MQ update

1

July 1999

In this issue

MQ Update

Published by

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

North American office

Xephon/QNA
1301 West Highway 407, Suite 201-405
Lewisville, TX 75077-2150
USA
Telephone: +1 940 455 7050

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*.

MQSeries Update on-line

Code from *MQ Update* is available from Xephon's Web site at www.xephon.com (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

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.

MQSeries announcements

The MQSeries family is built on a messaging foundation that's provided by the base MQSeries product. This article describes the key features of the latest MQSeries announcements, highlighting improvements to the base messaging and infrastructure products. In particular the discussion concentrates on the two major elements of the latest releases: Dynamic Workload Distribution (which is available for both OS/390 and a selected number of other platforms), and new ground-breaking changes to the installation and administration of the MQSeries for Windows NT product. Overall, the latest release of the MQSeries messaging product offers improved usability, failover, and availability compared with its predecessor.

The article is split into four sections, which focus on MQSeries for OS/390 V2.1; general improvements for MQSeries V5.1 for AIX, NT, Solaris, and HP-UX; details of specific improvements to the NT product; and an in-depth examination of Dynamic Workload Distribution (DWD). The DWD feature is delivered on both the OS/390 product and across the board on MQSeries V5.1.

MQSERIES FOR OS/390 V2.1

A number of new features have been added to MQSeries specifically for the OS/390 platform. These features generally integrate MQSeries more closely with the underlying OS/390 operating system, making use of the operating system's facilities to improve availability, and are part of the on-going support for Parallel Sysplex.

Resource Recovery Services (RRS)

In previous releases of the product, MQSeries for OS/390 operated as a resource manager under either CICS or IMS control. This allowed a single transaction to update both MQSeries resources and a database, such as DB2, under CICS or IMS control. RRS extends this ability to embrace MVS's batch environment, TSO, DB2 stored procedures, and Component Broker applications by providing the required sync point manager function. MQSeries operates as a resource manager under RRS in the same way as it previously operated under CICS or IMS.

Automatic Restart Manager (ARM)

Using ARM allows a failed MQSeries Queue Manager and/or message mover to be restarted automatically following a failure. The restart may occur on the original MVS image or (if the original is unavailable) a different MVS image. In addition, systems administrators can define the order in which resources are recovered. This makes it possible, for example, to restore CICS before MQSeries, should this be necessary.

General new developments

In previous releases of MQSeries for OS/390, the MQSeries Listener Program stopped when TCP/IP was recycled. The latest release has increased availability by being able to detect whether TCP/IP or ACCP/MVS has returned, thereby eliminating the need for operator intervention to restart the listener.

TCP/IP OE Sockets support, which was made available as a PTF on V2.0, is now integrated into the main release. This high performance API reduces CPU utilization for MQSeries TCP/IP message channel agents.

MQSERIES V5.1

The latest version of MQSeries for distributed platforms (AIX, HP-UX, Sun Solaris, Windows NT, and OS/2) has an important range of new messaging features. In addition to the introduction of Dynamic Workload Distribution (DWD) and some significant improvements to the version for NT (both are discussed later in this article), MQSeries V5.1 delivers improvements in:

- Performance
- Scalability
- Java capabilities
- Administration programming.

Performance

A valuable performance improvement is the result of reducing the amount of disk I/O that results from an MQGET WAIT call being processed for a persistent message. This makes a considerable difference to overhead associated with processing persistent messages on local queues.

Performance improvements were also made to Unix versions of MQSeries by reducing system memory requirements in terms of both real and virtual memory. This was achieved by implementing multithreaded client agents and multithreaded channels.

Scalability

The size of MQSeries implementations at customer sites has grown significantly in the past two years. In order to provide still further scalability, the maximum size of the disk file system used to store messages has been extended from 320 MB to 2 GB. This allows larger messages to be stored, also allowing the storage of a greater number of messages.

Other scalability improvements have been brought about by the use of threads. These include multithreaded agents for AIX, Sun Solaris, and HP-UX, which allow up to two-thirds more MQ clients to run. Multithreading has been implemented by enabling MQSeries channels to make use of multithreading on Sun Solaris, AIX, and HP-UX.

Java capabilities

Java support is a vital part of MQSeries. Further improvements to it were made in Version 5.1 by packaging both the MQSeries Java Client and Bindings together. The MQSeries Java Client can now communicate using IIOP in addition to the propriety TCP/IP-based protocol. This function is especially important for sites that intend to use MQSeries as a transport mechanism for Object Request Brokers.

Administration programming

IBM has received numerous requests from customers for a simple way to create scripts for automating MQSeries administration and write monitoring programs and process alerts. In order to address these requests, the MQAI (MQSeries Administration Interface) was created. This API is also useful in large-scale deployments and unattended operations in distributed environments.

Windows NT enhancements

Windows NT is a popular platform on which to run MQSeries, so significant new features were added to Version 5.1 to exploit NT's functionality and integrate MQSeries with the operating system. One of the most visible changes concerns ease of use – the installation procedure is changed so that a fully functioning MQSeries system (including membership of a default cluster) is now in place as soon as installation is complete. MQSeries can now be managed using the Microsoft Management Console (MMC), which may be used to manage queues and messages on queues. In addition, a Web administration tool was developed to assist in the definition and administration of the MQSeries infrastructure.

As well as improvements to ease of use, improvements were also made to the product's performance in a number of areas. These include a reduction in CPU utilization of over 20% and a similar improvement to the path length of non-persistent messages. By using an improved operating system compiler, overall message throughput performance has improved – while the improvement experienced depends on a number of factors, an improvement of around 10% is typical.

Publish-and-subscribe

Publish-and-subscribe is a natural step in the evolution of messaging. Simple messaging is 'point-to-point', where one application sends messages to another (known) application. MQSeries does not examine the content of messages as part of a point-to-point connection. However, in a publish-and-subscribe environment, applications issuing messages identify the subject of the message when it's published or 'emitted'. A specialized MQSeries node then examines the subject of the message and distributes it to applications subscribed to that topic stream. Applications can subscribe to multiple topics, and, by using MQSeries's assured delivery (a fundamental part of the system), subscriptions remain queued when the subscriber is disconnected.

Subsequent reconnection by the application enables queued messages to be retrieved.

Many messaging applications exhibit the attributes of publish-and-subscribe. Any application that sends messages to more than one application can be thought of as a publisher of messages. Subscribers may require messages that are published by a variety of applications.

DYNAMIC WORKLOAD DISTRIBUTION (DWD)

This ground-breaking functionality is a new development in MQSeries that allows a number of queue managers to be grouped, irrespective of geography and operating system. The members of the cluster (a group of MQSeries queue managers) can also discover each other's presence and self-define their interfaces, a major step forward for setting up and configuring systems. However, DWD is not just about administration – it also provides facilities for alternative routing and workload balancing.

MQSeries clusters shouldn't be confused with hardware clusters – an MQSeries cluster is a logical grouping of queue managers. It can span many operating systems and interconnectivity protocols over a wide geography.

Key features of DWD are:

1 Automatic configuration and dynamic discovery of queue managers and resources, including:

   – Queue managers can now advertise their queues to other queue managers. These queues then become public queues.

   – Channel and queue definitions are removed or automated, channels are defined automatically, and no remote queue definitions are required for remote queues. This improves the reliability of installations.

2 Automatic failover for queue managers within an MQSeries cluster.

3 Dynamic workload distribution, allowing the same queue to be available at multiple sites or on multiple machines.

Within a cluster the need for MQSeries definitions is significantly reduced. Previously, two queue managers wishing to communicate remotely required a transmission queue, a channel to the remote queue manager, and a remote queue for each target queue. In a DWD cluster any queue manager can send a message to any other queue manager in the same cluster without the need for explicit channel definitions, remote-queue definitions, or transmission queues for each target destination. This reduction in administrative overhead does not require users to forgo MQSeries's assurance of delivery for persistent messages. Queue managers can be part of more than one MQSeries cluster, thereby allowing a significant degree of flexibility in the system's configuration.

Essential to the operation of clusters of queue managers is the 'repository'. There should be two or more queue managers in the cluster defined as repository queue managers. These hold information (such as names, locations, channels, and hosted queues) about queue managers in the cluster. Each queue manager in the repository sends information about itself to the repository queue managers (which operate a mirroring system to remain synchronized). The repository is a queue called SYSTEM.CLUSTER.REPOSITORY.QUEUE. In addition to the two full repositories, all other queue managers host partial repositories. This includes information on a limited set of queue managers in the cluster that a given queue manager needs to communicate with. When a queue manager requires more information than is available locally, a request is made on the SYSTEM.CLUSTER.COMMAND.QUEUE.

