24

MQ update

June 2001

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/contnote.html.

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/mqupdate.html (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 ($33.50) each including postage.

Printed in England.

MQSeries V5.2: clusters update

When IBM announced the raft of MQSeries V5.2 products (with the notable absence of MQ for OS/2!) most people picked up on the performance enhancements as well as the new shared queue capability on the OS/390 platform. Very little was written about the clustering enhancements made and this article aims to rectify matters. It will only touch upon the new enhancements; if you require more detailed information please refer to the MQSeries clustering article that appeared in the September and October 2001 issues of *MQ Update*, and the following manuals:

- *MQSeries Release Guide GC34-5761-01*.
- *MQSeries Queue Manager Clusters SC34-5349-03*.

The specific enhancements added to the clustering capabilities include the ability to:

- Define a cluster-receiver channel without specifying the queue manager's network address.
- Define a cluster-sender channel without specifying the name of the repository queue manager.
- Dynamically allocate the cluster cache.

THE PROBLEM WHEN USING DYNAMIC IP ADDRESSES

If your environment uses the Dynamic Host Configuration Protocol (DHCP), which automatically generates a new IP address when a machine reconnects, prior to V5.2, you had to stop the channel, change its address as part of the *CONNAME* field, and restart it.

If an existing conversation was active when the IP address changed, then you may have needed to reset the message sequence numbers on the channels.

The purpose of these enhancements is to simplify further the task of the system administrator. It is no longer necessary for you to know the network address of your queue manager, nor the names of the other queue managers in the cluster.

THE SOLUTION

In an environment where TCP/IP is used as the network protocol, the cluster-receiver channel no longer requires the network address of the queue manager. In fact, when you make the definition, you can leave the *CONNAME* blank. MQSeries automatically generates a *CONNAME* for you using the current IP address of the system. The generated *CONNAME* is always in the dotted-decimal form (*x.x.x.x*) rather than in the form of an alphanumeric DNS host name. Note that this only works when the default port number – 1414 – is being used.

How does this work with the repository? The MQSeries-generated *CONNAME* is stored in the repositories, and, therefore, the other queue managers in the cluster do not know that the *CONNAME* was originally blank.

Be careful when you issue the **DISPLAY CHANNEL** command because the *CONNAME* shown is not the generated address, and will be blank. Instead, use the **DISPLAY CLUSQMGR** command.

A restart of the queue manager with a different IP address (courtesy of DHCP) causes MQSeries to change the *CONNAME* and store it in the repositories.

Auto-defined cluster-sender channels – careful!

As we know, clustering means having to define fewer channels. The cluster receiver of the remote queue manager is used as a template to auto-define the cluster sender – even manually-defined cluster-sender channels get modified.

It is possible, therefore, to define a cluster receiver (as mentioned above) without a *CONNAME*, which MQSeries then changes to the local IP address, and for the cluster sender to be defined with the same *CONNAME*. If the remote queue manager does not listen to the default port 1414, then the channel will go into a retry state.

STEPS REQUIRED TO USE DHCP WITH MQSERIES

- Suspend the local queue manager from the cluster.

```
SUSPEND QMGR CLUSTER(cccccc)
```

- Stop the cluster-receiver channel on the queue manager.

  ```
  STOP CHL(yyyy)
  ```

- Alter the *CLUSRCVR* definition to remove the *CONNAME* or set it to blank.

  ```
  ALTER CHL(yyyy) CHLTYPE(CLUSRCVR) CONNAME('')
  ```

- Put queue manager back in the cluster.

  ```
  RESUME QMGR CLUSTER(cccccc)
  ```

- Start the cluster receiver channel.

  ```
  START CHL(yyyy)
  ```

Do not use DHCP for queue managers with a *FULL* repository.

Queue managers with a *FULL* repository are known either by their IP address or *HOSTNAME*, as specified in the *CONNAME*. When another queue manager wants to join the cluster, its cluster-sender channel definition must refer to the *CONNAME* of the *FULL* repository. In an environment where DHCP is used, the IP address could have changed, so a *CONNAME* of the format *x.x.x.x* would not work.

Even if a *HOSTNAME* is specified you may not be able to rely upon DHCP to update the DNS directory with the new address for the host.

CHANGES TO THE CLUSTER-SENDER CHANNEL WHEN JOINING A CLUSTER

To further reduce the administrative task, when defining a cluster-sender channel in order to join a cluster, you do not need to know which queue manager hosts the *FULL* repository, provided that the channel-naming convention has the name of the queue manager that hosts the *FULL* repository.

For example, if queue manager *QM1* has a cluster-receiver channel called *QM1*, then on the queue manager joining the cluster, say *QM2*, you define the cluster-sender channel as follows:

```
DEF CHL(+QMNAME+) CHLTYPE(CLUSSDR) TRPTYPE(TCP) CONNAME(cccccc)
```

MQSeries uses the *CONNAME* to get to the queue manager, interrogates the repository, and substitutes *+QMNAME+* with the name of the

queue manager hosting the *FULL* repository – in this case with the name *QM1*.

HOW IT WORKS IN PRACTICE

We've seen the theory – now let's 'prove' it by using some worked examples. The environment I've used is Windows 2000 with MQSeries for Win NT/2000 V5.2 (no CSDs). Definitions were made via the MQSeries explorer GUI and confirmed via the **RUNMQSC** commands in a DOS window.

- Create a queue manager called *QM1*.
- On *QM1*, alter it to have a repository called *CLUS1*.
- On *QM1*, define a cluster-receiver channel called *QM1* within cluster *CLUS1* and a specific *CONNAME* with a TCP/IP port number, eg *Win2000-pc(1500)*.
- Create a queue manager called *QM2*.
- On *QM2*, define a cluster receiver called *QM2* within the cluster *CLUS1* but with a blank *CONNAME*. (See Figure 1.)

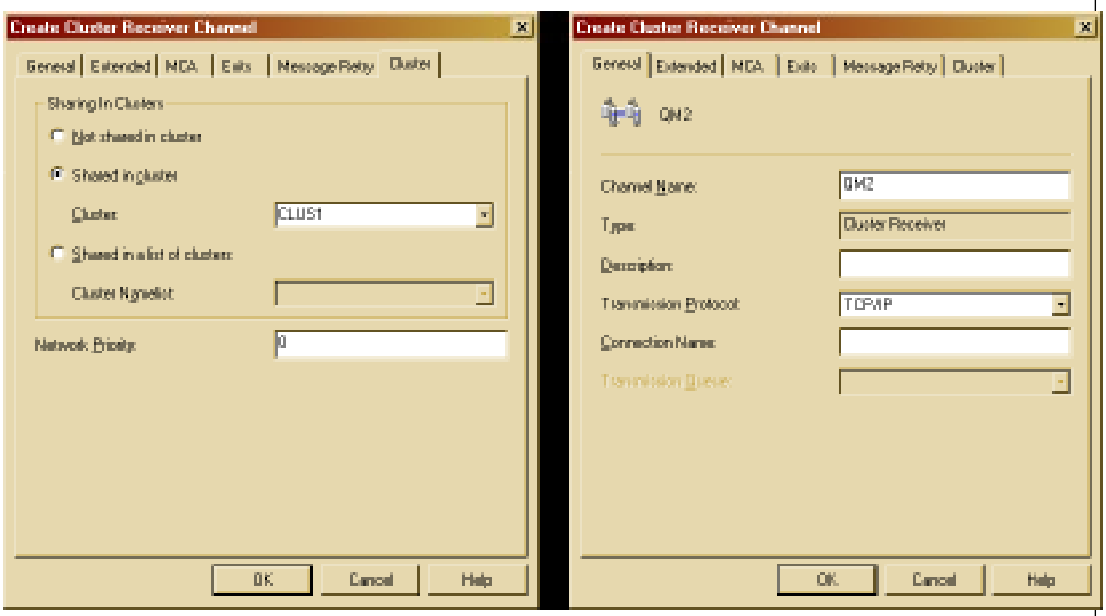

Figure 1: Defining QM1 and QM2

- On *QM2*, define a cluster-sender channel called *+QMNAME+* within the cluster *CLUS1* but pointing to a specific *CONNAME*, eg *Win2000-pc(1500)*. (See Figure 2.)
- When the screen is refreshed, the *+QMNAME+* has been changed to *QM1*, as Figure 3 illustrates.

