23

MQ update

May 2001

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/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

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.

Setting up DB2 for MQSI V2 on AIX

INTRODUCTION

As you may have realized, the MQSI installation manual has very little information on database configuration, seeming to assume that middleware practitioners are also database experts, or that a friendly database administrator will always be around when we install the product. This article aims to complement the installation manual, providing more information on how to set up and configure DB2 databases on AIX for use with MQSI V2.

DATABASES ASSOCIATED WITH MQSI V2

MQSI for AIX consists of two components:

- The Configuration Manager and Control Center, which runs on Windows NT, exploiting the GUI.
- The Broker, which runs on AIX.

For each component you need to create a corresponding database.

In the Windows NT part there is one database for message repository and another for configuration management.

In the AIX environment you have to create a broker database, and if you use NEON Nodes you have to create a database for the NEON Formatter and Rules.

THE CHALLENGE

Creating a DB2 database in the Windows NT environment is not ‘rocket science’ – with the nice GUI front-end tools on Windows a middleware integrator should have no problem, even if knowledge of DB2 commands is limited. The challenge lies with DB2 in the AIX environment.

DB2 on AIX is very much like MQ in the AIX environment; there is no GUI front-end to administer the DB2 systems. You have to use DB2 commands, or, to make life a little easier, you can use the DB2 Control

Center on Windows NT to administer remote DB2 systems, in much the same way that you would use MQ Explorer on NT to administer remote MQ systems.

The best place to start gaining DB2 knowledge is from the *Quick Beginning Manual for Unix*. A friendly DBA will definitely make your life a lot easier. Documented below are the requisite steps for setting up DB2 databases in the AIX environment in case a manual is not available.

CREATING A DB2 BROKER DATABASE ON AIX

MQSI does not put high demands on the database when it is used to manage a broker, so you can create the broker database using any existing database instance on the machine. It is recommended that you create a new database instance rather than use an existing one.

The following is taken from the *MQSI for AIX Installation Manual GC34-5841-01*, chapter four, page 51:

1 Log on as root, using the following commands:

```
/usr/lpp/db2_6_1/instance/db2icrt -u <username><username>
```

2 Logon as *<username>*, using the following command:

```
.~/sqllib/db2profile
db2 start database manager
db2 create database MQSIBKDB
db2 connect to MQSIBKDB
db2 bind ~/sqllib/bnd/@db2cli.lst grant public CLIPKG 5
```

Note that **.~/sqllib/db2profile** should be in the profile of all users.

3 In order for MQSeries Integrator brokers to use the database you must update the ODBC configuration file (*var/mqsi/odbc/.odbc.ini*) to contain definitions for the database. To do this, edit the file and add the lines listed below.

– at the top of the file add a definition for the database name:

```
MQSIBKDB=IBM DB2 ODBC Drive
```

– at the end of the file add the following lines:

```
Driver=<INSTHOME>/sqllib/lib/db2.o
Description=Broker Database
Database=MQSIBKDB
```

The path identified by the driver definition will be specific to your installation so you must replace the <*INSTHOME*> with the path to your DB2 instance directory.

Following the above steps, a DB2 instance named *mqsiuid* was created under the username *mqsiuid*; the new database *MQSIBKDB* was created in the home directory of the *mqsiuid*. An MQSI broker can then be created based on this broker database, and receive work (message flows) deployed from the MQSI configuration manager from the NT side.

SETTING UP DB2 ON AIX FOR REMOTE ADMINISTRATION

In order for the DB2 database on AIX to be administered by a remote client, such as the Windows NT workstation or server where the MQSI Configuration Manager runs, extra steps are need to make the newly created *mqsiuid* DB2 instance accessible to the NT client DB2 Control Center.

1 On AIX, edit the *etc/services* file – there should be two lines like these for the first *db2inst1* instance:

```
db2cdb2inst1 50000/tcp # Connection port for DB2 instance db2inst1
db2idb2inst1 50001/tcp # Interrupt  port for DB2 instance db2inst1
```

Create the next two entries like those below, using spare ports, eg:

```
db2cdb2inst2 50002/tcp # Connection port for DB2 instance mqsiuid
db2idb2inst2 50003/tcp # Interrupt  port for DB2 instance mqsiuid
```

2 Update the database management configure file for the SVCENAME for the second DB2 instance with the following command: (this variable was blank)

```
db2 update dbm cfg using SVCENAME db2cdb2inst2
```

You should see the following when you do a **DB2 get dbm cfg | grep SVCENAME**

```
TCP/IP Service name              (SVCENAME) = db2cdb2inst2
```

3 Do a **db2get –all** to display the DB2 variables set for this instance:

```
$ db2set -all
[i] DB2COMM=tcpip
```

```
[i] DB2AUTOSTART=TRUE
[g] DB2SYSTEM=unixd01
[g] DB2ADMINSERVER=db2as
```

4 If you do not see the variable set for **DB2COMM** and **DB2AUTOSTART**, you need to add this by issuing the following db2 commands:

```
db2set -i mqsiuid DB2COMM=tcpip
```

and

```
db2set -i mqsiuid DB2AUTOSTART=TRUE
```

where *mqsiuid* is the instance name.

5 Stop the instance *mqsiuid* by issuing **db2stop**.

6 Stop the *db2as* by issuing **db2admin stop**.

7 Start the *db2as* by issuing **db2admin start**.

8 Start the db2 instance *mqsiuid* by issuing **db2start**.

On the Windows NT side, start the DB2 Control Center, right-click on the system and choose ‘add’, type in the remote host name and retrieve, and then click ‘OK’. The remote AIX DB2 system will be added to your Control Center for administration.

Expand the AIX DB2 system, connect as *mqsiuid* and its corresponding password when prompted, and expand the DB2 instance – you should see the DB2 instance *mqsiuid* there, ready for remote administration.

CREATING A DB2 NEON DATABASE ON AIX

In order to use NEON support with MQSI V2, a NEON database must be created first. This database should be created on the same DB2 instance with the broker database *MQSIBKDB*.

Creating *MQSINEON* database

There are three steps required to configure the DB2 database for NEON Formatter and Rules. These are:

- Create a new database to contain the rules and formats.
- Configure a client connection to the new database.

- Create tablespaces within the database for the MQSeries Integrator tables.

1 To create a database, log on to the AIX box as *mqsiuid*, the DB2 database instance user, and run the **db2** command. At the command line, issue the following command:

```
db2=>create database MQSINEON
```

2 To connect to a DB2 database with an ODBC driver, use the Client Configuration Assistant found in: *Start->Programs->DB2 for Windows NT -> Client Configuration Assistant* and carry out the following steps:

– click the ‘Add Database’ button.

– a window called ‘Add Database SmartGuide’ appears. Select the ‘Manually configure a connection...’ option and click ‘Next’.

– connect to the AIX server.

– select TCPIP for the protocol and click ‘Next’.

– type the name of the host machine where the database instance *mqsiuid* is running and the port on which the instance is listening; in our case it is 50002.

– type the name of the database *MQSINEON* and click ‘Next’.

– change the alias if you want a different name from that of the database and click ‘Next’.

– the last screen should have a ‘tick’(✓) in the register database as an ODBC source, and the system data source should be selected.

– click ‘Done’ to create the service and ODBC driver. It asks whether to test the connection.

– click ‘Test Connection’.

– enter the *mqsiuid* as user and its associated password. You are then successfully connected.

3 To create the tablespace for the NEON Formatter and Rules, use

the DB2 command line processor found in *Start->Programs->DB2 for Windows NT-> DB2 command line processor* and issue the following commands:

```
db2=>connect to MQSINEON user mqsiuid using mqsipw
db2=>create tablespace FORMATTER_DATA managed by system using
     ('FORMATTER_DATA')
db2=>create tablespace FORMATTER_INDEX managed by system using
     ('FORMATTER_INDEX')
db2=>create tablespace RULES_DATA managed by system using
     (RULES_DATA ')
db2=>create tablespace RULES _INDEX managed by system using
     ('RULES _INDEX')
```

Use the quit command to exit the DB2 command line program:

```
db2=>quit
```

Install the database schema

1 At the command line prompt, change to the *install.sql* directory, for example:

```
cd c:\Program Files \IBM MQSeries Integrator 2..1 \install.sql
```

2 To build the MQSeries Integrator schema, type one of the following:

– open a Windows NT command prompt window and type the following: **db2cmd**.

– the **db2cmd** command opens another command prompt window. Switch to that window and run the following command:

```
inst_db.cmd mqsiuid mqsipw MQSINEON
```

3 As the script runs, answer the prompts and look for errors.

4 When the script completes the instantiation, a verification message appears.

5 For installation details look at the *inst_db.log* file located in the *c:\temp* directory.

When the above is done, you can use the NEON Formatter and Rules from NEON support found in *start->programs->IBM MQSeries Integrator 2.0.1->Neon Support->NEON Formatter* or *NEON Rules.*

Type in user-id *mqsiuid*; password *mqsipw*; DBMS ODBC – DB2; Driver *MQSINEON*; leave the qualifier blank and click 'OK'.

You should then be connected to the NEON Formatter or Rules GUI.

