# 34

# MQ

update

*April 2002*

## In this issue

# MQ Update


## Published by

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

## North American office

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


## Contributions

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

## *MQ Update* on-line

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


## Commissioning Editor

Peter Toogood
E-mail: PeterT@xephon.net

## Managing Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com


## Disclaimer

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

## Subscriptions and back-issues

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

# MQSeries and 64-bit applications

Most of today's hardware and operating system vendors are able to sell you a '64-bit' system. What does this mean? How does it affect your applications? And how does it relate to the use of MQSeries?

The simplest definition of a 64-bit system is one where the memory that a process can address directly is expressed as a 64-bit value. A 32-bit program can address at most 4GB; if a program needs to work with more data than that it needs to use tricks such as swapping portions of the data to disk. With the 64-bit address range the theoretical limit is enormous – approximately 18 exabytes (18x1018). It is sometimes said that this is larger than the total amount of disk space ever manufactured!

Claims are sometimes made about the performance benefits for 64-bit programs. In practice these benefits are limited or non-existent. Just recompiling a 32-bit program is not guaranteed to make it run faster; indeed, the fact that the compiled binary is often larger might even slow it down.

The biggest improvements are achieved for programs that can make efficient use of the expanded memory. One class of application in this category is a database server where complete indexes and even tables can now be pre-loaded into memory instead of needing to be read from disk. Some mathematical algorithms also require large amounts of working space and these too can take advantage of 64-bit addressing.

Machines might not today have sufficient real memory (RAM) to match the addressing range but simplifying the application and allowing the operating system's virtual memory paging to handle it will often improve performance. Adding more real memory as bigger machines are built will then immediately further improve performance with no changes required to the application.

## WRITING PROGRAMS

C programs can normally be easily recompiled in 64-bit mode, providing you have taken care over the datatypes used as the sizes of some change. All current Unix systems have decided that the *long*

datatype is 64-bit with *int* remaining at 32-bit (this is often called the LP64 model). Microsoft has decided once again to be different and the 64-bit version of Windows keeps both *long* and *int* at 32-bits wide (also known as a P64 model). In both cases *pointers* are 64-bits. This means that conversion between pointers and an integer datatype (which has never been recommended but has been common practice) may no longer work; casting a *pointer* to an *int* will definitely not work. You might like to look at some of the lesser-known ANSI C datatypes (for example *ptrdiff_t* and *size_t*) for places where programs need to use portable definitions.

A good compiler will be able to point out places where your code makes assumptions about the sizes of objects. Some warnings can be ignored if you choose; others will need fixing. My own experience has been that existing code has often been very loose in making the distinction between *int* and *long* for 'small' values such as loop counters. This leads to a lot of warnings about truncating long values to 32-bit integers but when the variable only holds a small number the truncation doesn't matter. Of course the best solution is to have code with no warnings at all but this is sometimes too expensive to achieve.

Normal rules for writing portable code – don't assume ordering of bytes, use your own type definitions which can easily be changed in one place to match system definitions – apply strongly here, especially if you want 64-bit code to be portable between Unix and Windows systems.

When you are building programs all of the libraries which you link to must be the same 'size' – either 32-bit or 64-bit. You cannot mix the two types of code in the same process but there is nothing to stop a 32-bit program communicating with a 64-bit one using an inter-process communications mechanism such as a TCP/IP socket or a Unix message queue. When Windows moved from its 16-bit APIs to 32-bit Microsoft had a 'thunking' layer to allow the different types of code to work together. This has not been done for any of the 64-bit environments; any interoperability can only be done via separate processes.

However, it is important to realize that you do not necessarily need a 64-bit version of MQSeries or its libraries in order for your applications to use other 64-bit products such as Oracle. While you cannot mix 32-

bit libraries with 64-bit in the same process, products which have a 64-bit 'engine' may still include 32-bit application libraries that work with it.

## WHAT DOES MQSERIES PROVIDE?

### Versions and platforms

The first 64-bit platform supported by MQSeries was Digital Unix, with a client-only SupportPac made available in 1996. Now renamed to Compaq Tru64, this operating system has only ever supported 64-bit programs. There is no 32-bit toleration or compatibility in the operating system so when the full queue manager was ported to this system it too was built as a 64-bit program. This version of MQSeries is presently at the V5.1 level.

Recently released are two further MQSeries clients with 64-bit support – SupportPac MACR (for Solaris) and SupportPac MACS (for AIX 4.3) based on MQSeries V5.2. These SupportPacs require that you install at least the base part of the corresponding 32-bit client packages in order to pick up some supporting components such as the header files. This in turn must be at CSD02 level or later so that a single *cmqc.h* can be used for developing both 32-bit and 64-bit programs.

It is important to remember that the AIX SupportPac will not run on AIX V5 as there is no binary compatibility for 64-bit programs between AIX V4 and AIX V5. The MQSeries client packages can all be downloaded from *http://www.software.ibm.com/ts/mqseries/txppacs*.

These new SupportPacs include only TCP/IP connectivity to the queue manager across CLNTCONN/SVRCONN channels. They also have only MQI libraries for C and C++ programs; other languages are not directly supported. Apart from these caveats they have exactly the same function as the 32-bit client SupportPacs. For example, the MQCONNX verb is supported.

But in common with all MQSeries clients there is no support for transaction coordination with other resource managers. So you cannot use these SupportPacs for applications that need to update both a database and a queue as part of the same unit of work. If you need that

function you will have to use other techniques to keep the MQSeries interactions within a 32-bit process.

### Coexistence

Most operating systems that offer 64-bit environments also support 32-bit programs transparently. Such operating systems include AIX, Solaris, and HP-UX. There are plenty of existing applications for these operating systems and it is essential that they continue to run without change when a new version of the OS or hardware is put in. As most applications do not benefit from the 64-bit address space, running MQSeries and its applications in 32-bit mode on these operating systems is normally sufficient. MQSeries is supported fully on these 64-bit platforms even though it is a 32-bit program.

My description of this way of running MQSeries is ‘coexistence’: while not itself 64-bit, MQSeries will run alongside 64-bit programs on 64-bit hardware and operating systems.

### Toleration

Being able to run as a 64-bit program is what I call ‘toleration’. This is the first stage of making it possible to write 64-bit MQSeries applications.

The client SupportPacs and the Tru64 queue manager fit in this category of support. The application libraries – and in the case of Tru64 the whole queue manager – have been compiled as 64-bit programs without any major redesigns to take account of the fact that they are now 64-bit.

An extended form of toleration is where one queue manager is able to accept connections from both 32-bit and 64-bit local applications without a client channel. This will eventually be needed as there will be some applications that would benefit from being built in 64-bit mode but also need functions such as global transaction coordination. However, doing this will need new design effort in some internal components of the queue manager.

Having a queue manager that works only in 64-bit mode (like the Tru64 product) is not believed to be desirable on most other operating systems as there will already exist 32-bit applications that must continue to be supported on the same machine.

**Exploitation**

Different people mean different things by 64-bit exploitation. In my terminology this is where the programs – either applications or middleware such as a database engine or the MQSeries queue manager itself – are able to make good use of the expanded address range.

When starting from a 32-bit implementation this will often mean redesigning components such as memory management functions within the programs. It may mean a large amount of work but can in turn lead to a large payback for both raw performance and scalability. This work has not yet been done for the queue managers but is being considered for the future.

Remember that most end-user applications do not directly manipulate large amounts of data and will not, therefore, gain anything from being compiled in 64-bit mode. For these applications the 'coexistence' level of running is perfectly satisfactory.

## APPLICATION PROGRAMMING WITH MQSERIES

The MQSeries datatypes have always been abstract with no explicit size associated with them. Only the MQBYTE datatype can be assumed to represent a known quantity. This makes it easy to map the MQSeries definitions to appropriate values for 64-bit systems and to keep application code clean.

If you study the header files you will see that MQLONG is always set to be a 32-bit quantity even on 64-bit systems. Apart from a couple of definitions showing the length of structures (and these definitions are rarely used by anyone), the 64-bit path through the header file is identical to the 32-bit. Switching between the paths is done automatically when you compile a program as each compiler sets pre-processor flags to show whether it is generating a 32- or 64-bit program.

It is important for MQSeries to keep its core datatypes the same size regardless of whether the application program is 32-bit or 64-bit, as structures may need to be interpreted by applications on other machines that have different sizes. The structures are also passed across client channels and keeping the datatypes the same size means that the channel protocols do not need to waste time manipulating the bytes.

Any pointers that are contained in MQI parameters are always kept local to the machine on which the application is running so the difference in size does not matter.

On AIX, for example, **cmqc.h** has

```
#if defined(__64BIT__)
 #define MQ_64_BIT
#endif
#if defined(MQ_64_BIT)
 typedef int MQLONG;
#else
 typedef long MQLONG;
#endif
```

Any application program that has kept to using types such as MQLONG or MQCHAR will be able to recompile in 64-bit mode easily. Changing the typedef for MQLONG to an *int* for the 32-bit route through the header file could have been done and would have removed #ifdef clauses, but might have introduced problems for applications that had not followed portability rules and which relied on the real representation of an MQLONG.

You might notice some differences in the printed appearance of an hConn or hObj on some platforms. They can look like memory addresses for 32-bit applications and small integers for 64-bit applications. However, as these correlators must never be interpreted directly by applications the differences are purely cosmetic.

### Compiling programs

The two client SupportPacs install new versions of the MQI libraries, libmqic, and its dependencies. These libraries have the same name as the 32-bit libraries but are put in new directories. The libraries are not symbolically linked into the standard directories as the 32-bit libraries are already in there.

So when you compile and link applications you must specify the directory name. For example, on AIX:

```
cc -q64 program.c -L/usr/mqm/lib64 -lmqm
```

If you do not name the directory the linker will not be able to resolve the MQI symbols as there will not be a compatible version of the library in its default search path. There is no need to name a directory

for the include files, as the same files are used for both 32-bit and 64-bit programs and these are already symbolically linked into the standard search path. The C++ libraries have also been installed in the same way.

### Java

The Java programming language was defined to be 64-bit capable from its inception. However, most implementations of the runtime environment – the JVM – are currently 32-bit and emulate 64-bit behaviour. This does not matter for programs that are pure Java but any attempt to access an external function may need to know about the JVM implementation.