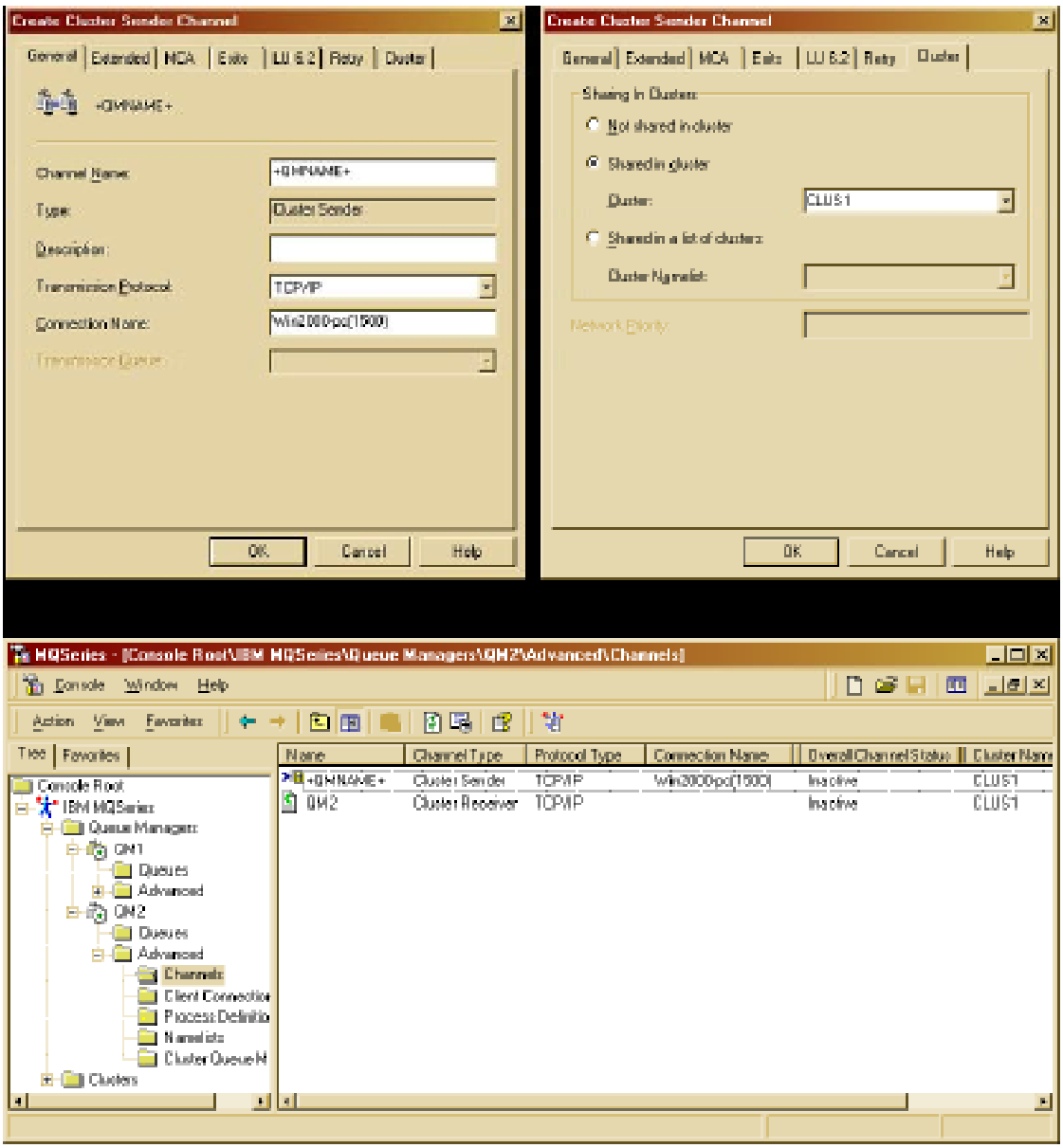

Figure 2: Defining the cluster-sender channel within CLUS1, pointing to a specific CONNAME

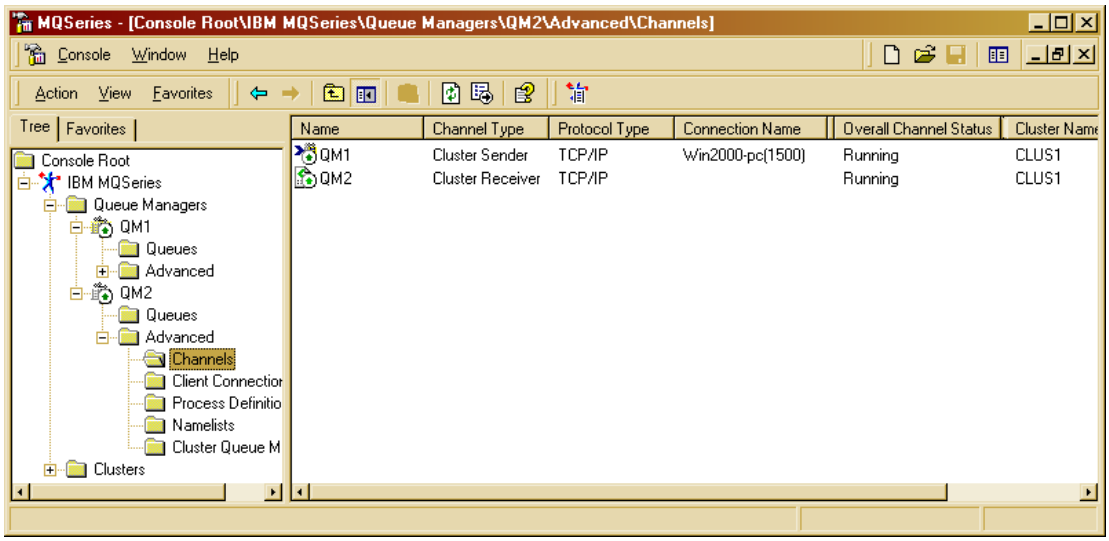

Figure 3: +QMNAME+ changes to QM1

To confirm what MQSeries has done, run **RUNMQSC** against queue manager *QM2*.

First, display the channel status:

```
C:\>runmqsc QM2
Ø784726, 5639-B43 (C) Copyright IBM Corp 1994, 2ØØØ.  ALL RIGHTS
RESERVED.
Starting MQSeries Commands.
dis chs(*)
     1 : dis chs(*)
AMQ8417: Display Channel Status details.
   CHANNEL(QM1)
XMITQ(SYSTEM.CLUSTER.TRANSMIT.QUEUE)
   CONNAME(Win2ØØØ-pc(15ØØ))               CURRENT
   CHLTYPE(CLUSSDR)                        STATUS(RUNNING)
AMQ8417: Display Channel Status details.
   CHANNEL(QM2)                            XMITQ( )
   CONNAME(169.123.123.123)                CURRENT
   CHLTYPE(CLUSRCVR)                       STATUS(RUNNING)
```

Notice that the *CONNAME*, which was left blank at definition time, is now prefilled with the local IP address.

(You can confirm this via the **IPCONFIG** command in a DOS window.) Notice also that the port has been left off, which then defaults to 1414.

The same sorts of detail are displayed when using the **DIS CLUSQMGR** command:

```
dis clusqmgr(*) status conname
     2 : dis clusqmgr(*) status conname
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(QM1)                           CLUSTER(CLUS1)
   CHANNEL(QM1)                            CONNAME(Win2000-pc(1500))
   STATUS(RUNNING)
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(QM2)                           CLUSTER(CLUS1)
   CHANNEL(QM2)                            CONNAME(169.123.123.123)
```

However, if we display the channel definition itself, the *CONNAME* is still blank:

```
dis chl(QM2)
     3 : dis chl(QM2)
AMQ8414: Display Channel details.
   CHANNEL(QM2)                            CHLTYPE(CLUSRCVR)
   TRPTYPE(TCP)                            DESCR( )
   MCANAME( )                              MODENAME( )
   TPNAME( )                               BATCHSZ(50)
   DISCINT(6000)                           SHORTRTY(10)
   SHORTTMR(60)                            LONGRTY(999999999)
   LONGTMR(1200)                           SCYEXIT( )
   SEQWRAP(999999999)                      MAXMSGL(4194304)
   PUTAUT(DEF)                             CONVERT(NO)
   SCYDATA( )                              MCATYPE(THREAD)
   CONNAME( )                              MREXIT( )
   MRDATA( )                               MRRTY(10)
   MRTMR(1000)                             HBINT(300)
   BATCHINT(0)                             NPMSPEED(FAST)
   MCAUSER( )                              CLUSTER(CLUS1)
   CLUSNL( )                               NETPRTY(0)
   ALTDATE(2001-02-07)                     ALTTIME(12.28.37)
   MSGEXIT( )
   SENDEXIT( )
   RCVEXIT( )
   MSGDATA( )
   SENDDATA( )
   RCVDATA( )
```

Just to emphasize that only port 1414 works on dynamically generated connection addresses, define a third queue manager, *QM3*, which listens to port 1415.

Define the cluster receiver and cluster sender as before.

Now display the result via **RUNMQSC** on *QM3*:

```
dis clusqmgr(*) conname status
     1 : dis clusqmgr(*) conname status
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(QM3)                           CLUSTER(CLUS1)
   CHANNEL(QM3)                            CONNAME(169.123.123.123)
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(SYSTEM.TEMPQMGR.Win2000-pc(1500))
   CLUSTER(CLUS1)                          CHANNEL(QM1)
   CONNAME(Win2000-pc(1500))               STATUS(RUNNING)
```

Notice that the full repository *QM1* name has not been resolved, and is still the 'temporary' name *SYSTEM.TEMPQMGR.Win2000-pc(1500)*.

If we use the same command from *QM1* we get:

```
dis clusqmgr(*) conname status
     1 : dis clusqmgr(*) conname status
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(QM1)                           CLUSTER(CLUS1)
   CHANNEL(QM1)                            CONNAME(Win2000-pc(1500))
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(QM2)                           CLUSTER(CLUS1)
   CHANNEL(QM2)                            CONNAME(169.123.123.123)
   STATUS(INACTIVE)
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(QM3)                           CLUSTER(CLUS1)
   CHANNEL(QM3)                            CONNAME(169.123.123.123)
   STATUS(RETRYING)
```

This time, the automatically-defined cluster-sender channel *QM3* is in 'retrying' state. This is not surprising, as the *CONNAME* does not have a port number.

It therefore defaults to 1414, but *QM3* is listening on 1415.

DYNAMIC ALLOCATION OF CLUSTER CACHE

Clustered MQSeries objects, such as queues and channels, are held in a cache. In V5.1 the size of the cache was fixed when the queue manager was started, which could cause problems when a large number of clustered objects was defined. The only way to resolve this was to define a smaller number of clustered objects, restart the queue manager, and repeat this process until all objects were defined.

