MQ update

20

February 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* is available from Xephon's Web site at www.xephon.com/ mqupdate.html (you'll need the user-id shown on your address label to access it). If you've a problem with your user-id or password call Xephon's subscription department on +44 1635 33886.

Commissioning Editor

Peter Toogood
E-mail: PeterT@xephon.net

Managing Editor

Madeleine Hudson
E-mail: MadeleineH@xephon.com

Subscriptions and back-issues

A year's subscription to *MQ Update*, comprising twelve monthly issues, costs £255.00 in the UK; $380.00 in the USA and Canada; £261.00 in Europe; £267.00 in Australasia and Japan; and £265.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the July 1999 issue, are available separately to subscribers for £22.50 ($33.50) each including postage.

MQSeries logging on AIX

CIRCULAR OR LINEAR LOGGING ON AIX

Circular logging

Circular logging is the default ‘out of the box solution’. Three files are created when a queue manager is created:

- *S0000000.LOG*
- *S0000001.LOG*
- *S0000002.LOG*.

These are used, as the logging style implies, in a circular fashion. When the last log is overwritten, MQSeries starts at the beginning, overwriting the contents of the first log unless there is a log entry contained therein, which would be required to recover any persistent data. More on this later.

Specified in the *QM.INI* file are the parameters which define the logging for that queue manager. A sample of a default is shown below.

LOG

```
LogPrimaryFiles=3
LogSecondaryFiles=2
LogFilePages=1Ø24
LogType=CIRCULAR
LogBufferPages=17
LogPath=/var/mqm/log/QPØ2/
```

The first parameter in the stanza specifies the number of log files created and used when the queue manager is first started. The second line of the stanza (*LogSecondaryFiles*) is the number of logs that will be created as an overflow, should the scenario occur when any of the first three cannot be overwritten because that log contains log records required for recovery.

LogFilepages specifies the number of 4K pages used as log file space; in this example (which is the default for Unix), each log file will be

LogBufferPages is a tuning parameter, which can be used to speed up writes to the logs by supplying larger buffers. Specified here is the default, which will supply 68KB of buffer pages. A greater number of buffer pages will speed up logging of large messages. Persistent messages are logged before they are written to the queue.

Circular logs contain sufficient information to recover from a queue manager failure or a controlled shutdown. If MQ objects are damaged or lost, circular logging will not contain the necessary records to recover them. However, the scheduling of a utility, such as *SAVEQMGR* (IBM Support PAC MS03), which captures an image of a queue manager with all its objects, will satisfy this requirement.

The PCF scripts created by *SAVEQMGR*, for example, can be used to recreate a completely 'gubbed' queue manager or to recreate a damaged object.

It is unlikely that a media failure could result in the complete loss of log if RAID 1+ mirroring is used; however, if this is not available, the log data needs to be backed-up along with the queue manager and its associated file system in order to provide a consistent point of recovery.

Linear logging

Linear logging can only be created at the same time as the queue manager. You cannot switch over to linear logging – the queue manager must be deleted and recreated with the linear logging parameter specified (**-ll**). The logs are contained in a continuous sequence of files, which MQ does nothing to manage.

Archiving or deletion must be carried out manually, as linear log space is never re-used. Linear logging allows the use of the MQ media back-up operation (**rcdmqimg**) to be carried out for a queue manager, which will write MQ images to the log files. This, in turn, allows the recover command (**rcrmqobj**) to be used to completely recover a queue manager, all its objects, and its data, to the point of recovery specified by the last media recovery command. However, this command must be carried out under a running queue manager, ie you would have to recreate and start the queue manager to effect a recovery.

Linear logging is, therefore, a requirement when no back-ups of a queue manager have been taken other than the use of the record media image command, and there is a requirement to provide forward recovery.

This requirement is unlikely to surface if RAID 1 mirroring or higher is being used for the DASD subsystem as the likelihood of complete media failure is significantly reduced.

