave(0);
 }
 /* reportCounts is a function to report
  * the counts and times of the messages which were
  * read and written
  */
 void reportCounts()
 {
  char formattedTime[30];
  printf("\nResults are as follows:\n");
  printf("Read  a total of %i messages (including skipped
messages)\n",totalMsgsRead);
  printf("Wrote a total of %i messages\n",totalMsgsPut);
      if (totalMsgsPut)
  {
    sprintf(formattedTime,"%0.2s Hr %0.2s Min %0.2s.%0.2s Secs",
firstPutTime, firstPutTime+2, firstPutTime+4, firstPutTime+6);
    printf("First message written at  %s\n", formattedTime);
    sprintf(formattedTime,"%0.2s Hr %0.2s Min %0.2s.%0.2s Secs",
lastPutTime, lastPutTime+2, lastPutTime+4, lastPutTime+6);
    printf("Last  message written at  %s\n", formattedTime);
  }
 }
 /* leave is a common exit point
  *
  * Function will close all queues and
  * disconnect from the queue manager
  */
 void leave(int rc)
 {
  reportCounts();
     if (rc)
    printf("MessageCopy exiting with return code %ld\n",rc);
  /*  Close the source queue (if it was opened) */
  if (openInput)
  {
    closeOptions = 0;
    MQCLOSE(handleConnection,
        &handleObjectInput,
        closeOptions,
        &compCode,
         &reason);
    /* Check the outcome */
    if (reason != MQRC_NONE)
       printf("MQCLOSE for %s ended with reason code %ld\n",
objDescInput.ObjectName, reason);
  }
    /*   Close the destination queue (if it was opened) */
  if (openOutput)
```

```
  {
    closeOptions = 0;
     MQCLOSE(handleConnection,
          &Hobjo,
         closeOptions,
         &compCode,
          &reason);
    /* Check the outcome */
    if (reason != MQRC_NONE)
      printf("MQCLOSE for %s ended with reason code %ld\n",
objDescOuput.ObjectName, reason);
  }
  /*   Disconnect if we connected in the first place */
  if(connected)
  {
      MQDISC(&handleConnection,
          &compCode,
        &reason);
    /* Check the outcome */
    if (reason != MQRC_NONE)
        printf("MQDISC ended with reason code %ld\n", reason);
  }
   /* If a reason code was returned from the close or disconnect
    * use that to exit the program with, otherwise use the value passed
    * into the function
    */
  if (reason)
    rc=reason;
  printf("\nEnd of MessageCopy\n");
  exit(rc);
}
```

A known limitation with MessageCopy is that it does not preserve the context of the original message when copying to the destination queue. This is because MessageCopy reads messages on the source queue with ‘browse’ in order to preserve the original message. This means that the following fields are not maintained from the original message: Put date and time; UserID under which the message was put; and the name of the application which put the original message. If maintaining these fields is important then one approach would be to modify the program to issue an MQGET without the browse option. The problem in doing this is that the original message is lost and the sequencing on the source queue is changed.

As coded, the maximum message size is 4MB. If you need to process larger messages you will need to increase the size of the buffer.

Here is an example of MessageCopy output:

```
MessageCopy QUEUE1 QUEUE2  BROKER1  12 1Ø 5Ø
Start of MessageCopy
Run parameters are as follows:
Copy 12 messages
Repeat each message 1Ø times
Skip 5Ø messages before starting to copy
Results are as follows:
Read a total of 62 messages (including skipped messages)
Copied a total of 12Ø messages
First message written at  14 Hr 43 Min 26.88 Secs
Last  message written at  14 Hr 43 Min 26.91 Secs
End of MessageCopy
```

Tim Dunn, Software Engineer
IBM Hursley (UK)

Connecting tightly coupled legacy to loosely coupled applications: part 2

THE END-TO-END DESIGN

Last month we discussed connecting an environment based on two-phase commit (2PC) to one based on message queueing (MQ). In this article we look at the end-to-end design of such a system.

Security considerations will not be included here; the scope is too large.

To allow the connection of the synchronous and asynchronous worlds we need a method which:

- Reduces the costs and timescales of integrating applications.
- Is generic enough to enable re-use across applications.
- Ensures minimal human intervention (business and IT support).
- Utilizes the integration broker service.
- Minimizes development effort in the legacy world.
- Is reliable and, if required, supports 24 hour operation.

- Is scalable to meet the demands of an expanding business.
- Supports both read and update processing.

Ideally, the design should be based on a service-oriented component architecture and be event-driven so that any application can be exploited to build a business process.

From the perspective of the initiator of a request for an application service, there is a need to consider each direction of communication separately.

DESIGN SCENARIOS

There are likely to be several permutations of the connectivity options available for tightly coupled and loosely coupled environments. The two we will focus on are:

- New message-based business applications connecting to the synchronous legacy applications.
- Legacy applications connecting to business functions that have been migrated to the new 'infrastructure' and now need to connect via messaging.

Each of these scenarios poses challenges that can be surmounted with good architectural design. The second option is the more difficult, since the goal is to be non-invasive.

Infrastructure design

The basic building blocks we identified last month are:

- 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 requirements, eg SAP access from the new business applications.)
- Messaging technology that is XA-compliant.
- A communications protocol technology (eg LU6.2, OSI/TP) which supports syncpoint coordination.

- A standard message header.

It is assumed that the network underpinning the design will be robust with alternative path strategies to cater for outages, and that invoking the back-up alternatives takes a matter of minutes in the worst case.

For simplicity we will refer to an application rather than transactions, of which there will be several making up the application.

Legacy to new business application update

The topology in Figure 1 is representative of the basic end-to-end design that is required to support updates to the new business application initiated by the legacy application. Note that the new business application is deemed the 'master database' and needs to reflect the true position of the business – even though the secondary database may be up-to-date on the data it shares with the new business application.

MQSeries provides a decoupled solution where messages can be secured within an intermediate transaction manager gateway database (a queue). In this scenario data for applications on transaction manager A will be sent via MQSeries. The intermediate transaction manager gateway will hold a logical unit of work (UOW) with the application on transaction manager B.

Software locks will be retained by the new business application until such time that transaction manager gateway B indicates they can be removed. This will in fact be done at the application level, since the gateway application intercepts system-level commit instructions and signals an application-level alternative to the new business application.

The commit from transaction manager A and the response from the intermediate gateway transaction manager will ensure the legacy database is updated before the new business database.

The initial conversations will be over an MQSeries Client connection (see *Exception handling* below) with a final commit message to request removal of the soft locks issued over an MQSeries server to the MQSeries server connection. This ensures the 'unlock' message will be delivered and the master database updated even if the network goes down for a short time.

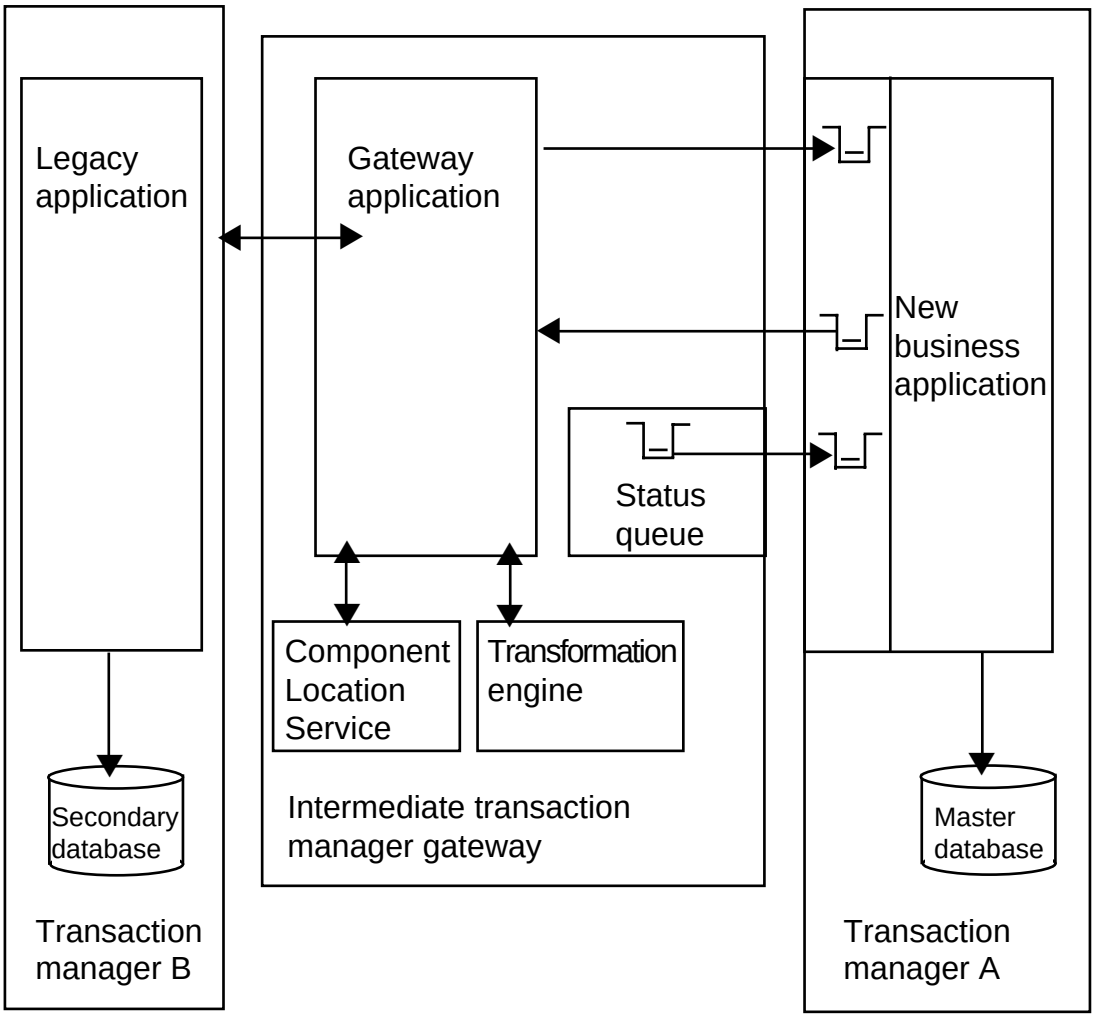


Figure 1: Update from legacy

The 2PC transaction identifier will flow between the legacy application and the new business application for the duration of the conversation – required for the soft lock mechanism previously described. This will ensure only actions relevant to the transaction will take place, thereby preserving its atomicity.