MQSeries V5.2 allows the cache to grow 'on demand'. If you have an existing exit in V5.1 that relies on a fixed layout of the cache and you

do not want to change the exit, then the cache size can be made 'static' via a tuning parameter. (See the *qm.ini* file with new parameter in the *TuningParameters* stanza: *ClusterCacheType=DYNAMIC|STATIC*.)

CONCLUSION

In a DHCP environment, MQSeries V5.2 further reduces the administrative task of setting up clusters. Users must be aware of the limitations regarding the TCP/IP port numbers for cluster-receiver channels, and must have an appropriate channel-naming convention for cluster-sender channels.

Those companies with a large – and growing – number of clustered objects will benefit from the dynamic cluster cache, which avoids the need to restart the queue manager.

Ruud van Zundert
Independent Consultant (UK)

How to improve message throughput in an MQSI V2 broker

INTRODUCTION

The performance you can obtain with a given message flow in MQSeries Integrator (MQSI) V2 is a function of the way in which the flow is written, the hardware on which the broker is located, and the performance of the broker queue manager. Given a particular hardware environment, it is still possible to make changes to improve performance. This article discusses a number of such changes.

The changes relating to queue manager log and data location and log parameter tuning will apply equally to any MQSeries queue manager and not just one used by an MQSI broker.

DETERMINE THE MESSAGE TYPE

There is a significant difference in the message rate that can be

achieved with non-persistent and persistent message rates. Non-persistent message processing is typically CPU-bound, whereas persistent message processing is I/O-bound (on the MQSeries queue manager log). Since the speed of I/O is much slower than a CPU, the message rate for persistent messages is similarly much lower than that for non-persistent messages.

The first requirement is to be absolutely sure that persistent messages are required before using them, since they are so expensive.

Here are some recommendations when using non-persistent or persistent messages.

Non-persistent messages

- Set 'transactionMode' on the MQInput nodes to either 'automatic' or 'no' – the default is 'yes'.
- Eliminate I/O where possible. Even with non-persistent messages it is possible to perform I/O within MQSeries. I/O arises when using a large number of records in a queue and/or large messages because the messages overflow the queue buffer (64K by default) for each queue. This buffer is held in memory. By keeping the size of the queue down, ie the number of messages on the queue at any point in time, it is more likely that messages will be held in memory rather than overflowing to disk.

It is more efficient to be continually processing messages rather than pre-loading the queue prior to the start of processing, so aim to continually top-up the queue. This mode of operation is also more realistic. It is unlikely that thousands of messages will be pre-loaded onto a queue.

Persistent messages

- Locate each MQSeries queue manager log on a dedicated disk.*
- Locate each MQSeries queue manager data on a dedicated disk.*
- When using an XA connection make sure the database log and data are each located on a dedicated disk.
- Look at the settings in effect for the queue manager log*: in

particular:

  – the number of LogBufferPages: this controls the maximum amount of data which can be written in one log I/O.

  – the number of LogFilePages: this controls the size of the log on disk and will affect the amount of log switching that takes place.

* See Appendix A, which details the procedure to follow in order to locate the queue manager data and log on different disks, as well as how to change log-related parameter settings.

- Use the fastest device possible for the queue manager log. Disks which have a fast-write, non-volatile cache can significantly improve message processing rates when compared with a SCSI disk. Measurements have shown that, when using an SSA Fast-Write disk with non-volatile cache for the queue manager log disk, it was possible to improve message throughput for persistent messages by between three and four times, when compared with using a SCSI disk.

- Do not use IDE disks. Although they may appear to offer better performance, the write I/O completion is signalled before the write is actually complete. This can compromise the integrity of the data because a write could subsequently fail and there is no non-volatile cache to back-up the data.

- The use of multiple brokers (and queue managers) will allow you to overcome the limitation of a single queue manager's maximum persistent message rate in order to achieve the required throughput rate.

You will see some benefits from locating the log and data on different disks – the greatest is likely to come from increasing the LogBufferPages and LogFilePages values. The benefits obtained will vary in each case, and will depend upon the size of messages. It is worth conducting tests with a simple program to repeatedly GET or PUT messages in order to see the benefits of making these changes.

IMPROVING NODE PERFORMANCE

- It is better to have fewer, more complex, compute nodes than a

larger number where the processing is spread out amongst the nodes. There are startup costs associated with each node and it is best to minimize these. Make sure that the nodes are manageable though, when combining ESQL, and that you do not have an excessive number of lines of ESQL to manage, for example.

- When using publication nodes where the number of subscribers receiving publications is greater than 30 for any published message, ensure that the size of the cache of open queue handles is set appropriately. The cache has a default of 30 entries. Ideally, the value should be at least the number of subscribers who will receive the publications. Be aware that increasing the size of the cache will have an effect on the amount of storage used by the execution group. If the number of entries in the cache is too small then one queue has to be closed and a new one opened before the message can be written to the required queue. This is repeated for each subscriber over the cache size. This becomes costly if there are many more subscribers than entries in the cache.

  In order to increase the size of the cache use the following command:

  ```
  mqsichangeproperties <broker>  -e <execution group >
       -o ComIbmMQConnectionManager  -n queueCacheMaxSize -v <size>
  ```

  where <*broker*> is the name of the message broker being used, <*execution group* > is the name of an execution group containing a publication node, and <*size*> is the size to which the cache is to be increased. The command is issued to a named execution group. Be aware that you may need to issue it to more than one execution group, depending upon the message flows you have.

- It is possible to modify the level of persistence on the published message for each subscriber. Although the input message from the publisher may be persistent, not all the published messages have to be. This provides the opportunity to reduce the cost of writing the published message to each queue, where appropriate.

  Deciding when this is appropriate will depend upon the data being published. Consider the following two cases: one where the current score in a football game is being published, and the

second where stock prices are being published. In both cases, publications take place twice a second.

In the case of the football score it is not essential to receive every message. There is unlikely to be any impact from not receiving a particular message, especially since there will be another publication in half a second. It is appropriate to make these messages non-persistent. With the stock price notification there could well be an impact, since the failure to act in a timely manner could result in financial loss as you failed to sell or buy at the correct point in time. In this case it is important that all messages are delivered and so it is appropriate to opt for persistent messages.

- Do not have the *CARDINALITY* function coded in an ESQL *WHILE* loop. Coding it in that manner means that the function will be invoked for every iteration of the loop. With a large message this could add a significant amount of processing, as *CARDINALITY* determines the size of the message.

  The approach you should use is to set a variable to the value of the message size prior to entering the loop. For example:

```
DECLARE I INTEGER;
DECLARE LIMIT INTEGER;
SET LIMIT = CARDINALITY(InputRoot.XML.A.B[]);
WHILE I < LIMIT DO
. . . . . .
. . . . . .
END WHILE
```

- Do not use trace nodes when trying to monitor performance. They are costly.

ESTIMATING THE BENEFITS OF SCALING

Although it is normally possible to achieve higher message throughput by using multiple execution groups (as opposed to using multiple message flows in an execution group or additional instances of a message flow), it is not possible to predict precisely what benefit will be obtained by running multiple execution groups. The results will vary between no additional benefit and significant benefit, with very good scaling – almost *N* times the throughput of a single execution group when running *N* execution groups.

No additional benefit arises when there is data conflict, all copies of the message are after the same rows in a database, or there is contention for queue access, for example.

Maximum benefit comes when there is additional processing in a message flow, such as compute nodes with ESQL, and there is no contention. In this case each message is processed independently, there is very little contention for common resources (queues or database data), and it is possible to add more parallel pieces of work (execution groups). As the processing of individual messages is independent, the scaling ratio is much higher.

With persistent messages you need to bear in mind that there is an upper limit, which is the maximum rate at which the MQSeries queue manager log is able to operate, and, as such, it will not be possible to continuously scale.

MAXIMIZING THROUGHPUT

Your aim should be to get the maximum possible CPU utilization possible, that is 100%. You do not have to run your servers at this level, but it is good to know that they are capable of it should you need to, rather than having them limited at 50% because of data contention, for example.

Depending upon conditions, full CPU utilization may or may not be possible. If it is not happening you need to find the limitations. Reasons for not achieving it would include:

- Data contention – multiple execution groups trying to update the same row in a database at the same time, for example.
  - The solution is to remove the data contention or, failing that, find some work-around.
- Contention on queue access – this is most likely to be a problem where very simple message-copying is taking place from one queue to another. A message consisting purely of an MQInput and MQOutput node would be an example of a flow that could encounter this problem.
  - Contention for queue access is best overcome by running

multiple pairs of input and output queues. This would mean multiple message flows.

- I/O – disks will go so fast and then no faster.
  - It is possible to reduce I/O times by using solid state disks or disks with caches – the fastest I/O is no I/O! I/O occurs with MQSI because of:
    - Queue data being written to or read from disk.
    - Queue manager logging to provide recovery.
    - Data being written to/read from a database causes the database to perform I/O.

If you are using persistent messages you need to follow the points detailed in the section *Persistent Messages*, above. It will not be possible to eliminate I/O; all you can do is minimize its effect.

Where the I/O is taking place in a database, tune the database wherever possible. For example, try to ensure the database is local, increase buffer sizes, or use faster disks for data and logs.

Whatever type of messages are in use you will derive benefit (increased message rate) by ensuring the number of messages on the queue is kept low. The arrival rate of messages needs to be the same as, or only slightly higher than, the rate at which they are being processed. If the number of messages increases more rapidly than the rate at which they are being processed, an increase in I/O occurs and slows the message rate. In a production environment the arrival rate is not always controllable, so you need to ensure that there is sufficient processing power available to cope with the highest arrival rate. In a test environment there is more control. Continually feed messages onto a queue rather than pre-load all the messages which are to be processed.

You will also derive benefit by immediately reading the messages off the queue as soon as they have been processed rather than allowing the queue depth to increase.