For MQSeries this will show up in the way that Java and JMS programs connect to the queue manager. If the Java classes are invoked inside a 64-bit JVM they should be expected to work with a client connection, but 'bindings mode' involves calls via JNI to a C library. If you try to start a local connection to a queue manager from a 64-bit JVM an error will be generated. This will probably show up as an exception which points to not being able to resolve an MQSeries internal symbol or load a shared library.

## CLIENT ADMINISTRATION

The MQSeries server does not know whether the client connecting to it is 32-bit or 64-bit; this is completely transparent. All clients can connect to all servers. However, an administrator might need to be aware of the application requirements when configuring the channel definitions as it could involve the naming and execution of channel exits.

### Channel exits

MQSeries client programs can have exits that are loaded at runtime into the application program. The decision to use these exits can be made either programmatically (using the MQCONNX verb) or by an administrator defining CLNTCONN channels and then distributing the *AMQCLCHL.TAB* file.

Because the same definition file can be used for all programs on a machine the format is usable by both 32-bit and 64-bit programs.

However, you will need to distinguish between channels that are used by the two types of program as they will not be able to use the same exit modules. The MQSeries client code will attempt to load the exit named in a channel definition and this exit must be compiled to match the size of the program loading it.

One choice for channel exits is to have an exit compiled twice with two channel definitions that point to the correct version, and the program makes use of the appropriate name in the MQCONN call.

An alternative (which is not recommended) is to only use 'base' names of the modules in a channel definition and then rely on LIBPATH processing to pick up the correct one.

Because of the sensitivity of channel exits, which can manipulate the security controls on a channel, I would always prefer to use fully-qualified pathnames in a definition so that it is clear which exit is being loaded and run, even with the inconvenience of needing two channel definitions.

## SUMMARY

The latest SupportPacs have extended MQSeries into the 64-bit world on two of the most popular Unix platforms. We have seen how and where these can be used and administered.

If you are considering building your applications as 64-bit you need to make sure you are doing it for the right reasons and that you have structured your code in a way which will actually take advantage of the possibilities of the memory or system architecture. It should not be done simply because it may be perceived as fashionable.

The SupportPacs should not be seen as the end of the 64-bit capabilities in MQSeries. There are more things that could be done and I've alluded to some of them here. The now-shipping code is just a stage on the path towards fuller support.

---

*Mark E Taylor*
*Technical Strategist, IBM Hursley (UK)*

# Dead letter queue handling: MQSeries for OS/390

## INSTALLATION

To install DLQ Handler follow the steps given below:

- Send the REXX code to your mainframe (ASCII mode in FTP or ASCII and CR/LF in Personal Communications file transfer).
- Store it as a library member (RECFM=FB, LRECL=80) named DLQHANDL.

Now you are ready to use it! To do so:

1) Read this article.
2) Prepare the Rules Table.
3) Use the supplied job (after tailoring to your site's requirements) to run DLQ Handler against your dead letter queue.

The REXX source code for DLQ can be found on the Xephon Web site at *www.xephon.com/extras/DLQhandler.txt*.

## USE

The dead letter queue is an important element of MQSeries' design because it helps to achieve the assured delivery feature. If a message cannot be delivered to its destination, perhaps because the destination queue does not exist or is full, it will be put to the dead letter queue. The MQSeries administrator can look at messages on the dead letter queue at an appropriate time and decide what to do with them.

The dead letter queue is not mandatory but it is strongly recommended that you have one. If you do use one do not allow it to fill up because this could, for example, cause communication problems with your queue manager.

The utility **runmqdlq** provides assistance with managing the dead letter queue, but unfortunately it is not available on OS/390 prior to MQSeries V5.2. My DLQ Handler fills this gap. I have written it in the REXX programming language. Since the standard MQSeries API

is not available for REXX I used a support pack (*MA18*) which implements just that. REXX is relatively simple and self-commenting, so I think that my DLQ Handler can be used as an example of manipulating messages on the dead letter queue. Feel free to change the code as you wish.

DLQ Handler behaves much like the **runmqdlq** utility known from other platforms for which MQSeries is available (with a few minor differences). It reads a message from the dead letter queue (in browse mode) and tries to apply rules supplied in the Rules Table.

A rule consists of pattern-matching keywords and action keywords. If a rule matches a message the required action is taken and the next message is read. If one rule doesn't match, DLQ Handler tries to apply the next one. When none of the rules match the message it is left on the dead letter queue and the next message is processed. Having finished processing all messages on the dead letter queue DLQ Handler will, depending upon the selected options, quit or wait for a new message to arrive. The wait time can be either a specified or an unlimited interval.

### Syntax

I use the following syntax to describe all keywords:

```
KEYWORD value1 | value2 | … | valuen | default_value
```

Note that **string[n]** means a user-supplied string of up to *n* characters and that **numeric[n..m]** means a user-supplied numeric in the range from *n* to *m.*

All parameters written in capital letters are DLQ Handler keywords.

## RUNNING DLQ HANDLER

DLQ Handler can be invoked either from the supplied job or run from a TSO or ISPF line command. The last two options are not recommended since, depending on the options selected, DLQ Handler can run for a long time (even infinitely!).

The job to run DLQ Handler is shown and explained below.

```
//DLQHANDL JOB NOTIFY=&SYSUID
```

```
//* parameters in SYSTSIN:
//*  QMgrName
//*  DLQName
//*  RetryInterval
//*  WaitOption
//GETDLQ    EXEC PGM=IKJEFT01,
//          PARM='%DLQHNADL'
//STEPLIB   DD DSN=MA18.LOAD,DISP=SHR
//          DD DSN=MQM.SCSQAUTH,DISP=SHR
//SYSEXEC   DD DSN=YOUR.DSN,DISP=SHR
//SYSTSPRT  DD SYSOUT=*
//SYSIN     DD DSN=YOUR.DSN(RULES),DISP=SHR
//SYSTSIN   DD *
QmgrName
DLQName
RetryInterval
WaitOption
/*
```

**Notes on the above code**

- Line 1: supply a valid job card.
- Line 9: MA18 load library.
- Line 10: MQSeries *hlq.SCSQAUTH* library.
- Line11: library containing the DLQHANDL source.
- Line 12: a class for sysout (DLQ Handler reports will be printed there).
- Line 13: Rules Table DSN (may be sequential file or a PDS member).
- Lines 15,16,17,18: parameters for DLQ Handler:
  - queue manager name (required)
  - dead letter queue name
  - retry interval in seconds
  - wait option (NO | YES | time.
- The parameters for DLQ Handler must be given in the order shown; the last three may be omitted (leave a blank line instead).
- The queue manager name is required – it has no default.

*Parameters*

The meaning of the first two parameters is self-explanatory; here are the other two:

- RetryInterval time | 60. This is the time between subsequent attempts to process a message that could not be processed at the first attempt; it has meaning only for ACTION RETRY.
- WaitOption NO | YES | time. This indicates whether DLQ Handler should wait for more messages to arrive on the dead letter queue after processing all messages on the queue; time is given in seconds.
  - NO: DLQ Handler will quit after processing the last message
  - YES: DLQ Handler will wait indefinitely.
  - time: DLQ Handler will wait a given number of seconds.

*Default values for DLQ Handler parameters*

- QMgr: none.
- DLQName: none. If blank, DLQ Handler will query the queue manager for it (it is stored in a queue manager attribute DEADQ); if it cannot be obtained DLQ Handler will quit.
- RetryInterval: 60 seconds:
  - WaitOption: 'NO'.

*Beware!*

The default value of WaitOption for **runmqdlq** on other platforms is 'YES'. I thought 'NO' would be more appropriate. You can still change it in the REXX code (line 16).

Another parameter which may require manual intervention is MaxMsgLen. It is hard-coded with a value of 1000. Again, you can change it in the code (line three). Using *MA18* (*REXX Interface to MQSeries*) it is quite awkward to get the actual message length so I decided to work with truncation.

## THE RULES TABLE

Please observe the following when you prepare the Rules Table.

- Keywords are coded one per line.
- Keywords can occur in any order.
- A keyword can be included only once in any rule (if repeated the last occurrence will overwrite the previous one).
- Keywords are case sensitive: everything should be coded in capital letters (except queue and queue manager names if they are mixed case).
- User-supplied values (like queue names) are case sensitive (as is MQSeries).
- A parameter value for each keyword must be coded in the same line separated by at least one blank.
- Each rule must end with a single asterisk (‘*’) in a new line in the first column.
- Each rule must have the ‘ACTION’ keyword with a proper value (see below); all other keywords are optional (default values are given below).
- Support of wildcards is limited (as opposed to the **runmqdlq** utility; I may add processing of them in future versions of DLQ Handler); you can use a wildcard character of ‘*’ but it must be the only character in the parameter.
- If a rule has more than one pattern-matching keyword a message to be processed must match all of them (logical AND is used); if you need to implement logical OR for patterns, build several rules with the same action but different patterns.

## KEYWORDS

### Pattern-matching keywords

The pattern-matching keywords are used to specify values that will be matched against messages to be processed. All pattern-matching keywords are optional so if you don’t code any of them the rule will

match every message. An asterisk (‘*’) coded as a value for any pattern-matching keyword means no checking. The same effect can be achieved by specifying a blank or by omitting the keyword.

- APPLIDAT string[16] | *

  Matched against the value of the ApplIdentityData field in the message descriptor, MQMD, of the message on the dead letter queue.

- APPLNAME string[28] | *

  Matched against the value of the PutApplName field in the message descriptor, MQMD, of the message on the dead letter queue.

- APPLTYPE numeric | *

  Matched against the value of the PutApplType field in the message descriptor, MQMD, of the message on the dead letter queue. You must use a numeric value, not MQSeries symbolic names like MQAT_MVS!

- DESTQ string[48] | *

  Matched against the value of the DestQName field in the dead letter header, MQDLH, of the message on the dead letter queue.

- DESTQM string[48] | *

  Matched against the value of the DestQMgrName field in the dead letter header, MQDLH, of the message on the dead letter queue.

- FEEDBACK numeric | *

  Matched against the value of the Feedback field in the message descriptor, MQMD, of the message on the dead letter queue. You must use a numeric value, not MQSeries symbolic names like MQFB_COA!