CONCLUSION

There are quite a lot of similarities between MQSeries and the DB2 products; the similarity is particularly obvious on the OS/390 platform. It is always beneficial for middleware professionals to master more DB2 skills, including system administration and management skills. The learning curve isn't as steep as one might think. Hopefully, this article will serve as a useful starting point.

Alex Au
I/T Architect, IBM (USA)

Using variable queue names in MQSI V2

One of the restrictions of MQSI V1 was its inability to vary the queue name when writing a message. When a message is processed by a V1 rules engine it must be written to a specific queue. That queue is hard-coded in the subscription of the rules definition. Various ways of achieving a variable queue name have been developed by customers so that the message itself can carry an indication of the queue to which to write the message. However, this involved either contrived formats and re-processing of messages through a rules engine, or writing post-processors to determine the ultimate destination queue.

Superficially, message flows within MQSI V2 appear to carry the same restriction. However, it is possible to manipulate the queue name (and the queue manager name if necessary) by using the destination list and by modifying advanced properties within a message flow.

The destination list is an array of queue name/queue manager name pairs, and, like the message properties, may be manipulated for use by subsequent nodes, particularly the MQOutput node.

By altering nodes within the message flow in the following way, the queue name can be set to any desired value, based, for example, on the content of the application data within the message body.

THE COMPUTE NODE

This is where the destination list must be set using appropriate ESQL statements, eg:

```
SET
OutputDestinationList.Destination.MQDestinationList.DestinationData[1].queueName
     = 'QUEUE.NAME.1';
SET
OutputDestinationList.Destination.MQDestinationList.DestinationData[1].queueManagerName
     = 'QUEUE.MANAGER.1';
SET
OutputDestinationList.Destination.MQDestinationList.DestinationData[2].queueName
     = 'QUEUE.NAME.2';
SET
OutputDestinationList.Destination.MQDestinationList.DestinationData[2].queueManagerName
     = 'QUEUE.MANAGER.2';
```

The advanced properties compute node setting must be changed to 'Destination and Message' so that the destination list will be transmitted to subsequent processing nodes along with other message data and properties.

THE MQOUPUT NODE

The advanced properties destination node must be changed from 'Queue Name' to 'Destination List'.

In the example above, the values are supplied as literals, however, in a real message flow these are likely to be values derived from data within the message body, either directly or as the result of data concatenation.

Compute nodes may also access the destination list set by previous compute nodes. In this case, the data must be referred to as *InputDestinationList….* etc.

Whilst the DestinationList is an array of queue names, it is possible to utilize just the first subscript of the array to denote a single queue.

Ken Marshall
MQSeries Consultant, MQSolutions (UK) *© MQSolutions*

System management for MQSeries OS/390 V5.2

INTRODUCTION

This article discusses the new facilities and features of MQSeries for OS/390 V5.2. It will not focus on the function or usage of these features, but on their impact from a systems management perspective. The following new features will be discussed:

- Shared queues.
- Command scope.
- Global repository.
- Shared channels.
- Enhanced accounting and statistics.
- Queue and connection status.
- Dead letter queue handler.

There are a number of additional features in MQSeries for OS/390 5.2 that do not specifically impact management and these will not be covered.

One item that should be addressed relates to the features of MQSeries for OS/390 5.2 and MQSeries for Distributed Systems 5.2. While the release numbers were synchronized to reflect the capabilities of each, the two releases are separate code bases and are not completely equivalent. The material covered in this article is unique to MQSeries for OS/390.

SHARED QUEUES

The key benefit in MQSeries for OS/390 5.2 is shared queues. The need to share messages across the sysplex has been a long-term requirement for MQSeries. For example, when used with CICS Transaction Server in a CICSplex, workload balancing could not be used because of the affinity that existed between the CICS region and the queues containing messages.

In this first release of shared queue support IBM has provided a basic level of support. There are restrictions on messages that can be put in shared queues: first, the messages must be non-persistent, and second, the maximum message size is 63K. This size limitation is the result of the maximum supported Coupling Facility size minus some MQSeries overhead. The Coupling Facility is the key element of shared queues. The queue definition and the messages are stored in the Coupling Facility. In order to use this support, the minimum level for the operating system is OS/390 V2.9. The Coupling Facility must be at level nine.

Figure 1 shows the basic concept of shared queueing, with four queue managers across three images in the sysplex, sharing a coupling facility. A shared queue called SQ1 has been created and three of the queue managers are sharing this queue, some putting and some getting messages. Each of the queue managers has a number of local queues that are not shared.

The set of queue managers that can access the same shared queues is called a Queue Sharing Group (QSG). Figure 1 shows that we can create a queue sharing group, QSGP, consisting of the three queue

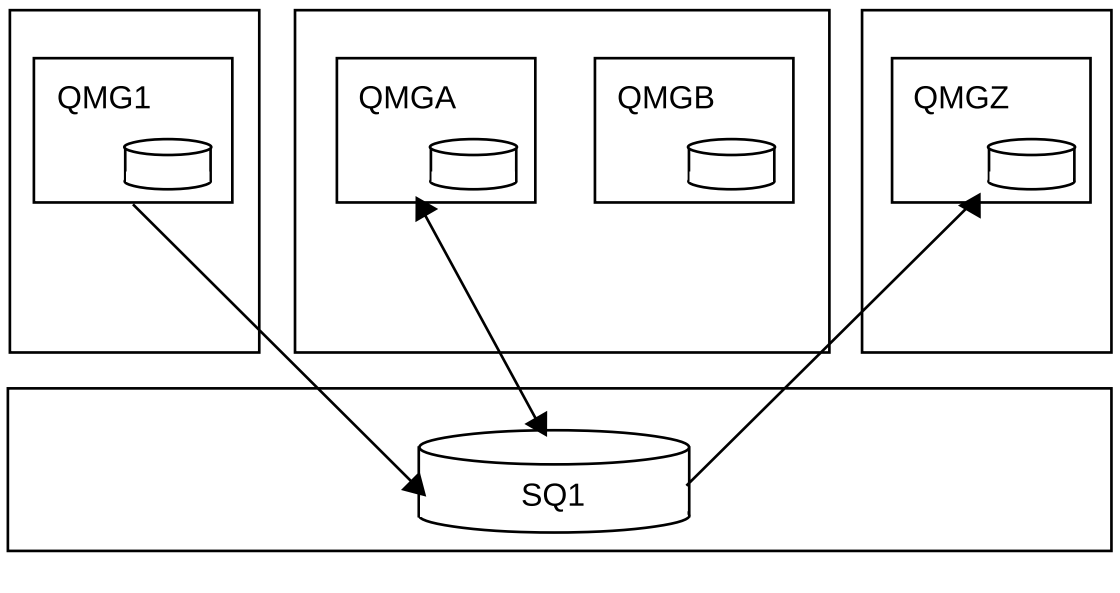

Figure 1: Basic concept of shared queueing

managers, *QMG1*, *QMGA*, and *QMGZ*. The *QMGB* queue manager is not participating in the queue sharing group.

Additionally, as mentioned in the restrictions above, the queue is created as part of a CF structure, and, in this case, *SQ1* could be placed in a structure called *QSGPSTR1*. Note that the CF structure name is preceded by the queue sharing group name.

Considerations

Coupling Facility usage

The Coupling Facility is a shared resource across the sysplex. While continually increasing in capacity and performance, usage of this should be well understood before implementing a shared queueing environment. A number of products provide detailed analysis of the Coupling Facility.

Because of restrictions within the Coupling Facility, the following maximum limitations are imposed:

- Eight million messages.
- 512 shared queues per CF structure.
- 512 CF structures per sysplex.
- 63 CF structures per queue manager.

Because of other usage of the Coupling Facility, the maximum number of objects needs to be assessed and limitations may be further reduced in real-life operation. Before being used by MQSeries, the CF structures must be defined to the Coupling Facility policy. These structures should be carefully managed to prevent problems such as excessive number of structures or orphaned structures. Deleting a structure once it’s been created means deleting all queues in the structure, restarting the queue managers, and updating the Coupling Facility policy. The structures created use a naming convention where the queue sharing group prefixes the referenced structure name.

A special structure, *qsgCSQ_ADMIN*, is created to hold MQSeries objects. User objects cannot use this structure. Note that the name contains an underscore (_), which is not supported in user-defined structures.

MQSeries commands

Since there is really only one instance of the queue, actions taken against the queue have a global affect. That is, if you alter the maximum depth of the queue all queue managers are aware of the change. You cannot restrict the queue from one of the queue managers; for example, if a problem was occurring on *QMGZ* and it was disabled from its queue manager, the queue would be disabled from all queue managers in the QSG.

Performance events

MQSeries generates performance events in several cases. For queue full, queue depth high, and queue depth low, the condition will be observed by one queue manager, but it may not be the queue manager that contributed to the condition. That is, in the example above, if queue manager *QMG1* was putting ten messages a second to *SQ1* and *QMGA* was putting one message a minute, it is still possible that *QMGA* could put the message that caused the event to occur. In order to address it, MQSeries will send multiple events. The first event will be raised at the observing queue manager – *QMGA* in this example. The remaining queue managers will then raise events that correlate back to the original event. These will include the activity statistics for the queue for each of the queue managers. Even queue managers that did not use the queue will raise events.

Service interval events are not raised for shared queues.