Benefits can be obtained even when there are only a few thousand records on the input queue.

- Waiting for a response from some other component – maybe an application on another queue manager.
  - When your configuration involves receiving messages from,

or passing them on to, another queue manager or piece of software, you need to ensure that those other components have sufficient resources and are well tuned.

GENERAL POINTS

- Use the fastest processors available, and multiples of them, so that you can benefit from the use of multiple instances and/or execution groups.
- Ensure there is plenty of memory available so that the chances of paging are minimized: a minimum of 0.5 GB – preferably 1GB or more for large configurations or where all components are on one machine.
- Ensure that any paging or swap space is located on a dedicated disk.
- Use as many execution groups as processors initially, and then adjust the number to get the maximum message rate. Each case is likely to vary.
- Use the minimum number of nodes in a message flow that can perform the task.
- Use a trusted broker where possible.
- Be careful when using ‘Logical Order’ on the MQInput node. Using a value of ‘By UserID’ can restrict message throughput. Messages for a given user will be restricted to a single thread of processing.
- Plan the level of data recovery required:
  - Only use an XA connection if needed, ie only if updating messages in MQSeries and data in a database.
  - Do you really want to make non-persistent messages transactional?
- Do not make any deployments when evaluating message throughput capabilities as they will cause the execution groups to be stopped and started.

- Ensure that you have the latest product level, CSD, and any e-fixes installed.
- Use MQSeries 5.2. It has improved logging performance which will be of particular help with persistent messages.

APPENDIX A: MQSERIES TUNING

This appendix details the procedure to follow in order to locate the MQSeries queue manager log and data on different disks. It also shows how to change the LogBufferPages and LogFilePages values for a queue manager.

The procedure is slightly different for the NT and the Unix platforms, although the principle is the same. Both methods are shown.

Some of the details in this section show how to modify the Windows NT registry. Such tasks should only be undertaken by experienced users. It is recommended that you take a copy of the registry prior to making any changes.

Locating the MQSeries data on a dedicated disk

This change must be made *before* the queue manager is created.

NT

An initial value for the data location is specified as part of the MQSeries installation process on NT. To change it subsequently, you will need to follow the procedures below.

1 On the disk that is to be used for the MQSeries data, create a directory to hold the data.

2 Run **regedit**.

3 Locate the key: *HKEY_LOCAL_MACHINE\SOFTWARE\IBM \MQSeries\CurrentVersion.*

4 In that key, change the setting of ‘WorkPath’ to point to the directory created in 1 above.

5 If no other changes are required, create the queue manager.

Unix

1 Allocate a file system that is to be used for the MQSeries data and create a directory that is to be used for the data.

2 Edit the file */var/mqm/mqs.ini*.

3 Change the value of 'DefaultPrefix' in the 'AllQueueManagers' stanza to the name of the directory that is to be used for the data.

4 Save the changes.

5 If no other changes are required, create the queue manager.

All queue managers that are subsequently allocated will use the same value of 'WorkPath' (or 'DefaultPrefix'), and, consequently, the data will be located on the same disk. Remember to update the value each time when allocating multiple queue managers.

Locating the MQSeries log on a dedicated disk

The easiest method of specifying the queue manger log location is to use the '-ld' flag when you issue the **crtmqm** command to create the queue manager.

NT

1 On the disk that is to be used for the MQSeries log, create a directory to hold the log.

2 Use the name of the directory created above for the '-ld' flag on the **crtmqm** command.

3 If no other changes are required, create the queue manager using the **crtmqm** command.

Unix

1 Allocate a file system that is to be used for the MQSeries log and create a directory that is to be used for the log.

2 Use the name of the directory created above for the '-ld' flag on the **crtmqm** command.

3 If no other changes are required, create the queue manager using the **crtmqm** command.

An alternative approach is to follow the procedure below. This change must be made *before* the queue manager is created.

NT

1 On the disk that is to be used for the MQSeries log, create a directory to hold the log.

2 Run **regedit**.

3 Locate the key *HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\Configuration\LogDefaults.*

4 In that key, change the setting of 'LogDefaultPath' to point to the directory created in 1 above.

5 If no other are changes required, create the queue manager.

Unix

1 Allocate a file system that is to be used for the MQSeries log and create a directory that is to used for the log.

2 Edit the file */var/mqm/mqs.ini*.

3 Change the value of 'LogDefaultPath' in the 'LogDefaults' stanza to the name of the directory that is to be used for the log.

4 Save the changes.

5 If no other changes are required, create the queue manager.

Changing the 'LogFilePages' value

It is possible to specify the log size when creating the queue manager by using the '-lf' flag on the **crtmqm** command. For NT, the minimum is 32 and the maximum is 4095. For Unix, the minimum is 64 and the maximum is 16384.

An alternative approach is as follows. This change must be made *before* the queue manager is created.

NT

1 Run **regedit**.

2 Locate the key *HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\Configuration\LogDefaults.*

3 In that key, change the setting of LogFilePages. The larger the value, the larger the size of the log extent on the disk. The minimum value is 32 and the maximum is 4095.

4 If no other changes are required, create the queue manager.

Unix

1 Edit the file */var/mqm/mqs.ini*.

2 Change the value of 'LogFilePages' in the 'LogDefaults' stanza. The larger the value, the larger the size of the log extent on the disk. The minimum value is 64 and the maximum is 16384. Ensure that there is sufficient space in the file system for the log to accommodate the value you are specifying.

3 Save the changes.

4 If no other changes are required, create the queue manager.

Changing the 'LogBufferPages' value

The value of 'LogBufferPages' can be set either before or after the queue manager has been created.

To set it before queue manager creation, follow the procedures detailed below.

NT

1 Run **regedit**.

2 Locate the key *HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\ CurrentVersion\Configuration\LogDefaults.*

3 In that key, change the setting of 'LogBufferPages' – try 512 if using MQSeries 5.2 (32 for MQSeries 5.1). The larger the value, the larger the size of the buffer that is written with each log I/O. Large values are required for large messages and/or busy systems.

4 If no other changes are required, create the queue manager.

Unix

1 Edit the file */var/mqm/mqs.ini*.

2 Change the value of 'LogBufferPages' in the 'LogDefaults' stanza. Try 512 if using MQSeries 5.2 (32 for MQSeries 5.1). The larger the value, the larger the size of the buffer that is written with each log I/O. Large values are required for large messages and/or busy systems.

3 Save the changes.

4 If no other changes are required, create the queue manager.

To change the value of 'LogBufferPages' after queue manager creation, follow the procedure below:

NT

1 Run **regedit**.

2 Locate the key *HKEY_LOCAL_MACHINE\SOFTWARE\IBM\MQSeries\CurrentVersion\Configuration\QueueManager\<qmgr name>\Log*.

3 In that key, change the setting of 'LogBufferPages' – try 512 if using MQSeries 5.2 (32 for MQSeries 5.1). The larger the value, the larger the size of the buffer that is written with each log I/O. Large values are required for large messages and/or busy systems.

4 If no other changes are required, restart the queue manager.

Unix

1 Edit the file */var/mqm/qmgrs/<qmgr name>/qm.ini*.

2 Change the value of 'LogbufferPages' in the Log stanza. Try 512 if using MQSeries 5.2 (32 for MQSeries 5.1). The larger the value, the larger the size of the buffer that is written with each log I/O. Large values are required for large messages and/or busy systems.

3 Save the changes.

4 If no other changes are required, restart the queue manager.

Tim Dunn
Software Engineer, IBM UK Laboratories (UK)

Copying queue contents from MQSeries V1.2 to V2.1 on OS/390

INTRODUCTION

Copying queue contents from V1.2 to V2.1 queue managers on OS/390 has an undocumented pitfall. This article describes the problem and the solution.

WHERE AND HOW THE PROBLEM ARISES

When migrating from MQSeries V1.2 to V2.1 on the OS/390 platform there is often a need to copy the queue contents from the old queue manager pagesets to the new queue manager's pagesets. For example, before an upgrade of an MQSeries production system is carried out, a test system with the new version will be installed on a separate LPAR and some data will be required in the queues to perform the test operations. This data needs to be copied from the production queue manager, which is installed on the production LPAR, and its operation need not be disturbed.

Using IBM's CSQUTIL utility is the obvious solution for this task. With it, you can COPY a queue's contents to a flat VBS dataset, transfer the file to the target system (or vary online the volume on which it resides), and LOAD it to the target queue manager. However, there is an undocumented pitfall in CSQUTIL's COPY and LOAD behaviour.

As a result of the addition of new features, such as clustering, to the queue and message properties, the flat VBS dataset format, which CSQUTIL COPY outputs, has changed from V1.2 to V2.1. Version 2.1. CSQUTIL LOAD is not backwards-compatible with V1.2 COPY format. This causes V2.1 CSQUTIL LOAD operations to fail when used with datasets that were created on a V1.2 queue manager. This incompatibility is not documented in IBM's manual – even the *Migrating from Version 1.2 to Version 2.1* section. Also, the message which CSQUTIL outputs when the LOAD operation fails is obscure, and does not specify the nature of the problem.

However, V2.1's CSQUTIL COPY function is backwards-compatible with V1.2's pagesets. So the solution is to use the new version of CSQUTIL when COPYing the older version's queues. This may sound trivial to experienced MVS people, but the rest of us may encounter difficulties, especially if the two versions are installed on different systems or LPARs.

The following method was used extensively in my shop to copy from a V1.2 production system to a V2.1 system between different LPARs. IBM support personnel have acknowledged that, although it's undocumented, this is the only way of transferring queue contents between the two versions.