- FORMAT numeric | *

  Matched against the value of the Format field in the message descriptor, MQMD, of the message on the dead letter queue. You must use a numeric value, not MQSeries symbolic names like MQFMT_STRING!

- MSGTYPE numeric | *

  Matched against the value of the MsgType field in the message descriptor, MQMD, of the message on the dead letter queue. You must use a numeric value, not MQSeries symbolic names like MQMT_REPORT!

- PERSIST numeric | *

  Matched against the value of the Persistence field in the message descriptor, MQMD, of the message on the dead letter queue. You must use a numeric value, not MQSeries symbolic names like MQPER_PERSISTENT!

- REASON numeric | *

  Matched against the value of Reason in the dead letter header, MQDLH, of the message on the dead letter queue. I recommend using MQSeries symbolic names like MQRC_Q_FULL for example. This one is different from all the other pattern-matching keywords!

- REPLYQ string[48] | *

  Matched against the value of ReplyToQ in the message descriptor, MQMD, of the message on the dead letter queue.

- REPLYQM string[48] | *

  Matched against the value of ReplyToQMgr in the message descriptor, MQMD, of the message on the dead letter queue.

- USERID string[12] | *

  Matched against the value of the UserId field in the message descriptor, MQMD, of the message on the dead letter queue.

**Action keywords**

The action keywords tell DLQ Handler what to do with a message which matched the patterns.

There is one mandatory action keyword – ACTION – and five optional ones. Depending on the value of the ACTION keyword some optional keywords may be required as well.

- ACTION FWD | RETRY | DISCARD | IGNORE

  This is the action to be taken for messages matching a pattern in this rule. The components break down as follows:

  – FWD – the message will be forwarded to another queue (and possibly another queue manager). This value requires the use of the FWDQ keyword. Other action keywords – FWDQM, HEADER, and PUTAUT – also have an impact on the behaviour of DLQ Handler if FWD is used. See details below.

  – RETRY – the message will be put on the destination queue after the time period specified in the RetryInterval parameter. Another action keyword related to the RETRY action is RETRY (the names are identical but these are two different parameters: one is a possible value of the ACTION keyword; another is a separate keyword). See details below.

  – DISCARD – the message will be unconditionally discarded from the dead letter queue.

  – IGNORE – the message will left on the dead letter queue. DLQ Handler will start processing for the next message on the queue.

- FWDQ &DESTQ | &REPLYQ | string[48]

  The above command specifies the name of the queue to which DLQ Handler should forward current messages. It has no default value. It is meaningful only when action FWD is requested (and required in that case!). You can code any queue or use two special values for this keyword:

  – &DESTQ – the current message will be forwarded to the original destination queue (its name will be taken from the DestQName field in the dead letter header, MQDLH, of the current message).

  – &REPLYQ – the current message will be forwarded to the reply queue specified in the ReplyToQ field in the message descriptor, MQMD, of the current message.

- FWDQM &DESTQM | &REPLYQM | string[48] | ' '

  The above command specifies the name of the queue manager to which DLQ Handler should forward current messages. It is meaningful only when action FWD is requested. You can code any queue manager name (when left blank, which is the default value, the local queue manager name will be used) or use two special values for this keyword:

  – &DESTQM – the current message will be forwarded to the original destination queue manager (its name will be taken from the DestQMgrName field in the dead letter header, MQDLH, of the current message).

  – &REPLYQ – the current message will be forwarded to the reply queue manager specified in the ReplyToQMgr field in the message descriptor, MQMD, of the current message.

- HEADER YES | NO

  The above command specifies whether the dead letter header of the current message should accompany it when the message is forwarded to another queue. Meaningful only for action FWD.

- PUTAUT DEF | CTX

  The above command defines the authority with which the message should be put by the DLQ Handler. Meaningful for actions FWD and RETRY.

  – DEF – the message will be put with the authority of DLQ Handler. Effectively, it is the authority of the user that runs DLQ Handler.

  – CTX – the message will be put with the authority of the user named in the UserId field of the message descriptor, MQMD, of the message being processed. Using CTX requires authority to use alternate user authority (see the chapter on security in the *MQSeries System Management* book).

- RETRY 1 | numeric

  This specifies the number of times DLQ Handler will try to put the current message to its destination queue. It is meaningful only

for action RETRY. Standard **runmqdlq** uses it also for FWD and DISCARD.

## CONCEPTS AND HINTS

Before you define any rules for processing messages on the dead letter queue make sure you know when and why messages can be put to it. I have outlined a few reasons below.

Messages that are put to the dead letter queue have a special format, namely an additional header called MQDLH that is appended in front of the application data. The format of a message sent to the dead letter queue is set to MQFMT_DEAD_LETTER_HEADER. The MQDLH header includes the following information (for a full description please refer to the *MQSeries Application Programming Reference*).

- Reason – reason the message arrived on the dead letter queue.
- DestQName – the name of the original destination queue the message was meant to go to.
- DestQMgrName – the name of the original destination queue manager the message was meant to go to.

Listed below are some of the reasons why a message can be put on the dead letter queue.

- The destination queue is full.
- The size of the message is greater than the MaxMsgLen attribute of the destination queue.
- Put is disabled for the destination queue.
- The User ID being used to open the destination queue is not authorized to do so.
- The message on the transmission queue is not in the correct format.
- The format of the trigger message is not correct.
- The program to be triggered cannot be found.

If you have a good idea of the reasons why messages are put to the dead

letter queue at your site you can create a Rules Table and run DLQ Handler. Initially, it is a good idea to run a test on the rules or use an 'artificial' dead letter queue. Bear in mind that DLQ Handler, depending on the options you select, can send a message from the dead letter queue to another queue (or another queue manager) and can even discard it completely.

**Warning**

Do not let the dead letter queue fill up!

DLQ Handler will process only messages that have a valid MQDLH header. Other messages will be reported to be in error but will be left on the dead letter queue indefinitely. So if you find any messages on your dead letter queue after running DLQ Handler they might be such invalid dead letter messages. You should process them manually. Either discard them or get and put them to another queue.

To ensure that any valid messages are not left on the dead letter queue after DLQ Handler processing include in your Rules Table the following rule (make it the last one):

```
ACTION FWD
FWDQ 'ANOTHER.QUEUE'
FWDQM ''
```

Thus any message that did not match any previous rule will be sent by the last rule to a queue called 'ANOTHER.QUEUE'.

**Another warning**

Watch security!

To open the dead letter queue for input and to open other queues for output (when action FWD or RETRY is requested) DLQ Handler uses by default the authority of the user that submitted the job or the user given in the job card. Make sure that your user has all the authority he needs. I will not discuss here specific RACF (or other security products) profiles – refer to *MQSeries for OS/390 System Management* for details. I will just outline the authorities DLQ Handler will need.

DLQ Handler needs authority to:

- Open the dead letter queue for browse – all actions.

- Open the dead letter queue for input – FWD, RETRY, and DISCARD actions.
- Open destination queue for output – FWD (for the queue specified in the FWDQ keyword) and RETRY (for the queue specified in the DestQName field of the dead letter header, MQDLH) actions.

Bear in mind that DLQ Handler can send a message to another queue manager. In that case authority for opening the proper transmission queue for output will be needed.

If you want to use context security (you do so by specifying PUTAUT CTX) remember that additional authorities will be necessary.

To open the output queue with alternate user authority the context user, which is taken from the UserId field of the message descriptor MQMD, must be authorized to open the destination or forward queue for output.

The use of context authority can be difficult to maintain if there are numerous messages from different platforms generated by different users.

**Forwarding messages to another queue manager**

DLQ Handler can forward a message to a remote queue manager (if the FWDQM keyword does not point to the local queue manager). Remember that some administrative work will be needed to achieve this. Normally, forwarding a message to another queue manager will require having a transmission queue with the name of the remote queue manager (and a channel that reads from that queue, needless to say).

The goal can be also achieved by a queue manager alias (you create a queue manager alias by defining a remote queue) or by configuring a cluster of queue managers. Please refer to the *MQSeries for OS/390 System Management* manual for details.

---

*Marcin Grabinski*
*System Engineer*
*SPIN (Poland)*

# Buying more hardware capacity for MQSeries Integrator

## OVERVIEW

The idea for this article started with a study I made on how maximum throughput varied across three different disk types:

- A fast/wide SCSI disk.
- An SSA disk with non-volatile fast write cache.
- A Solid State disk.

I have broadened the idea a little to include a very brief overview of the other hardware resources we buy to get ourselves out of a capacity problem, namely processors, memory, and network.

## NETWORK CAPACITY

This factor is the least likely to be a problem. Make the assumption that a network card will run well only if it stays under 50% of the theoretical maximum capacity.

The rule of thumb, therefore, gives a 100 megabit Ethernet card a maximum planned rate of 50 megabits/second. For example, a 50KB message (400 megabits) should have a planned message rate of 125 messages/second. Note that it is possible to get more (even double) if the card is configured ‘full duplex’ and the traffic each way is about equal.

## MEMORY

To adequately support MQSI brokers in a production environment I recommend at least 1GB of memory. Each dataflowengine process (equivalent to an execution group) is likely to take at least 50MB (the number and size of the flows contributes to the size) and the minimum needed to start with one execution group is 512MB.

## PROCESSOR

With one or more processors 100% busy the answer is simple: buy more or faster. Difficulties occur when all the processors on the machine are not 100% busy (or near enough) but the maximum message rate has been reached.

Unless ‘additional instances’ has been specified on an execution group a dataflowengine process can only use one processor.

The answer here may be to increase the number of execution groups and/or the number of additional instances specified for each message flow (as a general rule, up to the number of available processors). I said ‘may’ because if the bottleneck is file writing/reading the above will not help at all. This leads me nicely on to the main topic.

## DISK

If the messages driving MQSI are persistent the MQ log is a prime candidate for a potential bottleneck. It was to investigate this problem that I undertook a small study to see the effect on throughput of replacing the standard SCSI disk with an SSA disk with non-volatile fast write cache or a Solid State disk. The configuration for the study was:

- A Netfinity 8500R with four Intel 700Mhz processors and 3.2GB RAM.
- MQSeries V5.2.
- MQSI V2.01.
- The SSA disk was an Advanced SerialRAID/X Adapter (which had 32K write cache, 64K read cache) connected to an IBM 7133-D40 enclosure containing 8 IBM DMVC 9G drives in a single loop.
- The Solid State disk setup was a fast/wide differential ultra SCSI Adaptec 2944W adapter and a SolidData Excellerator disk with a 2GB capacity.