Andrew Miller
Senior Technician (UK)

MQSeries for OS/390 V5.2: CF and DB2 issues

By now, most readers will know about the latest and greatest updates to the MQSeries family of products, which were included in the 'common' release, V5.2, as announced by IBM, firstly in April 2000, and subsequently in October 2000.

For a comprehensive overview of the product updates, please refer to the Web site:

http://www-4.ibm.com/software/ts/mqseries/announceoct

Although IBM strives to make the whole product family look and feel the same, there are always going to be differences due to the underlying operating systems. This article concentrates on the differences on the OS/390 side of MQSeries, and in particular on the Coupling Facility (CF) and DB2, which are essential components required to support the new 'Shared Queues' facility. This article is based on actual hands-on experience.

DEFINITIONS

The introduction of shared queues in the new release was particularly significant. These queues are defined as normal queues, but they have an extra attribute known as 'disposition', which is set to 'shared'. To ensure that the shared queue can be accessed in full read and write mode from any of the OS/390 queue managers, which are all part of a new grouping arrangement called the Queue Sharing Group (QSG), two additional resources are required:

- DB2, which is IBM's flagship relational database system and is used to store the actual queue definition. 'Normal' queue definitions are stored in Pageset zero.
- The Coupling Facility, which is used to store the actual data.

The result is that, although a queue manager is needed to make the original definition of a shared queue, thereafter the queue is 'separated' from the queue manager – it no longer 'belongs' to it.

If the queue manager crashes, the shared queue definition and data are still available to the other queue managers in the QSG. In fact, MQ detects when a queue manager fails (and disconnects from the CF) while processing messages within a logical unit of work, and uses another queue manager within the QSG to complete the work. This is known as peer-recovery.

It is interesting to note that IBM intends to use shared queues in its NetView product. See *HTTP://www.s390.ibm.com/sa* for further details.

Figure 1 illustrates the set-up we used. By convention, the CF is shown as a triangle.

THE ROLE OF DB2

The required release of DB2 must be V5.1 or higher (in fact, we successfully tested with DB V6.1), which must be in full data-sharing mode. It could be said that, if your environment does not use DB2, then it may seem like overkill to have to install DB2 just to support shared queues. However, it is not totally surprising.

In order to improve further the availability of MQSeries (begun in the previous release V2.1, when IBM introduced clustering), IBM needed a 'system' to store the queue definitions in such a way that they were accessible to more than one queue manager at the same time – in full read and write mode – with the additional requirement that this 'system' had full recoverability itself.