First, you need access to the new version's binary datasets on the source (old version) system. This can be done either by copying the datasets to disks used by the source system, or by varying online the disks on which they currently reside (ask your system people for help). The three required datasets are:

- *thlqual.SCSQANLE*.
- *thlqual.SCSQLINK*.
- *thlqual.SCSQAUTH*.

(*thlqual* is the high level qualifier of the V2.1 binary datasets.)

Note that no changes to the runtime environment (LPA, linklist) or to the source queue manager are needed, so your production system can continue to operate normally while you do this.

The job to COPY the contents of the queue (or a whole pageset) from the source system is as follows:

```
//COPY   EXEC PGM=CSQUTIL,PARM='CSQ1'
//STEPLIB  DD DISP=SHR,UNIT=uuuuu,VOL=SER=vvvvvv,
//   DSN=thlqual.SCSQANLE
//         DD DISP=SHR,UNIT=uuuuu,VOL=SER=vvvvvv,
//   DSN=thlqual.SCSQLINK
//         DD DISP=SHR,UNIT=uuuuu,VOL=SER=vvvvvv,
//   DSN=thlqual.SCSQAUTH
//OUTPUTA  DD DSN=SAMPLE.UTILITY.COPYA,DISP=(NEW,CATLG),
//            SPACE=(CYL,(5,1),RLSE),UNIT=SYSDA,
//            DCB=(RECFM=VBS,BLKSIZE=23200)
//CSQUOUT  DD DSN=SAMPLE.UTILITY.COPY3,DISP=(NEW,CATLG),
```

```
//            SPACE=(CYL,(5,1),RLSE),UNIT=SYSDA,
//            DCB=(RECFM=VBS,BLKSIZE=23200)
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
* COPY WHOLE PAGESET TO 'CSQUOUT'
COPY PSID(03)
* COPY ONE QUEUE TO 'OUTPUT'
COPY QUEUE(ABC123A) DDNAME(OUTPUTA)
/*
```

This is the same job as in IBM's *System Management Guide*, except STEPLIB has three DD statements concatenated to it with the V2.1 binary datasets. Essentially, what it does is execute the V2.1 CSQUTIL with the V1.2 queue manager pagesets.

USAGE NOTES

- Replace *CSQ1* with the source queue manager subsystem name, *uuuuu* and *vvvvvv* with the unit and volser of the disk on which the V2.1 binary datasets reside, respectively, and *thlqual* with the high level qualifier of the V2.1 binary datasets.
- Note that the queue must not be opened by any application during the execution of CSQUTIL COPY or the operation will fail. This means you need to terminate all applications that are reading from or writing to the source queue before running the job. You can check this by issuing the following command to the source queue manager:

  ```
  DISPLAY QL(queuename) IPPROCS OPPROCS
  ```

- If the source queue manager is down, you can replace COPY with SCOPY. See the *System Management Guide* for details.
- For detailed CSQUTIL reference: *http://www-4.ibm.com/software/ts/mqseries/library/manuals99/csqrap/csqrap3e.htm - HDRQUMGT*)

The next step is simply to make the output dataset available to the destination queue manager (again, either by copying it, by varying the disk online, or any other method), and run a standard CSQUTIL LOAD job:

```
 //LOAD   EXEC PGM=CSQUTIL,PARM=('CSQ1')
//STEPLIB  DD   DISP=SHR,DSN=thlqual.SCSQANLE
```

```
//         DD   DISP=SHR,DSN=thlqual.SCSQAUTH
//OUTPUTA  DD DSN=MY.UTILITY.OUTPUTA,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
LOAD QUEUE(ABC123) DDNAME(OUTPUTA)
/*
```

That's it. Happy upgrading!

Roy Razon
MQSeries Consultant (Israel)

Monitoring a Unix queue manager's error logs

INTRODUCTION

In all Unix MQSeries installations, the queue manager's error logs are one of the main sources of information about what exactly has been occurring within the system. In addition to listing all of the regular (and perfectly normal) activities, such as channel starts and stops, it will also contain entries for any error conditions that have been detected. This makes error logs an essential tool in the diagnosis of any problems that might have occurred.

Each queue manager will have a set of three error logs which are usually written to the directory */var/mqm/qmgrs/<queue manager name>/errors*. Each log has a capacity of 256K, with *AMQERR01.LOG* always being the active one that is actively being written to by the queue manager. When *AMQERR01.LOG* is about to exceed 256K, it is copied to *AMQERR02.LOG*. However, before this operation can complete, *AMQERR02.LOG* is in turn copied to *AMQERR03.LOG*. The previous contents of *AMQERR03* are discarded, and in this way, the logs cycle.

Unfortunately, on very busy queue managers the error logs may cycle too fast, which can be a problem. This is especially true where there are a lot of channels with relatively short disconnect intervals. In these cases, entries that you need to see in order to resolve a problem that has occurred can be lost because the error logs are swamped by lots of channel starts and stops. If you are told by operational support staff

or a particular application's support team that a certain MQ system had problems overnight, key messages for the time period in question may well have gone from the logs by the time you check them. What is needed is a copy of any log that may contain error messages indicating a problem has occurred.

THE CHKMQLOGS UTILITY

This utility runs as a background task and monitors queue manager error logs. It does this by periodically checking to see if a particular error log has a new system date/time stamp. In the sample code below, this is *AMQERR02.LOG*, but this is a configurable setting within the startup shell script. If it finds that the file hasn't changed it will wait for a specified time interval before checking once again. However, if it finds that there has been a change – in this case if *AMQERRO1.LOG* has scrolled into *AMQERR02.LOG* – it launches a new script (*gochklog*) that analyses that log (ie *AMQERR02*), passing to it the parameters it needs.

Some points to note about *chkmqlogs* and its code

- The utility takes no parameters, but contains a block of code where all the variables needed must be set. These are:
  - The directory containing the queue manager's error logs.
  - The particular log file you want to monitor.
  - The parent directory for copies of error logs and the utility's console.
  - The queue manager name.
  - The sleep interval, ie the length of time it waits before checking the date/time stamp again.
- The date/time stamp is captured by routing the output of the Unix **ls -l** command to a shell script variable.

THE CHKMQLOGS SHELL SCRIPT

```
#!/bin/ksh
# chkmqlogs
```

```
# Script that checks if a new version of a Queue Manager's
# Error Log has been created - and if found, calls gochklog
# to scan it for specific AMQnnnn errors
#      Display Help Text for Utility?                    #
if [[ $1 == '?' ]]
then
     print "chkmqlogs:"
     print " "
     print "Utility to scan a Queue Manager's Error Logs"
     print "for error codes specified within the config"
     print "file chkmqlogs.cfg"
     print " "
     print "If any are found - the Error Log is copied..."
     print " "
     print "Variables needed are:"
     print "  Log File Directory"
     print "  Error Log Name"
     print "  Directory to copy Logs into"
     print "  Queue Manager Name"
     print "  Sleep Interval"
     print " "
fi
exit
#      Set all variable parameters here ....              #
#=========================================================================#
logdir=/var/mqm/qmgrs/MYQMGR/errors     # Directory containing QMs Logs
logname=AMQERRØ2.LOG                    # Log to analyse
copypdir=/var/mqm/copylogs              # Parent Directory for Copy Logs
qm=MYQMGR                               # Queue Manager Name
sleepfor=6ØØ                            # Check Interval (Secs)
#=========================================================================#
mqlog=$logdir/$logname
#      Capture timestamp of file and see if it has changed #
#      - if it has, call analysis script                  #
olddate=""
while :
do
     newdate=$(ls -l $mqlog)
        if [[ $olddate != $newdate ]]
     then
          print "New log found ....."
          print "Calling gochklog .."
          gochklog $logdir $logname $copypdir $qm
     fi
#      Wait for specified interval before checking again   #
     sleep $sleepfor
     olddate=$newdate
done
exit Ø
```

THE UTILITY'S CONFIG FILE

In analysing the error log, the utility needs to know what specific MQSeries error codes may indicate that a problem has occurred, and so trigger the copying of that file. This is done by means of a configuration file called *chkmqlogs.cfg*, which should reside in the same directory as the shell scripts.

An example of this file is shown below.

```
# Configuration File for chkmqlogs utility
# Enter all error message codes to be scanned for
# in the format AMQnnnn - separated by the pipe character
## NB - do NOT leave any blank lines below this header
AMQ9254|AMQ9558|AMQ67Ø9|AMQ8Ø03|AMQ8Ø04
```

Here, amongst other things, we are interested in whether the queue manager has been stopped or started at any time during the lifecycle of the log.

NOTES ON GOCHKLOG

- The utility puts out a series of messages as it analyses the log. Under normal circumstances these will be written to a file called *chkmqlogs.console* but if the variable 'debug' is set to 'y' they will be written to the screen. This will be located in a directory with the same name as the queue manager, under the directory specified in the variable 'copypdir' (a sample of its contents will be shown later in this article).
- The start and end date/times of the log are determined by issuing the Unix head and tail commands to find the appropriate lines.
- All unique *AMQnnnn* message identifiers are retrieved and routed to a temporary file on the system's */tmp* directory.
- This file is then processed in order to find out how many occurrences of each message have been found in the log.
- Finally, the message codes that have been found are checked against those specified in the config file. If any matches are found the entire log is copied and made unique by appending a date/time stamp as part of its name.