## SSD VERSUS SCSI MEASUREMENT

Single execution group lean and mean nodes (eg input – output nodes,

input – filter – output, and PubSub with a small number of subscribers) show large percentage benefits (in the 600-700% range).

As MQ V5.2 logging becomes more efficient given more parallel units of work (more execution groups) the percentage benefits are lower than above, although still in the 300-500% range.

Any node with a maximum non-persistent message rate of less than 50 messages per second will be limited by processor power rather than the SCSI disk so we expect much smaller benefits for these measurements.

Putting the MQ log rather than the MQ queues on SSD had by far the greatest benefit. The only measurements to improve by putting the MQ queues on SSD were the multiple subscriber measurements that had many MQPUTs.

**Results**

Table 1 shows the maximum message rates attained with non-persistent messages and then persistent messages. The first column shows the message rate for different nodes when both the MQ log and the MQ queues were placed on SCSI disks (marked SCSI). The next column shows the rates when the MQ log was moved to a Solid State

| | | Non-persistent messages | | | Persistent messages | | |
|---|---|---|---|---|---|---|---|
| | **MQ log on** | **SCSI** | **SSD** | **SSD** | **SCSI** | **SSD** | **SSD** |
| | **MQ queues on** | **SCSI** | **SSD** | **SSD** | **SCSI** | **SSD** | **SSD** |
| **Node** | **RUN** | **msgs/ sec** | **msgs/ sec** | **msgs/ sec** | **msgs/ sec** | **msgs/ sec** | **msgs/ sec** |
| In/out | 1 Exec grp | 2946 | 3138 | 2913 | 59 | 438 | 435 |
| In/out | 3 Exec grps | 3441 | 3400 | 3591 | 143 | 390 | 512 |
| Pub/sub | 1 Exec grp, 1 sub | 1243 | 1362 | 1414 | 55 | 360 | 332 |
| Pub/sub | 1 Exec grp, 10 subs | 489 | 488 | 478 | 32 | 102 | 127 |
| Pub/sub | 1 Exec grp, 100 subs | 58 | 61 | 62 | 7 | 11 | 15 |
| Pub/sub | 2 Exec grps, 1 sub | 2631 | 2301 | 2295 | 91 | 431 | 448 |
| Filter | 1 Exec grp | 1237 | 1261 | 1260 | 51 | 333 | 332 |
| Compute | 1 Exec grp, very complex | 50 | 48 | 48 | 30 | 45 | 46 |
| Compute | 4 Exec grps, very complex | 214 | 207 | 209 | 94 | 158 | 160 |

*Table 1: SSD versus SCSI – maximum message rates attained*

disk (marked SSD), and the next column shows the rate when the MQ queues are moved on to Solid State disks.

It shouldn't be too much of a surprise to see that moving the log to Solid State disks with persistent messages gave the most benefit.

I have provided a brief explanation of the Node and Run columns below.

- Node:
  - IN/OUT: a very simple flow with an input node connected to an output node
  - PUB/SUB: an input node connected to a PubSub node
  - FILTER : an input node connected to a filter node connected to an output node
  - COMPUTE: an input node connected to a compute node connected to an output node.
- Run
  - *n* subs: every published message has *n* subscribers' queues which will receive the message. The rates quoted are the published rates, which are the same as the receive rate per subscriber
  - very complex: a compute node with 'lots' of ESQL.

## SSA VERSUS SCSI MEASUREMENT

Table 2 shows the maximum message rates attained with non-persistent messages and then persistent messages. The first column shows the message rate for different nodes when both the MQ log and the MQ queues were placed on SCSI disks.

The next column shows the rates when the MQ log was moved to a SSA Disk and the next column shows the rate when the MQ queues are moved on to SSA Disks. As before, moving the log to SSA disk with persistent messages gave the most benefit.

| Node | MQ log on<br>MQ queues on<br>RUN | Non-persistent messages<br>SCSI<br>SCSI<br>msgs/<br>sec | <br>SSA<br>SCSI<br>msgs/<br>sec | <br>SSA<br>SSA<br>msgs/<br>sec | Persistent messages<br>SCSI<br>SCSI<br>msgs/<br>sec | <br>SSA<br>SCIS<br>msgs/<br>sec | <br>SSA<br>SSA<br>msgs/<br>sec |
|---|---|---|---|---|---|---|---|
| In/out | 1 Exec grp | 2946 | 2784 | 3148 | 59 | 397 | 384 |
| In/out | 3 Exec grps | 3441 | 3351 | 3427 | 143 | 542 | 570 |
| Pub/sub | 1 Exec grp, 1 sub | 1243 | 1407 | 14397 | 55 | 308 | 305 |
| Pub/sub | 1 Exec grp, 10 subs | 489 | 478 | 468 | 32 | 101 | 121 |
| Pub/sub | 1 Exec grp, 100 subs | 58 | 62 | 58 | 7 | 11 | 16 |
| Pub/sub | 2 Exec grps, 1 sub | 2631 | 2627 | 2366 | 91 | 451 | 458 |
| Filter | 1 Exec grp | 1237 | 1140 | 1234 | 51 | 348 | 308 |
| Compute | 1 Exec grp, very complex | 50 | 50 | 51 | 30 | 45 | 45 |
| Compute | 4 Exec grps, very complex | 214 | 215 | 210 | 94 | 164 | 165 |

*Table 2: SSA versus SCSI – maximum message rates attained*

## SSA OR SOLID STATE DISK?

This was not an exhaustive study but first glance shows that there was very little to choose between them in pure performance terms (see Table 3). I assume price and capacity will dictate the choice.

## SUMMARY

Message flows that use non-persistent messages are unlikely to benefit from these faster disks unless heavy use is made of PubSub. If persistent messages are used and the combined rate of all the execution groups is likely to be greater than 50 messages/second, faster disks should be considered. I had a brief look at placing a DB2 log on fast disk and the performance improvement was very encouraging

In summary, faster disks can improve maximum throughput considerably but a knowledge of the flows is needed in order to make the buying decision.

| Node | MQ log on<br>MQ queues on<br>RUN | SSA<br>SSA<br>msgs/sec | SSD<br>SSD<br>msgs/sec |
|---|---|---|---|
| In/out | 1 Exec grp | 2946 | 2784 |
| In/out | 3 Exec grps | 3441 | 3351 |
| Pub/sub | 1 Exec grp, 1 sub | 1243 | 1407 |
| Pub/sub | 1 Exec grp, 10 subs | 489 | 478 |
| Pub/sub | 1 Exec grp, 100 subs | 58 | 62 |
| Pub/sub | 2 Exec grps, 1 sub | 2631 | 2627 |
| Filter | 1 Exec grp | 1237 | 1140 |
| Compute | 1 Exec grp, very complex | 50 | 50 |
| Compute | 4 Exec grps, very complex | 214 | 215 |

*Table 3: SSA or Solid State disk?*

*R E Branagan*
*Software Engineer*
*IBM Hursley (UK)*



## Need help with an *MQSeries* problem or project?

Maybe we can help:

- If it's on a topic of interest to other subscribers, we'll commission an article on the subject, which we'll publish in *MQ Update*, and which we'll pay for – it won't cost you anything.
- If it's a more specialized, or more complex problem, you can advertise your requirements (including one-off projects, freelance contracts, permanent jobs, etc) to the hundreds of MQSeries professionals who visit *MQ Update*'s home page every month. This service is also free of charge.

Visit the *MQ Update* Web site, http://www.xephon.com/mqupdate.html, and follow the link to *Suggest a topic for an article* or *Opportunities for MQ specialists*.

# Experiences with MQSeries clustering

## INTRODUCTION

MQSeries V5.1 introduced the concept of clustering, where queue managers could be connected together to form a cluster. There are two significant benefits in using clusters:

- Simplified administration. Without clusters, setting up a complex distributed queueing network can be time-consuming and error-prone because of the sheer number of objects which need to be correctly defined (transmission queues, sender/receiver channels, remote queues).
- Increased system availability and workload balancing. Instances of the same queue can be placed on multiple queue managers within a cluster and the work can be shared across these queue managers automatically by MQSeries.

This article describes our organization's experiences with clusters and provides a number of hints, tips, and recommendations for exploiting this very powerful technology.

## PAST EXPERIENCES AND LESSONS LEARNT

We have been using MQSeries within our organization for over three years and have experimented with clusters since the concept was originally introduced into the product about two years ago.

Early experiences with using clusters were mixed. When you set up your clusters correctly without making any mistakes they function perfectly and really do simplify administration as well as providing the usual high availability and load-balancing features. However, many times in the early days, if a minor mistake was made in any of the cluster definitions or commands were run out of sequence, the cluster simply would not work. Worse than that, it was often impossible to rectify the mistake without deleting and recreating the queue manager. These early bad experiences made us very wary about implementing clusters across the organization.

Our main concern was that, if something went wrong within the cluster, how could we fix it? On the one hand clusters simplify administration as the software builds definitions for objects automatically and stores them away internally within its repositories; however, when things go wrong it is practically impossible to modify the internal data held within the repository.

The simple answer is not to make a mistake in the configuration (and this point has been voiced by a number of people from IBM). I have written a number of scripts to automate the tasks involved in getting queue managers to join clusters and this has made the probability of making mistakes much lower. However, when you do make a mistake (and you will) then MQ is just too unforgiving.

The implementation of MQ Clusters has improved significantly since the original release and now the product functions well and is reasonably robust. The problems which arise when errors are made during configuration still hold true, however.

## CURRENT USE OF CLUSTERING

Within our organization we have recently implemented a network of integration hubs based on MQSeries and MQSeries Integrator. These hubs are located around the world in order to satisfy our global business.

There were some requirements placed on the network of hubs and these are listed below.

- Even though multiple hubs make up the integration network it had to be viewed as a single logical entity. Applications always connected to their local hub and could bridge through to any other queue manager connected to the logical entity.
- Support for workload balancing within the context of a single hub.
- The network had to offer a 24x7 service by utilizing other hubs within the logical entity. If the local hub became unavailable then there would still be other hubs available to an external application server.