The gateway application will determine from the Component Location Service (CLS) which queue manager/queue will receive the transaction.

The application running under transaction manager B will send an update request to the gateway application on the intermediate transaction manager gateway, and several conversations may need to

flow within the UOW. The UOW cannot be committed straight away because a response is needed from the new business application running under transaction manager A.

If the message is produced within the UOW it will not be released to transaction manager A. To release the message the MQ calls will be issued outside the syncpoint of the main UOW. The response to the request is read under the syncpoint of the main UOW.

The gateway application uses an MQSeries client (ie synchronous connection) that provides an immediate response indicating success or failure to write to a queue on transaction manager A. It will also request the queue manager to create a permanent dynamic queue to receive the response.

Any enquiry requirement from legacy to new business will also use the topology detailed in Figure 1.

New business application update to legacy

In some situations the new business application may be required to connect to a legacy application that has not been migrated to the new, loosely coupled architecture. The business wants to see legacy applications continuing to provide business services in an event-driven manner.

The topology in Figure 2 is representative of the basic end-to-end design that is required to support updates from the MQ-oriented new business application, which is outwardly very similar to the legacy-to-new-business-application scenario described above.

Updates from the new business application will use MQSeries Server to MQSeries Server to communicate: MQ Client could be used although it introduces a synchronous component that is not necessary when the ‘master’ initiates the update request.

As transaction manager A is the master, the locks provided by the database management will be removed on writing the message to the queue and the response from legacy will be used solely as an audit trail for reconciliation purposes.

The gateway application will hold the legacy transaction open, handle all the conversation interactions, and map these to MQSeries messages

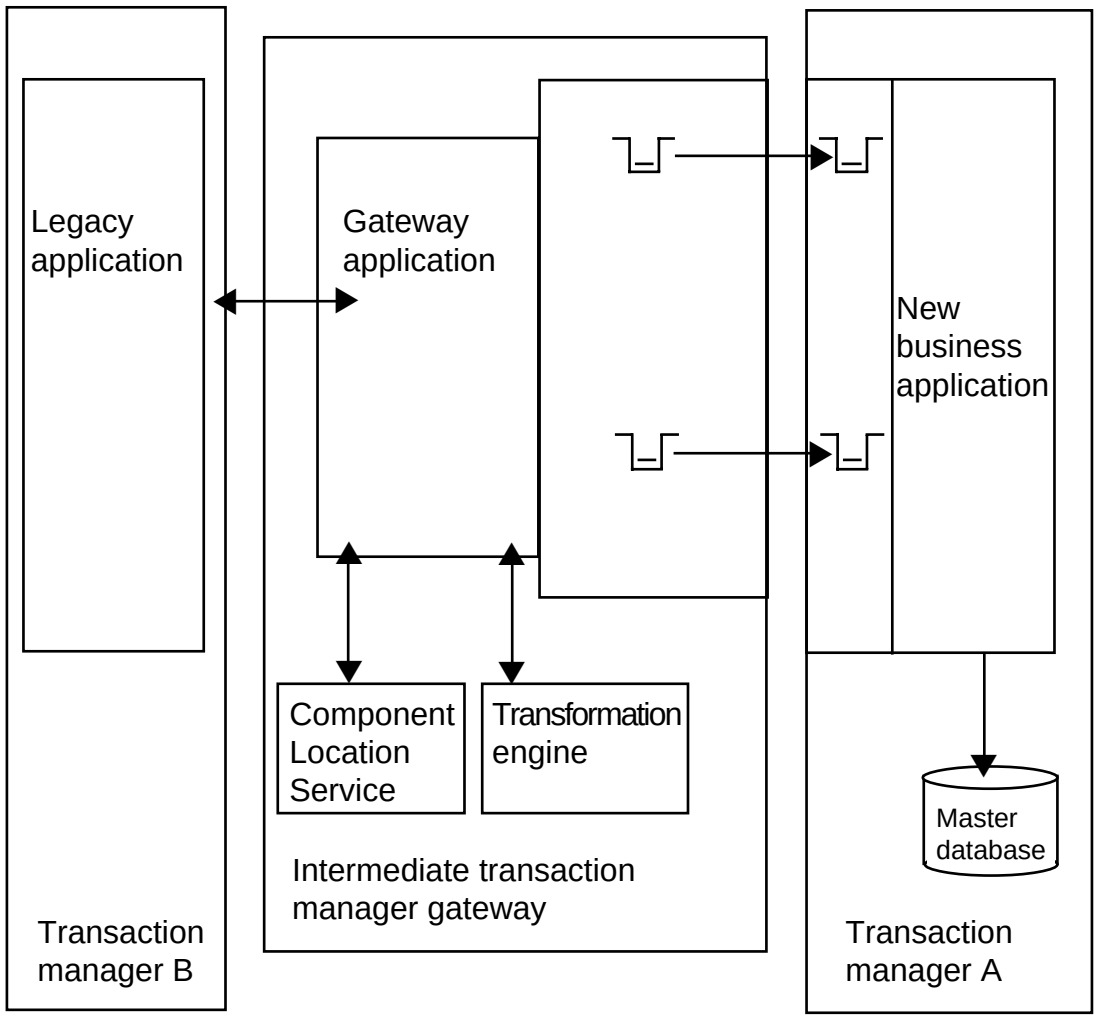


Figure 2: Update to legacy

request/replies using the correlation identifier. The CLS will contain the appropriate message to UOW/conversation mappings.

If the new business application decides to abort the update then transaction manager A will rollback the updates to the database and queue. Any enquiry requirement from a new business application to legacy will also use the topology detailed in Figure 2.

INFRASTRUCTURE CONSIDERATIONS

There are two key options to consider when interfacing with MQ:

- Persistent messages: normally used for updates – they survive system restarts but response times are affected.
- Non-persistent messages: normally used for enquiries – they do not survive system restarts.

If possible, the non-persistent option should be used for at least the MQ client-related interactions.

Care needs to be taken to ensure that the application level protocol details have been captured – these may be the most difficult to process. A decision will need to be taken as to where to handle these protocols; there are two options:

- Within the integration broker.
- Within the middleware infrastructure application (gateway).

A Component Location Service needs to be developed to map target systems to physical routing attributes – LDAP is a possibility here if the required look-up performance can be achieved.

It is also beneficial to develop a higher level MQ API (MQ Bridge) for the new business applications to shield them from the complexities of the underlying infrastructure APIs. The MQ Bridge will enable services to be built-in and configured independently of the application utilizing the bridge.

The gateway application will make the routing decisions by reference to the Component Location Service. This will enable the scaling up of the services as required – it is here that a form of workload management can be introduced to spread the load across several hosts.

An MQ Bridge supporting the new business application running under transaction manager A, as shown in Figure 2, can also make routing decisions by referring to the Component Location Service (a local version will be required). The MQ Bridge will consult the Component Location Service using a token submitted by the application, which provides an abstracted reference to the target application. Additional information will flow in a standard message header from the MQ Bridge to the gateway application to enable it to consult the Component Location Service for additional routing information (eg support for multiple-target platforms).