THE GOCHKLOG SCRIPT

```
#!/bin/ksh
# gochklog
# Script that actually does the checking of the specified
# Queue Manager Error log
# Parameters are:
# 1 Error Log Directory
# 2 Error Log Name
# 3 Parent Directory for Copy Logs and Console File
# 4 Queue Manager Name
debug=n
#      Are we running in Debug Mode ?                       #
if [[ $debug = 'y' ]]
then set -o xtrace
fi
print "gochklog running ..."
#      Set exact location and name of Queue Manager Log    #
logdir=$1
logname=$2
copypdir=$3
mqlog=$logdir/$logname
#      Set Queue Manager Name and Temporary File Name/Loc  #
qm=$4
tmp=/tmp/$LOGNAME.$$.tmp
#      Set Directory to copy Logs and Console File to ..   #
copyldir=$copypdir/$qm
ls $copyldir > /dev/null 2>&1
rc=$?
if [[ ${rc} != 0 ]]
then
     mkdir $copyldir
fi
console=chkmqlogs.console
consout="$copyldir/$console"
#      Get Error Codes from config file into variable      #
copymqlog=`cat chkmqlogs.cfg | grep -v '#'`
print "          Error Log Monitor for $qm" >> $consout
print " " >> $consout
print "Log Start at $(head -1 $mqlog)" >> $consout
print "Log End   at $(tail -100 $mqlog |grep '../../'| tail -1)" >>
$consout
grep ^AMQ $mqlog | cut -f1 -d':' | sort | uniq > $tmp
#      Examine Log to determine how many occurrences of    #
#      each error message there are ..                     #
print " " >> $consout
print " Number of                       MQSeries" >> $consout
print "Occurrences                       Message" >> $consout
print "===========                      ========" >> $consout
for err in $(cat $tmp)
```

```
do
   print "$(grep $err $mqlog| wc -l) - $(grep $err $mqlog |head -1)" >>
$consout
done
#      Compare each error to those specified in config     #
#      file - if any match, copy the log ..                #
egrep $copymqlog $mqlog > /dev/null 2>&1
rc=$?
if [[ ${rc} == 0 ]]
then
     newlog=$(date +%d%m%H%M)
        print " " >> $consout
        print "Specified error detected -" >> $consout
     print "Saving $mqlog to" >> $consout
     print "Saving Error Log ..."
        print " $copyldir/$logname.$newlog" >> $consout
     cp $mqlog $copyldir/$logname.$newlog
fi
print " " >> $consout
print "========  End Log File Analysis  ========" >> $consout
print >> $consout
rm $tmp
exit 0
```

After the utility has run, this is what the console file chkmqlogs.console shows:

```
Error Log Monitor for MYQMGR
Log Start at 24/02/01  02:58:51 AM
Log End   at 01/03/01  12:34:29 PM
No of Occurrences    MQSeries Message
    3 - AMQ8003:     MQSeries queue manager 'MYQMGR' started.
    1 - AMQ8004:     MQSeries queue manager ended.
    1 - AMQ8506:     Command server MQGET failed with reason code 2009.
    3 - AMQ9410:     Repository manager started
    3 - AMQ9411:     Repository manager stopped
    2 - AMQ9447:     Unable to backout repository changes.
    2 - AMQ9509:     Program cannot open queue manager object.
    5 - AMQ9510:     Messages cannot be retrieved from a queue.
    1 - AMQ9542:     Queue manager is ending.
   48 - AMQ9999:     Channel program ended abnormally.
Specified error detected -
Saving /var/mqm/util/AMQERR02.LOG to
 /var/mqm/copylogs/MYQMGR/AMQERR02.LOG.16030946
 ========  End Log File Analysis  ========
```

Chris Bell
Systems Consultant, British Airways (UK)

MQLST: a REXX utility to filter and list MQ objects and attributes for OS/390

This REXX creates a list of MQSeries objects according to given parameters and writes it to SYSPRINT. The list can be filtered to contain a specified string, and a certain attribute of the object can be printed along with its name.

POINTS TO NOTE

- Parameters are separated by commas, eg:
  - MQLST QMGR,ObjectType,FilterString,Attribute.
- QMGR: the queue manager's name.
- ObjectType: any legal MQ object type (QUEUE,CHANNEL, etc). Please refer to 'Usage notes', below.
- FilterString: only objects whose name contains the string will be returned.
- Attribute: optional, can be omitted. MQ object attribute (such as CURDEPTH for QUEUE objects), whose value is to be printed along with the object name.

Example 1: listing certain queues' depth

```
MQLST CSQ1,QUEUE,MYPROJ,CURDEPTH
```

will output:

```
searching for <MYPROJ> in list of all <QUEUE>
in MQSeries QManager <CSQ1>
also retrieving <CURDEPTH>
TEST.MYPROJ1                    0
TEST.FOO.MYPROJ.OUTPUT        100
search completed successfully
2 objects written to SYSPRINT
```

Example 2: listing all stopped channels

```
MQLST CSQ1,CHS,STOPPED
```

will output:

```
searching for <STOPPED> in list of all <CHS>
in MQSeries QManager <CSQ1>
also retrieving <>
CSQ1.TO.UNIX1
CSQ1.TO.PRODNT
TESTNT.TO.CSQ1
search completed successfully
3 objects written to SYSPRINT
```

Prerequisites

Support pack MA19 *REXX TSO interface for MQSeries* must be installed, and, specifically, the module *RXMQVC*, which it contains, must be in the *LNKLST* or in the *STEPLIB* of the current job.

Usage notes

The cause of a common problem is that the ObjectType parameter should be specified as it would appear in the MQSC DISPLAY command. For example, the command **MQLST CSQ1,QLOCAL,BLA** will output an empty list. This is because the output of the command **"DISPLAY QLOCAL(*)"** is as follows:

```
CSQM201I +CSQ1 CSQMDMSG  DIS QLOCAL DETAILS
QUEUE(MQ.PROJ1)
TYPE(QLOCAL)
CSQMDMSG END QLOCAL DETAILS
```

The word **QLOCAL** does not appear on the searched line and the program cannot draw the object name from it. The solution is simply to use the word **QUEUE** and not **QLOCAL** in your query.

Programming notes

This REXX contains a useful general procedure, **perform_mq_cmd()**, which performs an MQSC command through the TSO REXX Interface and returns the results in a stem. This procedure can be used wherever an MQSC command needs to be executed – it simplifies the use of the support pack for REXX programs.