The above requirements were implemented using:

- standard MQSeries distributed queueing facilities, specifically multi-hop capabilities implemented using queue manager aliases and clusters. The hubs which make up the single entity were grouped together in a 'Global Gateway Cluster'.
- a cluster which includes the application server queue manager along with the queue managers residing on the local hub.
- a cluster to include the queue manager on the application server along with the local hub queue manager and another 'standby' queue manager on another hub.

The last two types of cluster were merged into a single Application Availability Cluster. Figure 1 illustrates the implemented topology.

Before clusters were used intercommunication was implemented using *inetd*. Once multiple overlapping clusters were required we moved to using the MQ Listener program (*runmqlsr*) as this allowed us to isolate clusters by having cluster receiver channels listening on different ports.

## HINTS AND TIPS FOR PROBLEM SOLVING

Whilst we've been using clusters we have encountered many problems and below is a list of useful hints and tips which may be of potential use.

- If your cluster is not working correctly (eg the cluster queue is not visible) check to see if the cluster repository manager process (amqrrmfa) is running. If it is not running it can be easily restarted using the command **amqrrmfa –m QmgrName**.
- When you issue cluster commands such as 'refresh' they are written to the system queue known as *SYSTEM.CLUSTER.COMMAND.QUEUE*. If messages are not moving from this queue it is sometimes useful to clear the queue down.
- MQ maintains two other system queues associated with clusters:
  - *SYSTEM.CLUSTER.TRANSMIT.QUEUE* – which is used to talk to the other queue managers within the cluster.

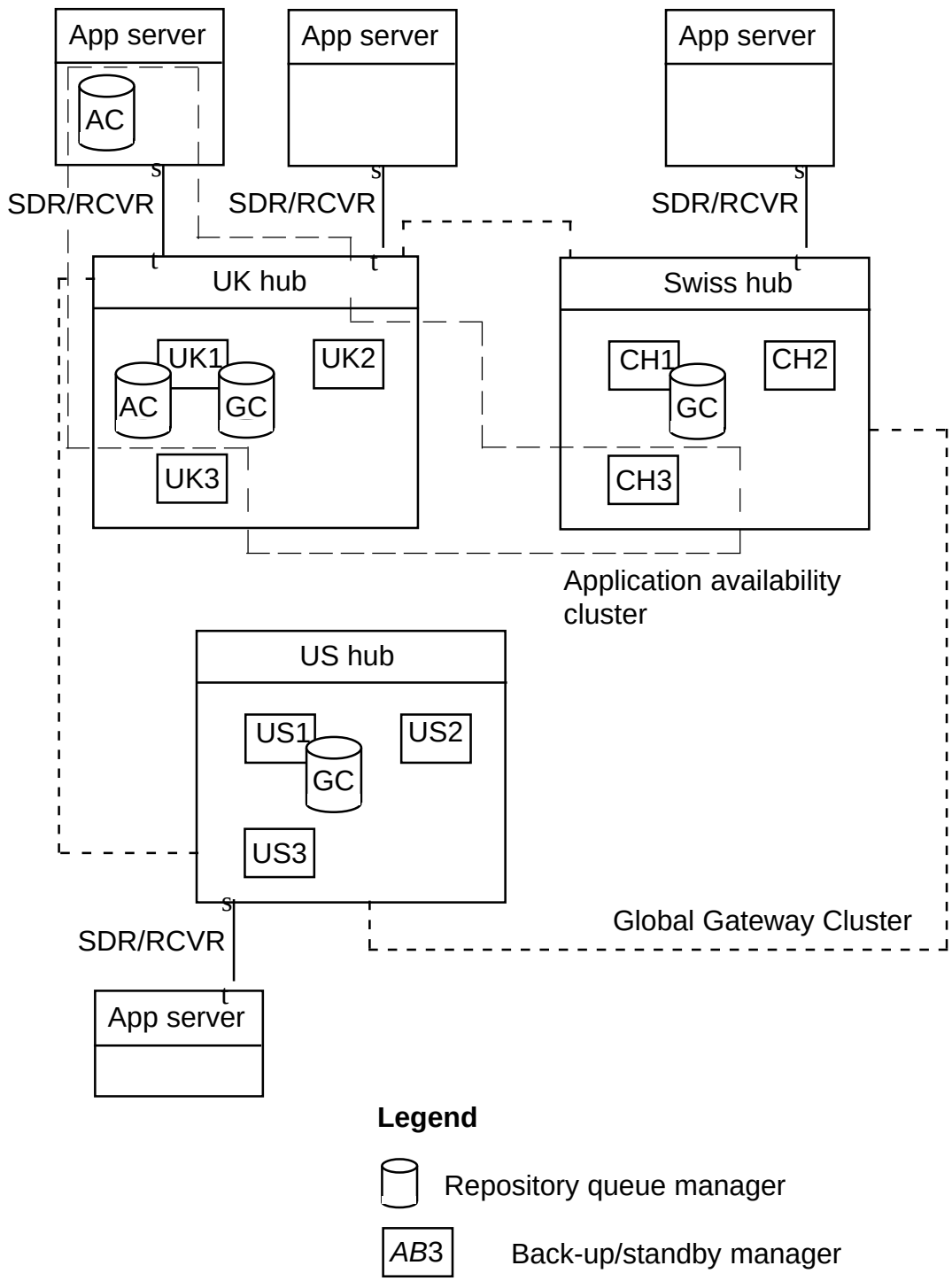


*Figure 1: Cluster topology*

– *SYSTEM.CLUSTER.REPOSITORY.QUEUE* – which is used to actually store the repository data, eg information on other participating queue managers within the cluster and queues shared within the cluster.

In desperate situations it is sometimes necessary to delete the contents of these queues. This should only be performed if your cluster really is broken and you cannot tolerate deleting and recreating the queue manager. You cannot clear these queues down whilst the repository queue manager process is running.

Often we have had to kill the repository manager process (amqrrmfa), clear down these queues, and then stop/start the queue manager in order to refresh the cache. Again, I must stress that this course of action is very much a last resort.

- I strongly recommend using DNS names when defining connection names for cluster channels. The repository stores this connection name and, if you use explicit IP addresses and a queue manager moves IP address, it is practically impossible to alter this within the cluster definitions.

  When defining a cluster receiver channel be very careful when using the DNS name in the connection definition as you cannot guarantee that participating queue managers within the cluster also have access to that DNS definition. If you don't have global DNS *do not* use the DNS name in the connection.

- If a cluster sender channel goes into retry state it may be because it has got out of step with the cluster receiver channel at the other end. In this case reset the cluster channel at either end to message number one. Failing this, delete and recreate the channel definition and restart it. If this still doesn't work then restart the queue manager (which is a last resort).

- If a cluster sender channel goes into retry state check the queue manager error log. If the error log indicates that another cluster sender channel is in an 'in-doubt' state then even if that cluster sender channel no longer exists you should recreate the channel, resolve the in-doubt condition with a commit, and then delete the channel.

- Automatically-defined cluster sender channels are modelled on the cluster receiver channel 'at the other end' and have a DEFTYPE of CLUSSDRA when issuing a DIS CLUSQMGR(*) DEFTYPE command. This is useful to know when diagnosing problems.
- When setting up multi-hopping using clusters you may need to create a remote queue definition to a clustered queue. However, this is not possible within the product!

  For example, if queue A defined on queue manager CH01DM1 is shared within a cluster so that queue manager UK01DM1 can see it, you cannot create a remote queue definition on queue manager NB01, which points to queue A on queue manager UK01DM1.
- For performance reasons cluster queue managers maintain a repository cache which contains the clustering definitions (such as cluster queues, cluster channels, etc).

  The undocumented command **amqrfdm** can be used to get information from the repository and cache. This program can be executed using the command **amqrfdm –m QmgrName**.

  The program isn't terribly friendly to use and does not allow you to amend the repository or the cache, but does provide useful information on how it believes the cluster is configured. The really useful option is **"o"** to output the entire cache.

## THE WAY FORWARD AND OTHER RECOMMENDATIONS

We are using clusters now across multiple platforms and they perform well. However, a number of areas need to be examined, ideally as part of the core MQSeries product, but perhaps there is an opening for ISV products. These areas include the following topics.

- The structure and architecture repository needs to be documented and published so that users can understand what MQ believes to be configured.
- Some form of repository view/edit tool would allow the repository to be dynamically inspected and, in cases of emergency, modified.
- A *Redbook* aimed specifically at MQSeries clusters would be

much appreciated by all those users out there.

From personal experience, I advocate the following:

- Use clusters when appropriate:
  - for very large MQ networks with complex distributed queueing requirements
  - when high availability is required (although clusters on their own do not solve this problem)
  - when load-balancing is required.
- Ensure that you are running the latest releases of MQ on all your platforms with the latest CSDs and APARs applied. Invariably, the PTF list for a CSD contains references to fixes to clustering problems.
- Use clusters with measured caution. They do simplify administration but you need to know what you are doing. Encourage your MQ administrators to experiment extensively with clusters in a large variety of configurations across all your different platforms.

  Try to get your administrators trained – IBM does not offer a clustering course *per se*, but there are MQ training partners, such as Alphacourt in the UK, who offer a specific course on clusters.
- Always follow the instructions in the clustering manual (*MQ Clusters SC34-5349-00*) to the letter. There were a number of major errors in the manual in early editions so ensure that you have the latest version of the manual.
- Where possible, build scripts to perform common tasks such as:
  - adding a queue manager to an existing cluster
  - removing a queue manager from a cluster
  - removing a queue from a cluster.
- Define naming standards for:
  - cluster names

- namelists (useful when overlapping clusters are required)
- cluster sender channels
- cluster receiver channels.

## APPENDIX A: THE PERL SCRIPT TO BUILD CLUSTER DEFINITIONS

This script was written in Perl and has been tested on AIX. Whilst it may not be possible to use 'as-is' at another site it forms a good basis for your own script development.

The script refers to Global Gateway Clusters (GGC) and Application Availability Clusters (AAC), which are described in the section earlier entitled *Current use of clustering*.