Using clusters changes the number of queues required for communication significantly. Every queue manager in a cluster has one transmission queue that it can use to transmit messages to any other queue manager in the cluster. Each queue manager is required to have only the following channel definitions:

- *Cluster receiver channel*
  This is similar to an MQSeries receiver channel. When the cluster receiver channel is defined, it's added to the repositories along with the owning queue manager. Defining this channel is part of the process of introducing a queue manager to the cluster. When

other queue managers in the cluster become aware of a new queue manager, they automatically add their own definitions for the cluster sender channel.

- *Cluster sender channel*
  This is similar to an MQSeries sender channel. The process of defining the first cluster sender channel on a queue manager introduces the queue manager to a cluster. Subsequently, all required cluster-sender channels on the queue manager are started automatically (they take their attributes from the corresponding target cluster receiver channel, thereby reducing the administrative overhead).

- *Cluster transmission queue*
  Every cluster queue manager has a queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE, which holds all messages from that queue manager to other queue managers in the cluster.

Not only do MQSeries clusters simplify both administration and installation, they can also offer increased availability and workload balancing. Clusters may have more than one instance of a queue spread across multiple queue managers. For example, multiple instances of the same application in a System/390 Sysplex can be used to read and/or update a shared DB2 database. This configuration can provide greater message throughput and increased availability. If a destination queue manager fails while there is a message on a transmission queue for it, the system attempts to re-route the message (without the risk either of losing it or of creating a duplicate). Only 'in-doubt' messages are not re-routed (these are messages that have been sent, but for which no acknowledgement has been received from the target queue). Re-routing in-doubt messages could clearly lead to duplicates.

The algorithm used to select a target queue manager in a cluster takes into account such attributes as availability of the queue manager or queue, the state of the channel, and the priority of the message. Using the 'cluster-workload' exit, workload management within an MQSeries cluster can be tailored. Such customization may take account of network cost, system load, and the capacity of channels.

A word of caution: applications that rely on message sequence (messages must arrive in strict order, this also known as 'message affinity') are not suitable for MQSeries clusters without modification.

SUMMARY

The latest announcements from MQSeries (V2.1 for OS/390, and V5.1 for NT, AIX, HP-UX, Sun Solaris, and OS/2) add significant new functionality to the product set. In particular, Dynamic Workload Distribution enhances the ease of administration, availability, and workload balancing of the product. Major improvements were also made to the installation and management of MQSeries for Windows NT. Performance improvements were realized, particularly on local queueing operations where the target application is available and can accept messages. Finally, the availability of publish-and-subscribe functionality as a Web download for MQSeries V5.1 brings message queueing technology to a new range of applications.

Dr Dave Watson
MQSeries Technical Strategy Manager
IBM Hursley (UK)

Using RACF and the OAM for end-to-end security

It's a commonly held view that MQSeries is not a secure product and that to install it in your network infrastructure is to give hackers a free rein. In this article I'll demonstrate that this isn't necessarily so.

Security is a general term that covers such tasks as sender and receiver authentication, encryption and privacy, non-repudiation, and message integrity and data authentication.

When communication between companies occurs, MACs, digital signatures, and public key encryption may be employed to enforce security, perhaps by means of third-party products, such as Baltimore

Secure MQ. However, many companies consider that, when it comes to communication within an enterprise, such measures are not required as all machines in their infrastructure are managed by administrators whom they trust. All that's required is to provide administrators with the means to prevent unauthorized users from accessing the network and creating messages, while still providing access to authorized users.

This can be done using third-party products, many of which have the potential to secure communications completely. Nevertheless, there is a lot that can be achieved using just the security mechanisms provided by MQSeries itself – that is, using RACF on MVS and the Object Authority Manager (OAM) on distributed platforms.

This article covers the policies that a company would need to put in place and the configuration that administrators would need to implement in order to establish an acceptable level of security in an environment where administrators are trusted but users are not.

EXAMPLE ENVIRONMENT

Consider the following example: for a number of years a large company has successfully used techniques such as file transfer (FTP) to carry out point-to-point communication. However, in order to improve speed of development, they decide to move to an MQSeries-based infrastructure, also deciding to use a central hub managed by a trusted group, as this yields benefits in manageability and allows new connections to be added quickly.

Most of the machines are based in a secure machine room (while it's possible to log on to the machines from outside the room, a discussion of how to secure this type of access is beyond the scope of this article). Each business unit owns one machine. Business units don't trust users (who could be disgruntled employees), and they don't trust administrators of machines belonging to other business units, though they do trust their own administrators. Most security problems, such as 'sniffers' on the communication lines, were addressed when FTP was set up (possible solutions include using encryption at the communications layer, splitting SNA packets into so many parts that they are virtually impossible to read, and using security calls within applications themselves).

Consider a situation in which *A* and *B* need to communicate with each other using MQSeries, as do *C* and *D* (see Figure 1). Most of the security issues that exist in this environment also apply to FTP, though a major new one is introduced.

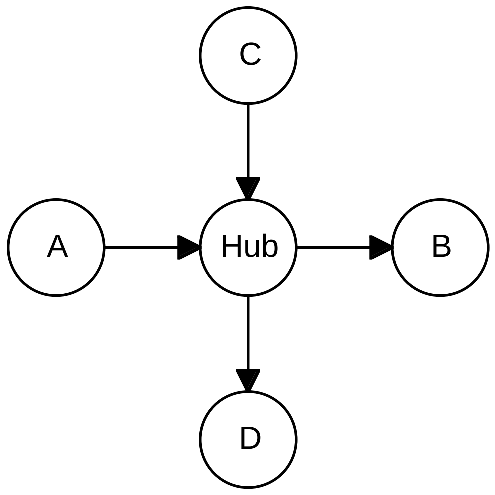

Figure 1: An example network

With the environment shown above, if business *A* decides that it needs to talk to business *D*, the infrastructure is already in place and only the application development needs to be done. This is a very strong reason for using MQSeries and a hub environment. However, it also introduces the problem that an unauthorized person on *A* could send a message to *D*.

In order to secure the end-to-end connection, including preventing the generation of unauthorized messages, it is necessary to carry out the measures detailed in this article.

COMMUNICATIONS LAYER SECURITY

Firstly, security needs to be set up at the level of the communication layer. SNA bind or session-level security can be used to ensure that, when an SNA bind request comes from *A*, the hub knows that the request really does originate at *A*. This is a default with most communications packages (but not ones from Tandem) and involves providing the same password at each end. Obviously the password must be kept secret and should be accessible only by the machines' administrator. Something similar can be done for TCP/IP using secure domain units or some form of Virtual Private Network.

CHANNEL INITIATION SECURITY

We've now ensured that no boxes are connected to the hub that shouldn't be connected to it. However, it's still possible for a user on *A* to define a queue manager called *C* and a channel on *A* called *C.TO.HUB* (for example, by knowing the naming convention or by querying the hub's command server), and then connect to the hub by impersonating *C* and having messages routed to *D*.

If the channel is a sender/receiver channel, the only way around this is to use a security exit provided by a third-party product (such as Baltimore, mentioned earlier). However, if *A* is a secure machine, users won't have the authority necessary to add these definitions to the system. An alternative is to use requester/sender channels. This is similar to a call-back system: the hub acts as a requester and thus needs to initiate the conversation. It calls out to the known LU/IP address stating that it wants to start a channel. *A*, acting as the sender, would then initiate the channel back to the hub. If *A* were to try to start a channel to the wrong requester, the request would not be accepted. Similarly *D*, acting as a requester, could initiate a conversation asking the hub, as a sender, to call it back. As the hub's sender channel contains *D*'s *CONNAME* and calls it back, the most that *A* could do in this set-up would be to get the hub to call *D*.

ROUTING SECURITY

So now we have a system where we can be confident that all messages coming to the hub on any channel are from the machine that they should be from.

The hub is merely a queue manager, looking after transmission queues and running the associated channels. Each transmission queue is named after the queue manager that it points to. The next problem, as mentioned above, is that a user on *A* could, by default, do an *MQPUT* specifying as its target the queue manager of *D*. The message would be put on *A*'s default transmission queue (to the hub); when it reaches the hub, it would automatically be put on transmission queue *D*, and thus get to a destination that it shouldn't be able to reach.

The way around this is to specify the *MCAUSER* parameter on the receiver/requester channel definitions. By default the inbound channel at the hub puts messages on its target transmission queue using the *userid* running the channel. This *userid* has full access to put messages on all queues. However, if you change the channel's *MCAUSER* parameter, the message will be put on the queue using the *userid* specified by the parameter.