MQLST

```
/*  This REXX creates a list of MQSeries objects according to    */
/*   given parameters and writes it to screen.                   */
/*  The list created is a list of objects where a certain string */
```

```
/*   (given as parameter) exists in its name or the parameter    */
/*   requested (MQparm).                                         */
/* EXAMPLE OF USAGE :                                            */
/*  MQLST <Qmanager>,<MQobj>,<String>,<MQparm>                   */
/*  For example, the command :                                   */
/*   'MQLST CSQ1,QUEUE,PROJ1,CURDEPTH'                           */
/*  Will print a table of local queues and their depth with the  */
/*   string 'PROJ1' in their name (example 'MQ.PROJ1.INPUT').    */
/*  Another example of usage, is:                                */
/*   'MQLST CSQ1,CHS,RUNNING,STATUS'                             */
/*  Will print a list of all running channels...                 */
/* PARAMETERS  :                                                 */
/*  Qmanager    - MQSeries queue manager name (without prefix).  */
/*  MQobject    - MQSeries legal object name(QUEUE,CHANNEL etc). */
/*  String      - String to search in MQSeries object names or   */
/*                in parameter MQparm.                           */
/*  MQparm      - MQSeries object parameter (CURDEPTH, STATUS)  .*/
/* MODULES USED:                                                 */
/*  RXMQVC - Rexx TSO interface for MQSeries                     */
/*           supplied as Support Pack MA19.                      */
/* REXX'S USED :                                                 */
/*  NONE.                                                        */
/* IMPORTANT NOTE:                                               */
/*  The REXX gets the objects name using the string you typed as */
/*  your 'MQobject' parameter, this causes a problem im several  */
/*  cases.                                                       */
/*  for example, if you type 'MQLST CSQ1,QLOCAL,BLA', you will   */
/*  get a blank list, this is because the output of the command  */
/*  "DISPLAY QLOCAL(*)" is as follows:                           */
/*  CSQM201I +CSQ1 CSQMDMSG  DIS QLOCAL DETAILS                  */
/*  QUEUE(MQ.PROJ1)                                              */
/*  TYPE(QLOCAL)                                                 */
/*  CSQMDMSG END QLOCAL DETAILS                                  */
/*  the word "QLOCAL" does not appear in the searched line,      */
/*  and the program cannot draw the object name from it.         */
/*  the solution is simple enough, use the word "QUEUE" and not  */
/*  QLOCAL in your query.                                        */
/* RETURN VALUE:                                                 */
/*  0/8 - (RC) whether creating the list succeeded               */
/* Updates                                                       */
/* 23/10/2000 MQACID RRNO                                        */
/*            Creation                                           */
/* 02/11/2000 MQACID RRNO                                        */
/*            Final Version                                      */
/* 19/11/2000 MQACID RRNO                                        */
/*            Added mqprm parameter                              */
/* 12/04/2001 MQACID RRNO                                        */
/*            Further commenting added                           */
   /*                  Constants Definition                     */
   TRUE                  = 1;
   FALSE                 = 0;
```

```
   NULL                    = '';
   INDEX_STRING_NOT_FOUND = Ø;
   OUTPUT_TO_SCREEN        = 'SYSPRINT';
   /*                 Arguments Section                          */
   qmgr_name  = NULL;
   mqobj      = NULL;
   mqobj_str  = NULL;
   mqobj_prm  = NULL;
   PARSE ARG qmgr_name','mqobj','mqobj_str','mqobj_prm
/*
   qmgr_name = 'QM1';
   mqobj     = 'QUEUE';
   mqobj_str = 'NERL';
   mqobj_prm = 'CURDEPTH'
*/
/*
   say 'FOR DEBUG PURPOSES';
   say 'qmgr_name   <'qmgr_name'>';
   say 'mqobj_str   <'mqobj_str'>';
   say 'mqobj       <'mqobj'>';
*/
   /*                Variables Initialization                    */
   /*                      MAIN                                  */
   /* Strip parameters                                           */
   qmgr_name  = STRIP(qmgr_name);
   mqobj_str  = STRIP(mqobj_str);
   mqobj      = STRIP(mqobj);
   mqobj_prm  = STRIP(mqobj_prm);
   /* Check parameters validity                                  */
   if( qmgr_name  = NULL |,
       mqobj_str  = NULL |,
       mqobj      = NULL) then do;
   say 'MQLST Parameters Invalid.'
   say 'Rexx Usage:'
   say 'MQLST <qmgr>,<mqobj>,<string>,<mqobj_prm>';
   say 'qmgr       = MQSeries queue manager name (without prefix)'
   say 'mqobj      = CHANNEL/QLOCAL/PROCESS etc.';
   say 'string     = string to be searched in MQSeries objects';
   say 'mqobj_prm  = (OPTIONAL) MQS object parameter (exp. CURDEPTH)'
   say 'NOTE: USE UPPER CASE LETTERS'
      parms_valid = FALSE;
   end;
   else,
      parms_valid = TRUE;
   if (parms_valid = TRUE) then do;
      /* Notify user of search at hand                           */
      say 'searching for <'mqobj_str'> in list of all <'mqobj'>'
      say 'in MQSeries QManager <'qmgr_name'>'
      say 'also retrieving <'mqobj_prm'>';
      /* create object list                                      */
      objlist_received =,
```

```
         cre_objlist(qmgr_name,mqobj,mqobj_str,mqobj_prm);
      /* Write results to screen                                    */
      "EXECIO * DISKW SYSPRINT (STEM new_objlist. FINIS";
   end;
   else,
      objlist_received = FALSE;
   /* Check if object list received                                 */
   if (objlist_received = TRUE) then do;
      say 'search completed successfuly';
      say new_objlist.Ø' objects written to SYSPRINT';
      MQlst_RC = Ø;
   end;
   else do;
      say 'search was unsuccessful';
      say 'no objects objects written to SYSPRINT';
      MQlst_RC = 8;
   end;
return (MQlst_RC);
/*-------------------------------------------------------------------*/
/*                    get_all_objlist                                */
/* DESCRIPTION:                                                      */
/*  This procedure gets all wanted object names from queue           */
/*   manager by using a DISPLAY * MQSeries command.                  */
/*  It then creates the object list.                                 */
/* PARAMETERS:                                                       */
/*  qmgr_name - MQSeries queue manager name (without prefix).        */
/*  mqobj     - MQSeries object type (CHANNEL etc.)                  */
/*  mqobj_prm - MQSeries object's parameter (exp. CURDEPTH)          */
/* RETURN VALUE:                                                     */
/*  Ø/1 whether procedure succeeded.                                 */
/*-------------------------------------------------------------------*/
get_all_objlist: procedure,
                 expose TRUE,
                 expose FALSE,
                 expose INDEX_STRING_NOT_FOUND,
                 expose mqerr_msg.,
                 expose objlist.;
PARSE ARG qmgr_name,mqobj,mqobj_prm;
   /*                 Constants Definition                          */
   /*                Variables Initialization                       */
   proc_success   = FALSE;
   NO_OBJ_FOUND   = '';
   NO_PRM_FOUND   = '';
   objlist.Ø      = Ø;
  /*                        MAIN                                  */
   /* Perform mq display all mq objects command                     */
   /* Build basic command                                           */
   mq_dis_cmd = "DIS "mqobj"(*)";
   /* If mqobject parameter stated, add to command                  */
   if (mqobj_prm ^= NULL) then,
      mq_dis_cmd = mq_dis_cmd' 'mqobj_prm;
```

```
   /* Perform built command and check results                    */
   mq_cmd_success = perform_mq_cmd(qmgr_name,mq_dis_cmd);
   if (mq_cmd_success = TRUE) then do;
      /* Go through all lines in command output and extract      */
      /* queue names - from them create mq obj list              */
      do line_counter = 1 to mq_cmd_output.0;
        /* Parsing string to find object name                   */
         obj_name = NO_OBJ_FOUND;
         obj_prm  = NO_PRM_FOUND;
         /* Parse string to object name and object parameter     */
         PARSE VALUE mq_cmd_output.line_counter with,
          dummy VALUE(mqobj)'('obj_name')',
          dummy VALUE(mqobj_prm)'('obj_prm')',
          dummy;
         /* Check if string parmsed correctly                   */
         if(obj_name ^= NO_OBJ_FOUND) then do;
            next_objlist_line = objlist.0 + 1;
            objlist.next_objlist_line =,
               LEFT(STRIP(obj_name),40,' ')||' '||STRIP(obj_prm);
            objlist.0 = objlist.0 + 1;
         end;
      end;
      proc_success = TRUE;
   end;
   else do;
      /* Put error message received in stem to notify user       */
      mqerr_msg.0 = mq_cmd_output.0;
      do mqerr_line = 1 to mq_cmd_output.0;
         mqerr_msg.mqerr_line = mq_cmd_output.mqerr_line;
      end;
      proc_success = FALSE;
   end;
return (proc_success);
/*-----------------------------------------------------------------*/
/*                     cre_objlist                                 */
/* DESCRIPTION:                                                    */
/*  This procedure creates the wanted object list.                 */
/*  This is done by going through all objects in queue manager     */
/*   and finding the requested string (given as parameter).        */
/* PARAMETERS:                                                     */
/*  qmgr_name - MQSeries queue manager name (without prefix).      */
/*  mqobj     - MQSeries object (CHANNEL, QLOCAL etc.)             */
/*  mqobj_str - The string searched in all object names.           */
/*  mqobj_prm - MQSeries object's parameter (exp. USAGE)           */
/* RETURN VALUE:                                                   */
/*  0/1 procedure success.                                         */
/*-----------------------------------------------------------------*/
cre_objlist:     procedure,
                 expose TRUE,
                 expose FALSE,
                 expose INDEX_STRING_NOT_FOUND,
```

```
                expose new_objlist.;
PARSE ARG qmgr_name,mqobj,mqobj_str,mqobj_prm;
   /*                 Constants Definition                        */
   /*                Variables Initialization                     */
   proc_success = TRUE;
   /*                       MAIN                                  */
   /* Get queue list                                              */
   objlist_received = get_all_objlist(qmgr_name,mqobj,mqobj_prm);
   if (objlist_received = TRUE) then do;
      /* Go thorugh all lines in object list and find string.       */
      /* Create object list from those lines.                       */
      new_objlist.0 = 0;
      do line_counter = 0 to objlist.0;
/*
         say 'CREobjlist: search for <'mqobj_str'>';
         say 'CREobjlist: in <'objlist.line_counter'>';
*/
        cfgidx = INDEX(objlist.line_counter, mqobj_str);
         /* if string found, add it to new object list              */
         if (cfgidx ^= INDEX_STRING_NOT_FOUND) then do;
            next_cfg_line = new_objlist.0 + 1;
            new_objlist.next_cfg_line = objlist.line_counter;
            new_objlist.0 = new_objlist.0 + 1;
/*
            say 'CREobjlist: << string found';
*/
         end;
      end;
      proc_success = TRUE;
   end;
   else,
      proc_success = FALSE;
return (proc_success);
/*-------------------------------------------------------------------*/
/*                     perform_mq_cmd                                */
/*-------------------------------------------------------------------*/
/* This procedure performs and MQSeries command and returns results  */
/*  All this using the RXMQVC module supplied by MQSeries MA19 SP    */
/*                                                                   */
/* Parameters: qmgr_name - MQSeries queue manager name (NO PREFIX)   */
/*             mq_cmd    - MQSeries command                          */
/* Return code: command success (TRUE/FALSE)                        */
/* Exposed parameters:  NONE.                                        */
/*-------------------------------------------------------------------*/
perform_mq_cmd: procedure,
                expose TRUE,
                expose FALSE,
                expose RXMQWX_mq_cmd_rcc,
                expose mq_cmd_return_code,
                expose mq_cmd_reason_code,
                expose mq_conn_return_code,
```

```
               expose mq_conn_compl_code,
               expose mq_conn_reason_code,
               expose mq_cmd_output.;
PARSE ARG qmgr_name, mq_cmd;
   /*                 Constants Definition                      */
   RXMQVC_NO_TRACE     = '';
   /*                Variables Initialization                   */
   /*                      MAIN                                 */
   /* Init RXMQVC and associated variables                         */
   rcc = RXMQVC('INIT');
   RXMQVCTRACE = RXMQVC_NO_TRACE;
   mq_cmd_output.Ø = Ø;
   mq_cmd_successful = FALSE;
   /* Call RXMQVC to execute command and get output               */
   mq_cmd_rcc =,
    RXMQVC('command ',qmgr_name,mq_cmd,'mq_cmd_output.');
   mq_cmd_rcc = STRIP(mq_cmd_rcc);
   /* Init exposed variables                                      */
   RXMQVC_cmd_rcc      = mq_cmd_rcc;
   mq_cmd_return_code  = mq_cmd_output.CC;
   mq_cmd_reason_code  = mq_cmd_output.AC;
   mq_conn_return_code = word(mq_cmd_rcc,1);
   mq_conn_compl_code  = word(mq_cmd_rcc,2);
   mq_conn_reason_code = word(mq_cmd_rcc,3);
   /*                                                             */
   /* Check if connection to MQSeries successful and              */
   /* Check if MQSeries command successful                        */
   if (mq_conn_compl_code ^= 'Ø' |,
       mq_cmd_return_code ^= 'ØØØØØØØØ') then,
      mq_cmd_successful = FALSE;
   else,
      mq_cmd_successful = TRUE;
   /* If command not successful - print messages                  */
   if (mq_cmd_successful = FALSE) then do;
      say 'MQLST | ------------------------------------------';
      say 'MQLST | Error executing cmd <'mq_cmd'>';
      say 'MQLST | on queue manager    <'qmgr_name'>';
      say 'MQLST | MQCONN return code  <'mq_conn_return_code'>';
      say 'MQLST | MQCONN compl code   <'mq_conn_compl_code'>';
      say 'MQLST | MQCONN reason code  <'mq_conn_reason_code'>';
      say 'MQLST | return code         <'mq_cmd_return_code'>';
      say 'MQLST | reason code         <'mq_cmd_reason_code'>';
      say 'MQLST | MQ err msg number   <'mq_cmd_output.Ø'>';
      say 'MQLST | ------------------------------------------';
      do line_counter = 1 to mq_cmd_output.Ø;
         say 'MQLST | 'mq_cmd_output.line_counter;
      end;
      say 'MQLST | ------------------------------------------';
      say 'MQLST | RXMQVC rcc message  <'RXMQVC_cmd_rcc'>';
      say 'MQLST | ------------------------------------------';
   end;
```

```
   /* Terminate RXMQVC                                                  */
   rcc = RXMQVC('TERM');
return (mq_cmd_successful);
```