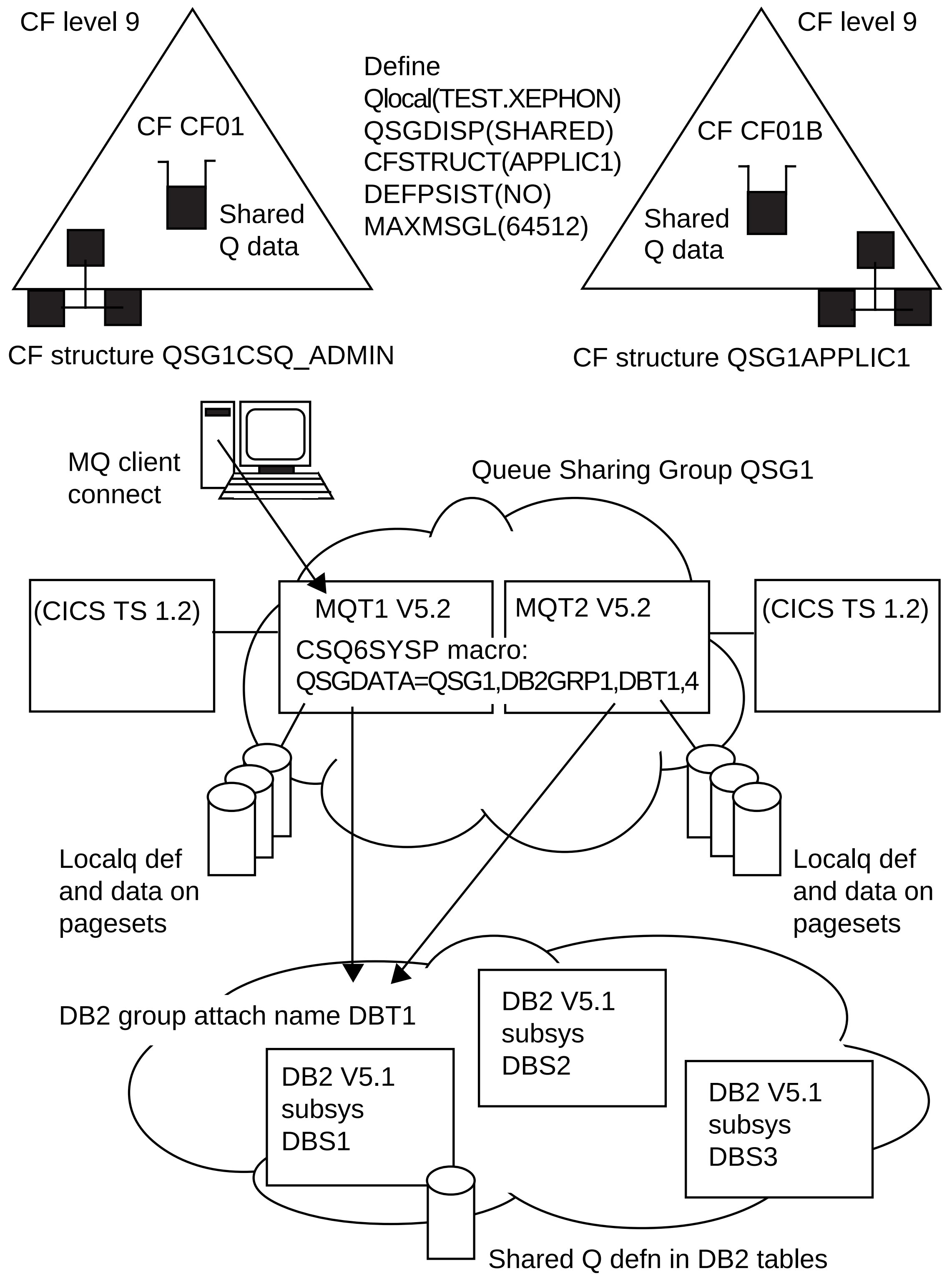
CF level 9
CF CF01
Shared Q data
CF structure QSG1CSQ_ADMIN
Define Qlocal(TEST.XEPHON) QSGDISP(SHARED) CFSTRUCT(APPLIC1) DEFPSIST(NO) MAXMSGL(64512)
CF level 9
CF CF01B
Shared Q data
CF structure QSG1APPLIC1
MQ client connect
Queue Sharing Group QSG1
(CICS TS 1.2)
MQT1 V5.2
MQT2 V5.2
CSQ6SYSP macro: QSGDATA=QSG1,DB2GRP1,DBT1,4
(CICS TS 1.2)
Localq def and data on pagesets
Localq def and data on pagesets
DB2 group attach name DBT1
DB2 V5.1 subsys DBS2
DB2 V5.1 subsys DBS1
DB2 V5.1 subsys DBS3
Shared Q defn in DB2 tables

Figure 1: Coupling Facility set-up

Also, DB2 is already in use on other platforms, for example IBM's message broker product MQSeries Integrator, where it is used to store 'state' information (V2.0.1).

There are a number of steps to be followed before DB2 can be used.

1 Decide whether to create a DB2 subsystem just for MQSeries, or to use an existing one. We decided to use the existing environment.

   MQSeries' use of DB2 is for queue definitions and holding the state of the 'shared channels'. This suggests MQSeries does not need a huge amount of tables. On the other hand, to ensure that MQSeries is not affected by DB2 maintenance, a separate subsystem may be required.

2 Ensure that DB2 is running in full data-sharing mode, with preferably two or more DB2 subsystems combined in a data-sharing group.