The legacy applications will continue to have hard-coded configuration data.

Just as with the tightly coupled model, the application may be communicating success or failure in user mode, so the interactions with MQ may not necessarily be fully transactional.

The transaction manager will have to provide the unique identifier of the data sub-set on which the applications work to enable identification of each data flow, which will in turn need to be conveyed over the messaging infrastructure to the message-based application. It will assist in reconciling message to conversation and for exception handling purposes.

There will be large numbers of services running simultaneously and all maintaining locks within the resource managers so care will be required when sizing and configuring the hardware they will run on.

The key area of exception handling is discussed later in this article.

An alert should always be raised to the systems management sub-system when exceptions occur.

The systems management sub-system will be aware of remote queue manager outages through regular testing of queue manager link availability; this will ensure detection when there is no application-level activity.

BUSINESS APPLICATION DESIGN ISSUES

When delivering business functionality in the new system you need to decide whether the design should comprise:

- several discrete transactions, or
- a single transaction which co-ordinates with other transactions: this option gives the impression of a single transaction to the caller.

It is also necessary to ensure that the applications are aware of progress through the infrastructure to a point when they can be assured messages are deemed as ‘committed’ and therefore delivered from the

sender's viewpoint. As stated previously, to achieve this you need to determine which is the master and which the secondary.

Depending upon which end initiated the transaction the following will inform the application of a 'good' state.

- For the application driving the syncpoint protocol there will be:
  - a 'good' return code to the two-phase commit request.
- For the application using queues there will be:
  - a 'good' return code to the request to put a message on a queue. Additionally, the application can request confirmation of delivery and confirmation of arrival messages from the remote queue manager.

Routing decisions will be made primarily by the infrastructure; however, the application will have to provide some routing token to enable mapping from an abstract relationship to a physical delivery address held within the Component Location Service.

EXCEPTION HANDLING

Just as with the pure, tightly coupled model, the update transaction program may be communicating success or failure in user mode, so update interactions via MQ may not have to be fully transactional.

Unlike pure transactional communications, which may have rollback initiated by either party, the asynchronous world has to have a different strategy. A key design issue is the recovery strategy for which there are three options:

- Compensating transactions.
- Manual unpicking.
- Forward recovery.

The first is difficult to achieve, as testified by industry consensus, and is probably best avoided. The preference is for either the second option, which is self-explanatory, or the third option, which forces the transaction to be completed and will introduce additional middleware infrastructure recovery and business recovery applications at the target.

It is assumed that the applications will run on clustered systems and that recovery to the alternative platform will be a matter of seconds for the key components.

Recovery of update failures from new business applications

For recovery of update failures to a legacy environment, if a message cannot be delivered the gateway application will put a message on a recovery queue to trigger a scavenger application, as shown in Figure 3. For messages stuck in transit at the intermediate server because of a network fault, for example, the scavenger application will attempt to resend the data over a syncpoint connection to the legacy application when the network has recovered.

Timeout values are required and they should be set to reflect:

- The business process needs – static values.
- System management failure notifications – dynamic values.

A review of the systems management and business processes will be required to ensure they can cope with the changing circumstances.

A new application within the legacy system will process the message issued by the scavenger according to the existing processing rules and any enhancements to the business process.

There may be a need to enhance the existing legacy application to save the ‘state’ at the time of the abort, so that the new legacy application can determine what course of action to take in support of the business process, which may be a request for human intervention.

Recovery of update failures from legacy

For recovery of failures to a new business application-related update, failure to write to a queue will result in the legacy UOW being aborted and all its potential updates being rolled back.

If this is not the first leg of the update the gateway application will insert a failure status message on the local status queue, which will signify to the new business application that the associated UOW was aborted. The application will tidy up any outstanding locks and roll back any data updates.

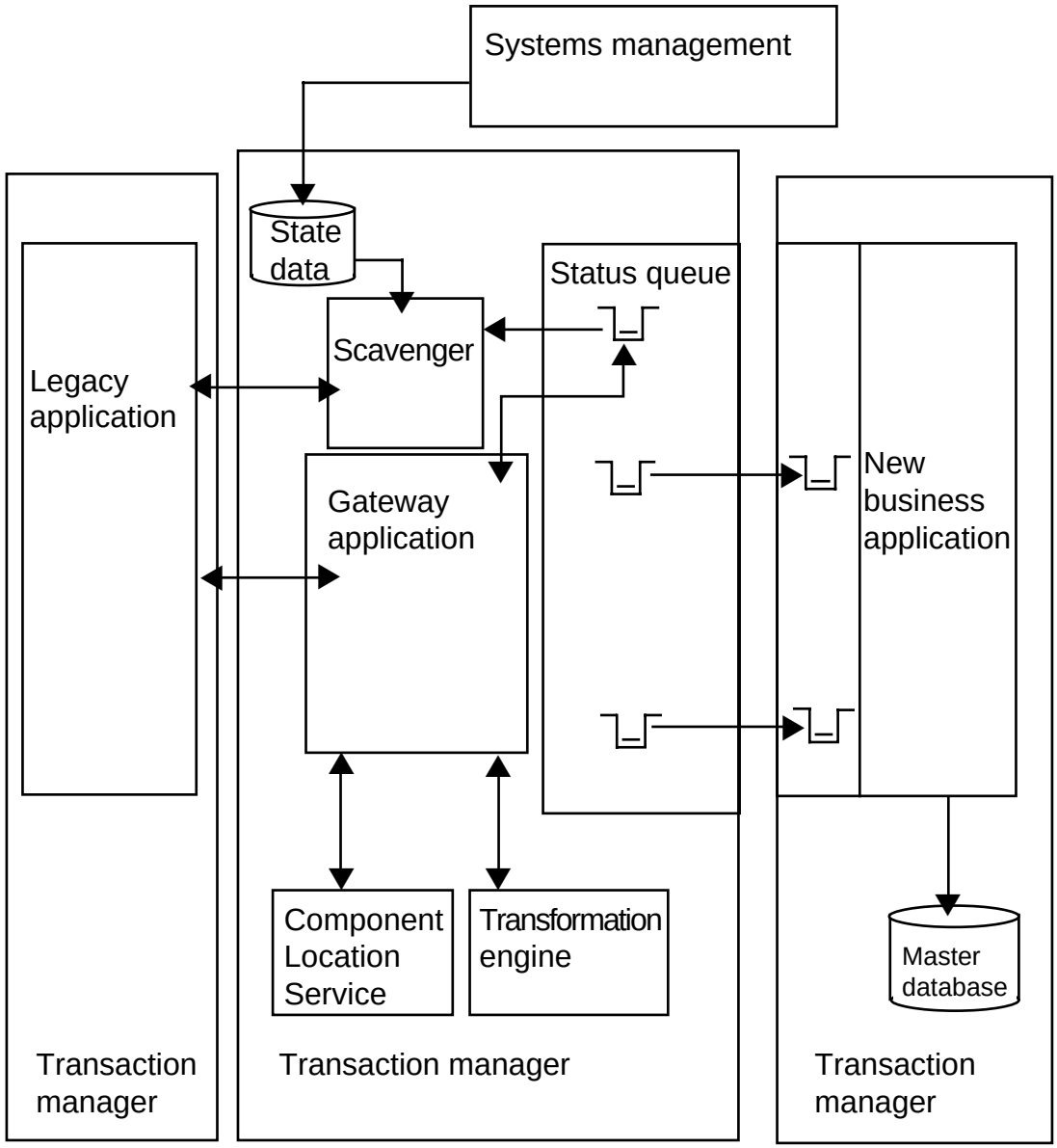


Figure 3: Recovery of updates from a new business application

If it is the last leg of the update, ie the update has been signalled as successful by the legacy application, an update commit status message is written to the local status queue. This is subsequently processed by the new business application, which ensures the update takes place and all locks are removed.

In the event of a 'final commit' being delayed by the transaction, which could be the last in a chain as shown in Figure 4, the locks are left in place and a recovery message put on a queue to invoke a scavenger application.

The scavenger application will wait for the arrival of the message following reinstatement of the network and will attempt to update the database and remove any outstanding locks. Again, timeout values need to be set to reflect:

- The business process needs – static values.
- System management failure notifications – dynamic values.

Systems management and business processes must also be reviewed again to ensure they can cope with the changing circumstances.

The gateway application will communicate with the new business application using a synchronous connection. This will ensure that if the legacy application aborts the UOW the gateway application will delete the dynamic reply queue and put a failure message on the status queue, as previously discussed.