Statistics

Some of the shared queue information is kept in the local queue manager. This includes the statistics for the queue, including input and output counts and queue depth. Input and output count will only represent the counts from a specific queue manager, not all queue managers. Queue depth will show the queue depth at the last access to that queue by the queue manager. Note that this can affect the statistics reported in the service level event described above.

Triggering

Triggering is handled somewhat differently with shared queues. The triggered queue or the initiation queue, or both, can be shared queues.

When using triggering types of ‘first’ and ‘depth’, a CF facility is used for monitoring. If triggering is required, the CF will notify the queue managers, which will in turn generate the trigger message.

For triggering of type ‘every’, the queue monitor that is putting the message will check the trigger conditions and generate the trigger message.

One difference with shared queues is that uncommitted messages are not included in the trigger count. This addresses a problem in non-shared queues when uncommitted messages are counted, potentially causing a trigger event to be raised – but no messages will be available to the triggered application.

If the trigger monitors are monitoring the same queue it is possible that multiple trigger messages will be produced, initiating a race within the QSG to service the message.

Indexed queues

Another difference in behaviour has to do with indexed queues. Queues can be indexed by message id, indexed by correlation id, or not indexed. In non-shared queues, if an application does a ‘get’ by message id or by correlation id and the queue does not have the matching index, MQSeries will default to an inefficient sequential technique to retrieve the message. With shared queues, the index is built in the Coupling Facility. If the index does not match on the ‘get by’ request, an error will be returned to the application because the impact to the Coupling Facility, which is a cross-system resource, would be considered too large.

Recovery

One of the benefits of shared queueing is that, in the case of a queue manager failure, sufficient information is contained in the Coupling Facility for another queue manager to perform recovery. Partial recovery is possible even when the unit of work contains updates to non-shared queues.

Persistence

One change that should be considered is that non-persistent messages will outlive a queue manager. That is, with non-shared queues, non-persistent messages are deleted when the queue manager terminates and restarts. However, for shared queues, the messages are stored in the Coupling Facility and are independent of the queue manager that created them. Thus, they will remain even if the queue manager is restarted. In fact, the messages will remain except for a Coupling Facility failure. This has the advantage of almost being persistent, as the Coupling Facility is highly reliable. If your application can live with the small potential for failure, you may be able to use non-persistent messages instead of persistent ones. Another implication is that, if your applications rely on queue manager restart cleaning up non-persistent messages, this will no longer occur.

Duplicate queues

One small potential problem is the occurrence of two queues of the same name – one being a shared queue and the other being a non-shared queue – within the same QSG. This is not allowed. However, it is possible to create the situation by having an existing non-shared queue defined in a queue manager that is added to an existing queue sharing group that contains a shared queue of the same name. It is also possible for this to occur if a queue is defined as a shared queue when the queue manager that contains the non-shared queue is inactive. At all other times, MQSeries will return an error if duplicate names exist.

However, once the condition is recognized, you will not be able to open either queue and an error will be returned. The only option is to fix the problem by deleting one of the queues. If the queue contains messages that you don’t want to lose, the ‘move messages’ command will move them to another queue.

QUEUE SHARING GROUPS

A queue manager is joined to a queue sharing group by specifying the *QSGDATA* parameter in the *CSQ6SYSP* parameter macro:

```
    QSGDATA=(qsgname,dsgname,db2name,db2servers )
```

Where *qsgname* is the name of the queue sharing group, *dsgname* is the name of the data sharing group to be used, *db2name* is the name of the DB2 subsystem to connect to, and *db2servers* is the number of server tasks to create for activity in DB2.

Before joining the queue sharing group, the DB2 databases must be in place and the queue manager registered as a member of the queue sharing group using the queue sharing group utility, *CSQ5PQSG*.

Benefits

In addition to shared queues, there are several benefits to using a queue sharing group. These include:

- Command scope.
- Inter-group communication without channels.
- Group definitions.
- Shared channels.

Each of these will be explored in more detail in later sections.

One benefit to mention briefly here is that, when connecting (**MQCONN**), batch and TSO applications can specify the queue sharing group name rather than the queue manager name. This allows applications to connect to any queue manager available on the local system in the queue sharing group. This is not available for CICS or IMS connections, which can only specify a queue manager name.

Considerations

Association

A queue manager can belong to one, and only one, queue sharing group. A queue sharing group is a single set of queue manager. Unlike clustering – where clusters can overlap – a given queue manager can belong to multiple clusters, and each cluster can contain a different set of queue managers. Once added to a queue sharing group it is not easy to remove the queue manager from it. Changing the name of the associated queue sharing group will cause the queue manager to fail at startup.

DB2

An important consideration is that, in order to use a queue sharing group, you must also use DB2 (V5.1 or higher). While not overly complex this may require an additional level of skills that perhaps isn’t available among the MQSeries group. The queue sharing group is tied to a DB2 data sharing group (DSG). During queue manager startup, if DB2 is not active the queue manager will wait for it. RRS (Resource Recovery Services) is also required for coordination of recovery. Once active, intermittent outages of DB2 are tolerated, but if DB2 services are required the queue manager will not be able to perform any actions that require access to DB2, including starting shared channels and opening shared-queues it has not previously accessed.

Authority to access DB2, the data sharing group, and the DB2 resources is required.

Security

A new model of security is available for the queue sharing group. To simplify authority management, the queue sharing group name can be used to create resources instead of the MQSeries subsystem name. For example, using our example, *QSGP.SQ1* could be used instead of *QMG1.SQ1*, *QMGA.SQ1*, and *QMGZ.SQ1*. However, you may still want to restrict access on a queue manager basis so you can mix and match the definitions used. To eliminate some ambiguity across the queue sharing group, you can also use switch classes to customize which set of definitions to use. If you are not familiar with switch classes, they are basically options set via RACF classes under the control of the security administrator, rather than MQSeries parameters that could be set by the MQSeries administrator.

Security messages have been enhanced to improve ease-of-use of information. Because of the increased options, messages that explain the state of security are displayed at queue manager start as well as after a refresh security command.

COMMAND SCOPE

Command scope is a new feature that provides a mechanism to route commands around the queue sharing group. After connecting to any member of the queue sharing group the following options are available:

- **cmdscope(' ')** – (the default) indicates that the command is destined for the current queue manager only. Basically, this is the same as today.

- **cmdscope(qmgr)** – allows a command to be targeted to another queue manager. For example, if connected to *QMG1*, entering the command **DEFINE QL(AAAA) CMDSCOPE(QMGZ)** would route the command to *QMGZ* to create the local queue.
- **cmdscope(*)** – routes the command to all queue managers in the queue sharing group. For example, as in the above example, **DEFINE QL(AAAA) CMDSCOPE(*)** would route the command to *QMG1*, *QMGA*, and *QMGZ*, and three queues would be created.

After completion, the responses from each of the queue managers participating would be displayed back to the requestor at the originating queue manager.

Considerations

Sysplex management

When using command scope, the requests can only be routed with the queue sharing group associated with the current queue manager. It cannot be used to route work to queue managers that are outside of the queue sharing group or in another queue sharing group. As a result, the end user entering the commands must have a good working knowledge of the topology to connect to the proper queue manager to invoke the command. A queue manager in the required queue sharing group is needed on the same system as the user because MQSeries does not support a remote connect. Vendor products that provide sysplex support are typically not restricted in this fashion.

INTER-GROUP COMMUNICATION

This is a similar concept to that used in command scope and event propagation. Queue managers in a queue sharing group are able to communicate directly with one another. Given the right usage, messages can also use this communication layer. If the message is destined for one of the queue managers in the queue sharing group and it meets the requirements for shared queueing (non-persistent and no bigger than 63K), the message will be routed to the target queue manager without the use of MQSeries channels. If the message does not meet these requirements it will be placed in the appropriate transmission queue and routed via traditional MQSeries channels.

Considerations

Existing structure

If messages put by the same application vary widely in size or are very close to 63K it is possible that some messages will use the inter-group communication and others the traditional methods. MQSeries on OS/390 does not currently support any of the grouping methods supported on the distributed platforms. This could result in messages being delivered out of sequence to the retrieving application.

Unless all messages are 'shared queue-capable', which is very unlikely, you will still need to define and manage your normal transmission queues and channels.

GROUP DEFINITIONS

Group definitions extend the current object definition facilities to allow objects to be stored in a shared repository. For this repository MQSeries uses a DB2 database. This repository can be used to store all types of MQSeries objects – not just queues. A side-benefit is that SQL could be used to interrogate the definitions to create reports or for correlating definitions. Update of the data in the database is not supported except via the standard MQSeries command interfaces.

Figure 2 shows the potential location of the object instances.

When creating objects, a new keyword, **QSGDISP** (queue sharing group disposition), determines where it will be created.

- **QSGDISP(QMGR)** – the default; the object will be created only within the queue manager page sets. This is the same as is done for existing levels of MQSeries. In the example above, *QueueB, QueueC*, *QueueX*, and *ChnlZ* were defined as QMGR objects.
- **QSGDISP(GROUP)** – definition is to reside in the DB2 repository and the object is instantiated in each of the queue managers in the queue sharing group. In the example above, *QueueA*, *ProcA,* and *ChnlX* were defined as GROUP objects. For each, a copy object was created in each of the queue managers.
- **QSGDISP(SHARED)** – definition is a shared queue. In the example above, *SQ1* and *SQ2* were defined as shared queues.