   The advantage here is that, should one DB2 subsystem become unavailable (through a crash or a controlled shutdown), then another subsystem in the same group will automatically take over its functions.

   This function was successfully tested. Our set-up had three DB2 subsystems within a DB2 data-sharing group. When one DB2 was 'forced', MQ reconnected to the next DB2 subsystem.

3 Ensure that the OS/390 Resource Recovery Services (RRS) facility is enabled.

4 A number of standard DB2 jobs need to be run. Please see below.

5 Update the *CSQZPARM* parameter module to tell MQSeries which DB2 Group Attach name to connect to (see *QSGDATA*).

   This is actually quite an 'advanced' connection. MQSeries connects to the 'group attach' name, and has no knowledge of – or interest in, for that matter – which DB2 subsystems are available.

It is a marked improvement over the CICS-DB2 connection, where CICS still connects to a specific DB2 subsystem. If that subsystem goes down, then CICS-DB2 activity stops. IBM has acknowledged this to be a long-standing requirement, but at the time of writing it will not be available in CICS TS 2.1 or the following release. In addition, the CICS-MQ connection also suffers from this limitation and the shared queue facility does not resolve this! Maybe we need a CICS-to-MQ-group-attach!

Please note that the DB2 data-sharing group must be available to all MVS images (LPARs) where an MQ queue manager in the group will run. If multiple DB2s are available in the same data-sharing group in an MVS image, MQ can use any of them because it connects using the DB2 data-sharing group name, which also makes it simple to move MQ queue managers between MVS images. However, for MQSeries to connect to a queue-sharing group, a DB2 in the data-sharing group must be active in the same MVS image where the queue manager is running. At start-up, MQSeries will automatically connect to DB2. If it fails to do so (either DB2 or RRS is down), then it will attempt to reconnect until it is successful. If you have a product such as PDSMAN, you will see the following two lines repeated (every 10 seconds):

```
PDSMØ18U-E,ELEMENT DSN3IDØØ LVL ØØØ DATE ØØØØØ/ØØØØ LIB xxx.LOAD
PDSMØ18U-E,ELEMENT DSNARRS LVL ØØØ DATE ØØØØØ/ØØØØ LIB xxx.LOAD
```

Until MQ reconnects to DB2, you will not be able to display any shared queue information. It is worth trapping this failure using one of the automation tools (eg NetView).

STEPS REQUIRED TO USE DB2

For each DB2 data-sharing group (see sample JCL in *thlqual.SCSQPROC*):

```
Note: all DB2 jobs were run under batch TSO using the following JCL :
//CREATEDB EXEC PGM=IKJEFTØ1,REGION=4M,DYNAMNBR=2Ø
//STEPLIB   DD DISP=SHR,DSN=xxxxxxxxxxxx
//SYSPRINT  DD SYSOUT=*
//SYSTSPRT  DD DUMMY,SYSOUT=*
//SYSTSIN   DD *
    DSN SYSTEM(DBT1)                      (Group attach name)
    RUN PROGRAM(DSNTIAD) -
        PLAN(DSNTIAD)
//*     LIB('DSN++DB2VER++.RUNLIB.LOAD')    (not needed if linklsted)
//SYSIN     DD *
```

1 Create the storage group for the MQSeries database, tablespaces, and tables.

```
CREATE STOGROUP "MQDEFLT" VOLUMES('*');
```

2 Create the database to be used by all queue managers that will be connecting to this DB2 data-sharing group:

```
CREATE DATABASE "MQDB1"
BUFFERPOOL BP32K
STOGROUP MQDEFLT;
```