Recovery of enquiry failures from legacy or new business applications

The 'read' capability need not be handled with anywhere near the elegance required for updates. If the message or transaction cannot be delivered the application will timeout and issue a failure alert, probably to an end-user screen and to systems management.

Within the MQSeries world the 'message expiry value' will have to be set to reflect the application timeout and the message will have disappeared; this is necessary because of the resilience capabilities of MQSeries.

MQ application rollbacks

If the new business application fails, a rollback will be issued for a persistent message, which will result in the message being left on the queue. The MQ application will handle this scenario by allowing only a limited number of retries before moving the message to a 'poison message' queue and returning an error code to the calling program.

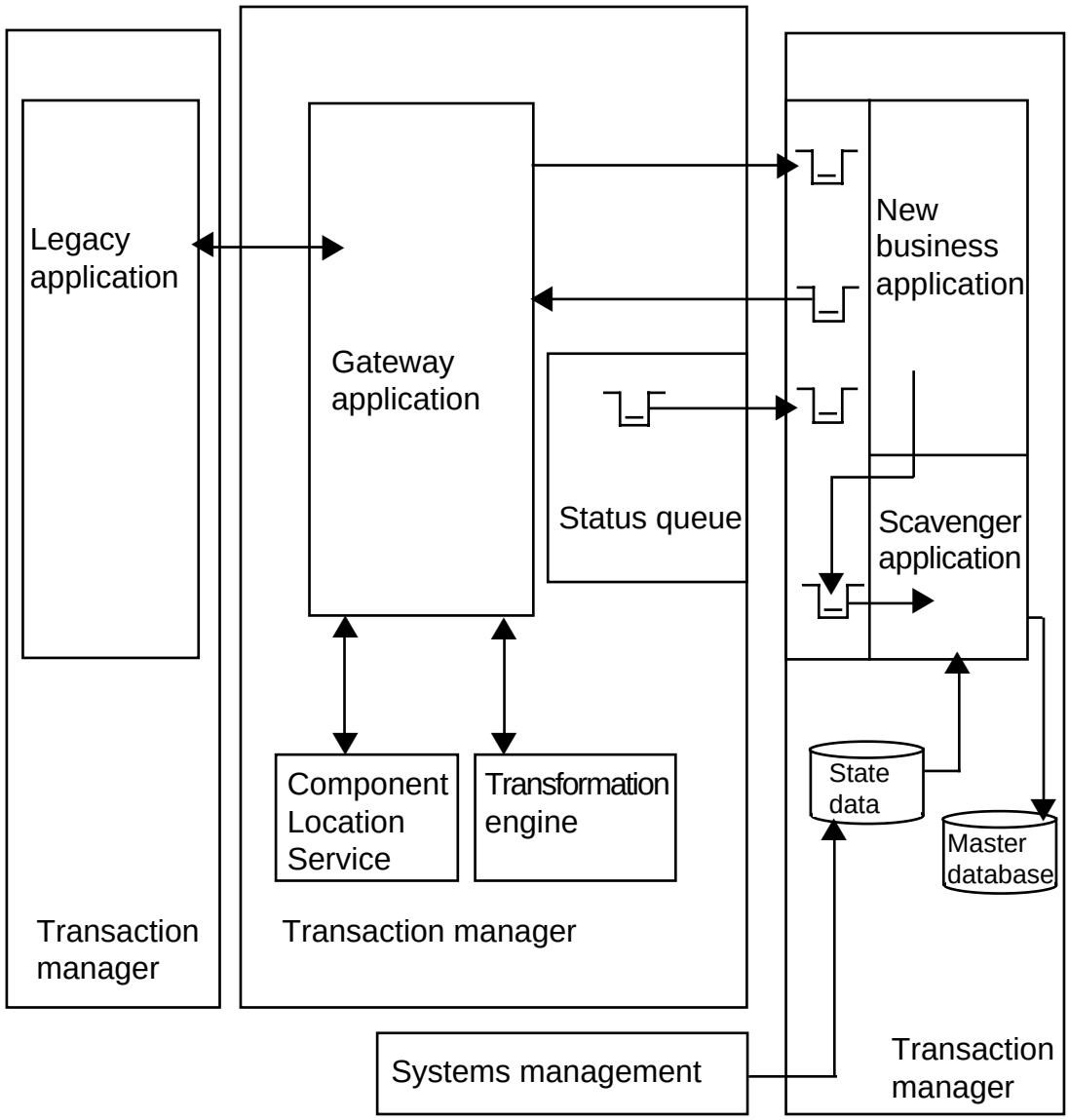


Figure 4: Recovery of update failures from legacy

Supporting systems

The systems management system will be monitoring for a number of conditions:

- Messages stuck on a queues.
- Network failures.

- Server failures.
- Transaction failures.

When it detects a failure state a ‘rule’ should process the ‘state’ and issue an alert to both the new business application/user and the scavenger program. These programs will adjust their behaviour according to the message content.

SUMMARY

The task of linking tightly and loosely coupled systems whilst ensuring the smooth adoption of new architecture is challenging, but it can be done.

Business processes must be reviewed to determine the level of impact exceptions will have and what course of action is required to handle them. It is inevitable that human intervention will be required to recover from exception conditions, but this can be minimized.

The two-phase protocol with syncpoint is complex and requires specialist support, as do the application level protocols, where knowledge has been lost over time.

Nothing can be introduced without some ‘pain’ and inevitably there is development required both within the middleware infrastructure and the existing legacy environment.

There are business benefits in moving towards the loosely coupled environment and readers will no doubt be aware of them; this article highlights an approach that goes some way to ensuring these benefits are achieved.

Mick Broderick
Solutions Architect (UK)

MQ back-up and recovery on open system platforms

This article discusses back-up and recovery of MQSeries queue managers on open systems platforms, ie Unix and NT. Most of the points made also apply to MQSeries on platforms using a common code base such as OpenVMS. Although MQSeries for AS/400 has about 75% of its code in common with the Unix and NT versions, its logging and recovery uses the AS/400's transactional file system, so most of what is said here does not apply to this platform.

We will look at how MQSeries logging works, the differences between circular and linear logging, and how the logs are used for restart and media recovery.

From the first 'real' version of MQSeries – V2.0 – transactional integrity was a key feature. MQSeries provides the ability to recover the state of its queues to a consistent point following a failure (of an application, the queue manager, or the machine). The guarantee that persistent messages would be delivered only once distinguished MQSeries from other rival products and helped earn MQSeries its reputation for reliability.

Transactional integrity requires that all updates to queues involving persistent messages be recorded in non-volatile storage. This transactional capability is very similar to that offered by database management systems (DBMSs) and transaction monitors, such as CICS, and the methods used to provide it are also very similar. Like a DBMS, an MQSeries queue manager stores information about the resources it controls (ie its queues) in two parts:

- A snapshot, which is periodically updated.
- A list of changes since the last snap-shot.

For a DBMS, the snapshots are known as table spaces and the list of changes as the 'redo' log. For MQSeries they are known as object files and log files.

The similarity between MQSeries logging and that performed by DBMS products is more than coincidental. Version 2.0 of MQSeries

reused the transaction logging code developed for DB2. As a messaging product MQSeries is designed to hold only transient data (eg messages waiting delivery). Therefore, a number of facilities available on DB2 were seen as being unnecessary and were removed. These facilities included point-of-time recovery and tools to examine logs to provide activity audits.

The snapshot, known as the table space in a DBMS, is called the object data in MQSeries. This is stored as a series of files, one for each queue and process defined to the queue manager. The typical location for the file for a queue named *SYSTEM.DEFAULT.LOCAL.QUEUE* on a Unix system for a queue manager called qmgrA is: */var/mqm/qmgrs/qmgrA/queues/SYSTEM!DEFAULT!LOCAL!QUEUE.*

Notice the 'character folding', ie the substitution of certain characters in the object name by other characters in the related file name. This is required due to differences in the permitted character set in an MQSeries object and a Unix or NT file name.

The equivalent location for the file for a process named *SYSTEM.DEFAULT.LOCAL.PROCESS* would be: */var/mqm/qmgrs/qmgrA/procdef/SYSTEM!DEFAULT!LOCAL!PROCESS.*

The log files, known as the transactional log or redo log in a DBMS, keep a record of changes to the objects.

Together, the object files and log files are used to store a record of queue manager changes (ie configuration changes) and queue updates (ie puts and gets). This information can then be used to bring the queue manager back to a consistent state following a system crash. As part of this recovery all persistent messages will be recovered, provided the unit of work to which any operations belong is complete. As a transactional product MQSeries provides options in its API which allow the units of work to which queue operations belong to be controlled.