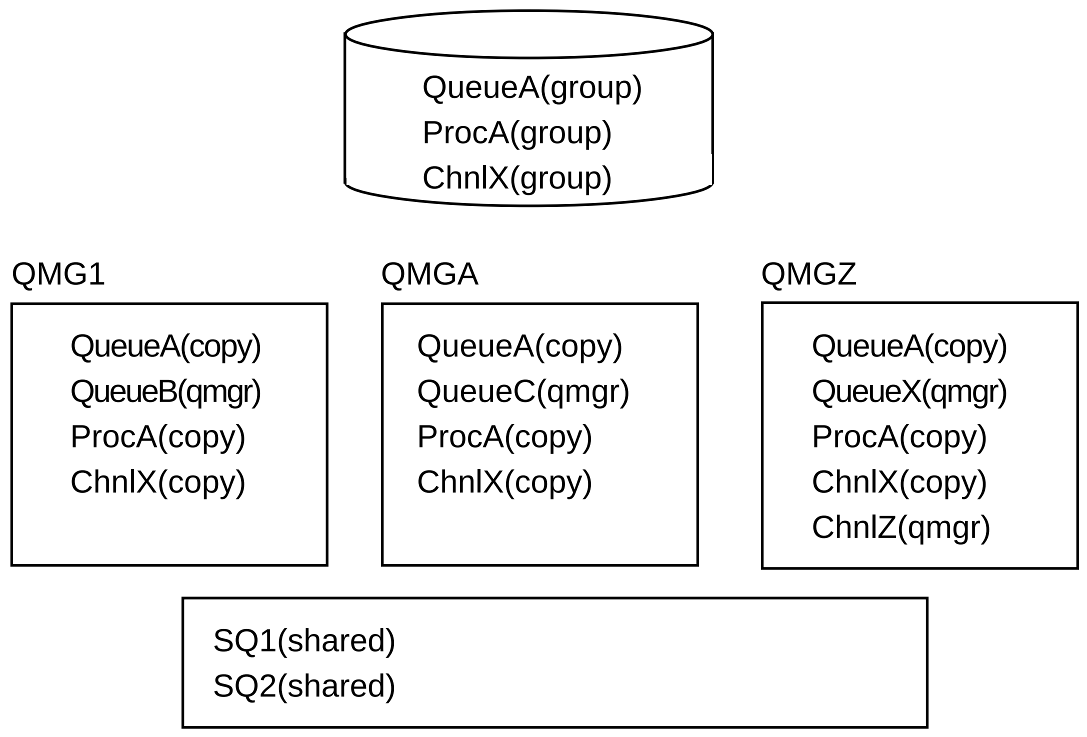

Figure 2: Potential location of the object instances

Considerations

QSGDISP

When issuing other commands, such as 'alter', additional **QSGDISP** values apply and results will vary depending on which is used.

- **QSGDISP(LIVE)** – the default, and only applicable for display commands. The response will include one entry for each object and will be a qmgr, a group (copy), or a shared definition (in the case of a shared queue). In the example above, if issued with a command scope of *QMGZ*, it would display *QueueA*, *QueueX*, *ProcA*, *ChnlX*, *ChnlZ*, *SQ1*, and *SQ2*.
- **QSGDISP(QMGR)** – applicable on all commands and the default for all commands except display. The target of the command is a definition defined to the queue manager. If the definition is another type (group or shared), the command will fail with an 'object not found' error. In the example above, if issued with a command scope of *QMGZ,* the alter command

would be valid for *QueueX*, and *ChnlZ*. If used or defaulted for *QueueA*, the command would result in an 'object not found' error.

- **QSGDISP(GROUP)** – applicable to all commands. The target of the action is the group definition. Once altered, the change will be broadcast to the queue sharing group. In the example above, regardless of command scope, the valid objects would be the *QueueA*, *ProcA*, and *ChnlX* group objects. Note that it is possible that these object definitions may not match the current copy definitions in use in each queue manager.
- **QSGDISP(COPY)** – applicable to update commands only. The target of the action is the local copy of the object created from a group definition. Altering the object will remain in effect until the queue manager is restarted or a change to the group object is broadcast. In the example above, based on command scope, the valid objects would be the *QueueA*, *ProcA*, and *ChnlX* copy objects. Note that it is possible that these object definitions may not match the global definition or the other copy definitions in use in each queue manager.
- **QSGDISP(SHARED)** – only applicable to local queue commands. The target of the command is a queue that will exist within the Coupling Facility; when defining a queue a DB2 version will also be created for tracking. In the example above, regardless of command scope, the valid objects would be *SQ1* and *SQ2*.
- **QSGDISP(PRIVATE)** – a generic request applicable to alteration commands. The target of the command is either a queue manager local definition **QSGDISP(QMGR)** or a copy of a group object **QSGDISP(COPY)**. In the example above, if issued with a command scope of *QMGZ*, it would represent *QueueA*, *QueueX*, *ProcA*, *ChnlX*, and *ChnlZ*.
- **QSGDISP(ALL)** – a generic request applicable to display commands. The command will display all objects. For group objects, the object will be shown once for the group definition and once for each queue manager copy. Queue manager objects and shared queues will be listed once. In the example above, if issued with a command scope of *QMGZ*, it would display the group

objects for *QueueA*, *ProcA*, and *ChnlX*, the copy objects for *QueueA*, *ProcA*, and *ChnlX*, the queue manager objects *QueueX* and *ChnlZ*, and the shared objects *SQ1* and *SQ2*.

Group broadcast

As noted above, commands that use **QSGDISP(GROUP)** will be broadcast within the queue sharing group. So it is not necessary to use **cmdscope** to route the requests. If a queue manager is not active at the time of the request it will get the current definition at queue manager startup. Conditions at any of the queue managers could prevent the command from completing. For example, a delete command will not complete if the local copy has messages. The request will be re-attempted at queue manager start. One unlikely but possible situation may arise here. If the user making the request did not have the authority to make the change on all of the queue managers the initial request will fail at those queue managers. However, the command will be retried at the next queue manager start under the authority of the queue manager rather than the user, and it will succeed.

Canned definitions

QSGDISP as a parameter is not supported on all releases of MQSeries. And in fact, for 5.2, it is not valid except as **QSGDISP(QMGR)** on a queue manager outside of a queue sharing group. As such, canned definitions from vendors or other sources will most likely not specify it. If you want to create these objects as group definitions in a queue sharing group, you will need to modify the canned definitions manually.

SHARED CHANNELS

Shared channels were also introduced in MQSeries for OS/390 5.2. It is somewhat misleading to refer to channels as ‘shared’ because it implies that an instance of a channel could be used by multiple queue managers, like queues, but in fact this is not the case. It is the synchronization queue associated with the channel that is shared. A shared channel is not defined explicitly as such. Its shared state is determined by its usage. As with normal channels, only a single instance of a shared channel is active at any given time. However, another queue manager could host the channel if the existing one were to fail.

Inbound shared channels

Inbound shared channels provide fault tolerance and load balancing. They are receiver channels that refer to a generic port or LU name. Sending channels on remote queue managers refer to the generic port. When the connection is made, the communications logic determines which receiving queue managers reference this generic port and then routes the connection request to one of them. If the queue manager were to fail, during retry logic the channel may be connected with another queue manager. Synchronization data stored in the shared synchronization queue is used to synchronize with the remote sender. The queue manager will also have a specific port or LU name that will be used for normal channels.

Considerations

Message routing

It is possible to define as ‘shared’ channels that are only used for queues. Alternatively, shared channels could be used for all communication. In this case, from the remote queue manager’s point of view, the connection is with a specific queue manager, but in reality, the connection may not be with that queue manager. Thus, when messages arrive in the associated queue manager, they may be targeted to one of the others in the queue sharing group. In this case, the message will be routed to that queue manager, either using inter-group routing or via normal channels. If the target is a shared queue the message will put directly to the queue.

Clustering

When used with clustering, with or without shared channels, shared queues take on an interesting perspective. From the cluster perspective, each member of the queue sharing group will have an instance of that queue, so remote systems will attempt to route work to all members of the queue sharing group. For example, if the remote system is doing round-robin to members of the queue sharing group, as each puts its message it will place it in the shared queue. Since the queue is shared, the message may be processed by applications on any of the queue managers, thereby increasing the value of workload balancing. If the channel is shared, it is interesting to note that the remote queue manager may intend the message to be processed by one queue manager, but in fact it might be processed by a different one.

Channel disposition

A new keyword is available for channel action commands, **CHLDISP** (channel disposition), and while it may appear to be the same, it has nothing to do with **QSGDISP**. Channel disposition is used to indicate the scope of channel requests. 'Private' indicates the request is for normal MQSeries channels and 'shared' indicates it is for shared channels.

Command Scope (**CMDSCOPE**) can be used with these commands to direct a command to a given queue manager in the queue sharing group or to broadcast it to all queue managers.

Outbound shared channels

Outbound shared channels are channels that refer to a transmission queue that is a shared queue. At any given time, the channel will be active within one queue manager in the queue sharing group. If that queue manager were to fail, another queue manager could take over the channel. Synchronization data stored in the shared synchronization queue is used to synchronize with the remote receiver.