So, define one *userid* for each inbound channel on the hub. For example, define a *userid* called *A* for the channel from *A*, a *userid* called *C* for the channel from *C*, etc. Alter the inbound channels to put messages on queues using their corresponding userid – for example:

```
ALTER CHL(A.TO.HUB) CHLTYPE(RQSTR) TRPTYPE(LU62) MCAUSER(A)
ALTER CHL(C.TO.HUB) CHLTYPE(RQSTR) TRPTYPE(LU62) MCAUSER(C)
```

Next set the permissions on the hub's transmission queues to accept only messages from authorized channels. How you do this depends on your set-up – on distributed platforms, use the following commands:

```
SETMQAUT -M HUB -T QMGR -P A +CONNECT +SETALL
SETMQAUT -M HUB -T QMGR -P C +CONNECT +SETALL
SETMQAUT -M HUB -T Q -N B -P A +PUT +SETALL
SETMQAUT -M HUB -T Q -N D -P C +PUT +SETALL
```

If you use RACF, then the following commands are needed:

```
RDEFINE MQQUEUE HUB.B UACC(NONE)
PERMIT HUB.B CLASS(MQQUEUE) ID(A) ACCESS(UPDATE)
RDEFINE MQQUEUE HUB.D UACC(NONE)
PERMIT HUB.D CLASS(MQQUEUE) ID(B) ACCESS(UPDATE)
```

TARGET QUEUE SECURITY

So now *B* and *D* can be confident that all the messages they receive are from authorized queue managers. The next problem is to make sure that the right messages go to the right queues. For example, user *UserX* on *A* might be allowed to send messages to queue *QueueQ* on

B, and user *UserY* on *A* might be allowed to send messages to queue *QueueR* on *B* (see Figure 2). However, we need to ensure that *UserX* cannot send messages to *QueueR*. To do so without either using security exits or changing applications, business *B* needs to trust *A*'s administrator (but not *C*'s, etc). Also a system-wide naming convention of *userid*s needs to be enforced.

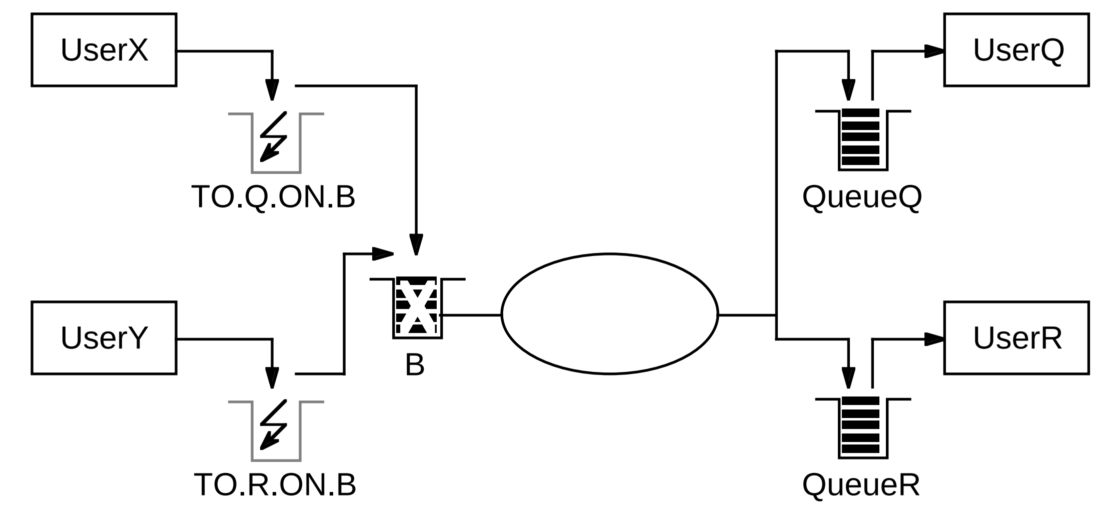

Figure 2: An example implementation using MQSeries

By default, when a user on *A* sends a message, the user's *userid* is put in the *USERID* field of the message descriptor. The user is not allowed to change this. Also, by default the inbound channel at the receiver (for instance, *B*) puts messages on its target queue using the *userid* that's used to run the channel. This *userid* has sufficient access rights to put messages on all queues. If you change the *PUTAUT* parameter of the channel from *PUTAUT(DEF)* to *PUTAUT(CTX)*, messages are placed on the queue using the authority of the *userid* specified in the message descriptor.

So *queues* can now be secured by defining *userid*s on the receiving machines that have the same names as the *userids* on the sending machines. The receiving *userids* do not need authority to log on. So, in this example, define two users, *UserX* and *UserY*, on *B*:

```
ALTER CHL(HUB.TO.B) CHLTYPE(RQSTR) TRPTYPE(LU62) PUTAUT(CTX)
```

On distributed platforms, issue the following command:

```
SETMQAUT -M B -T QMGR -P UserX +CONNECT
SETMQAUT -M B -T Q -N QueueQ -P UserX +PUT +SETALL
SETMQAUT -M B -T QMGR -P UserY +CONNECT
SETMQAUT -M B -T Q -N QueueR -P UserY +PUT +SETALL
```

If you use RACF, the following commands are needed (assuming MQM runs the channel):

```
RDEFINE MQQUEUE B.QueueQ UACC(NONE)
PERMIT B.QueueQ CLASS(MQQUEUE) ID(MQM) ACCESS(UPDATE)
PERMIT B.QueueQ CLASS(MQQUEUE) ID(UserX) ACCESS(UPDATE)
RDEFINE MQQUEUE B.QueueR UACC(NONE)
PERMIT B.QueueR CLASS(MQQUEUE) ID(MQM) ACCESS(UPDATE)
PERMIT B.QueueR CLASS(MQQUEUE) ID(UserY) ACCESS(UPDATE)
```

As mentioned previously, security is a bit more complex on Tandem systems. On Tandem, the '*userid*' in the message descriptor is actually a *groupid*. For the above to work when a Tandem system is the receiver, it is necessary to use a group that's defined and authorized with the same name as the sending *userid*. When a Tandem system is the sender, the receiver needs a *userid* defined and authorized with the same name as the sending group.

LOCAL QUEUE SECURITY AT THE SENDER

If users on the sender machine do not trust one another, some additional work is necessary to set up security.

If *QREMOTE* queues are not used, and users specify the target queue manager in the *MQPUT* call, then messages from *UserX* and *UserY* on *A* are put directly on the transmission queue and there is no way for MQSeries to stop them specifying one another's target queues. It is also possible, when the channel is not running, for them to remove one another's messages before the messages are sent.

The best way to solve this problem is to restrict access to transmission queues (this is the default) and to allow users to put messages only on *QREMOTE* queues that point to the target queues. Using this approach, a secure structure can be set up such that *UserX* and *UserY* cannot put messages on one another's queues.

For instance, using RUNMQSC, enter the following definitions:

```
DEFINE QR(TO.Q.ON.B) RNAME(QueueQ) RQMNAME(B)
DEFINE QR(TO.R.ON.B) RNAME(QueueR) RQMNAME(B)
```

The commands below are the ones to use on distributed platforms.

```
SETMQAUT -M A -T QMGR -P UserX +CONNECT
SETMQAUT -M A -T QMGR -P UserY +CONNECT
SETMQAUT -M A -T Q -N TO.Q.ON.B -P UserX +PUT
SETMQAUT -M A -T Q -N TO.R.ON.B -P UserY +PUT
```

while the ones below are for use with RACF.

```
PERMIT A.BATCH CLASS(MQCONN) ID(UserX) ACCESS(READ)
PERMIT A.BATCH CLASS(MQCONN) ID(UserY) ACCESS(READ)
RDEFINE MQQUEUE A.TO.Q.ON.B UACC(NONE)
RDEFINE MQQUEUE A.TO.R.ON.B UACC(NONE)
PERMIT A.TO.Q.ON.B CLASS(MQQUEUE) ID(UserX) ACCESS(UPDATE)
PERMIT A.TO.R.ON.B CLASS(MQQUEUE) ID(UserY) ACCESS(UPDATE)
```

However, if you are happy to allow applications to write to the transmission queue, you could use either the following commands on distributed platforms:

```
SETMQAUT -M A -T Q -N B -P UserX +PUT
SETMQAUT -M A -T Q -N B -P UserY +PUT
```

or this one with RACF:

```
RDEFINE MQQUEUE A.B UACC(UPDATE)
```

LOCAL QUEUE SECURITY AT THE RECEIVER

If users on the receiving machine do not trust one another, then it’s necessary to set up some additional security.