3 Create the 3 tablespaces that will contain the queue manager and channel initiator tables that are used for queue-sharing groups (to be created in step 4). The sizes you choose may be different:

```
REATE TABLESPACE "MQTS1"
USING STOGROUP MQDEFLT
PRIQTY 720  SECQTY 720  PCTFREE 20  SEGSIZE 64
BUFFERPOOL BP10  LOCKSIZE ANY  CLOSE NO
IN MQDB1;
CREATE TABLESPACE "MQTS2"
USING STOGROUP MQDEFLT
PRIQTY 720  SECQTY 720  BUFFERPOOL BP32K
LOCKSIZE ANY  CLOSE NO
IN MQDB1;
CREATE TABLESPACE "MQTS3"
USING STOGROUP MQDEFLT
PRIQTY 720  SECQTY 720  FREEPAGE 10  PCTFREE  30
SEGSIZE 64  BUFFERPOOL BP10  LOCKSIZE ANY  CLOSE NO
IN MQDB1;
```

4 Create the ten DB2 tables and associated indexes. Do not change any of the row names or attributes. I have just listed the table names and main keys here, as there are a large number of columns.

```
CREATE TABLE CSQ.ADMIN_B_QSG
    (
    QSGNAME        CHAR(4)  NOT NULL   ,
    etc …
    PRIMARY KEY (QSGNAME)
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_QSG
    ON CSQ.ADMIN_B_QSG (QSGNAME ASC)
    USING STOGROUP MQDEFLT
    PRIQTY 720  SECQTY 720  CLOSE NO;
CREATE TABLE CSQ.ADMIN_B_QMGR
    (
    QMGRNAME     CHAR(48) NOT NULL   ,
    QSGNAME      CHAR(4) NOT NULL    ,
    etc …
    VSOBJECT     VARCHAR(256Ø)       ,
    PRIMARY KEY (QMGRNAME),
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE RESTRICT
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_QMGR
    ON CSQ.ADMIN_B_QMGR (QMGRNAME ASC)
    USING STOGROUP MQDEFLT PRIQTY 72Ø  SECQTY 72Ø  CLOSE NO;
CREATE TABLE CSQ.ADMIN_B_STRUCTURE
    (
    STRUCNAME    CHAR(12) NOT NULL   ,
    QSGNAME      CHAR(4)  NOT NULL   ,
    etc…
    PRIMARY KEY (STRUCNAME, QSGNAME) ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_STRUCTURE
             ON CSQ.ADMIN_B_STRUCTURE (STRUCNAME ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT PRIQTY 72Ø  SECQTY 72Ø  CLOSE NO;
CREATE TABLE CSQ.OBJ_B_QUEUE
    (
    QNAME          CHAR(48) NOT NULL   ,
    QSGNAME        CHAR(4)  NOT NULL   ,
    etc…
    PRIMARY KEY (QNAME, QSGNAME)       ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_QUEUE
    ON CSQ.OBJ_B_QUEUE (QNAME ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT
    PRIQTY 72Ø  SECQTY 72Ø  FREEPAGE 1Ø  PCTFREE  2Ø CLOSE NO;
CREATE TABLE CSQ.OBJ_B_PROCESS
    (
    PROCNAME     CHAR(48) NOT NULL   ,
    QSGNAME      CHAR(4)  NOT NULL   ,
    etc…
    PRIMARY KEY (PROCNAME, QSGNAME)  ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_PROCESS
    ON CSQ.OBJ_B_PROCESS (PROCNAME ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT PRIQTY 72Ø  SECQTY 72Ø  CLOSE NO;
CREATE TABLE CSQ.OBJ_B_STGCLASS
    (
    STGCNAME     CHAR(48) NOT NULL   ,
    QSGNAME      CHAR(4)  NOT NULL   ,
    etc…
    PRIMARY KEY (STGCNAME, QSGNAME)  ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_STGCLASS
    ON CSQ.OBJ_B_STGCLASS (STGCNAME ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT PRIQTY 72Ø  SECQTY 72Ø  CLOSE NO;
CREATE TABLE CSQ.OBJ_B_NAMELIST
    (
    NLNAME       CHAR(48) NOT NULL   ,
    QSGNAME      CHAR(4)  NOT NULL   ,
    etc…
    PRIMARY KEY (NLNAME, QSGNAME)    ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_NAMELIST
    ON CSQ.OBJ_B_NAMELIST (NLNAME ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT PRIQTY 72Ø  SECQTY 72Ø  CLOSE NO;
CREATE TABLE CSQ.OBJ_B_CHANNEL
    (
    CHLNAME      CHAR(48) NOT NULL   ,
    QSGNAME      CHAR(4)  NOT NULL   ,
    etc…
    PRIMARY KEY (CHLNAME, QSGNAME)   ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS1;
CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_CHANNEL
    ON CSQ.OBJ_B_CHANNEL (CHLNAME ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT PRIQTY 72Ø  SECQTY 72Ø  CLOSE NO;
CREATE TABLE CSQ.ADMIN_B_SCST
    (
    CREATESTAMP     TIMESTAMP NOT NULL        ,
    XMITQ           CHAR(48) NOT NULL         ,
    CHLNAME         CHAR(2Ø) NOT NULL         ,
    REMOTEQMGR      CHAR(48) NOT NULL         ,
    QSGNAME         CHAR(4)  NOT NULL         ,
    etc…
    PRIMARY KEY (XMITQ, CHLNAME, REMOTEQMGR, QSGNAME) ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
  ON DELETE CASCADE
  )
  IN MQDB1.MQTS2;
```

```
CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SCST_IX1
    ON CSQ.ADMIN_B_SCST
    (XMITQ ASC, CHLNAME ASC, REMOTEQMGR ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT
    PRIQTY 72Ø  SECQTY 72Ø  FREEPAGE 5  PCTFREE  3Ø CLOSE NO;
CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SCST_IX2
    ON CSQ.ADMIN_B_SCST
    (CREATESTAMP ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT
    PRIQTY 72Ø  SECQTY 72Ø  FREEPAGE 5  PCTFREE  3Ø CLOSE NO;
CREATE TABLE CSQ.ADMIN_B_SSKT
    (
    XMITQ          CHAR(48) NOT NULL         ,
    CHLNAME        CHAR(2Ø) NOT NULL         ,
    REMOTEQMGR     CHAR(48) NOT NULL         ,
    KEY            INT      NOT NULL         ,
    QSGNAME        CHAR(4)  NOT NULL         ,
    etc…
    PRIMARY KEY (XMITQ, CHLNAME, REMOTEQMGR, QSGNAME) ,
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
    ON DELETE CASCADE
    )
    IN MQDB1.MQTS2;
CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SSKT_IX1
    ON CSQ.ADMIN_B_SSKT
    (XMITQ ASC, CHLNAME ASC, REMOTEQMGR ASC, QSGNAME ASC)
    USING STOGROUP MQDEFLT
    PRIQTY 72Ø  SECQTY 72Ø  FREEPAGE 5  PCTFREE  3Ø CLOSE NO;
```