CIRCULAR AND LINEAR LOGGING

One of the first choices to be made when creating a queue manager on a Unix or NT system is that between circular and linear logging. Using circular logging the log files are linked into a ring and old log files that are no longer needed are reused automatically. If more log space is

needed than is available in the original ring of primary log files the ring will be increased by creating more log files up to the secondary log file limit defined when the queue manager was created.

For linear logging the log files form a continuous sequence with new files created as they are needed. Thus, the number of log files continuously increases unless files that are no longer needed for recovery are deleted. To facilitate this house-keeping task, if linear logging is used, messages appear in an error log to indicate which log files are no longer needed for restart and media recovery. There is also a support pack available which finds these messages and automatically deletes or archives old log files.

Restart and media recovery

An important difference between the circular and linear logging is the level of recovery that is provided. Circular logs provide only restart recovery; linear logs provide both restart and media recovery.

Restart recovery

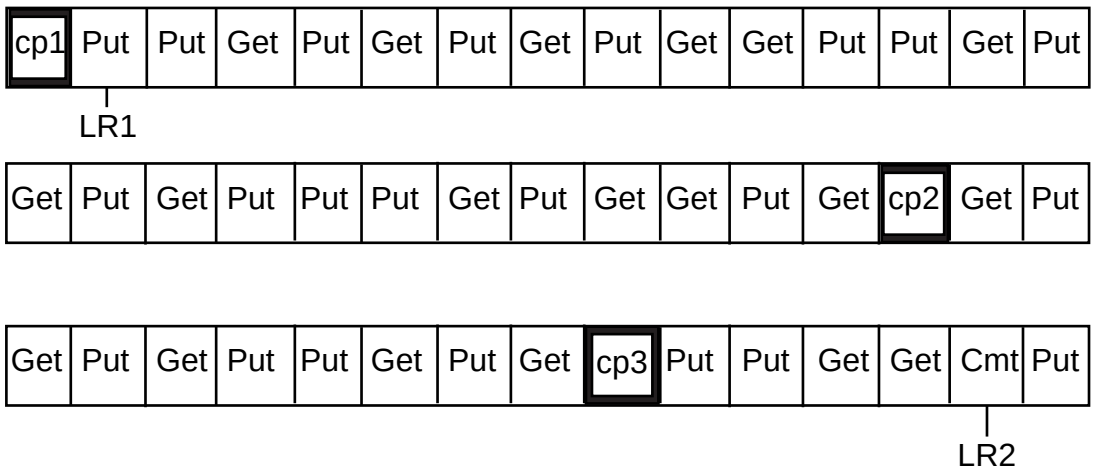
Restart recovery is the restoration of a queue manager to the state before a crash using both log files and object files. This requires the following log file entries (see Figure 1):

- Most recent checkpoint.
- All entries after the most recent checkpoint.
- All the records for any uncompleted transactions.

A checkpoint is a point in time when the object file data is up-to-date (ie it matches the data in the log). Transactions are written first to the log files and only later copied to the object files. Checkpoints are created periodically by updating the object files from the transaction logs. This reduces the number of records needed for recovery, thereby saving log space and reducing the time to restart following a crash.

Checkpoints are generated automatically by the queue manager:

- When the queue manager starts.
- When logging space is running low.
- Every 1,000 operations logged.



Notes:

1 cp 1, 2, and 3 are checkpoints.

2 Until the transaction is completed (LR2) with an MQCMIT the record LR1 is needed for restart recovery.

3 Without record LR1 the queue manager will not restart.

4 The first log record for a long-running transaction may be before the most recent check-point.

Figure 1: Log records required for restart recovery

When a transaction is started a log record is created. Each new operation relating to the same unit of work causes a new record to be written to the log. Finally, when the transaction is completed a completion record is written to the log. The transaction may be completed by being committed, using MQCMIT, or backed-out using MQBACK because of the failure of the program performing the transaction.

Until the transaction is completed all the records relating to the transaction are needed for restart recovery. There can be a long time and many megabytes of log space between the start and completion of a transaction. If an application were to perform an MQPUT (with SYNC-POINT) then sleep for a month before doing the MQCMIT the transaction will last for a month. Similarly, if a transaction performs a large number of operations under a single UOW (eg copying the

contents of a large file to a queue as part of a file transfer operation) a large amount of log space will be used by the transaction. These are examples of long-running transactions. In extreme cases a long-running transaction can prevent the space being released, causing the log files to fill up. New transactions will then fail and the error message AMQ7465 will be generated.

The queue manager will not restart until all the records relating to all uncompleted transactions are available. If a transaction that was started a month ago has not completed, the original log record will be in a log file that is a month old. This log file will be needed to restart the queue manager. Although this outstanding transaction will be backed out anyway (because it is not committed) it will prevent the queue manager being restarted unless all the log files containing all its records are available. There is no facility in MQSeries to ignore an outstanding long-running transaction in order to recover other, more recent transactions. This is particularly annoying because incomplete transactions will be rolled back anyway.

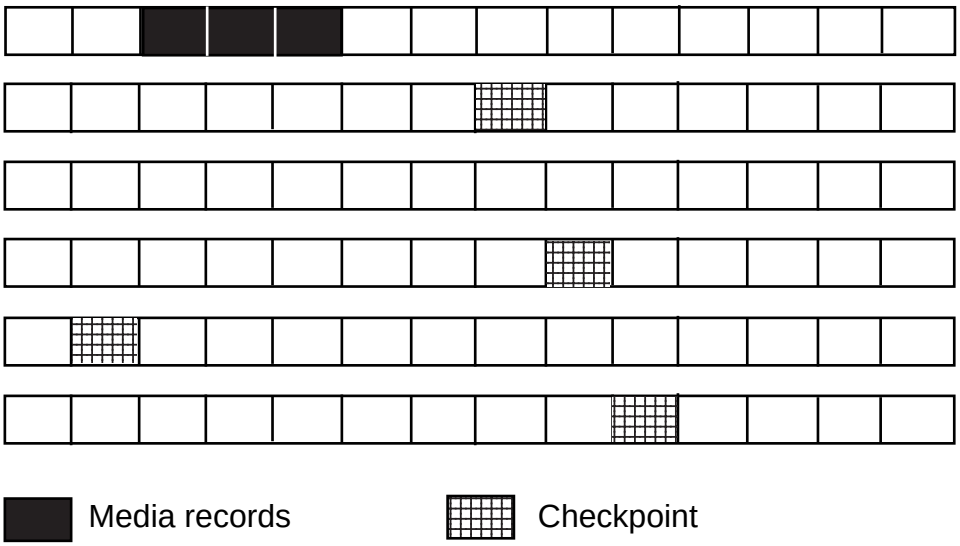
On distributed platforms the queue manager will sometimes perform an operation called ‘pull-up’. This involves finding old log files that contain only a few active records and copying them to a new log file. This saves log space by allowing the old, previously sparsely used log files to be discarded (or reused in the case of circular logging). The algorithm used for pull-up is not documented and should not be relied upon to prevent problems occurring due to long-running transactions.

Media recovery

Media recovery is the restoration of a queue manager to the state before a crash, using log files only. This is necessary to restart a queue manager if the object files have been lost or damaged and involves recreating the object files. This is done by restoring an old copy of the object files from the media image stored in the log and then replaying the contents of the log to bring the object files back up-to-date.

Media recovery requires the following log entries (see Figure 2):

- The most recent check-point.
- All entries after the most recent check-point.
- Records for any uncompleted transactions.



Notes:

1 Media records are needed for all objects.

2 All transactions logged must be re-applied to return to state before the failure – this requires all records after the media image.

3 The first log record required to recreate an object is known as its media recover record; it is the start of the latest media image for the object.

Figure 2: Records required for media recovery

- The media image for all damaged objects.

A media image is a set of log records that saves the state of a queue manager's objects from the object files. They can be created manually using the **rcdmqimg** command and are also are created automatically in some circumstances. The MQSeries manuals give some information about when media images are created, specifically:

- Images of all process objects and queues that are not local are taken at each shutdown.
- Images of empty local queues are taken at each shutdown.

Media images are also created at other times, eg sometimes if a queue goes empty; however, practical experience shows that the automatic creation of media images cannot be relied on to prevent the number

of logs needed for media recovery getting very large. The experience of a large bank illustrates this point. The bank uses 4MB linear log files and runs **rcdmqimg** manually, once a week. Before running **rcdmqimg**, typically, 20 log files are needed for recovery. After **rcdmqimg** is run this number is reduced to just one.

A media image creation and media recovery can only be done using linear logs. For queue managers with circular logging the **rcdmqimg** command will fail with an error message 'AMQ7044 - Media recovery not allowed'.

The media image records the object's state when the media image is taken. Media recovery involves recreating the object files from the media image and then re-applying all the operations performed since then (using the records in the log). As well as using a large amount of log space, recovery from an old media image also takes a long time, which increases the amount of time needed to restart the queue manager.