Considerations

Start channel

One of the major differences with outbound shared channels is that only a single instance is active in the queue sharing group at a time. Issuing a start channel in one queue manager may result in the channel being started in another. The channel start may be routed around the queue sharing group to another queue manager.

Channel commands

The display channel status command issued in one queue manager may reflect the status in another. This will be covered in more detail in the following section.

As with inbound channels, channel commands must include the **CHLDISP** command to indicate the scope of the request.

Command Scope (**CMDSCOPE**) can be used with these commands to direct a command to a given queue manager in the queue sharing group.

Display channel status

Depending on how the command is issued, the results will be based on data kept in one of four locations. Table 1 shows the source of a ‘display channel status’ command based on the options used.

If the command uses the **SHORT** and **CHLDISP(PRIVATE)** keywords, the channel status will be obtained from the local queue manager’s local memory. If the channel is a shared channel, no status will be displayed.

If the command uses the **CURRENT** keywords and any channel disposition, the result will be from the local queue manager’s cache. If the channel is a shared channel, but it is not active in the current queue manager, no status will be displayed.

If the command uses the **SAVED** and **CHLDISP(PRIVATE**) keywords, the status will be obtained from the local channel sync queue. Only private channel status is kept in this queue.

If the command uses the **SAVED** and **CHLDISP(SHARED)** keywords, the status will be obtained from the shared channel sync queue. Only shared channel status is kept in this queue.

If the command uses the **SAVED** and **CHLDISP(ALL)** keywords, both of the above will be returned.

Command source	Options used		
Local queue manager	Short Current	Private ALL l Shared l Private	
Local sync queue	Saved	Private	ALL
Shared sync queue	Saved	Shared	ALL
Sync table (DB2)	Short	Shared	

Table 1: Source of ‘display channel status’ command

If the command uses the **SHORT** and **CHLDISP(SHARED)** keywords, the data will be obtained from a status kept in a DB2 database.

When used in conjunction with **QSGDISP** and **CMDSCOPE**, the results will vary based on the location of the channel instance and may produce some unexpected results.

OTHER 5.2 ENHANCEMENTS

This section will look at some enhancements that affect systems management but do not require a queue sharing group.

SMF statistics

The type 115 SMF records, which contain queue manager statistics, have been expanded to include statistics from the Coupling Facility services and the DB2 services running within the queue manager. The Coupling Facility statistics include the requests made, and elapsed times. For DB2, the statistics are more detailed and provide basic server statistics plus detailed request type information and associated timings.

Another change is that, instead of defining an SMF interval for the queue manager, the SMF data collection can be tied to the system-wide SMF collection (triggered by an SMF broadcast). This simplifies correlation of MQSeries data with data from other subsystems.

Performance collection

Additional detail performance collection is available. This is collected as extensions to the type 116 SMF records. These statistics are cut at the end of each task (job) and are also collected at the SMF collection interval for long-running tasks. The statistics include task-specific information, such as type, identity, call totals, plus queue level statistics indicating which queues were accessed, and timings. The queue level statistics are very detailed and include information commonly needed by MQSeries support.

While the statistics are fairly detailed, the primary focus was charge-back, although the data can be collected and correlated to provide additional insights into MQSeries operation. Partly because of the level of information collected, the activation of the detailed performance data will add an overhead of between seven and thirteen per cent.

‘Clear queue’ command

The ‘clear queue’ command available on distributed queue managers is now supported on OS/390. Without this command, clearing messages from a queue required an application program to read the queue. This is much more efficient, especially if there are a lot of messages on the queue.

Queue status

A new command is available to view the status of queue manager requests. This is commonly needed when troubleshooting problems and you need answers to questions such as, ‘who is using this queue?’ or ‘what is this application doing?’.

- **DISPLAY QSTATUS() TYPE(QUEUE)** – displays status of each queue (one line per queue) showing current input and output counts, and whether any uncommitted messages exist.
- **DISPLAY QSTATUS() TYPE(HANDLE)** – displays status of each thread (one message per thread per queue) identifying the connection name, the type, and current connection details. Using this information you can determine the origin of the request and cancel it. The cancel would have to be done using a local command to the originating subsystem (unless you had a vendor product that permitted cancelling from MQSeries).
- **DISPLAY THREAD() TYPE(REGIONS)** – the display thread command is summarized to display a single line per address space. This is most useful for threads that originate from IMS or CICS regions.

Dead letter queue handler

A dead letter handler has been available on distributed systems. This has now been made available on OS/390. The handler comprises a utility program and a set of rules for processing the messages. Rules can consist of queue names, application names, return codes, and message characteristics. Messages can be routed to alternative queues, retried after a duration to the same queue, discarded, or left on the dead letter queue.

SUMMARY

MQSeries for OS/390 5.2 provides some advanced features but potential users should note that new management concepts and facilities are required. Move carefully, and make sure you clearly understand the implications of your implementation. The potential impact on the Coupling Facility should be carefully evaluated.

Additional information

For additional information, the following Web sites are recommended:

- *www.software.ibm.com/mqseries*
- *www.bmc.com*

The documentation set that comes with MQSeries for OS/390 V 5.2 contains a wealth of detailed information on all of these topics:

- *Concepts and Planning Guide, GC34-5650.*
- *System Setup Guide, SC34-5651.*
- *System Administration Guide, SC34-5652.*
- *Intercommunication Guide, SC33-1872.*

Richard G Nikula
Product Author and Strategic Planner
BMC Software (USA)

Useful MQSeries clustering configurations

MQSeries clustering provides many benefits for systems administrators and developers looking to utilize a simplified network. However, it does have its limitations. In this article I will detail 'work-arounds' for two of the shortcomings of MQSeries clustering.

CLUSTERED AND NON-CLUSTERED QMR COMMUNICATION

It is often desirable to have an array of identical applications reading from a series of queues to provide a level of resilience and workload balancing. The clustering feature of MQSeries can provide this. Using multiple queue managers running on the same or different physical machines with each hosting an instance of a particular named queue, MQSeries can be set up to provide the required load balancing and resilience.

It should be noted that the resilience provided by this configuration is not a replacement for using a hardware cluster with cluster-aware applications. These two forms of clustering are often confused. Utilization of Microsoft Cluster Server with MQSeries Version 5.1 is provided in *SupportPac MC74*. Figure 1 shows this configuration.

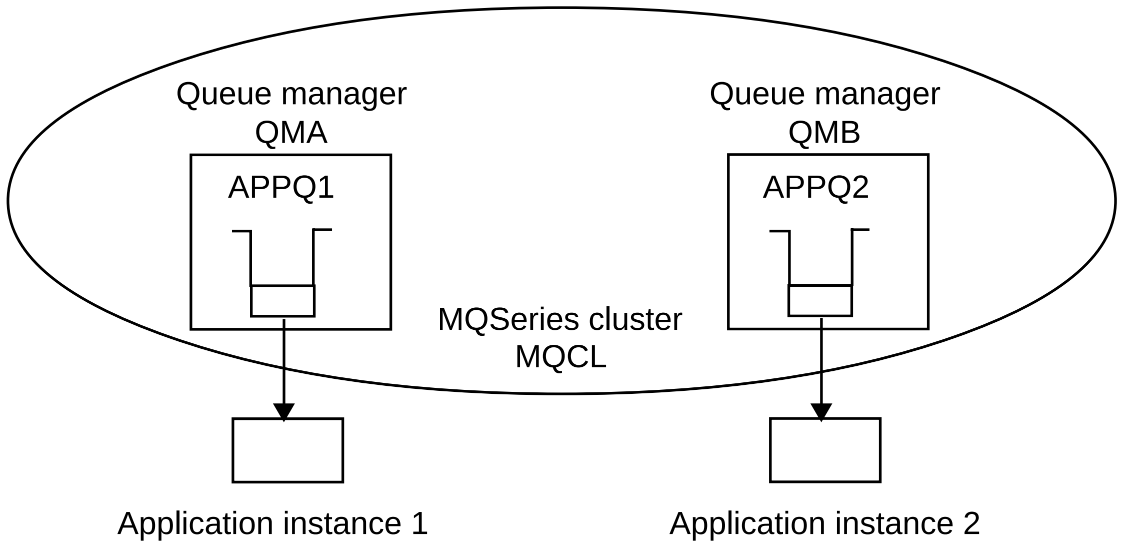

Figure 1: Providing load balancing and resilience

The problem arises when the requirement is to link this up with the rest of the legacy MQSeries network.

For instance, suppose that you have a system running on a Tandem NSK machine and there is a requirement for it to send messages for processing to the applications that are reading from queues in the MQSeries cluster. At present, Tandem NSK does not support the clustering feature of MQSeries so we have to find a way of getting the messages into the cluster and then workload-balancing the messages between each instance of the application in the cluster.

To do this, we need to introduce a gateway into the cluster. This will be a queue manager hosting a queue manager alias. This queue manager alias will be the queue manager that the remote queue definitions on the Tandem NSK point to as the remote queue manager.

This architecture is shown in Figure 2.

Queue manager *GATEWAY* is a member of the MQSeries cluster *MQCLUS*. It hosts a queue manager alias named *ANY.Q*.