5 Bind the DB2 plans for the queue manager, utilities, and channel initiator. Your plan names may be different.

```
DSN SYSTEM(DB2A)
BIND PLAN(CSQ5D22Ø) MEM(CSQ5D22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5L22Ø) MEM(CSQ5L22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5R22Ø) MEM(CSQ5R22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5U22Ø) MEM(CSQ5U22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5W22Ø) MEM(CSQ5W22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5B22Ø) MEM(CSQ5B22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
```

```
BIND PLAN(CSQ5222Ø) MEM(CSQ5222Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5S22Ø) MEM(CSQ5S22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
BIND PLAN(CSQ5K22Ø) MEM(CSQ5K22Ø) ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) ACT(REP) RETAIN ISOLATION(CS)  -
LIB('xxxxxxxx.SCSQDEFS')
```

6 Grant 'execute' authority to the respective plans for the user IDs that will be used by the queue manager, utilities, and channel initiator. The user IDs for the queue manager and channel initiator are the user IDs under which their started task procedures run. The user IDs for the utilities are the user IDs under which the batch jobs can be submitted. Here, we chose to use PUBLIC for ease of set-up.

```
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
GRANT EXECUTE ON PLAN CSQ522Ø TO PUBLIC;
```

Should anything go wrong during the set-up, you can drop the tables, tablespaces, storage group, and the database, and start again. For further information, please refer to the *DB2 for OS/390 Administration Guide*.