The space needed for a media image is clearly dependent on the depth of the queues and size of the messages. Queues containing a lot of large messages will require a large number of log records for the media image and it will take a long time to create the media image.

This is one of the reasons for the generally accepted principle that MQSeries queues should not be used as a database to store non-transient data. However, this principle seems to have been forgotten at IBM with both the cluster repository and the OAM authorization data now being held in local queues.

MIRRORED DISKS

Any open systems installation that is serious about data integrity will ensure that critical data is held on RAID or mirrored disks. This stores multiple copies of data so that it can be recovered in the event of a disk failure. Whether linear or circular logging is used the log files should be stored on RAID or mirrored disks.

You have a choice to make about the object files. If the object files are located on non-mirrored disks they may be lost because of a disk failure and media recovery will needed. This in turn requires linear logging.

Alternatively, the object files can also be stored on the mirrored disk. They are then no more likely to be lost than the log files. This means that media recovery will not be required and circular logging can be used. This has several benefits: less log space is needed; it is not necessary to force the creation of regular media images; and log files do not need to be archived.

A word of warning about mirrored disks: whilst it is true that a failure of the disk-drive hardware can be recovered relatively easily, it should be remembered that mirrored disks work at a level below the operating system and merely update two copies of the data as directed via operating system commands. A software bug in the queue manager or in the operating system (or an operator error) could lead to the log or object files becoming corrupted. If this happens the mirrored disk will duly keep two identical copies of the corrupted data.

Dual logging provides a better way of keeping multiple copies of the queue manager data than mirrored disks. Using dual logging, the queue manager performs a write to two separate logs for each update. As far as the operating system is concerned these are two unrelated files and a problem with the operating system is unlikely to corrupt both copies of the log. Unfortunately dual logging is currently only available on MQSeries for MVS.

NEW IN V5.2

IBM has realized that many MQSeries sites use automated scripts to archive linear log files by searching the error log for AMQ7467 and AMQ7468 messages. These messages give the name of the oldest log files needed for restart and media recovery. They can be used to identify which log files can be deleted or archived.

Unfortunately, before version 5.2 only the automatic creation of a checkpoint caused these messages to be written to the log. When the **rcdmqimg** command is run, many log files are no longer needed and can be archived, but before version 5.2 this was not reflected by the AMQ7467 and AMQ7468 messages until a checkpoint was created.

MQSeries version 5.2 has introduced a new flag, '-l', for the **rcdmqimg** command that causes AMQ7467 and AMQ7468 messages to be issued when the **rcdmqimg** command has completed.

Something else introduced by version 5.2 for Windows NT is less welcome. There is a bug which causes the **rcdmqimg** command to fail if it is attempted after the **rcrmqobj** command is run without the queue being accessed by an application between the two commands. Happily, there is a fix for this obscure problem on *CSD01*.

RECOMMENDATIONS

- Always locate log files on mirrored or RAID disks.
- To reduce the total file space needed by your queue manager and simplify the task of log-file management, use circular logging and locate object files on mirrored or RAID disks.
- If linear logging is used ensure that regular and frequent media images are taken using the **rcdmqimg** command.

Michael Locke, Senior Solution Architect
Candle Service (UK)

Determining end-to-end response time in a multi-tier application environment

Today, we are experiencing a proliferation of development and production processing platforms. As Unix servers reach the power of mainframes and the cost of IBM's z/Series processors scales down to client/server prices, operating systems are chosen for developmental reasons.

This may sound like the realization of a Utopian dream – the flexibility to create a customized enterprise IT environment based on market or application requirements. However, flexibility always has a price, which in this case is a lack of compatibility in both networking and operating system application services. The differences in services and communication infrastructures make it difficult to measure end-user application response times.

THE INFRASTRUCTURE

Many companies have attempted to solve the communications

problems created by disparate processing platforms by implementing tools such as IBM's MQSeries. MQSeries masks the complicated process of writing network-enabled applications by providing connectivity across platforms with a simplified application interface. Instead of training applications developers in SNA or TCP/IP protocols, a simple get or put call to a local MQSeries queue manager or MQSeries client allows applications developers to concentrate on business models instead of network architectures.

Similarly, the decision to convert new application development to portable languages such as Java (as well as migrating legacy applications when maintenance is required), allows corporations to decide where key production applications can be best supported. More and more, processing platforms, regardless of operating system, are seen as enterprise servers to be slotted into the overall enterprise IT strategy. By encapsulating application modules into JavaBeans or Java applets the requirement for specific operating services can be removed as an execution consideration. This puts the choice of execution platform into the hands of business managers instead of technology support organizations.

As technicians we may not have been allowed to set up our environments or networks but we are still expected to keep all the pieces working. In the past decade we have become experts in disparate platforms and operating systems: we have learned how to support Unix servers; configure NT workstations; specify virtual memory for Windows; and make RAID work on OS/390.

We have, however, avoided addressing one of the most confusing issues: how do we combine processing across multiple platforms, networks, and applications into a single unit of work? Without the ability to connect processing on behalf of the end user through multiple systems, databases, network connections, and operating system environments we cannot truly define end-to-end response times. Without the ability to track application module movement we are back to the tried and tested method of the stopwatch to determine end-user response times.

Some might say that the old ways are the best, but there has to be a better one. Consider the end-user environment illustrated in Figure 1.

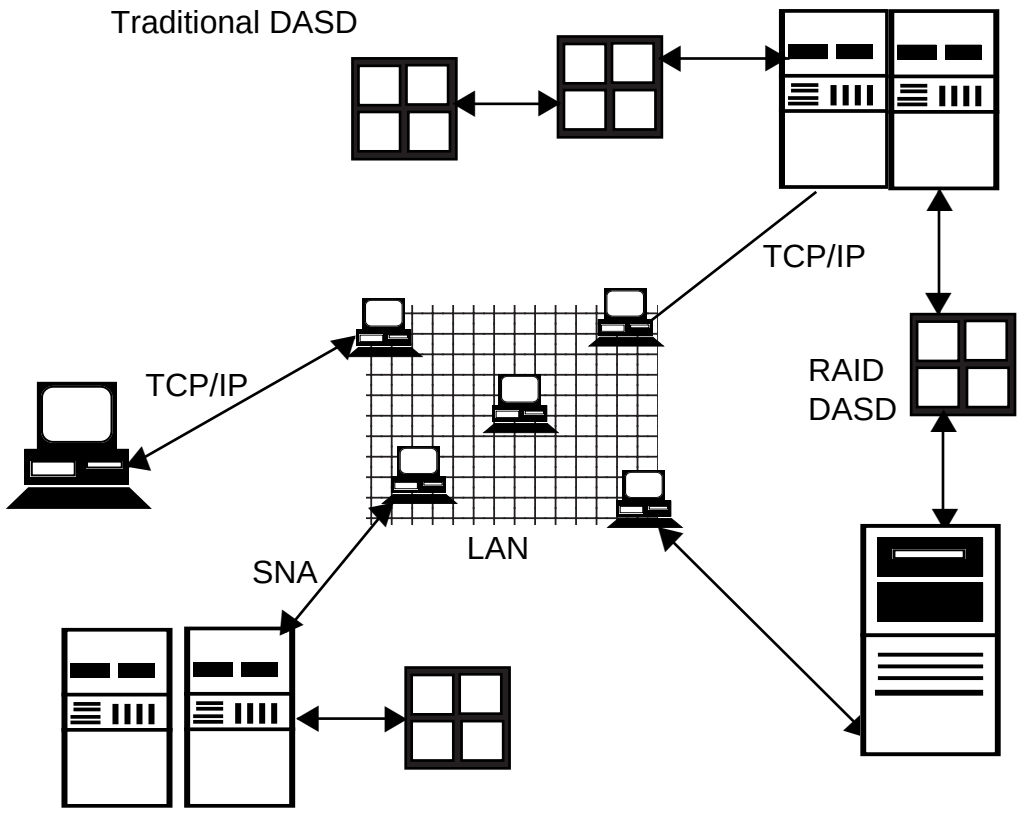


Figure 1: End-user environment

Despite what happens in the background of this modern implementation, the end-user's perspective is remarkably similar to that of a user in the old SNA/3270 environment. They start an application, add their input, and when they hit 'enter' they wait for the result to be produced. The routing of the application through various pieces of the enterprise is not of the slightest interest to the end user. They want an answer to a business question, not a lecture on how the enterprise architecture supports their processing request.

FUTURE SOLUTIONS

As vendors develop new performance solutions the determination of true end-to-end response times may be built into product suites or application processing protocols. New networking paradigms may result in applications being dispatched across multiple platforms and multiple operating systems. The concept of the future is 'the network is the computer and the computer is the network'.