Say *UserQ* is able to read messages on queue *QueueQ* and *UserR* is able to read messages on queue *QueueR*. If the users are not considered trustworthy, then one needs to guard against the possibility that *UserR* may put a message on queue *QueueQ* and for *UserQ* to receive it believing it to have come from *A*. Similarly *UserR* could get messages from queue *QueueQ* before *UserQ* gets them. To prevent this, it is necessary to run the following OAM commands:

```
SETMQAUT -M B -T QMGR -P UserQ +CONNECT
SETMQAUT -M B -T QMGR -P UserR +CONNECT
SETMQAUT -M B -T Q -N QueueQ -P UserQ +GET
SETMQAUT -M B -T Q -N QueueR -P UserR +GET
```

With RACF, the following commands would be needed:

```
PERMIT B.BATCH CLASS(MQCONN) ID(UserQ) ACCESS(READ)
PERMIT B.BATCH CLASS(MQCONN) ID(UserR) ACCESS(READ)
RDEFINE MQQUEUE B.QueueQ UACC(NONE)
RDEFINE MQQUEUE B.QueueR UACC(NONE)
PERMIT B.QueueQ CLASS(MQQUEUE) ID(UserQ) ACCESS(UPDATE)
PERMIT B.QueueR CLASS(MQQUEUE) ID(UserR) ACCESS(UPDATE)
```

Note that on MVS a problem still remains. *UserQ* (or perhaps a member of the same group) can run an application that puts messages on queue *QueueQ* that the main *UserQ* application then reads off in the belief that they came from *A*. On distributed platforms, the OAM command *SETMQAUT* can be used to ensure that *UserQ* can get messages from a queue but not put them on it. RACF does not have this facility. A user is either able to both get and put messages on a queue or neither. One solution to this is to use ‘alias’ queues.

For example:

```
DEFINE QA(ACCESS.BY.USERQ) TARGQ(QueueQ) PUT(DISABLED)
```

RACF could then be used to be used to prevent *UserQ* from directly accessing queue *QueueQ* while giving the user full access to the *ACCESS.BY.USERQ* alias queue. The *PUT(DISABLED)* attribute ensures that the user can’t put messages on the queue. Note that the *PUT(DISABLED)* attribute could not have been used directly on queue *QueueQ*, as this would have stopped the channel from being able to write messages.

While this method works, it’s a bit of overkill. As it’s common in MVS for a user to have read/write access to a dataset, allowing them also to have read/write access to a queue is usually seen as a natural extension. Another consideration is that, in MVS, it’s less likely that the same *userid* is used to run different applications.

Note that, on all platforms, such measures are unnecessary if the administrator has secured the machine so that users cannot add or run their own applications.

Sam Garforth
MQSeries Consultant
SJG Consulting Limited (UK)

MQSeries and Windows NT security

This article concerns MQSeries Security in a Windows NT environment. It contains information and suggestions provided by IBM MQSeries Level 2 Support, whose assistance in clarifying this confusing subject is much appreciated. For more information regarding MQSeries, please try the following Web address:

```
http://www.software.ibm.com/ts/mqseries/support/tandts/
```

As you begin your foray into MQSeries 5.0 for Windows NT, there are several important areas to be aware of: naming standards for queue managers and related objects, system resources available on the various Windows NT platforms where MQSeries is to run, and network connectivity of these platforms are just three of the more important that need to be considered before venturing too far down the MQSeries road. The most important area, however, is to verify that Windows NT security is properly set up. The set-up program takes care of most of this on your behalf (as it did in version 2.0 of MQSeries), but it may fail in some situations. The first indication of a Windows NT security problem is that the MQSC command **crtmqm**, or one of the other MQSeries programs, abends, and creates a first-failure support technology record (FFST) with a header similar to the one below:

```
+-------------------------------------------------------------------+
|                                                                   |
| MQSeries First Failure Symptom Report                             |
| =====================================                             |
|                                                                   |
| Date/Time         :- Thursday September 24 11:01  1998            |
| Host Name         :- PRODMQ                                       |
| PIDS              :- 5697177                                      |
| LVLS              :- 120                                          |
| Product Long Name :- MQSeries for Windows NT                      |
| Vendor            :- IBM                                          |
| Probe Id          :- XY204105                                     |
| Application Name  :- MQM                                          |
| Component         :- xcsCreateN                                   |
| Build Date        :- Sep 19 1997                                  |
| Userid            :- MQUSR1                                       |
| Process Name      :- C:\MQM\BIN\crtmqm.exe                        |
| Process           :- 00000191                                     |
| Thread            :- 00000000                                  |
| Major Errorcode   :- xecF_E_UNEXPECTED_SYSTEM_RC                |
| Minor Errorcode   :- OK                                        |
| Probe Type        :- MSGAMQ6119                                |
| Probe Severity    :- 2                                         |
| Probe Description :-                                           |
| Comment1          :- WinNT error 1332 from LookupAccountName.  |
|                                                                |
+----------------------------------------------------------------+
```

In MQSeries for Windows NT, FFST information is recorded in a file in the directory *c:\mqm\errors*. These errors are normally severe, unrecoverable errors, and indicate either a configuration problem with the system or an MQSeries internal error.

FFST files are named *AMQnnnnn.mm.FDC*, where:

nnnnn	Is the ID of the process reporting the error
mm	Is a sequence number, normally '0'.

When a process creates an FFST record it also sends a record to the Event Log. The record contains the name of the FFST file to assist in automatic problem tracking. The Event Log entry is made to the 'application' log.

It is important to remember, when setting up Windows NT security for MQSeries v5.0, that MQSeries authenticates user-ids on the same machine on which it itself is installed. Users wishing to have administrative authority over MQSeries must have their user-ids directly or indirectly in the local *mqm* group on the same physical machine on which MQSeries is installed. In V2.0, MQSeries could authenticate user-ids in different places, depending on how the user logged in.

As your MQSeries environment grows, you will undoubtedly have MQSeries installed on several different machines on the domain. Maintaining user-ids on each of those machines can be an administrative nightmare. This task can be avoided by including user-ids indirectly in the local *mqm* groups on MQSeries machines. In other words, add the user-ids to one or more global groups on the primary domain controller (PDC) of the domain and these global groups may then be included in the proper local *mqm* groups.

As is usually the case with security administration, this method can sound confusing. The following scenario should hopefully serve to clear the fog. The steps below assume there are two Windows NT systems in the same NT domain that have MQSeries v5.0 queue managers installed. One of them, *DEVLMQ*, is used for MQSeries development and testing by an application development group. The other one, *PRODMQ*, is the MQSeries production machine. The users *MQUSR1*, *MQUSR2*, *MQUSR3*, *MQUSR4*, and *MQUSR5* need administrative access to *DEVLMQ*, but only *MQUSR1* and *MQUSR2* are allowed to have administrative access to *PRODMQ*.

The steps below guide you through setting up the security structure detailed above in a Windows NT security environment. Thanks again to IBM MQSeries Level 2 for the procedure.

1 Let’s start from scratch to avoid confusion. First delete the local groups called ‘mqm’ on DEVLMQ and PRODMQ.

2 On the Primary Domain Controller (PDC), use the User Manager for Domains tool (usually found under *Start*, *Programs*, *Administrative Tools (Common)* on Windows NT 4.0 machines). Select *New Global Group* from the *User* menu to create a global group with the name ‘MQDev’. Repeat this operation to create a second global group called ‘MQProd’. (You can choose any other names for these groups that are meaningful in your installation, though *MQDev* and *MQProd* are used in the remainder of this example.)

3 Highlight the *MQDev* global group and select the *Add User* function from the *User* menu to add *MQUSR1* to the *MQDev* global group. Repeat this procedure to add *MQUSR2*, *MQUSR3*, *MQUSR4*, and *MQUSR5* to the *MQDev* global group. (As your user base grows, as it undoubtedly will, you can add other user-ids in this same manner.) On the successful completion of the remaining steps, the user-ids included in the *MQDev* global group will have administrative authority for all queue managers on the *DEVLMQ* machine.

4 Highlight the *MQProd* global group and select *Add User* from the *User* menu to add *MQUSR1* to the *MQProd* global group. Repeat

this procedure to add *MQUSR2* to the *MQProd* global group as well. (As indicated in step 3, you can add other user-ids in this same manner.) The user-ids included in the *MQProd* global group will have administrative authority for all queue managers on the PRODMQ machine at the successful completion of the remaining steps.

5 On *DEVLMQ*, enter the *User Manager for Domains* tool (see Step 2 above for its location). Select *New Local Group* from the *User* menu to create a local group called *mqm*. This group name is case-sensitive and must be lower-case only – I have seen MQSeries administrators spend hours trying to debug an access problem only to find that they set up the 'mqm' group as 'MQM'.

6 Double-click the *mqm* group. In the window that pops up, select the *Add* button and then the *MQDev* global group in the dialogue that follows. Click the *Add* button in the middle of the pop-up window, then *OK*.

7 On *PRODMQ*, launch *User Manager for Domains* as in step 2 above. Select *New Local Group* from the *User* menu to create a local group called *mqm*. Again, the group name is case-sensitive and must be lower-case only.

8 Double-click the *mqm* group; in the window that pops up, click the *Add* button. Next select the *MQProd* global group, click the *Add* button in the middle of the pop-up window, and select *OK*.