The script file will prompt for names of queue managers, IP addresses, etc and will generate *.mqsc* files which can then be run against the relevant queue managers in order to create the required definitions.

In addition, a *cluster.log* file is created with additional information (eg instructions on how to start listener programs). A *cluster-schematic.log* file is also created, which contains a graphical view of the configurations that were requested.

## BUILDING CLUSTER DEFINITIONS

```
#!/usr/bin/perl -w
# Filename    : cluster.pl                                        #
# Description : Used to generate definitions for clusters     #
# Created     : 07/02/01 Nick Breeds                          #
# Modified    :                                               #
$Version = "1.6.0.0";
use Time::Local;
use File::Find;
use IO::Handle;
# some constants
$False = 0;
$True  = 1;
$Info = "[INFO]";
$Err  = "[ERROR]";
# define main menu   .
format STDOUT =
          ========================
```

```
          =  Cluster Definitions  =               v @<<<<<<
$Version
          ==========================              ---------
(1) Add Hub queue manager into the Global Gateway Cluster (GGC)
(2) Bridge in to GGC from a local queue manager
(3) Bridge out to a local queue manager from the GGC
(4) Add a queue manager into Application Availability Cluster (AAC)
   (Ø) Exit
Enter option [Ø-4]:
#### START OF MAIN LOGIC ####
$LogFile = "cluster.log";
open (LOGFILE, "> $LogFile") or die "$Err: cannot open $LogFile: $!";
$SchematicFile = "cluster-schematic.log";
open (SCHEMATIC, "> $SchematicFile") or die "$Err: cannot open
$SchematicFile: $!";
SCHEMATIC->autoflush(1);
MainMenu();
close (SCHEMATIC) or die "$Err: cannot close $SchematicFile: $!";
close (LOGFILE) or die "$Err: cannot close $LogFile: $!";
exit Ø;
#### END OF MAIN LOGIC ####
sub MainMenu
{
my $option = "";
while ($option ne "Ø") {
   run_cmd("clear");
   write STDOUT;
   $option = uc(<STDIN>);
   chomp ($option);
   if ($option eq "1")      {
      AddIntoGGC ();
                           }
   elsif ($option eq "2")   {
      BridgeIntoGGC ();
                           }
   elsif ($option eq "3")   {
      BridgeFromGGC ();
                           }
   elsif ($option eq "4")   {
      AddIntoAAC ();
                           }
   elsif ($option eq "Ø")   {
      printf "$Info: Remember to check contents of $LogFile and
$SchematicFile\n";}
   else                     {
      AnyKeyToContinue ("$Err: invalid option");
                           }
   }
}
```

```
sub run_cmd
{
   local ($cmd) = @_;
   my $rc = Øxffff & system ($cmd);
   if ($rc & Øxff ØØ) {
     $rc = $rc / 256;
      system("echo $Err: failed to run *$cmd*, reason: $rc ");
      return Ø;
      }
   return 1;
}
sub AnyKeyToContinue
{
   local ($prompt) = @_;
   print $prompt . ", press any key to continue";
   $key = <STDIN>;
   chomp ($key);
}
sub write_header
{
   local ($filename) = @_;
   printf OUTFILE
("************************************************************\n");
   printf OUTFILE ("* FILENAME    : %s\n", $filename);
   printf OUTFILE ("* CREATED     : %s\n", scalar localtime);
   printf OUTFILE
("************************************************************\n");
}
sub AddIntoGGC
{
   run_cmd("clear");
   PromptFor ("Cluster");
   print "Specify details for this queue manager         ..\n";
   PromptFor ("ThisQmgr");
   PromptFor ("ThisIPAddr");
   PromptFor ("ThisPort");
   PromptFor ("Repository");
   print "Specify details for another queue manager holding a full
repository      .\n";
   PromptFor ("ReposQmgr");
   PromptFor ("ReposIPAddr");
   PromptFor ("ReposPort");
   print "\n";
   DisplaySchematic ("AddIntoGGC");
   if (!(YToContinue ("Is this configuration OK"))) {
      return;
   }
   $ThisFile = $ThisQmgr    . ".AddIntoGGC.mqsc";
   open (OUTFILE, "> $ThisFile") or die "$Err: cannot open $ThisFile:
```

```
$!";
   run_cmd ("chmod 777 $ThisFile");
   write_header ($ThisFile);
   printf OUTFILE ("\n* Define cluster receiver channel for this
qmgr\n");
   printf OUTFILE ("DEF CHL($Cluster.$ThisQmgr)              -\n");
   printf OUTFILE ("       CHLTYPE(CLUSRCVR)                 -\n");
   printf OUTFILE ("       CONNAME('$ThisIPAddr($ThisPort)') -\n");
   printf OUTFILE ("       CLUSTER($Cluster)                  \n");
   printf OUTFILE ("\n* Define cluster sender channel for this qmgr\n");
   printf OUTFILE ("DEF CHL($Cluster.$ReposQmgr)             -\n");
   printf OUTFILE ("       CHLTYPE(CLUSSDR)                  -\n");
   printf OUTFILE ("    CONNAME('$ReposIPAddr($ReposPort)')  -\n");
   printf OUTFILE ("    CLUSTER($Cluster)                     \n");
   if (uc($Repository) eq "Y") {
      LogFileDivider ();
      printf LOGFILE ("$Info: On queue manager '$ThisQmgr', a new
namelist\n");
printf LOGFILE ("$Info: called HUB.$ThisQmgr.NLØ1 will be createdby\n");
printf LOGFILE ("$Info: the script file - check that one didn't    \n");
printf LOGFILE ("$Info: already exist;  if it did then you need to \n");
printf LOGFILE ("$Info: manually add the cluster $Cluster to the   \n");
printf LOGFILE ("$Info: existing namelist.                         \n");
printf LOGFILE ("$Info: Also, the script file alters the qmgr to   \n");
printf LOGFILE ("$Info: make it a repos qmgr referring to above    \n");
printf LOGFILE ("$Info: namelist.                                  \n");
printf OUTFILE ("\n* Define namelist for this qmgr\n");
printf OUTFILE ("DEF NL(HUB.$ThisQmgr.NLØ1)                  -\n");
printf OUTFILE ("      NAMES($Cluster)                      \n");
printf OUTFILE ("\n* Make this a full repository\n");
printf OUTFILE ("ALTER QMGR REPOSNL(HUB.$ThisQmgr.NLØ1) \n");
      }
close (OUTFILE) or die "$Err: cannot close $ThisFile: $!";
LogFileDivider ();
printf LOGFILE ("$Info: You'll need to create a listener task for  \n");
printf LOGFILE ("$Info: queue manager '$ThisQmgr' as follows:      \n");
printf LOGFILE ("$Info: runmqlsr -t tcp -p $ThisPort -m $ThisQmgr  \n");
   LogFileDivider ();
   printf LOGFILE ("$Info: $ThisFile created\n");
   AnyKeyToContinue ("$ThisFile created");
}
sub BridgeIntoGGC
{
run_cmd("clear");
print "Specify details for this local queue manager               \n";
PromptFor ("ThisQmgr");
print "Specify details for the primary HUB queue manager          \n";
PromptFor ("ReposQmgr");
PromptFor ("ReposIPAddr");
```

```
PromptFor ("ReposPort");
print "\n";
DisplaySchematic ("BridgeIntoGGC");
if (!(YToContinue ("Is this configuration OK"))) {
      return;
   }
# First do the file for the local queue manager
$ThisFile = $ThisQmgr    . ".BridgeIntoGGC.mqsc";
open (OUTFILE, "> $ThisFile") or die "$Err: cannot open $ThisFile: $!";
   run_cmd ("chmod 777 $ThisFile");
   write_header ($ThisFile);
printf OUTFILE ("\n* Define sender channel for this qmgr\n");
printf OUTFILE ("DEF CHL($ThisQmgr.$ReposQmgr)                  -\n");
   printf OUTFILE ("       CHLTYPE(SDR)                           -\n");
   printf OUTFILE ("       CONNAME('$ReposIPAddr($ReposPort)')  -  \n");
printf OUTFILE ("       XMITQ('$ReposQmgr')                    -\n");
printf OUTFILE ("       DESCR('Sender channel')                 \n");
printf OUTFILE ("\n* Define xmitq to the HUB                    \n");
printf OUTFILE ("DEF QL($ReposQmgr)                            -\n");
printf OUTFILE ("    TRIGGER                                   -\n");
printf OUTFILE ("    TRIGTYPE(FIRST)                           -\n");
printf OUTFILE ("    TRIGDATA($ThisQmgr.$ReposQmgr)            -\n");
printf OUTFILE ("    INITQ(SYSTEM.CHANNEL.INITQ)               -\n");
printf OUTFILE ("    DEFPSIST(YES)                             -\n");
printf OUTFILE ("    USAGE(XMITQ)                               \n");
close (OUTFILE) or die "$Err: cannot close $ThisFile: $!";
# Now do the file for the repos queue manager
$ThatFile = $ReposQmgr   . ".BridgeIntoGGC.mqsc";
open (OUTFILE, "> $ThatFile") or die "$Err: cannot open $ThatFile: $!";
   run_cmd ("chmod 777 $ThatFile");
   write_header ($ThatFile);
   printf OUTFILE ("\n* Define receiver channel for this qmgr\n");
   printf OUTFILE ("DEF CHL($ThisQmgr.$ReposQmgr)              -\n");
   printf OUTFILE ("       CHLTYPE(RCVR)                       -\n");
   printf OUTFILE ("       DESCR('Receiver channel')           \n");
   close (OUTFILE) or die "$Err: cannot close $ThatFile: $!";
   LogFileDivider ();
printf LOGFILE ("$Info: You'll need to create a listener task for\n");
   printf LOGFILE ("$Info: queue manager '$ReposQmgr' as follows:
\n");
   printf LOGFILE ("$Info:    runmqlsr -t tcp -p $ReposPort -m
$ReposQmgr  \n");
   LogFileDivider ();
   printf LOGFILE ("$Info: $ThisFile and $ThatFile created\n");
   AnyKeyToContinue ("$ThisFile and $ThatFile created");
}
sub BridgeFromGGC
{
   run_cmd("clear");
```

```
   PromptFor ("Cluster");
   print "Specify details for this local queue manager         ..\n";
   PromptFor ("ThisQmgr");
   PromptFor ("ThisIPAddr");
   PromptFor ("ThisPort");
   print "Specify details for the primary HUB queue manager
..\n";
   PromptFor ("ReposQmgr");
   print "\n";

   DisplaySchematic ("BridgeFromGGC");
   if (!(YToContinue ("Is this configuration OK"))) {
      return;
   }
   # Do the file for the local queue manager
   $ThisFile = $ThisQmgr    . ".BridgeFromGGC.mqsc";
   open (OUTFILE, "> $ThisFile") or die "$Err: cannot open $ThisFile:
$!";
   run_cmd ("chmod 777 $ThisFile");
   write_header ($ThisFile);
   printf OUTFILE ("\n* Define receiver channel for this qmgr\n");
   printf OUTFILE ("DEF CHL($ReposQmgr.$ThisQmgr)        -\n");
   printf OUTFILE ("       CHLTYPE(RCVR)                 -\n");
   printf OUTFILE ("       DESCR('Receiver channel')            \n");
   close (OUTFILE) or die "$Err: cannot close $ThisFile: $!";
   # Now do the file for the repos queue manager
   $ThatFile = $ReposQmgr   . ".BridgeFromGGC.mqsc";
   open (OUTFILE, "> $ThatFile") or die "$Err: cannot open $ThatFile:
$!";
   run_cmd ("chmod 777 $ThatFile");
   write_header ($ThatFile);
   printf OUTFILE ("\n* Define sender channel to local qmgr\n");
   printf OUTFILE ("DEF CHL($ReposQmgr.$ThisQmgr)               -\n");
   printf OUTFILE ("       CHLTYPE(SDR)                         -\n");
   printf OUTFILE ("       CONNAME('$ThisIPAddr($ThisPort)')    -\n");
   printf OUTFILE ("       XMITQ(TO.$ThisQmgr)                  -\n");
   printf OUTFILE ("       DESCR('Sender channel')            \n");

   printf OUTFILE ("\n* Define xmitq to local qmgr\n");
   printf OUTFILE ("DEF QL(TO.$ThisQmgr)               -\n");
   printf OUTFILE ("    TRIGGER                        -\n");
   printf OUTFILE ("    TRIGTYPE(FIRST)                -\n");
   printf OUTFILE ("    TRIGDATA($ReposQmgr.$ThisQmgr) -\n");