Middleware solutions such as MQSeries Workflow will enable IT departments to define and monitor applications as they flow through the enterprise and to define units of recovery for application recovery purposes.

We expect to see this kind of progress at an ever-increasing rate over the next few years. However, these solutions may require the re-engineering of your business practices in order to fully implement the potential benefits. In most cases it will still be necessary for the business to define a unit of recovery which spans multiple platforms and/or multiple applications.

Let's consider...

Consider a client relations support centre for a financial planning firm. A telephonist takes a customer call in a local branch office. Using a workstation that supports either a browser or an X-Windows interface, the telephonist enters the client identification information.Validation of the customer account takes place at the local AIX server and includes a list of investments that are currently in place.

The client has decided to diversify and wants a different basket of investments based on personal investment goals. The telephonist needs to identify a list of potential investments along with the financial requirements and the rules of the particular funds involved. Fund information is sourced in the corporate legacy systems which have served the investment firm for more than a decade.

The customer profile starts in the workstation with the invocation of a Java program which queries the local AIX server. After reviewing the account with the customer the parameters of the new investment funds are prompted by a second Java application.

When the parameters are finalized they are formatted into the terminal input format of a CICS transaction, which is executed on the OS/390 environment. The transaction returns a result set, consisting of potential funds that meet the customer's specifications, along with links to the rules and investment requirements for each particular fund. As each fund is considered a new transaction is spawned, retrieving the fund requirements for inclusion in the customer's profile.

When the call is finished the customer's local profile has been updated

and any trades or redemptions required for the transfer have been scheduled for finalization at the end of the trading day.

Definition of the problem

Let's first discuss what we are trying to achieve. In Figure 1, we have several elements to consider:

- The end-user workstation and its connection to...
- ...the enterprise intranet/Internet and the routing to...
- ...the client/server platforms and the subsequent transaction processing on...
- ...the OS/390 server.
- Network response times across the enterprise.

Reviewing the local performance from the workstation through the AIX processing platform is fairly straightforward. We have an application running in a single-server system and if there are problems with response times we can deal with them in the same fashion that we deal with performance issues in any single-tier application model.

Consider the processing profile of the new fund selection process: workstation to AIX to OS/390 and the return.

Definition of application component response time in any of the given platforms is as straightforward as was intimated in the single-tier application example. However, how do we find the problems with the whole application if the user has a problem in our hypothetical application once the fund selection process starts?

The solution

In initial application design or in the re-engineering of legacy applications we need to recognize that the application is going to be multi-tiered. This means that we have to instrument each application program to collect and publish the performance metrics, which are essential to application response time management.

Creating the instrumentation within an application is relatively simple. A routine that stores a time value at the start of application processing and then subtracts the stored start time from the finishing time and

publishes the result is a basic reusable module. Inclusion of the instrumentation module works as a planned part of new application design as well as retrofitting legacy applications. We also need to consider the fact that adding the timing module is only part of the requirement for our methodology to work. Additionally, the application must publish identification, authorization, the amount of data transmitted between programs, plus any other instrumentation metrics necessary to correlate the timing statistics with the associated platform and module purpose.

Each program, along the application execution path, must also be modified to accept performance instrumentation from its predecessor and publish its performance, along with any additional performance metrics previously generated. Finally, the performance metrics need to be combined with the OS/390 performance data.

Product note

For sites that own Landmark's TMON for CICS/ESA, the performance information collected along the application path can be embedded in the TMON for CICS transaction records by implementing the TMON for CICS DCSUI exit. This guarantees that the collected performance statistics are correlated and retrievable in the future for performance analysis and capacity planning.

The TMON for CICS DCSUI exit allows application data to be recorded by accessing the user exit from within the CICS transaction, and inserting up to 256 bytes' worth of data into the record that is logged by TMON. This data can be viewed within the online portion of TMON for CICS by accessing TMON option 6.TA. Select a set of transactions from the collected data, and you will see the TRANSACTION ACTIVITY screen. Select a transaction to be reviewed and press ENTER. Cursor-select the MENU option, and you will get the following display:

```
**JOBNAME: EDU2ØC41***** DETAIL TRANSACTION DATA *****DATE: 11/12/Ø1**
* APPLID : EDU2ØC41                                      TIME:  4:43:32 *
* COMMAND: ____________________________________________ CYCLE: MMSS    *
* LMRKØ1226I - COLOR OPTION DISABLED            _NEXT? _GNXT? _MENU?   *
* TRANID:   CLS1     TERMID:    SYSID:  T2Ø4      START DATE: 11/11/Ø1 *
* ORIG TRN: CLS1     LUNAME:    SMFSID: EDUC       TIME: 19:26:52.2314 *
* BASE PGM: DFHZLS1  REMOTE:    IMAGE:  EDUC       STOP DATE: 11/11/Ø1 *
* CTASK#:     32     USERID:    ASID:   ØØ47       TIME: 19:26:52.4383 *
```

```
* MTASK#:    +------ HOT MENU SCREEN NAVIGATION ------+   EDU2ØC41 *
* ELAP:     | A=DETAIL DATA   E=WAIT EVENTS    I=DISPATCH/TCBS |S:
* DISP:     | B=TRANS FLAGS   F=FILE/DB ACTV   J=USER SEGMENTS |:  254 *
* WAIT:     | C=TRANS TIMINGS G=MRO/ISC ACTV                   |
*           | D=EIP REQUESTS  H=USER TARGETS                   |        *
PAGEIN:     |                                                  |INS:  Ø *
* PAGOUT:  +- ENTER MENU LETTER + PRESS ENTER : PF3 TO EXIT  -+HWM:  Ø *
* TERMTYPE:   ØØ   INPUT CNT:    Ø INPUT SIZE:       Ø ABEND CODE:     *
* 327Ø AID:   ØØ   OUTPUT CNT:   Ø OUTPUT SIZE:      Ø ABEND PGM:
* FLAGS: IBM ATI EIP EXHT BKG ISO C41Ø TM2Ø TØ1
*                                                    USERFLD:
* UOW: LANDMRK1.EDU2ØC41    : B9545FB947C9 : ØØØ1    LOC UOW:
B6B9545FB947C9A2 *
** HELP INFORMATION = PF1***NASTCE  **** PF KEY ASSIGNMENTS = PA1***
```

Select option ‘J’ from the ‘HOT MENU SCREEN NAVIGATION’ roster and press enter to see the USER SEGMENTS screen. This screen will allow you to review the data that has been passed to the DCSUI for inclusion with the transaction data collected by TMON.

Other monitoring products may have similar interfaces that will allow the inclusion of externally published data into the records collected for performance analysis and capacity planning. If so, see the documentation for your particular performance monitoring product for details on how to include the collected data.

The solution (continued)

If you are going to be altering existing applications or changing the design of new applications and you are a user of IBM’s MQSeries, then you might want to consider the following data collection methodology.

Instead of writing data collection modules and worrying about passing growing strings of performance data from program to program along the path of an application, you might want to collect the same information as was described previously, and then incorporate that information into MQSeries messages.

Each stage in the processing of an application can be instrumented by a collection subroutine that formats the data into an MQSeries message and puts it onto a queue defined for performance metric collection. The benefit of using MQSeries for message management is that, each time a message is put onto a queue, not only is the message content saved but the message itself is time-stamped automatically.