The tandem queue manager *TANDEMQM* has a sender channel that connects directly to queue manager *GATEWAY* inside the cluster.

The remote queue definition *APPQ1* on *TANDEMQM* has an *RQMNAME* of *ANY.Q*.

The message flow is as follows:

- The application connects to *TANDEMQM*.
- The application writes messages specifying remote queue definition *APPQ1*.
- The messages are sent across the dedicated sender channel to queue manager *GATEWAY*.
- MQSeries attempts to resolve the queue name *APPQ1* (from the *RNAME* property of the remote queue definition) on queue manager *GATEWAY*.
- Since a queue manager alias was specified in the *RQMNAME* of the remote queue definition, the next step is for MQSeries to attempt to resolve the queue name in the cluster.

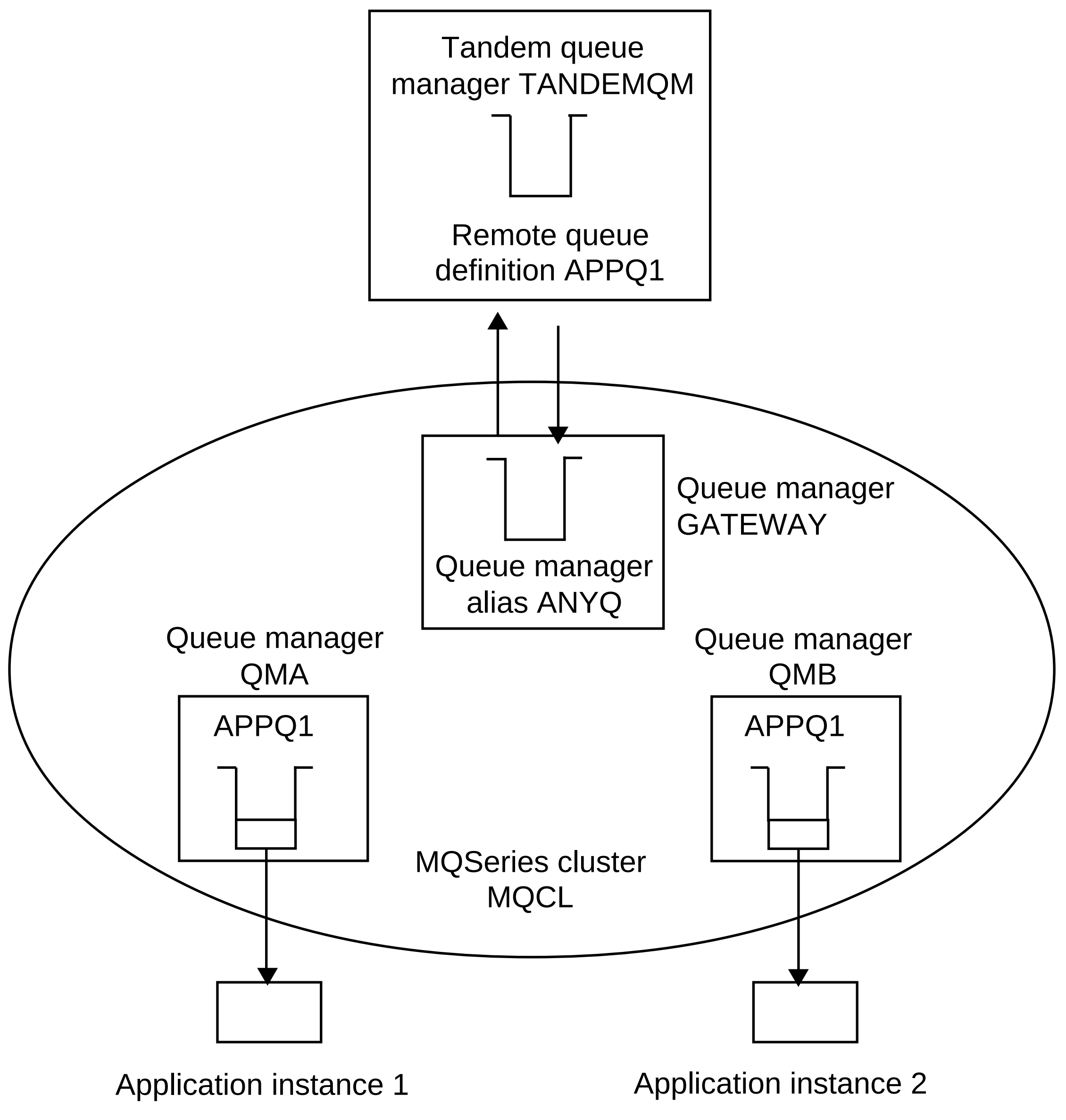

Figure 2: Introducing a gateway into the cluster

- There are multiple instances of the queue in the cluster, so 'round-robin' workload balancing is used to distribute the messages.

The full MQSeries definitions for this configuration are provided below.

THE MQSERIES CLUSTERED QUEUE MANAGER DEFINITIONS

Two queue managers in the cluster should be defined as repositories. Repositories are queue managers that store information about the location of other queues and queue managers in the cluster. All other queue managers are partial repositories. In this case I have arbitrarily chosen the repositories to be *QMA* and *GATEWAY.*

SETTING UP THE CLUSTER

On *QMA*

```
ALTER QMGR +
REPOS(MQCLUS) +
FORCE
```

Firstly, the cluster channels on *QMA* must be created. It is possible to create the cluster using the MQSeries Explorer in a Windows NT/2000 environment, however, for the benefit of users of other environments, I include the definitions.

```
DEFINE CHANNEL(TO.QMA) +
CHLTYPE(CLUSRCVR) +
TRPTYPE(TCP)
CONNAME(localhost) +
```

(Where local host is the IP address or hostname of the machine that hosts *QMA*.)

```
CLUSTER(MQCLUS)
DEFINE CHANNEL(TO.GATEWAY) +
CHLTYPE(CLUSSDR) +
TRPTYPE(TCP) +
CONNAME(IP Address of machine hosting GATEWAY) +
CLUSTER(MQCLUS)
```

On *QMB*

```
DEFINE CHANNEL(TO.QMB) +
CHLTYPE(CLUSRCVR) +
TRPTYPE(TCP) +
CONNAME(localhost) +
```

(Where local host is the IP address or hostname of the machine that hosts *QMB*.)

```
CLUSTER(MQCLUS) +
REPLACE
```

```
DEFINE CHANNEL(TO.QMA) +
CHLTYPE(TCP) +
TRPTYPE(TCP) +
CONNAME(IP Address of machine hosting QMA) +
```

(This sender channel could also have been pointed at *GATEWAY*.)

```
CLUSTER(MQCLUS) +
REPLACE
```

On *GATEWAY*

```
ALTER QMGR +
REPOS(MQCLUS) +
FORCE
DEFINE CHANNEL(TO.GATEWAY) +
CHLTYPE(CLUSRCVR) +
TRPTYPE(TCP) +
CONNAME(localhost) +
```

(Where local host is the IP address or hostname of the machine that hosts *QMB*.)

```
CLUSTER(MQCLUS) +
REPLACE
DEFINE CHANNEL(TO.QMA) +
CHLTYPE(CLUSSDR) +
TRPTYPE(TCP) +
CONNAME(IP address or hostname of the machine hosting QMA) +
CLUSTER(MQCLUS) +
REPLACE
```

This is enough to configure the cluster in preparation for the creation of the application queues.

QUEUE AND QUEUE MANAGER ALIAS DEFINITIONS

APPQ1 is simply a standard local queue that is hosted by multiple queue managers. Therefore, the queue will be created on each queue manager that an application will connect to in the cluster.

On *QMA* and *QMB*

```
DEFINE QLOCAL(APPQ1) +
DEFPSIST(YES*) +
*If it is deemed important that message survive a restart.
CLUSTER(MQCLUS) +
SHARE +
DEFBIND(NOTFIXED) +
REPLACE
```

On *GATEWAY*

```
DEFINE QREMOTE(ANY.Q) +
RQMNAME() +
RNAME() +
XMITQ() +
CLUSTER(MQCLUS) +
REPLACE
```

On *TANDEMQM*

```
DEFINE QREMOTE(APPQ1) +
RNAME('APPQ1') +
RQMNAME('ANY.Q') +
XMITQ('HUB') +
```

The transmission queue has been arbitrarily named 'HUB' in this case.

WORKLOAD BALANCING FROM WITHIN A CLUSTER

At a recent MQSeries User Group an organization suggested the requirement that load balancing be implemented between the local queues of the cluster when the application that is writing to the queues is local to one of the clustered queue managers.

The scenario here is that there are multiple applications that write to local queues hosted by queue managers inside an MQSeries cluster. The messages are read by multiple instances of another application. Figure 3 shows this configuration.

In this configuration, when an application writes to an instance of a local queue, the default behaviour of MQSeries is to write the message to the local instance of the queue.

The problem arises with the default behaviour of MQSeries in regard to queue resolution in a cluster. When the application writes a message and does not specify a queue manager, the message will always be written to the local instance. There are three ways to get around this problem.

- Write a cluster workload exit.
- Write the workload balancing logic into the application.
- Use a queue manager alias to fool MQSeries into thinking that the local queue exists on a remote queue manager.