   printf OUTFILE ("    INITQ(SYSTEM.CHANNEL.INITQ)    -\n");
   printf OUTFILE ("    DEFPSIST(YES)                  -\n");
   printf OUTFILE ("    USAGE(XMITQ)                    \n");
   printf OUTFILE ("\n* Define queue manager alias\n");
   printf OUTFILE ("DEF QR($ThisQmgr)              -\n");
   printf OUTFILE ("    DEFPSIST(YES)              -\n");
```

```
   printf OUTFILE ("    RNAME(' ')                  -\n");
   printf OUTFILE ("    RQMNAME($ThisQmgr)          -\n");
   printf OUTFILE ("    XMITQ(TO.$ThisQmgr)         -\n");
   printf OUTFILE ("    CLUSTER($Cluster)            \n");
   close (OUTFILE) or die "$Err: cannot close $ThatFile: $!";
   LogFileDivider ();
   printf LOGFILE ("$Info: $ThisFile and $ThatFile created\n");
   AnyKeyToContinue ("$ThisFile and $ThatFile created");
}
sub AddIntoAAC
{
   run_cmd("clear");
   PromptFor ("Cluster");
   print "Specify details for this queue manager         ..\n";
   PromptFor ("ThisQmgr");
   PromptFor ("ThisIPAddr");
   PromptFor ("ThisPort");
   PromptFor ("Repository");
   print "Specify details for another queue manager holding the
repository         \n";
   PromptFor ("ReposQmgr");
   PromptFor ("ReposIPAddr");
   PromptFor ("ReposPort");
   print "\n";
   DisplaySchematic ("AddIntoAAC");
   if (!(YToContinue ("Is this configuration OK"))) {
      return;
   }
   # Do the file
   $ThisFile = $ThisQmgr    . ".AddIntoAAC.mqsc";
open (OUTFILE, "> $ThisFile") or die "$Err: cannot open $ThisFile: $!";
   run_cmd ("chmod 777 $ThisFile");
   write_header ($ThisFile);
   LogFileDivider ();
   printf LOGFILE ("$Info: If there any existing sender/receiver
channels\n");
printf LOGFILE ("$Info: between $ThisQmgr and $ReposQmgr, then they
will\n");
printf LOGFILE ("$Info: need to be stopped and deleted.            \n");
printf OUTFILE ("\n* Define cluster sender channel to the repos
qmgr\n");
printf OUTFILE ("DEF CHL($Cluster.$ReposQmgr)                    -\n");
printf OUTFILE ("    CHLTYPE(CLUSSDR)                            -\n");
printf OUTFILE ("    CONNAME('$ReposIPAddr($ReposPort)')         -\n");
printf OUTFILE ("    CLUSTER($Cluster)                           -\n");
printf OUTFILE ("    DESCR('Cluster Sender channel to the repos')  \n");
printf OUTFILE ("\n* Define cluster receiver channel\n");
printf OUTFILE ("DEF CHL($Cluster.$ThisQmgr)                     -\n");
printf OUTFILE ("    CHLTYPE(CLUSRCVR)                           -\n");
```

```
printf OUTFILE ("    CONNAME('$ThisIPAddr($ThisPort)')            -\n");
printf OUTFILE ("    CLUSTER($Cluster)                            -\n");
printf OUTFILE ("    DESCR('Cluster Receiver channel')             \n");
if (uc($Repository) eq "Y") {
      LogFileDivider ();
printf LOGFILE ("$Info: The script '$ThisFile' alters qmgr
'$ThisQmgr'\n");
printf LOGFILE ("$Info: to make it a repos qmgr for the cluster
'$Cluster'.\n");
printf LOGFILE ("$Info: However, queue manager $This Qmgr may
already\n");
printf LOGFILE ("$Info: have a namelist set up, in which case this will
need\n");
printf LOGFILE ("$Info: to be manually extended to include the cluster
name '$Cluster'.\n");
printf LOGFILE ("$Info: In this case, edit the script to remove the
ALTER QMGR\n");
printf LOGFILE ("$Info: statement.\n");
printf OUTFILE ("\n* Make this a full repository\n");
printf OUTFILE ("ALTER QMGR REPOS($Cluster) \n");
      }
close (OUTFILE) or die "$Err: cannot close $ThisFile: $!";
LogFileDivider ();
printf LOGFILE ("$Info: $ThisFile created\n");
AnyKeyToContinue ("$ThisFile created");
}
sub LogFileDivider
{
printf LOGFILE ("=====Entry created %s =======>\n", scalar localtime);
}
sub PromptFor
{
   local ($what) = @_;
   if ($what eq "Cluster")      {
      print "Enter name of cluster :";
      $Cluster = <STDIN>;
      chomp ($Cluster);
   elsif ($what eq "ThisQmgr")   {
      print "Enter name of queue manager :";
      $ThisQmgr = <STDIN>;
      chomp ($ThisQmgr);
                        }
   elsif ($what eq "ThisIPAddr")   {
      print "Enter IP address :";
      $ThisIPAddr = <STDIN>;
      chomp ($ThisIPAddr);
                        }
   elsif ($what eq "ThisPort")   {
      print "Enter port :";
```

```
      $ThisPort = <STDIN>;
      chomp ($ThisPort);
                        }
   elsif ($what eq "Repository")   {
      print "Does it hold a full repository? (Y/N) :";
      $Repository = <STDIN>;
      chomp ($Repository);
                        }
   elsif ($what eq "ReposQmgr")   {
      print "Enter name of queue manager :";
      $ReposQmgr = <STDIN>;
      chomp ($ReposQmgr);
                        }
   elsif ($what eq "ReposIPAddr")   {
      print "Enter IP address :";
      $ReposIPAddr = <STDIN>;
      chomp ($ReposIPAddr);
                        }
   elsif ($what eq "ReposPort")   {
      print "Enter port :";
      $ReposPort = <STDIN>;
      chomp ($ReposPort);
                        }
   else {
      AnyKeyToContinue ("$Err: invalid call >>$what");
      }
}
sub DisplaySchematic
{
   local ($what) = @_;

   run_cmd ("clear");
   printf SCHEMATIC "\n";
   if ($what eq "AddIntoGGC")      {
      printf SCHEMATIC
":::::::::::::::::::::::::::::::::::::::::::::::::::::::::::\n";
printf SCHEMATIC ":  Global Gateway Cluster = %-8s                :\n",
$Cluster;
printf SCHEMATIC ":                                                :\n";
printf SCHEMATIC ":  +-----------------+     +-----------------+   :\n";
printf SCHEMATIC ":   /                 /|    /                 /| :\n";
printf SCHEMATIC ":  +-----------------+ +   +-----------------+ + :\n";
      my $IsItRepos = "       ";
      if (uc($Repository) eq "Y") {
         $IsItRepos = "(Repos)";
         }
printf SCHEMATIC ":  |%-8s %-7s |/=====>|%-8s (Repos) |/| :\n",
$ThisQmgr, $IsItRepos, $ReposQmgr;
printf SCHEMATIC ":  +-----------------+ |   +-----------------+ | :\n";
```

```
printf SCHEMATIC ":  |%16s | |     |%16s | + :\n", $ThisIPAddr,
$ReposIPAddr;
printf SCHEMATIC ":  |         (%5s) |<==+   |         (%5s) |/   :\n",
$ThisPort, $ReposPort;
printf SCHEMATIC ":  +---------------+-+  |  +----------------+   :\n";
printf SCHEMATIC ":                  |     |                       :\n";
printf SCHEMATIC ":                  +=====+                       :\n";
printf SCHEMATIC
":::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::\n";
      }
   elsif ($what eq "BridgeIntoGGC")   {
printf SCHEMATIC "
::::::::GGC::::::::::::::::::\n";
printf SCHEMATIC "                           :                          \n";
printf SCHEMATIC "    +-----------------+ :    +-----------------+  \n";
printf SCHEMATIC "   /                 /| :   /                 /|  \n";
printf SCHEMATIC "  +-----------------+ + :  +-----------------+ +  \n";
printf SCHEMATIC "  | %-8s        |/=====>| %-8s        |/|\n",
$ThisQmgr, $ReposQmgr;
printf SCHEMATIC "  +-----------------+ | :  +-----------------+ |  \n";
printf SCHEMATIC "  |                 | + :   |%16s | +\n", $ReposIPAddr;
printf SCHEMATIC "  |                 |/  :   |        (%5s)  |/\n",
$ReposPort;
printf SCHEMATIC "  +-----------------+  :   +-----------------+    \n";
printf SCHEMATIC "                         :\n";
printf SCHEMATIC "                         :\n";
printf SCHEMATIC "                         :\n";
printf SCHEMATIC "                         ::::::::::::m
printf SCHEMATIC "                         :                          \n";
printf SCHEMATIC "    +-----------------+ :    +-----------------+  \n";
printf SCHEMATIC "   /                 /| :   /                 /|  \n";
printf SCHEMATIC "  +-----------------+ + :  +-----------------+ +  \n";
printf SCHEMATIC "  | %-8s        |/=====>| %-8s       |/|\n",
$ThisQmgr, $ReposQmgr;
printf SCHEMATIC "  +-----------------+ | :  +-----------------+ |  \n";
printf SCHEMATIC "  |                 | + :   |%16s | +\n",
$ReposIPAddr;
printf SCHEMATIC "  |                 |/  :   |        (%5s)  |/\n",
$ReposPort;
printf SCHEMATIC "  +-----------------+   :   +-----------------+   \n";
printf SCHEMATIC "                         :\n";
printf SCHEMATIC "                         :\n";
printf SCHEMATIC "                         :\n";
printf SCHEMATIC "
:::::::::::::::::::::::::::::\n";
         }
     elsif ($what eq "BridgeFromGGC") {
printf SCHEMATIC ":::::::::::::::::::::::::\n";
printf SCHEMATIC " GGC = %-8s         :\n", $Cluster;
```

```
printf SCHEMATIC "                          :\n";
printf SCHEMATIC "    +-----------------+ :     +-----------------+\n";
printf SCHEMATIC "   /                 /| :    /                 /|\n";
printf SCHEMATIC "  +-----------------+ + :   +-----------------+ +\n";
printf SCHEMATIC "  | %-8s        |/=====>| %-8s        |/|\n",
$ReposQmgr, $ThisQmgr;
printf SCHEMATIC "  +-----------------+ | :   +-----------------+ |\n";
printf SCHEMATIC "  |                 | + :   |%16s | +\n", $ThisIPAddr;
printf SCHEMATIC "  |                 |/  :   |        (%5s)  |/\n",
$ThisPort;
printf SCHEMATIC "  +-----------------+   :   +-----------------+\n";
printf SCHEMATIC "                          :\n";
printf SCHEMATIC "                          :\n";
printf SCHEMATIC "::::::::::::::::::::::::::\n";
         }
     elsif ($what eq "AddIntoAAC")    {
printf SCHEMATIC":::::::::::::::::::::::::::::::::::::::::::::::::::\n";
printf SCHEMATIC ":  Application Availability Cluster = %-8s       :\n",
$Cluster;
printf SCHEMATIC ":                                                 :\n";
printf SCHEMATIC ":   +-----------------+      +-----------------+ :\n";
printf SCHEMATIC ":  /                 /|     /                 /| :\n";
printf SCHEMATIC ": +-----------------+ +    +-----------------+ + :\n";
         my $IsItRepos = "       ";
         if (uc($Repository) eq "Y") {
              $IsItRepos = "(Repos)";
              }
printf SCHEMATIC ": |%-8s %-7s |/=====>|%-8s (Repos) |/| :\n",
$ThisQmgr, $IsItRepos, $ReposQmgr;
printf SCHEMATIC ": +-----------------+ |    +-----------------+ | :\n";
printf SCHEMATIC ": |%16s | |     |%16s | + :\n", $ThisIPAddr,
$ReposIPAddr;
printf SCHEMATIC ": |         (%5s) |<==+   |         (%5s) |/     :\n",
$ThisPort, $ReposPort;
printf SCHEMATIC ": +---------------+-+   |   +-----------------+  :\n";
printf SCHEMATIC ":                 |     |                        :\n";
printf SCHEMATIC ":                 +=====+                        :\n";
printf SCHEMATIC "::::::::::::::::::::::::::::::::::::::::::::::::::\n";
         }
     else {
          AnyKeyToContinue ("$Err: invalid call >>$what");
          }
     # N.B. It is very important that each schematic is 15 lines long!!!
     printf SCHEMATIC "\n";
     run_cmd("tail -15 $SchematicFile");
}
############################################################
sub YToContinue
{
```