7 Add a QSG record (eg QSG1) into the table *CSQ.ADMIN_B_QSG* (DB2GRP1=DB2 data-sharing group; DBT1=DB2 group attach name).

```
     //ADDQSG   EXEC PGM=CSQ5PQSG,REGION=4M,
//         PARM='ADD QSG,QSG1,DB2GRP1,DBT1'
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DISP=SHR,DSN=xxxxxxxx.SCSQANLE
//         DD DISP=SHR,DSN=xxxxxxxx.SCSQAUTH
//         DD DISP=SHR,DSN=xxxxxxxx.SCSQLOAD
//         DD DISP=SHR,DSN=xxxxxxxx.DB2LOAD
```

8 Add a queue manager record into the table *CSQ.ADMIN_B_QMGR*. Repeat for each queue manager in the

QSG (here MQT1 and MQT2).

```
          //ADDQMGR1 EXEC PGM=CSQ5PQSG,REGION=4M,
     //         PARM='ADD QMGR,MQT1,QSG1,DB2GRP1,DBT1'
     //SYSPRINT DD SYSOUT=*
     //STEPLIB  DD DISP=SHR,DSN=xxxxxxxx.SCSQANLE
     //         DD DISP=SHR,DSN=xxxxxxxx.SCSQAUTH
     //         DD DISP=SHR,DSN=xxxxxxxx.SCSQLOAD
     //         DD DISP=SHR,DSN=xxxxxxxx.DB2LOAD
          //*
          //ADDQMGR2 EXEC PGM=CSQ5PQSG,REGION=4M,
     //         PARM='ADD QMGR,MQT2,QSG1,DB2GRP1,DBT1'
     //SYSPRINT DD SYSOUT=*
//STEPLIB  DD DISP=SHR,DSN=xxxxxxxx.SCSQANLE
     //         DD DISP=SHR,DSN=xxxxxxxx.SCSQAUTH
     //         DD DISP=SHR,DSN=xxxxxxxx.SCSQLOAD
     //         DD DISP=SHR,DSN=xxxxxxxx.DB2LOAD
```

DB2 COMMANDS SHOWING MQ CONNECTIONS (DB2 SUBSYSTEMS DBS1,DBS2,DBS3)

```
-DBS1 DISPLAY THREAD(*)
——————————-
E NCAC7    DSNV4Ø1I -DBS1 DISPLAY THREAD REPORT FOLLOWS -
" NCAC7
DSNV4Ø2I -DBS1 ACTIVE THREADS -
NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
RRSAF    N       12 MQT1DB2MSTØØ MQT1MSTR          ØØ66     Ø
RRSAF    N      367 MQT1DB2SRVØ1 MQT1MSTR          ØØ66     Ø
RRSAF    N     8442 MQT1DB2SRVØ2 MQT1MSTR          ØØ66     Ø
RRSAF    N       19 MQT1DB2SRVØ3 MQT1MSTR          ØØ66     Ø
RRSAF    N        7 MQT1DB2SRVØ4 MQT1MSTR          ØØ66     Ø
DISPLAY ACTIVE REPORT COMPLETE
E NCAC7    DSN9Ø22I -DBS1 DSNVDT '-DISPLAY THREAD' NORMAL COMPLETION

-DBS2 DISPLAY THREAD(*)
——————————-
E NCAC7    DSNV4Ø1I -DBS2 DISPLAY THREAD REPORT FOLLOWS -
" NCAC7
DSNV4Ø2I -DBS2 ACTIVE THREADS -
NAME     ST A   REQ ID           AUTHID   PLAN     ASID TOKEN
RRSAF    N       12 MQT2DB2MSTØØ MQT2MSTR          Ø1F6     Ø
RRSAF    N      157 MQT2DB2SRVØ1 MQT2MSTR          Ø1F6     Ø
RRSAF    N     8495 MQT2DB2SRVØ2 MQT2MSTR          Ø1F6     Ø
RRSAF    N        1 MQT2DB2SRVØ3 MQT2MSTR          Ø1F6     Ø
RRSAF    N        7 MQT2DB2SRVØ4 MQT2MSTR          Ø1F6     Ø
DISPLAY ACTIVE REPORT COMPLETE
E NCAC7    DSN9Ø22I -DBS2 DSNVDT '-DISPLAY THREAD' NORMAL COMPLETION
-DBS3 DISPLAY THREAD(*)
——————————-
```

```
E NCAC7    DSNV4Ø1I -DBS3 DISPLAY THREAD REPORT FOLLOWS -
E NCAC7    DSNV419I -DBS3 NO CONNECTIONS FOUND
E NCAC7    DSN9Ø22I -DBS3 DSNVDT '-DISPLAY THREAD' NORMAL COMPLETION
```