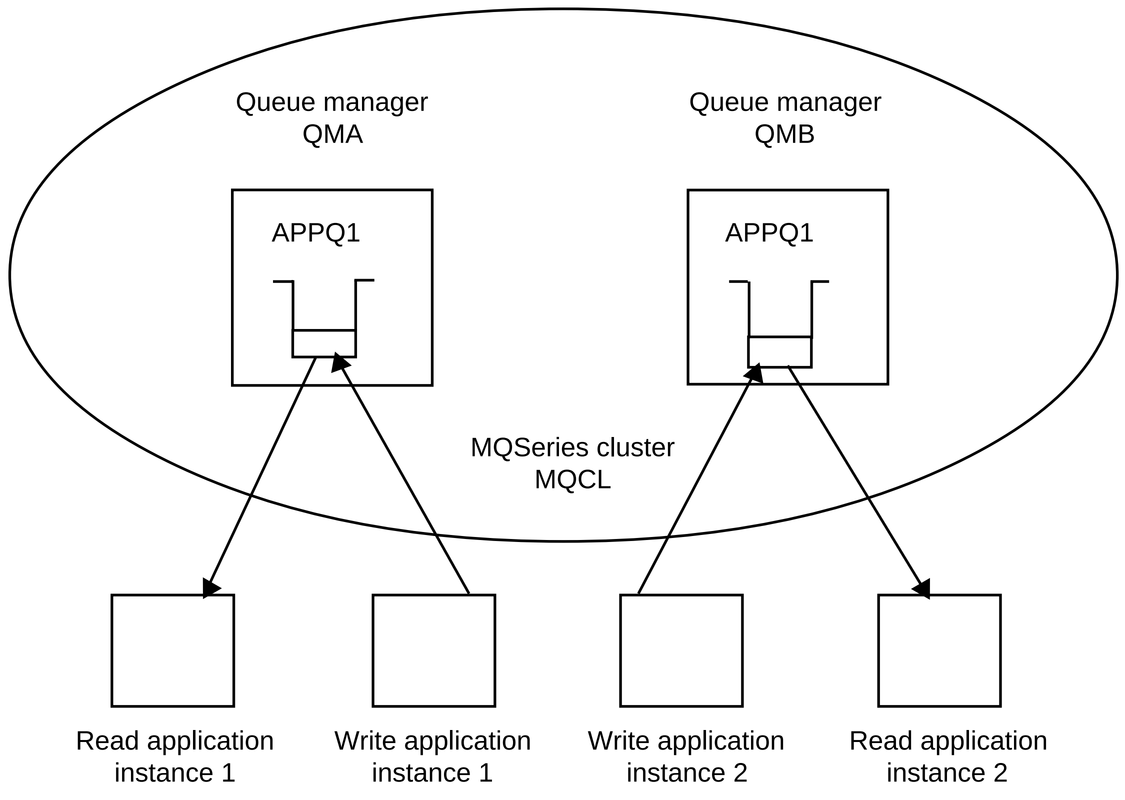
Queue manager QMA
Queue manager QMB
APPQ1
APPQ1
MQSeries cluster MQCL
Read application instance 1
Write application instance 1
Write application instance 2
Read application instance 2

Figure 3: Workload balancing from within a cluster

Writing a cluster workload exit will be a time-consuming exercise requiring specialist skills in coding MQSeries exits.

Incorporating the workload balancing logic into the application may not be possible if the application is already written.

The following work-around makes the requirement possible. This is a variation on the same theme detailed above, which allowed us to workload-balance messages sent from a queue manager outside the MQSeries cluster.

This time, the messages are sent from within the cluster. The way in which we prevent MQSeries from automatically writing the messages to the local instance of the application queue is by defining a remote queue on each queue manager in the cluster that hosts an application queue.

In a similar configuration to above, an extra queue manager is introduced, hosting a queue manager alias. The remote queue definitions point to this queue manager alias and have an *RNAME* property of the application queue name. The queue manager alias is a member of the cluster so MQSeries then resolves the *RNAME* to the instances of the local application queues in a round-robin manner.

This is shown in Figure 4.

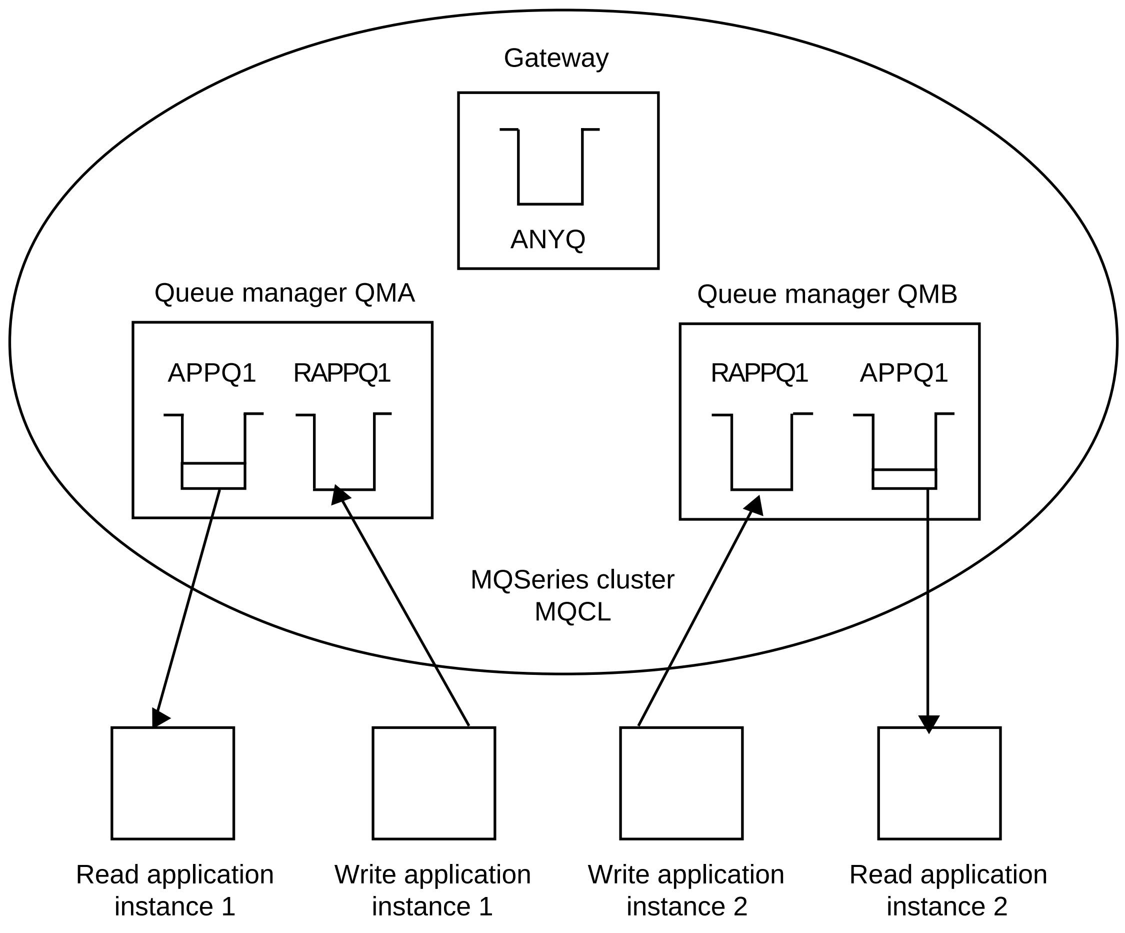
Gateway
ANYQ
Queue manager QMA
APPQ1
RAPPQ1
Queue manager QMB
RAPPQ1
APPQ1
MQSeries cluster
MQCL
Read application instance 1
Write application instance 1
Write application instance 2
Read application instance 2

Figure 4: Introducing an extra queue manager

CLUSTER DEFINITIONS

The cluster should be set up as previously defined above.

The following extra definitions should be made:

On *QMA*

```
DEFINE QREMOTE(RAPPQ1)
RNAME(APPQ1)
RQMNAME(ANY.Q)
XMITQ()
```

On *QMB*

```
DEFINE QREMOTE(RAPPQ1)
RNAME(APPQ1)
RQMNAME(ANY.Q)
XMITQ()
```

Considerations

This will either increase the network traffic if queue managers exist on separate machines or, if they are all hosted on the same machine, extra processing will be required. This is because messages must be sent to queue manager *GATEWAY* and then back again to either *QMA* or *QMB*.

Depending on the workload or time constraints, this solution could be permanent or a short-term work around while a cluster workload exit is written.

These techniques are not suitable if message sequence is an issue. Since MQSeries routes messages based on a ‘round-robin’ algorithm, message affinities will be destroyed.

If a machine hosting one of the queue managers was to fail, then the cluster compensates by offering an alternative path for the messages through the cluster. However, if there were any messages on a local queue on this queue manager, those messages would be unavailable until the machine and the queue manager are brought back online.

To compensate for this, the *SupportPac MC74 for Microsoft Windows NT Cluster Server* could be utilized. Other platforms, such as Sun Solaris, can also be hardware-clustered with MQSeries.