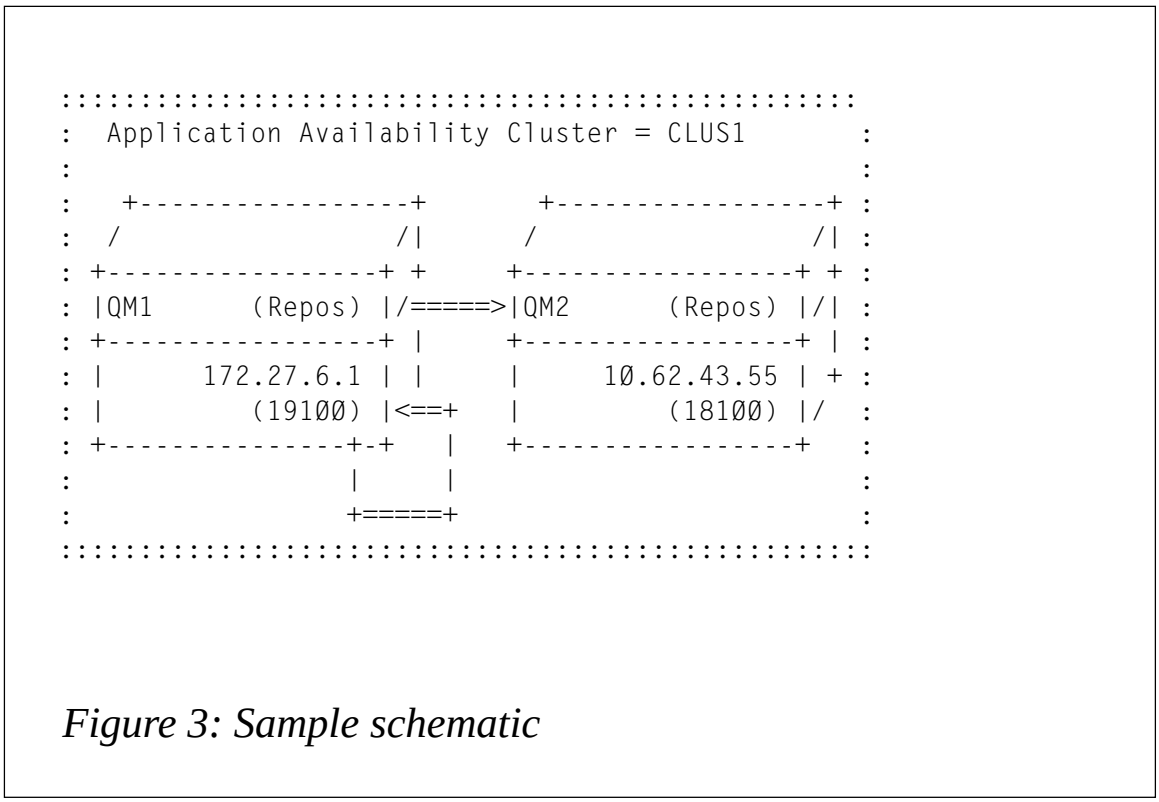


*Figure 3: Sample schematic*

```
     local ($prompt) = @_;
     my $key = "";
     print $prompt . ", enter 'Y' to continue, any other key to quit: ";
     $key = uc(<STDIN>);
     chomp ($key);
     if ($key eq "Y") {
          return $True;
          }
     else {
          return $False;
          }
}
```

Figure 3 shows a sample schematic generated by the Perl script.

*Nick Breeds*
*IT Consultant*
*Zurich Financial Services (UK)*

# MQ news

IBM has recently released a number of WebSphere MQ Family SupportPacs. These are detailed below.

*WD02: MQSeries Workflow – Considerations for production rollout.* This SupportPac describes production architectures for MQSeries Workflow. It also describes various tasks that have to be performed to keep an MQSeries Workflow system in a healthy state. IBM says that it should be read by system administrators who are ready to deploy a production MQSeries Workflow implementation as well as system architects who are designing a new MQSeries Workflow system.

*MA0J: MQSeries – Put message utility.* IBM says this SupportPac provides a simple and convenient method for MQSeries V5.1 and V5.2 users to put messages to a queue. Using the graphical user interface provided, users can connect locally or remotely to an MQSeries server, create message data from a file or from manually entered text, modify the message descriptor fields, and control the context information associated with a message. The installation instructions have been updated to address MQSeries classes for Java Version 5.2.

The following clients have all been updated to incorporate CSD 3.

*MACI: MQSeries Client for Linux for Intel – V5.2*

*MACJ: MQSeries Client for Windows 95, 98 and Me – V5.2*
*MACK: MQSeries Client for Windows NT and Windows 2000 – V5.2*
*MACL: MQSeries Client for AIX – V5.2*
*MACM: MQSeries Client for HP-UX V10 – V5.2*
*MACN: MQSeries Client for HP-UX V11 – V5.2*
*MACO: MQSeries Client for Sun Solaris – V5.2*
*MACP: MQSeries Client for Windows NT and Windows 2000 – V5.2.1*

*WD01: Business Process Modelling with MQSeries Workflow*. This SupportPac provides an introduction to the creation of business process modelling using IBM MQSeries Workflow. It gives a step-by-step overview of how to implement simple processes using MQSeries Workflow and also covers more advanced process models. IBM says this SupportPac is intended for business analysts and process modellers with a basic knowledge of MQSeries Workflow. Two new examples have been added and many descriptions have been modified.

IBM MQSeries SupportPacs may be accessed at *http://www.ibm.com/software/mqseries/txppacs*.

*For further information contact:*
Your local IBM representative.

* * *

xephon