Steps 2-4 make administering MQSeries security easy. Instead of maintaining and coordinating lists of user-ids in multiple local 'mqm' groups in a domain, only a single set of global groups (in some shops, this will boil down to just one global group) needs to be administered. It is still possible to add users directly to the individual 'mqm' groups, though, and the exact way in which security is administered is left to the discretion of the Windows NT domain administrator. Step 6 provides users in the global group *MQDev* with access to the development MQSeries queue managers on *DEVLMQ*.

Step 8 provides users in the global group *MQProd* with access to the production MQSeries queue managers on *PRODMQ*.

There are a few other points that are important to check if you are having difficulty setting up MQSeries 5.0 on Windows NT:

1 Make sure that your user-id contains 12 or fewer characters. Anything longer than 12 characters (eg 'ADMINISTRATOR') won't work with MQSeries.

2 Make sure that the Server service is running. To do this, open up the Control Panel and double-click on *Services*. The Server service should be listed as *Started* and should also be *Automatic*.

3 If your machine is not on a domain, you simply need to create a local group called 'mqm' on that machine (MQSeries v5.0 will build this group during installation). All user-ids to be given MQSeries rights can then just be added to the local 'mqm' group.

4 If you are running MQSeries v2.0, then you are looking at the wrong set of instructions! The security set up that is required by MQSeries v2.0 on Windows NT is different from that required by MQSeries v5.0. Call IBM MQSeries support for the proper instructions.

Terrence House, IBM Certified MQSeries Specialist
Senior Systems Software Engineer
Boole & Babbage (USA)

Tackling enforced thread affinity

MQSeries provides a simple, platform-independent API for moving large amounts of data across heterogeneous networks. While the API is simple, you may encounter some not-so-simple hurdles where you least expect them. This article provides some insights into one of the thorniest of them.

The quote below, from the *MQSeries Programmers'Reference Manual*, describes the connection handle returned by the *MQCONN* call.

"The scope of the handle is restricted to the smallest unit of parallel processing within the environment concerned; the handle is not valid outside the unit of parallel processing from which the *MQCONN* call was issued."

What this means is that a connection handle returned by an *MQCONN* call can be used on subsequent MQ API calls only from within the thread that made the original call to *MQCONN*. If an attempt is made to use the connection handle on an *MQOPEN* call from another thread, for example, the 'reason code' returned by *MQOPEN* will be 2018 (*MQRC_HCONN_ERROR*).

To see this for yourself try running the following program on Windows NT (the code was developed for Microsoft Visual C++, but should compile on other vendors' C++ compilers with little modification).

PROG1.CPP

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <iostream.h>
#include <cmqc.h>

VOID MQOpenInSecondThread(void* pHcon)
{
    // setup MQOPEN parameters
    MQHOBJ   Hobj;
    MQLONG   CompCode, Reason;
    MQLONG   O_options;
    MQOD     od = {MQOD_DEFAULT};
    ::CopyMemory(od.ObjectName, "SYSTEM.DEFAULT.LOCAL.QUEUE",
                 MQ_Q_NAME_LENGTH);
    O_options = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING;

    // What process and thread are we in?
    cout << "Process ID: " << ::GetCurrentProcessId() <<
         " Thread ID: " << ::GetCurrentThreadId() << endl;

    // attempt to open the system default local queue using the
    // handle from primary thread
    MQOPEN( *((MQHCONN*) pHcon), &od, O_options, &Hobj, &CompCode,
           &Reason);
    cout << "MQOPEN reason code: " << Reason << endl << endl;
}
```

```
int main(int argc, char* argv[])
{
    MQLONG    CompCode, Reason;
    MQHCONN*  pHcon = new MQHCONN;
    char      QMName[50];
    QMName[0] = 0;
    HANDLE th1;

    // What process and thread are we in?
    cout << "Process ID: " << ::GetCurrentProcessId() <<
         " Thread ID: " << ::GetCurrentThreadId() << endl;

    // open a connection to the default queue manager
    MQCONN (QMName, pHcon, &CompCode, &Reason);
    cout << "MQCONN reason code: " << Reason << endl << endl;

    if(Reason == MQRC_NONE)
    {
        // Start a thread and pass it the connection handle
        th1 = CreateThread(NULL, 0,(LPTHREAD_START_ROUTINE)
                           MQOpenInSecondThread,
                           pHcon, 0, NULL);

        // wait for the second thread to complete
        ::WaitForSingleObject(th1, 2000);

        // now try it within the primary thread
        MQOpenInSecondThread(pHcon);
    }
    return 0;
}
```

The output for my run was:

```
Process ID: 254 Thread ID: 124
MQCONN reason code: 0

Process ID: 254 Thread ID: 89
MQOPEN reason code: 2018

Process ID: 254 Thread ID: 124
MQOPEN reason code: 0
```

As you can see, when an *MQOPEN* is attempted from the new thread using the connection handle obtained from the primary thread, MQSeries complains. However, when the same code is run from the primary thread, the *MQOPEN* operation is successful. This behaviour suggests that IBM made the API thread-safe simply by not allowing calls from multiple threads.

WHEN IS ENFORCED THREAD AFFINITY A PROBLEM?

Enforced thread affinity for MQSeries connection handles is not a problem for many applications. Many single-user GUI applications generally don’t have multiple threads, and so don’t need to overcome this limitation. Similarly, thread affinity may not be a problem for server applications that are triggered by the receipt of a message they need to service. Thread affinity is also not generally a problem for applications running on platforms such as CICS, where the management of connection handles is done under the covers.

Where this limitation really requires a solution is in the development of multi-threaded server applications and components that use component architectures such as MTS and COM, which handle concurrence under the covers.

My experience of this type of problem has mostly been during the development of Web-based systems that require multi-threaded server applications conforming to the classic ‘boss/worker’ threading model. Two principal cases occur: in one the server receives MQ messages as requests in a boss thread and dispatches them to a worker thread obtained from a thread pool – in such cases the worker thread services the request and returns a reply to a reply-to-queue; in the other scenario a worker thread sends an MQ message as a request to another server and waits for the reply. The goal in both cases is to minimize both the latency associated with obtaining MQ connections and the total number of MQ connections required. If these were not issues, each thread could simply maintain its own connection or establish a new connection for each request. Minimizing the number of connections facilitates scaling in applications where this is important.

SOLUTIONS

Three solutions come to mind:

1 If this limitation isn’t a problem for your application, you could just live with it.

2 You could create a wrapper for the MQ API.

3 You could use Message Manager Objects that expose a few methods to your application.

I have used both the first and last options, with Message Manager Objects being the preferred choice.

Live with it

If the number of connections or the time required to obtain them is not a problem for your application, simply maintain a connection for every thread that needs one, or obtain a new connection each time a thread needs one. Messages can be passed between threads using the heap.

Wrap the MQ API

Another solution is to maintain a pool of thread and MQ connection pairs and wrap the MQ API with your own calls that hand out connections from the pool. When programs call your wrapper functions, parameter data is put on the heap and the thread associated with the connection is sent an event. When the event wakes up the corresponding thread, it looks at its input buffer on the heap to discover which function needs to be implemented and to read the input parameters. One warning: never try to pass data between threads using the stack.

Message Manager Objects (the preferred solution)

My solutions have always used 'Message Manager Objects'. These generally involve three types of object: a *send manager*, a *receive manager*, and a *send/receive manager* (a send/receive manager contains a send manager and a receive manager). Both the receive manager and send manager maintain their own thread and MQ connection. The send manager's job is to send messages, the receive manager's job is to receive them. The send/receive manager's job is to coordinate the two efforts.

When worker threads need to send messages but don't need replies, they invoke the *SendMessage* method on the send/receive manager. This causes the message to be added to a collection and the send manager's thread to be sent an event. The send manager finds the message in the collection and calls *MQPUT* to put the message on the appropriate queue.

When worker threads need to send messages requiring matching responses, they invoke the *talk* method on the send/receive manager. This causes the send manager to be sent an event, as before, though in this instance a unique message identifier is entered into the collection of messages waiting a response. The response is received by the receive manager, which then uses the correlation identifier as an index to the collection of messages waiting for a response. If the receive manager finds a match, it passes the response to the waiting worker thread.

The receive manager continually listens for messages and can receive unsolicited messages in addition to message responses. Unsolicited messages can be useful for unscheduled broadcasts to all application servers, which can be used by client applications to request services from the application servers.

The message manager approach isn't as generic as the API wrapper approach, but its benefits include allowing many worker threads to share just two MQ connections, insulating threads from the MQ API, and being more efficient as fewer synchronization calls are required.

The following sample code demonstrates how a thread that doesn't have affinity with an MQ connection handle can *MQPUT* a message. While this is a trivial example, it nevertheless provides insights that may help you to build the more complete solution using Message Manager Objects. Note that, in this example, error handling and code elegance have been sacrificed for brevity.

PROG2.CPP

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <iostream.h>
#include <cmqc.h>

#pragma warning(disable:4786)
#include <vector>
using namespace std;

class CMQmessage
{
public:
    CMQmessage(const char* msg, long msgLen, const char* destQ);
    char m_msg[100];
    long m_msgLen;
    char m_destQ[MQ_Q_NAME_LENGTH];
};

typedef vector<CMQmessage> VECTMQMESSAGE;

CMQmessage::CMQmessage(const char* msg, long msgLen, const char* destQ)
{
    ::ZeroMemory(m_msg, sizeof(m_msg) );
    ::ZeroMemory(m_destQ, sizeof(m_destQ) );
    ::CopyMemory(m_msg, msg, msgLen);
    m_msgLen = msgLen;
    ::CopyMemory(m_destQ, destQ, strlen(destQ) );
}

class CMQSendMgr
{

public:
    CMQSendMgr();

    static void statRun(void* pThis);
    void Run();
    void SendMessage(CMQmessage* msg);

    MQHCONN          m_hConn;
    MQLONG           m_CompCode, m_Reason;
    char             m_QMName[50];
    HANDLE           m_th1;
    HANDLE           m_msgEvent;
    VECTMQMESSAGE    m_msgs;
    CRITICAL_SECTION m_msgs_cs;
};

CMQSendMgr::CMQSendMgr()
{
    cout << "CMQSendMgr::CMQSendMgr" << endl;

    InitializeCriticalSection(&m_msgs_cs);

    // Start a thread and pass it a pointer to this object instance
    m_th1 = CreateThread(NULL, 0,
                        (LPTHREAD_START_ROUTINE) CMQSendMgr::statRun,
                        this, 0, NULL);
    // wait for the Run method to get going
    ::WaitForSingleObject(m_th1, 1000);
}

void CMQSendMgr::statRun(void* pThis)
```

```
{
    cout << "CMQSendMgr::statRun" << endl;
    // start waiting for requests
    ((CMQSendMgr*) pThis)->Run();
}

void CMQSendMgr::Run()
{
    // What process and thread are we in?
    cout << "CMQSendMgr::Run           Process ID: " <<
         ::GetCurrentProcessId()  <<  " Thread ID: " <<
         ::GetCurrentThreadId() << endl;

    m_QMName[0] = 0;

    // open a connection to the default queue manager
    MQCONN (m_QMName, &m_hConn, &m_CompCode, &m_Reason);
    cout << "CMQSendMgr::Run           MQCONN reason code: " <<
         m_Reason << endl << endl;

    // create an event to wait for messages
    m_msgEvent = ::CreateEvent(NULL, FALSE, FALSE, NULL);

    // loop waiting for send requests
    while(1)
    {
        cout << "CMQSendMgr::Run           Entering Run Loop..."
             << endl;

        // wait for a message to arrive
        ::WaitForSingleObject(m_msgEvent, INFINITE);

        cout << "CMQSendMgr::Run           Message arrived..."
             << endl;

        EnterCriticalSection(&m_msgs_cs);

        VECTMQMESSAGE::iterator itmsgs;
        itmsgs = m_msgs.begin();
        if(itmsgs != m_msgs.end() )
        {
            cout << "CMQSendMgr::Run           Putting the message"
                 << endl;
            MQOD ObjDesc       = {MQOD_DEFAULT};
            strncpy(ObjDesc.ObjectName, (*itmsgs).m_destQ,
                    MQ_Q_NAME_LENGTH);
            MQMD MsgDesc       = {MQMD_DEFAULT};
            MQPMO PutMsgOpts   = {MQPMO_DEFAULT};
            PutMsgOpts.Options = MQPMO_NO_SYNCPOINT +
                                 MQPMO_FAIL_IF_QUIESCING;
            MQPUT1(m_hConn, &ObjDesc, &MsgDesc, &PutMsgOpts,
                   (*itmsgs).m_msgLen, (*itmsgs).m_msg,
                   &m_CompCode, &m_Reason);
            cout << "CMQSendMgr::Run           MQPUT1 reason code: "
                 << m_Reason << endl << endl;

            m_msgs.erase(itmsgs);
        }
        LeaveCriticalSection(&m_msgs_cs);
    }
}

void CMQSendMgr::SendMessage(CMQmessage* pMsg)
{
    cout << "CMQSendMgr::SendMessage   Process ID: " <<
         ::GetCurrentProcessId()  <<  " Thread ID: " <<
         ::GetCurrentThreadId() << endl;

    EnterCriticalSection(&m_msgs_cs);
    m_msgs.insert(m_msgs.end(), *pMsg );
    LeaveCriticalSection(&m_msgs_cs);

    ::SetEvent(m_msgEvent);
}

int main(int argc, char* argv[])
{

    // What process and thread are we in?
    cout << "main                      Process ID: " <<
         ::GetCurrentProcessId()  <<  " Thread ID: " <<
         ::GetCurrentThreadId() << endl;

    // Create a message to send - it must be on the heap
    CMQmessage* pMsg = new CMQmessage("TEST MESSAGE", 12,
                                  "SYSTEM.DEFAULT.LOCAL.QUEUE");

    // Instantiate a CMQSendMgr
    CMQSendMgr* pSender = new CMQSendMgr();

    // Send a message
    pSender->SendMessage(pMsg);

    // Continue the process long enough for send manager
    // to find the message
    Sleep(5000);

    return 0;
}
```

A sample output of this program is shown below.

```
main                        Process ID: 262 Thread ID: 217
CMQSendMgr::CMQSendMgr
CMQSendMgr::statRun
CMQSendMgr::Run             Process ID: 262 Thread ID: 270
CMQSendMgr::Run             MQCONN reason code: 0

CMQSendMgr::Run             Entering Run Loop...
CMQSendMgr::SendMessage     Process ID: 262 Thread ID: 217
CMQSendMgr::Run             Message arrived...
CMQSendMgr::Run             Putting the message
CMQSendMgr::Run             MQPUT1 reason code: 0

CMQSendMgr::Run             Entering Run Loop...
```

You can see from the above that a call to the *SendMessage* method comes from the main thread, which is different from the thread used to obtain the MQ connection handle used by the *MQPUT1* call.

CONCLUSION

Evaluate your application and environment carefully before designing your solution, as you may not need to invest significant effort to overcome the enforced thread affinity hurdle. Develop 'use cases' for your application – this will help you establish whether you'll ever need to send or receive messages from a thread that doesn't establish a connection to an MQSeries queue manager.

Neil Harvey
Independent Consultant
Neil Harvey Information Technologies (USA)

Recovery procedures

This article presents a number of utilities for recovering MQSeries on a mainframe. The utilities are concerned with such tasks as ensuring that logging, log files, and datasets are handled correctly on restarting

MQSeries. Each utility is prefaced by a brief description of what it does and there are also comments in the code that should be read before any of the utilities are run. Note that some of the utilities rely on the successful completion of others – for this reason it's a good idea to read through all the comments before using any of the utilities.

First, though, here are some general comments to give you some background on the subject.

PAGE DATASETS

Page datasets are VSAM linear datasets (LDS) that are used to store messages and object definitions (page set '00' is used to store object definitions).

LOG FILES

A log dataset is made up of data records, each of which is handled by the system as a single unit, identified by the Relative Byte Address (RBA) of the first byte of its header – the RBA is the offset from the beginning of the log.

Three types of log record are written, which are associated with the maintenance and recovery of the following:

- *Unit of recovery*
  Any change to a queue is made within a unit of recovery.
- *Checkpoint*
  Checkpoints are taken by the MQSeries subsystem at the end of successful restarts, when the program terminates normally, and during normal operation when a pre-defined number of log records have been written.
- *Page set control*
  These records register page sets that are known to the MQSeries subsystem at each checkpoint.

If log files become corrupt, then back-up or archive logs (or new log files) that are to be used must be validated to the page sets. MQSeries' CSQUTIL utility handles this task.

BOOTSTRAP DATA SET (BSDS)

A BSDS is a VSAM keyed sequence (KSDS) dataset that stores information about log files. This information allows the MQSeries subsystem to locate log records so that it can handle restart processing and satisfy log read requests during normal processing. For active logs, the information kept shows which logs are full and which are available for reuse.

MQSeries supports ‘dual BSDS’, with a copy being written to the archive log.

ARCHIVE LOG

A copy of the BSDS is written to all archive logs. This comprises two datasets written in one operation; the first is the most recent image of the BSDS and the second is the archive itself. The names of the two are the same except that the lowest-level qualifier name for the archive log begins with ‘A’, while the BSDS copy begins with ‘B’. For example:

Archive Log:

```
    MQARCH.LOG1.D98279.T1016438.A0000021
                                ^
```

BSDS copy name:

```
    MQARCH.LOG1.D98279.T1016438.B0000021
                                ^
```

Listed below are various sample utilities (which need customizing to your installation’s needs) that may be used to handle failures in which any of the above datasets become corrupted. As stated before, it’s important to read all the comments in the JCL and perform the necessary actions.