Ross Harrison Coundon
Software Development Engineer
Dataroam (UK)

Application design considerations with MQSeries

Developers new to MQSeries often ask "How do I talk to MQSeries?", indicating that they need to get acquainted with the MQSeries API (MQI), ideally, from a familiar programming language. As an experienced MQSeries developer I find this question less important than "What about an application communications protocol?", since your MQSeries application program is really talking to another program. This article discusses some considerations that will enable the development team to ask the right questions – and to answer them.

THE EASY QUESTION: LEARNING MQSERIES

The typical learning curve when 'talking' MQSeries has three distinct phases.

- You discover that your favourite language is supported – great! For many languages, some sample programs even come with the product. Be aware that these programs are often simple utilities, but useful nevertheless. Provided that somebody installed MQSeries and did some setup you can be up and running with your first program surprisingly quickly. That is, if you don't mind grabbing pieces of other people's code to make up your program.
- When trying to use this new knowledge in a real project, you suddenly discover how many options MQSeries has, and that they can be combined in so many ways that you can't count them. Hopefully, your struggle with these options combined with the real-life requirements of your project, leads you to phase three.
- There is a limited number of real-life scenarios. Data transmission, with MQSeries or in other ways, almost always fits into one of these four models:
  - put data on a queue and forget it – called send-and-forget.
  - read data from a queue that somebody else puts data on – the receiver end of send-and-forget messages.
  - a client model, sending a request and waiting for a reply.
  - a server model, reading requests and producing replies.

In addition, other matters must be considered, ie message integrity, clean-up of messages, and code-page conversions.

THE HARD QUESTION: THE NEED FOR A PROTOCOL

It is simple to 'talk to' MQSeries, but remember your program is really 'talking to' another program, which is the program reading your output, or *vice versa*. No matter how many protocol layers you have beneath MQSeries, remember that you need to invent one for the communication between two programs talking to each other over MQSeries. Is your program 'talking to' an existing program? You need to know its protocol and stick to it. Are you designing both ends of a communication? You need to define a protocol for both the programs involved. Protocol considerations are:

- Is a unit of work (UOW) one message or more? If it's more, how do you determine when the UOW is finished, and how do you avoid handling a partial UOW, when, for example, the line breaks in the middle of transmission?
- Determine a maximum message length so receiver programs won't be disturbed by unexpectedly long messages.
- A client application will sometimes time-out without getting a reply. Probably, it will never get back and fetch its reply message, so you are left with a clean-up problem. The MQSeries expiry option might help you here. Please note that message integrity and expiry usually conflict.
- There is a convention, supported by MQSeries Message Descriptor fields, for servers to know who to reply to, and one for clients to pick a specific reply message out of a queue with more messages on it – the copy Message Id to Correlation Id convention. Another way of getting your own reply and not somebody else's is to create a dynamic reply queue. In both situations, the Reply-to-Queue field can be used. Such parts of the protocol must be determined and implemented at both ends of a communication. If implemented properly, a server program can serve new client machines with no changes except in the MQSeries setup, and it can serve new client applications with no change at all on the server side.

- Code-page conversion can be done by MQSeries, provided that all data is of the format ‘string’ and that messages are marked as format ‘string’. This can be very useful, since you should be prepared for the possibility that one end of a communication could be replaced by another application on another platform; a straightforward thing to do because of the architectural openness of MQSeries. You might be using the same ASCII code page on Unix and Windows, but remember that there are EBCDIC platforms too, such as IBM mainframes.
- MQSeries is an asynchronous transport mechanism. A client/ server application is conceptually synchronous, but can nonetheless, use MQSeries as the transport mechanism for request and reply messages. Remember, however, that the client and server programs are never ‘in contact’ because of the asynchronous nature of MQSeries and are unable to detect whether or not the other program is up and running. Therefore, problems in a client/ server setup tend to show up as time-out at the client end. Such time-outs can mean one or more of the following:
  - the line between the client and server machines is down
  - the MQSeries channels – a sender and receiver channel at each end – have not all started properly
  - the server machine or its MQSeries Queue Manager is not running properly
  - the server application has not started or, in case of triggering, the triggering is not defined properly
  - some machine in between the client and server machines, such as a gateway or a message broker, might be down
  - the server application is too busy to respond within the expected amount of time.

Here are some questions to ask yourself.

- Is message integrity important? If so, use the persistence and possibly, syncpoint options. Remember that message integrity and speed conflict. If your application is of the ‘if it doesn’t happen within 30 seconds, it doesn’t matter’ type, ie Web inquiries, typically, you should set the integrity options to off.

- If working with more MQSeries messages in one UOW and integrity matters, you should use the syncpoint option to synchronize one or more MQSeries 'read and write' activities. When running under a transaction monitor, MQSeries updates can be synchronized with updates to databases.
- For programs reading messages under syncpoint control, consider using the back-out count, back-out threshold, and back-out queue name fields from the definition of the input queue. The purpose is to prevent a 'poison' message from stopping your program prematurely, which in turn makes MQSeries back-out the message to the input queue, and your program reads it again. If you do not use this feature to get rid of 'poison' messages, a message or program failure might effectively stop the reader program.
- Similarly, receiver programs should be able to handle unexpectedly long messages by using the 'accept truncated message' option. The part actually read should be put to the back-out queue for diagnostic reasons and your program can go on handling OK messages.

HOW TO SIMPLIFY MATTERS

For those who want to make MQSeries data communications what application programmers want it to be – a way of sending or receiving data without too many problems – there are various options.

You can make or buy 'wrappers' that hard-code most of the options in suitable combinations. Typically, you would need wrappers for the four data transfer models defined above. A potential benefit of wrappers is a common API for MQSeries and other middleware tools.

The traditional MQSeries interface is called the Message Queue Interface (MQI). Today, MQSeries comes with an alternative application interface, the Application Messaging Interface (AMI). The purpose of this is to simplify the tasks of the application programmer and leave most of the options to administrators. To accomplish this, the AMI offers three components:

- The message: what is sent from one program to another.
- The service: where the message is sent.
- The policy: how the message is sent.

Services and policies are defined by a systems administrator and stored in a repository. The application programmers need only concern themselves with messages – when writing calls, they refer the appropriate service and policy to the AMI. The application programmer is relieved of the options burden and a particular application program can be used in different set-ups – simultaneously or over time.

Besides encompassing the four general data transmission models, the AMI offers distribution to multiple receivers using a distribution list, and it offers a publish/subscribe service. The AMI can be used by application programs communicating with other application programs using AMI or MQI.

BENEFITS

- Not only can programs on different platforms talk to each other, programs of different languages can talk to each other. To benefit from this, messages should be designed so that they are not bound to any particular language. Everything in plain string format works fine, and on top of that, gives you code-page conversion when required and no big/little endian trouble.
- For conventional database applications, multi-threading should be seriously considered. It can be very simple to implement a multi-threading MQSeries server program. In case of triggering, you can let MQSeries use the 'trigger every' option. For some platforms, eg Windows NT, remember to define the trigger command as **start <program>** rather than just **<program>** because this will start an independent thread and the trigger monitor will immediately go on looking at your trigger queues. For non-triggering, you can simply start several instances of your program, since MQSeries will take care of the multi-threaded aspects of the competition between programs working on a queue.
- Because MQSeries is so well-established there are many products available – message brokers and publish-and-subscribe tools, for example – that will add value to your existing applications.

Christian Sorensen
Senior Systems Engineer, Maersk Data (Denmark)

MQ news

IBM has launched its MQSeries Integrator Agent for CICS Transaction Server, providing application integration on the mainframe with CICS and IMS applications. The software enables functional migration from the existing Message Driven processor (Mdp) product originally available from Early, Cloud & Company.

It consists of two components: a build-time component that runs on Windows NT and uses tools that look and feel similar to those of MQSeries Integrator, and a run-time component that runs under CICS TS for OS/390 V1R3 as a CICS application on an OS/390 server.

The run-time bit uses CICS Business Transaction Services to manage the interrelationship, commit scope, recovery, and restart of the actions that make up a business transaction.

Together, they enable the construction and execution of adapters to process requests from controlling applications for business transactions running on CICS and IMS host systems.

Target IMS and CICS applications can be driven via 3270 data streams. CICS applications can also be driven through a Distributed Program Link, while MQSeries-enabled applications are accessed through MQSeries. There are server adapter programs that handle all three classes of applications.

Out on 27 April, it costs USD100,000.

For further information, contact your local IBM representative.

* * *

Progress Software's SonicMQ has begun shipping Version 3.1 of its SonicMQJava messaging server, with a new family of bridges for improved connectivity and integration with other messaging types and products, including MQSeries.

The software enables developers to route messages to other Java message services and disparate legacy systems. In particular, the bridge for MQSeries allows connection to third parties through an MQSeries interface, as well as to virtually any existing enterprise application built on top of an MQSeries infrastructure.

It natively supports Internet standards and is based on the Java Message Service specification.

For further information contact:
Progress Software, 14 Oak Park, Bedford MA 01730, USA
Tel: +1 781 280 4000
Fax: +1 781 280 4095
Web: http://www.progress.com

Progress Software, The Square, Basingview, Basingstoke, Hants RG21 2EQ, UK
Tel: +44 1256 816668
Fax: +44 1256 463226

* * *