Roy Razon
MQSeries Consultant (Israel)

The CSQUTIL utility

CSQUTIL is a utility provided with MQSeries to help issue commands, perform backups and restores, and reorganize tasks.

MAKECLNT COMMAND

In this example the dataset referenced by DDname COPYWHAT, member (COPYMSMQ) contains an MQSeries **DISPLAY CHANNEL** command. The **MAKECLNT** keyword causes this to be converted into corresponding sets of client channel definitions. These are put into the dataset referenced by DDname **MAKECLNT**, which is ready to be downloaded to the client machine. The Queue Manager is *MQM1* and the job is run in batch on the OS/390 platform.

```
//USERID JOB (SØ9P91T),'DAVIES',CLASS=T,MSGCLASS=B,
//    NOTIFY=&USERID
*********************************************************************
//*  PROGRAM CSQUTIL ISSUES COMMANDS TO QMGR IDENTIFIED BY PARM='NNNN'
//*  ON THE EXEC CARD
//*  MESSAGES ARE PRINTED TO DD SYSPRINT
//*  USING MAKECLNT CREATS AN OUTPUT FILE THAT CAN BE USED TO
//*  BUILD A CLIENT CHANNEL DEFINITION FILE IN BINARY FORMAT
//MAKECLNT EXEC PGM=CSQUTIL,PARM='MQM1'
//STEPLIB  DD   DSN=SYS1.SCSQAUTH,DISP=SHR
//         DD   DSN=SYS1.SCSQANLE,DISP=SHR
//         DD   DSN=SYS1.MQM1.LINKLIB,DISP=SHR
//COPYWHAT DD DISP=SHR,DSN=QMGRMQM1.CHNL.JCL(COPYMSMQ)
//MAKECLNT DD DISP=SHR,DSN=QMGRMQM1.CHNL.MAKECLNT
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
COMMAND DDNAME(COPYWHAT) MAKECLNT(MAKECLNT)
COPYMSMQ sample definitions
DISPLAY CHANNEL(MSMQTOMQS) ALL
```

Saida Davies
IBM (UK)

Customizing CSQXMQxx

CSQXMQxx is an example of how the file *CSQ4XPRM*, which is supplied in the *MQM.SCSQPROC* dataset, can be customized to meet specific requirements. Here you can define the correct LU name and TCP/IP task name.

This job is used to create a channel initiator options module. It assembles and links a new channel initiator parameter module for distributed queueing without CICS. Edit the parameters for the *CSQ6CHIP* macro to determine your parameters and relink module *CSQXPARM*.

JOB CSQX4PRM

```
//JOBCARD
********************************************************************************
//* Customize Channel Initiator options module
********************************************************************************
//* The Macros in this deck are tracked by OS/390 SMPE usermod
********************************************************************************
//* CUSTOMIZE THIS JOB HERE FOR YOUR INSTALLATION
//* YOU MUST DO GLOBAL CHANGES ON THESE PARAMETERS USING YOUR EDITOR
********************************************************************************
//*               IBM MQSeries for MVS/ESA
//*    This job assembles and links a new channel initiator
//*    parameter module for distributed queueing without CICS.
//*    Edit the parameters for the CSQ6CHIP macro to determine
//*    your parameters.
//*    See MQSeries for MVS/ESA System Management Guide
//*    for a full description of the parameters.
//*    Replace   ++THLQUAL++
//*                        with the high level qualifier of the
//*                        SCSQMACS target library.
//*    Replace   ++HLQ.USERAUTH++
//*                        with the data set name of the authorized
//*                        load library in which to store your
//*                        channel initiator parameter module.
//*    Replace   ++NAME++
//*                        with the name of your channel initiator
//*                        parameter module.
//*                        Note - do NOT use the default version
//*                        name of CSQXPARM if you are using the
//*                        IBM library SCSQAUTH to store your
//*                        channel initiator parameter module.
//*****************************************************************
```

```
//*             Assemble step for CSQXPARM
//CHIP   EXEC PGM=ASMA90,PARM='DECK,NOOBJECT,LIST,XREF(SHORT)'
//SYSLIB   DD DSN=MQM.SCSQMACS,DISP=SHR,UNIT=3390,VOL=SER=SYS001 <=vol?
//         DD DSN=SYS1.MACLIB,DISP=SHR,UNIT=3390,VOL=SER=SYS001
<=vol?
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPUNCH DD DSN=&&CHIP,
//         UNIT=SYSDA,DISP=(,PASS),
//         SPACE=(400,(100,100,1))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
       CSQ6CHIP ADAPS=8,     ADAPTER SUBTASKS          0-9999        X
           ACTCHL=200,       MAX ACTIVE  CHANNELS      1-9999        X
           CURRCHL=200,      MAX CURRENT CHANNELS      1-9999        X
           DISPS=5,          DISPATCHERS               1-9999        X
           LUNAME=MVSMQLU1,  LU NAME FOR OUTBOUND DATA (LPARx)       X
           LU62CHL=200,      MAX LU6.2   CHANNELS      0-9999        X
           TCPCHL=200,       MAX TCP/IP  CHANNELS      0-9999        X
           TCPKEEP=NO,       TCP/IP KEEPALIVE OPTION   YES|NO        X
           TCPNAME=TCPIPx,   TCP/IP ADDRESS SPACE NAME (LPARx)       X
           TRAXSTR=YES,      START TRACE AUTOMATICALLY  YES|NO       X
           TRAXTBL=2         TRACE DATASPACE SIZE IN MB 0-2048
       END
//*  LINKEDIT CSQXPARM into a parameter module.
//*//LKED   EXEC PGM=IEWL,COND=(0,NE),
//      PARM='SIZE=(900K,124K),RENT,NCAL,LIST,AMODE=31,RMODE=ANY'
//*   OUTPUT AUTHORIZED APF LIBRARY FOR THE NEW CHANNEL INITIATOR
//*   PARAMETER MODULE.
//*YSLMOD  DD DSN=MQM.SCSQAUTH,DISP=SHR
//SYSLMOD  DD DSN=MQM.MQxx.PARMODS,DISP=SHR       <= MQxxCHIN steplib
//SYSUT1   DD UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=*
//CHIP     DD DSN=&&CHIP,DISP=(OLD,DELETE)
//*
//SYSLIN   DD *
   INCLUDE CHIP
 ENTRY CSQXPARM
 NAME  CSQXMQxx(R)     MQxx channel initiator parameter module name
/*
```

Saida Davies
IBM (UK)

CommerceQuest has announced version 4.0 of its patent-pending software, enableNet Data Integrator, a message-to-file-to message broker for MQSeries.

Introduced in 1997, enableNet Data Integrator provides assured delivery of mission-critical information. The software supports a comprehensive range of platforms including MVS, Unix, NT, Windows 2000, AS/400, IBM 4690, TPF, and others.

Enhancements in version 4.0 include a wider range of platform support, greatly improved customization capabilities, and easier installation and configuration for complex information integration projects.

An evaluation copy of the software is available for free trial at http:/ www.commercequest.com/eval

For further information contact:
CommerceQuest, 2202 N Westshore Blvd, Tampa, FL, 33607, USA
Tel: +1 813 639 6300
Fax: +1 813 639 6900
Web: http://www.CommerceQuest.com

CommerceQuest (UK), Doncastle House, Doncastle Road, Bracknell, Berkshire, RG12 8PE, UK
Tel: +44 (0)1344 861010
Fax: +44 (0) 1344 861011

* * *

Science Applications International Corporation's (SAIC) Broadway and Seymour Group has announced the release of version 7.0 of its TouchPoint customer relationship management (CRM) solution. Built on a browser-based architecture, TouchPoint 7.0 provides enhanced sales and service functionality across multiple delivery channels for financial services institutions.

Specifically, this new release has added support for MQSeries as a transaction processing middleware option. Other new features include significantly expanded report extraction and report generation capabilities, and additional sales and services features such as product presentation and user messaging.

Along with MQSeries and enhanced reporting, TouchPoint 7.0 features additional sales tools such as user messaging, rates, fees, and scripting that enable users to view on-line product footnotes and benefits, while the product presentation component assists users in providing detailed information to customers about the products and services available throughout their institution.

For further information contact:
SAIC, 10260 Campus Point Drive, San Diego, CA 92121
Tel: +1 858 826 6000
Web: http://www.saic.com

* * *