ADDLOG

If archiving is delayed or stopped, use this utility to add a new active log. This enables MQSeries to continue logging and stops it from terminating.

ADDLOG

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* Add new active log in case of excessive logging or delayed
//* archiving.
//*
//* Define LOG file
//*******************************************************************
//*
//ADDLOG   EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
    DEFINE CLUSTER                                              -
        (NAME (SYS1.MQS.LOGCOPY3.DS01)                          -
        LINEAR                                                  -
        VOLUMES(SYS000)                                         -
        RECORDS(36750))                                         -
    DATA                                                        -
        (NAME(SYS1.MQS.LOGCOPY3.DS01.DATA))

    DEFINE CLUSTER                                              -
        (NAME (SYS1.MQS.LOGCOPY3.DS02)                          -
        LINEAR                                                  -
        VOLUMES(SYS000)                                         -
        RECORDS(36750))                                         -
    DATA                                                        -
        (NAME(SYS1.MQS.LOGCOPY3.DS02.DATA))

/*
//*
//*******************************************************************
//* Register new LOG file in the BSDS.
//*
//* Add to MQ start-up procedure.
//*******************************************************************
//*
//JU003     EXEC PGM=CSQJU003
//STEPLIB   DD DSN=SYS1.SCSQANLE,DISP=SHR
//          DD DSN=SYS1.SCSQSNLE,DISP=SHR
//          DD DSN=SYS1.SCSQAUTH,DISP=SHR
//SYSUT1    DD DISP=OLD,DSN=SYS1.MQS.BSDS01
//SYSUT2    DD DISP=OLD,DSN=SYS1.MQS.BSDS02
//SYSPRINT  DD SYSOUT=*,DCB=BLKSIZE=629
//SYSIN     DD *
 NEWLOG DSNAME=SYS1.MQS.LOGCOPY3.DS01,COPY1
 NEWLOG DSNAME=SYS1.MQS.LOGCOPY3.DS02,COPY2
/*
//*
```

COPYPAGE

This may be used to define new page sets or expand existing ones.

COPYPAGE

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* Define new page sets or expand existing ones.
//*******************************************************************
//*
    DEFINE CLUSTER                                              -
        (NAME(SYS1.MQS.NEW.PSID00)                              -
        RECORDS(50000 2500)                                     -
        LINEAR                                                  -
        VOLUMES(SYS000)                                         -
        SHAREOPTIONS(2 3))                                      -
    DATA                                                        -
        (NAME(SYS1.MQS.NEW.PSID00.DATA))

    DEFINE CLUSTER                                              -
        (NAME(SYS1.MQS.NEW.PSID01)                              -
        RECORDS(50000 2500)                                     -
        LINEAR                                                  -
        VOLUMES(SYS000)                                         -
        SHAREOPTIONS(2 3))                                      -
    DATA                                                        -
        (NAME(SYS1.MQS.NEW.PSID01.DATA))
/*
//*
//*******************************************************************
//* Format and copy old page sets to new.
//*
//* Amend page set reference in MQ start-up procedure.
//*******************************************************************
//*
//CSQUTIL EXEC PGM=CSQUTIL
//STEPLIB   DD DSN=SYS1.SCSQANLE,DISP=SHR
//CSQP0000  DD DISP=OLD,DSN=SYS1.MQS.NEW.PSID00
//CSQP0001  DD DISP=OLD,DSN=SYS1.MQS.NEW.PSID01
//CSQS0000  DD DISP=OLD,DSN=SYS1.MQS.PSID00
//CSQS0001  DD DISP=OLD,DSN=SYS1.MQS.PSID01
//CSQT0000  DD DISP=OLD,DSN=SYS1.MQS.NEW.PSID00
//CSQT0001  DD DISP=OLD,DSN=SYS1.MQS.NEW.PSID01
//SYSPRINT  DD SYSOUT=*
//SYSIN     DD *
* format new datasets (CSQP0000 and CSQP0001) as new page sets
```

```
FORMAT
/*
* copy old page sets CSQS0000 and CSQS0001 to new page sets
* CSQT0000 and CSQT0001.
COPYPAGE
/*
```

CSQ1LOGP

This job depends on the successful completion of job RECARCH.

When a BSDS is recovered from an archive log, the RBAs of the active log in the BSDS are inconsistent and this prevents MQSeries from starting. Run this job to print a summary report of the active logs and record starting and ending RBAs.

CSQ1LOGP

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* From job RECARCH
//*******************************************************************
//* When a BSDS is recovered from an archive log, the RBAs of the
//* active log in the BSDS are inconsistent and MQ will not start.
//*
//* Run this job to print a summary report of the active logs and
//* record starting and ending RBAs.
//*
//* Edit the JCL wherever necessary.
//*******************************************************************
//*
//LOGP      EXEC PGM=CSQ1LOGP
//STEPLIB   DD DSN=SYS1.SCSQANLE,DISP=SHR
//          DD DSN=SYS1.SCSQLOAD,DISP=SHR
//ACTIVE1   DD DISP=OLD,DSN=SYS1.MQS.LOGCOPY1.DS01
//ACTIVE2   DD DISP=OLD,DSN=SYS1.MQS.LOGCOPY1.DS02
//SYSPRINT  DD SYSOUT=*,DCB=BLKSIZE=629
//SYSSUMRY  DD SYSOUT=*,DCB=BLKSIZE=629
//SYSIN     DD *
Insert control cards here
/*
//*******************************************************************
//* Go to RECARCH1
//*******************************************************************
```

JOBCARD (THE JOB HEADER STATEMENT)

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
```

PRTBSDS

This job depends on the successful completion of job RCBSDS2.

PRTBSDS prints the contents of the BSDS. This enables RBAs to be synchronized, which is a necessary step in the recovery process.

PRTBSDS

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* From job RCBSDS2
//*******************************************************************
//* Print the contents of the BSDS.
//*
//* Edit the JCL wherever necessary.
//*******************************************************************
//*
//JU004     EXEC PGM=CSQJU004
//STEPLIB   DD DSN=SYS1.SCSQANLE,DISP=SHR
//          DD DSN=SYS1.SCSQSNLE,DISP=SHR
//          DD DSN=SYS1.SCSQAUTH,DISP=SHR
//SYSPRINT  DD SYSOUT=*,DCB=BLKSIZE=629
//SYSUT1    DD DISP=SHR,DSN=SYS1.MQS.BSDS01
//*
//*SYSUT2  DD DISP=SHR,DSN=SYS1.MQS.BSDS02
//*
//*******************************************************************
//* Go to job RECARCH
//*******************************************************************
```

RCBSDS1

If the BSDS is damaged and MQSeries has not terminated, delete the existing BSDS and define a new one with the same name as the one that’s damaged.

Issue command RECOVER BSDS after this job completes. A copy of the valid BSDS is made in the newly allocated dataset.

RCBSDS1

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* Damaged BSDS: READ ALL COMMENTS BEFORE RUNNING THIS JOB
//*******************************************************************
//* Delete existing BSDS and define a new one with the same name as
//* the damaged one.
//*
//* Edit the JCL wherever necessary.
//*******************************************************************
//* Issue the command RECOVER BSDS after this job completes.
//*
//* A copy of the valid BSDS is created in the newly allocated dsn.
//*******************************************************************
//*
//RECBSDS  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
    DELETE (SYS1.MQS.BSDS01)      ERASE CLUSTER
    SET MAXCC = 0
    DEFINE CLUSTER                                          -
        (NAME(SYS1.MQS.BSDS01)                              -
        VOLUMES(SYS000)                                     -
        SHAREOPTIONS(2 3))                                  -
    DATA                                                    -
        (NAME(SYS1.MQS.BSDS01.DATA)                         -
        RECORDS(3000 3000)                                  -
        RECORDSIZE(4089 4089)                               -
        CONTROLINTERVALSIZE(4096)                           -
        FREESPACE(0 20)                                     -
        KEYS(4 0))                                          -
    INDEX                                                   -
        (NAME(SYS1.MQS.BSDS01.INDEX)                        -
        RECORDS(5 5)                                        -
    CONTROLINTERVALSIZE(1024))
/*
//
```

RCBSDS1N

If the BSDS is damaged and MQSeries terminated before it could be recovered, the next attempt to restart MQSeries will also fail. So delete the existing BSDS and define a new one with same name as the damaged one. ‘REPRO’ the valid BSDS to the redefined BSDS.