Consider the header for the following message placed on an MQSeries queue:

```
* Queue Name: SYSTEM.CLUSTER.REPOSITORY.QUEUE                    _Prev *
* Description System clustering repository queue     _ Move _ Delete   *
* Msg Data : .qA.WDR ...@...QCQ2.B2AD08C17Ø3AEØØ5    _ Requeue         *
+-------------------------  Message Header  ----------------------+ *
* PUT Time(GMT): 14:2Ø:33.ØØ Report Option: ØØØØØØØØ
* PUT Date     : 2ØØ1-Ø5-22  Feedback Code: Ø
* Message Type : DATAGRAM    Message ID   : CSQ QEQ1        ..$...o
* Format       :             Correl ID    : CACHE_CHECKPOINT_RECORD
* Persistent   : YES         Userid       : LSCSTC
* Expiry Time  : UNLIMITED   ACCT Token   : ____________________________   *
* Msg Length   : 5192        APPL ID Data :
* Priority     : 5           APPL Origin Data :
* Backout Count: Ø           PUT APPL Type: MVS
* Data Encoding: 785         PUT APPL Name: QEQ1CHIN
* CodeCharSetID: 5ØØ
* Reply-To Queue       :
* Reply-To QMGR        : QEQ1
* Remote Queue (XMITQ) :  ****** NO XMITQ HEADER *****
* Remote QMGR (XMITQ)  :
```

Add to that information the actual data of the message itself:

```
* 12F4D2F4  ØØØØ  Ø598C1ØØ E6C4D94Ø ØØØØØØØ1 ØØØØØØ7C * .qA.WDR ...@* *
* 12F4D3Ø4  ØØ1Ø  ØØØØØ472 D8C3D8F2 4BC2F2C1 C4F0F8C3 *...QCQ2.B2AD08C*
* 12F4D314  ØØ2Ø  F1F7F0F3 C1C5F0F0 F5404040 40404040 *17Ø3AEØØ5    * *
* 12F4D324  ØØ3Ø  40404040 40404040 40404040 40404040 *             * *
* 12F4D334  ØØ4Ø  40404040 D8C3D8F2 40404040 40404040 *    QCQ2     * *
* 12F4D344  ØØ5Ø  40404040 40404040 40404040 40404040 *             * *
* 12F4D354  ØØ6Ø  40404040 40404040 40404040 40404040 *             * *
* 12F4D364  ØØ7Ø  40404040 ØØØØØØØØ ØØØØØØØØ ØØØØØØA4 * ...........u* *
* 12F4D374  ØØ8Ø  ØØØØØØØØ ØØØØØ152 35CF28AF ØØØØØØØØ *............* *
* 12F4D384  ØØ9Ø  ØØØØØØØØ ØØØØØØØØ ØØØØØØØØ ØØØØØØØØ *............* *
* 12F4D394  ØØA0  ØØØØØØØØ ØØØØØØØ1 E3D64BD8 C3D8F24Ø *.....TO.QCQ2 * *
* 12F4D3A4  ØØB0  40404040 40404040 40404040 ØØØØØØ5 *         ....* *
* 12F4D3B4  ØØC0  ØØØØØØØ9 ØØØØØØØ2 C3888195 95859340 *.....Channel * *
* 12F4D3C4  ØØD0  8184A585 99A389A2 89958740 D8C3D8F2 *advertising QCQ2*
* 12F4D3D4  ØØE0  40A39640 D8C1C3D3 E4E2E3C5 D9404040 * to QACLUSTER * *
* 12F4D3E4  ØØF0  40404040 40404040 40404040 4044040  *
```

The series of messages sent as the application processing progresses can then be read by the CICS transaction on the OS/390 platform, processed, and the resulting performance data written for later analysis.

Adding the network transit times

In addition to the application processing times being captured by the program performance modules, there is the time spent in transit across

the network to consider as well. In today's enterprise, intranet paths can be complex. Firewalls, routers, LANs, and gateways all add to the complexity of the route between the end user and the servers providing application data and services.

Consider the network connections involved in our hypothetical business application:

```
*                                Sent    Recv    Sent    Recv     Rt    TCP *
* Host-ID     OpSys Status       Pkts    Pkts    Segs    Segs    Segs   Est *
*                                                                        *
* MVSSYSE     OS390 Active     282014  1.65M 120414  1.25M  1454     6 *
* gw-nb3-2-to Cisco Active      7.99M 1.484G   6747   9215     1     0 *
* cisco2ch    Cisco Active     458811  4.23M   3038   3691     0     0 *
* zeus        Win2k Active     41.84M 0.109G 36.94M 32.23M 0.35M    24 *
* mork        HP-UX Active     92.74M 0.183G 12.75M 24.20M  3808     4 *
* selab03     SunOS Active     0.170G 0.100G 0.150G 22.20M 56572    10 *
* saturn      SunOS Active     0.297G 0.417G 0.148G 99.03M 72275     8 *
* MVSSYSA     OS390 Active      93223 277574  14236  18514    29    28 *
* mvssnac      zOS  Active       6571  76771   2501   4131     1     5 *
```

End users run Win/32 workstations connected to a LAN. The gateway from the LAN into the corporate backbone network is the Win/2000 server named 'zeus'. Access to the Unix environment (systems 'mork', 'selab03', and 'saturn') is routed from 'zeus' through the CISCO router named 'cisco2ch'. Access to the OS/390 and z/OS systems is controlled, along with access to the Internet, by the CISCO router named 'gw-nb3-2-to'. Both OS/390 systems, MVSSYSE and MVSSYSA, are VTAM-connected as well as TCP/IP connected. Access to the z/OS system 'mvssnac' is available only through TCP/IP.

End users accessing the business application would use TCP/IP for connections to their LAN and to the Unix box as well as the mainframe servers.

To get a fair approximation of end-user network response times we need to know two things: the route between the user workstation and each of the server components; and the maximum data transmission rate for each segment of the route. Matching this information with each program's data requirements will allow the calculation of expected network response times.

Most framework products as well as network performance management tools will allow the systems personnel to run the **TCP/IP TRACERT**

command from any given node in the corporate network. By tracing the route selected by the network for data traffic between nodes you can determine the number of hops expected each time the application executes. Once you know the network routing you can review it with your networking analysts and ensure that each leg of the path is appropriate.

Using your framework or performance monitor you should then create an automatic execution of the **TRACERT** command at specified intervals to ensure that there are no failures of key network components that will affect data transmission between nodes.

Once you know the path that will be taken by the application data and the size of the program data being shared by those programs you can get the bandwidth between nodes by using the **TCP/IP DMTU** (Determine Maximum Transmission Unit) command. DMTU will return the bandwidth specifications for each segment of the intranet path between the various servers that support the application processing. Once you know the path data will take, the amount of data that each program within the application will transmit, and the bandwidth of each segment of the path, simple mathematics will yield a best case estimate of actual network response times for the application.

Determining the actual network response times would require sophisticated network monitoring using existing tools. However, if you are using the methodology suggested either through parameterized or MQSeries message-based performance data collection, you can calculate the delay between the end of each program's processing and the start of the subsequent program's processing. This time will include both network transit and any latency delays based on server activity. If the calculated network/latency times are excessive you can flag for performance review the appropriate network segments as well as the servers involved.

CONCLUSIONS

If you are going to design and implement multi-tiered applications in today's corporate environment application end-to-end response time calculation will remain problematical. As vendor products mature point solutions may emerge that will solve the management problems.

But for the moment, application design will need to include a performance methodology as well as the data collection to support your goals.

Frameworks and their server-based monitoring agents may yield parts of a solution; however, specification of the components of an application will still need to be completed for management purposes. That specification will need to have in place the methodology outlined above for it to be effective.

Advanced middleware solutions such as MQSeries Workflow or OS/390 and z/OS Workload Manager may yield a more complete solution as they mature, but specification of the application design will still be required.

Performance management and user service level agreements are only one of the corporate issues addressed by this methodology. If your organization wants to be able to charge end users for usage of multi-tiered applications and the network traffic involved in those applications, you need the requisite information.

We have moved past the days when we could implement complex applications that span the enterprise and allow users to consume infrastructure resources without accountability. Expansion of network and server resources will require the ability to calculate usage and charge-back information at the end user level. As we move into the next generation of processing, where the computer is the network and the network is the computer, we will need more detail on hand to support increasingly complicated applications and their end-user communities.

Planning for application instrumentation now will eliminate the need to retrofit applications as corporations grow.

J Bailie
PConsulting (UK)

MQ news

Mercator Software has recently completed its suite of new product offerings, which includes: Mercator Integration Broker Version 6.5; a Java-class adapter; and a JMS (Java Messaging Service) adapter. The company has also added WebSphere MQ messaging as an option available with Mercator Integration Broker.

Mercator claims that the inclusion of a WebSphere MQ messaging option along with the introduction of Mercator Integration Broker 6.5, Mercator Process Integrator V1.0, and new standards-supporting adapters, provides an enterprise-wide integration solution.

For further information contact:
Mercator Software, USA
Tel: +1 800 234 5566
Web: http://www.mercator.com

Mercator Software, UK
Tel: +44 (0) 20 7314 9600

* * *

Peregrine Systems and IBM have announced the expansion of their existing relationship. Under the reseller agreement, IBM will add a broad range of Peregrine Integration adapters to its offerings.

The Peregrine Integration adapters are designed to integrate WebSphere MQ Integrator 2.1 and other WebSphere family products with a variety of packaged applications, legacy systems, databases, and technologies. Additionally, IBM will increase the number of Peregrine products offered as WebSphere Partner Agreement Manager.

For further information contact:
Your local IBM representative.

Peregrine Systems, 3611 Valley Center Drive, San Diego, CA 92130, USA
Tel: +1 800 638 5231
Fax: +1 858 481 1751
Web: http://www.peregrine.com

Peregrine Systems, Ambassador House, Paradise Road, Richmond, Surrey, TW9 1SQ, UK
Tel: +44 (0) 20 8332 9666
Fax: +44 (0) 20 8332 9533

* * *