RCBSDS1N

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* Damaged BSDS:     READ ALL COMMENTS BEFORE RUNNING THIS JOB
//*******************************************************************
//* If MQ terminates before the damaged BSDS can be recovered,
//* the next restart also fails. To correct this, delete the existing
//* BSDS and define a new one with same name as the damaged one.
//*
//* Edit the JCL wherever necessary.
//*******************************************************************
//*
//RECBSDS  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
    DELETE (SYS1.MQS.BSDS01) ERASE CLUSTER
     SET MAXCC = 0
    DEFINE CLUSTER                                              -
        (NAME(SYS1.MQS.BSDS01)                                  -
        VOLUMES(SYS000                                          -
        SHAREOPTIONS(2 3))                                      -
    DATA                                                        -
        (NAME(SYS1.MQS.BSDS01.DATA)                             -
        RECORDS(3000 3000)                                      -
        RECORDSIZE(4089 4089)                                   -
        CONTROLINTERVALSIZE(4096)                               -
        FREESPACE(0 20)                                         -
        KEYS(4 0))                                              -
    INDEX                                                       -
        (NAME(SYS1.MQS.BSDS01.INDEX)                            -
        RECORDS(5 5)                                            -
        CONTROLINTERVALSIZE(1024))
/*
//
//*******************************************************************
//* Copy the valid BSDS to the redefined BSDS.
//*
//* Restart MQ subsystem
//*******************************************************************
//*
//REPBSDS  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//DD1      DD DISP=SHR,DSN=SYS1.MQS.BSDS02
//DD2      DD DISP=SHR,DSN=SYS1.MQS.BSDS01
//SYSIN    DD *
    REPRO                                                       -
        INFILE(DD1)                                             -
        OUTFILE(DD2)
/*
//
```

RCBSDS2

If both BSDSs are damaged, the several jobs have to be run in sequence to recover the BSDSs (refer to ‘Order of jobs’ at the end of this article for the order).

Locate the BSDS associated with the most recent archive log by looking up the last occurrence of message *CSQJ0031* in MQSeries’ STC output, which indicates whether off-loading completed successfully.

For example:

```
BSDS01 ==> MQARCH.LOG1.xxxxxx.xxxxxxxx.Bxxxxxxx
BSDS02 ==> MQARCH.LOG2.xxxxxx.xxxxxxxx.Bxxxxxxx
```

Then either rename or delete and redefine the damaged BSDSs and ‘REPRO’ the BSDS from the archive log to *one* of the replacement BSDSs.

RCBSDS2

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*****************************************************************
//* Both BSDS damaged: READ ALL COMMENTS BEFORE RUNNING THIS JOB.
//*
//* See also pages 306 to 308 of MQ Systems Management Guide
//*****************************************************************
//*
//* Locate the BSDS associated with the most recent archive log by
//* looking up the last occurrence of message CSQJ0031 in the MQ STC
//* output, which indicates whether off-loading was successful.
//*
//* eg  BSDS01 ==> MQARCH.LOG1.xxxxxx.xxxxxxxx.Bxxxxxxx
//*     BSDS02 ==> MQARCH.LOG2.xxxxxx.xxxxxxxx.Bxxxxxxx
//*
//*****************************************************************
//* Either rename the damaged BSDSs or delete and redefine them.
//*
//* Edit the JCL wherever necessary.
//*****************************************************************
```

```
//*
//RECBSDS  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
    DELETE (SYS1.MQS.BSDS01)      ERASE CLUSTER
    DELETE (SYS1.MQS.BSDS02)      ERASE CLUSTER
    SET MAXCC = 0
    DEFINE CLUSTER                                           -
        (NAME(SYS1.MQS.BSDS01)                               -
        VOLUMES(SYS000)                                      -
        SHAREOPTIONS(2 3))                                   -
    DATA                                                     -
        (NAME(SYS1.MQS.BSDS01.DATA)                          -
        RECORDS(3000 3000)                                   -
        RECORDSIZE(4089 4089)                                -
        CONTROLINTERVALSIZE(4096)                            -
        FREESPACE(0 20)                                      -
        KEYS(4 0))                                           -
    INDEX                                                    -
        (NAME(SYS1.MQS.BSDS01.INDEX)                         -
        RECORDS(5 5)                                         -
        CONTROLINTERVALSIZE(1024))

     DEFINE CLUSTER                                          -
        (NAME(SYS1.MQS.BSDS02)                               -
        VOLUMES(SYS000)                                      -
        SHAREOPTIONS(2 3))                                   -
    DATA                                                     -
        (NAME(SYS1.MQS.BSDS02.DATA)                          -
        RECORDS(3000 3000)                                   -
        RECORDSIZE(4089 4089)                                -
        CONTROLINTERVALSIZE(4096)                            -
        FREESPACE(0 20)                                      -
        KEYS(4 0))                                           -
    INDEX                                                    -
        (NAME(SYS1.MQS.BSDS02.INDEX)                         -
        RECORDS(5 5)                                         -
        CONTROLINTERVALSIZE(1024))
/*
//*
//*********************************************************************
//* Copy the archive log's BSDS to one of the replacement BSDSs.
//*********************************************************************
//*
//REPBSDS  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//DD1      DD DISP=SHR,DSN=MQARCH.LOG1.xxxxxx.xxxxxxxx.Bxxxxxxx
//DD2      DD DISP=SHR,DSN=SYS1.MQS.BSDS01
//SYSIN    DD *
```

```
    REPRO                                                      -
        INFILE(DD1)
        OUTFILE(DD2)
/*
//
//*******************************************************************
//* Go to job PRTBSDS
//*******************************************************************
```

RCBSDS3

This job depends on the successful completion of job RECLOG3.

If both BSDSs are damaged, copy the recovered BSDS to the second replacement BSDS, pre-defined in job RCBSDS2.

RCBSDS3

```
//JOBID JOB CLASS=S,NOTIFY=&SYSUID
//*
//*******************************************************************
//* From job RECLOG3
//*******************************************************************
//* Both BSDSs damaged: READ ALL COMMENTS BEFORE RUNNING THIS JOB.
//*
//* Copy the recovered BSDS to the second replacement BSDS, which is
//* pre-defined in job RCBSDS2
//*******************************************************************
//*
//REPBSDS  EXEC PGM=IDCAMS,REGION=4M
//SYSPRINT  DD SYSOUT=*
//SYSUDUMP  DD SYSOUT=*
//DD1       DD DISP=SHR,DSN=SYS1.MQS.BSDS01
//DD2       DD DISP=SHR,DSN=SYS1.MQS.BSDS02
//SYSIN     DD *

    REPRO                                                      -
        INFILE(DD1)
        OUTFILE(DD2)
/*
//
```

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

Saida Davies (UK) © Xephon 1999

MQSeries news

IBM has unveiled MQSeries 5.1, with improvements to the base product and the introduction of two new products: MQSeries Integrator (the IBM-branded version of NEON's MQIntegrator product) and MQSeries Workflow. Version 5.1 for AIX, Solaris, HP-UX, NT, and OS/2 Warp incorporates new dynamic workload distribution across distributed platforms and MVS. Also new are publish-and-subscribe facilities.

Also announced was MQSeries for OS/390 Version 2.1 and MQSeries Workflow for OS/390 Version 3.1.

For further details see the 'MQSeries announcements' in this issue.

* * *

IBM's Tivoli subsidiary has released Tivoli Manager for MQSeries for OS/390 Version 2.2, which has added functionality and now offers administrators the ability to manage a company-wide MQSeries infrastructure from a single desktop PC. Support for the Tivoli Enterprise Console (TEC) Adapter for OS/390 has also been improved to enable NetView event notification and automation, bringing the product more in line with the versions available for Unix and Windows.

Tivoli Manager for MQSeries can correlate data from components, such as network hardware, with events generated by other middleware and applications and present the information in a unified context. The product also allows MQSeries requests to pass to an OS/390 system.

Out now, prices remain the same as the previous version.

For further details contact:
Tivoli Systems, 9442 Capital of Texas Highway, N Austin, TX 78759, USA
Tel: +1 512 436 8000
Fax: +1 512 794 0623
Web: http://www.tivoli.com

* * *

Convoy has announced Convoy/DM Version 3.0, the latest version of its product for developing and managing application interfaces. It also announced the integration of MQSeries into the product. Convoy/DM uses meta data about business applications to generate code for data conversion and interfaces. New features in Version 3.0 are aimed at merging multiple units of work to multiple targets, extracting meta data from ODBC-compatible sources, creating user-definable code exits, and meta data reporting.

Out now, Version 3.0 starts at US$32,000.

For further details contact:
Convoy Corporation, 2000 Powell Street, Suite 1380, Emeryville, CA 94608, USA
Tel: +1 510 601 4950
Fax: +1 510 601 4940
Web: http://www.convoy.com

Convoy Corporation Limited, 268 Bath Road, Slough, Berks, SL1 4DX, UK
Tel: +44 1753 708887
Fax: +44-1753 708810