MQSERIES COMMAND SHOWING DB2 AND CF DETAILS

DISPLAY GROUP shows details of the QSG, the DB2 connection, and the size of the CF structures.

THE ROLE OF THE COUPLING FACILITY (CF)

A Coupling Facility is a special logical partition (LPAR) that provides high-speed cacheing, list processing, and locking functions within a sysplex.

For those not familiar with the Coupling Facility (available since MVS 5.1), in the context of this article, think of it as a large data-store, available to all OS/390 images in a sysplex. Data is stored in ‘structures’, of which there are three types: cacheing, queueing, and locking. It is not new, and some examples of products that use the CF include DB2, CICS transaction server (TS) for its log streams, IMS for data sharing (and shared message queues), and the OS/390 Automatic Restart Manager (ARM).

The CF must be at level 9 in order to support the structures used by MQSeries (and OS/390 V2.9) (see Figure 2). The CF is extremely stable, partly because no user software runs inside it.

Despite that fact, it is recommended that the CF’s power supply be backed-up (with, for example, battery power). However, should the CF fail, a secondary CF can take over its functions, thereby further improving availability. Our tests showed that, after a CF was ‘reset’, all structures were copied to the back-up CF.

Although this implementation of shared queues only allows non-persistent messages, it does mean that a queue manager restart (controlled or otherwise) does not lose non-persistent messages because they are kept in the CF. This may well be good enough for some installations provided they can live with the shared queue maximum size restriction of 63K.

STEPS REQUIRED TO USE THE COUPLING FACILITY

1 Define the structures. Use the **IXCMIAPU** utility to define the CF structures used by the queue managers in the QSG. They are stored in the Coupling Facility Resource Management (CFRM) policy data set. All structures for the QSG start with the name of the queue-sharing group. Two types are required:

- A single administrative structure with fixed name *qsg-name*CSQ_ADMIN. This structure is used by MQSeries itself, and does not contain any user data.
- One or more structures (up to 63) used to hold messages for shared queues.

  These can have any name you choose up to 16 characters long, again, starting with a 4-character QSG name (QSG1 in example below), whilst the fifth character must be alphabetic; subsequent characters can be alphabetic or numeric. Characters 5-16 comprise what is specified for the **CFSTRUCT** attribute when defining a shared queue (**APPLIC1** in example below).

  (Note that, when defining a shared queue, MQ does not check whether the **CFSTRUCT** is valid. It is only when writing to the shared queue that this check is made; if it is incorrect, you cannot change the queue definition. Instead, delete and redefine it.)

Read the *MQSeries for OS/390 Concepts and Planning Guide* to decide how large to make the structures (to store 200,000 messages of size 1K would require approximately 256MB of CF storage).

Example utility statements – see *thlqual.SCSQPROC(CSQ4CFRM)*:

```
STRUCTURE NAME(QSG1CSQ_ADMIN)
SIZE(10000)
INITSIZE(10000)              (size(Kb) initially alloc)
PREFLIST(CF01)
STRUCTURE NAME(QSG1APPLIC1)
SIZE(10000)
INITSIZE(10000)
PREFLIST(CF01)
```

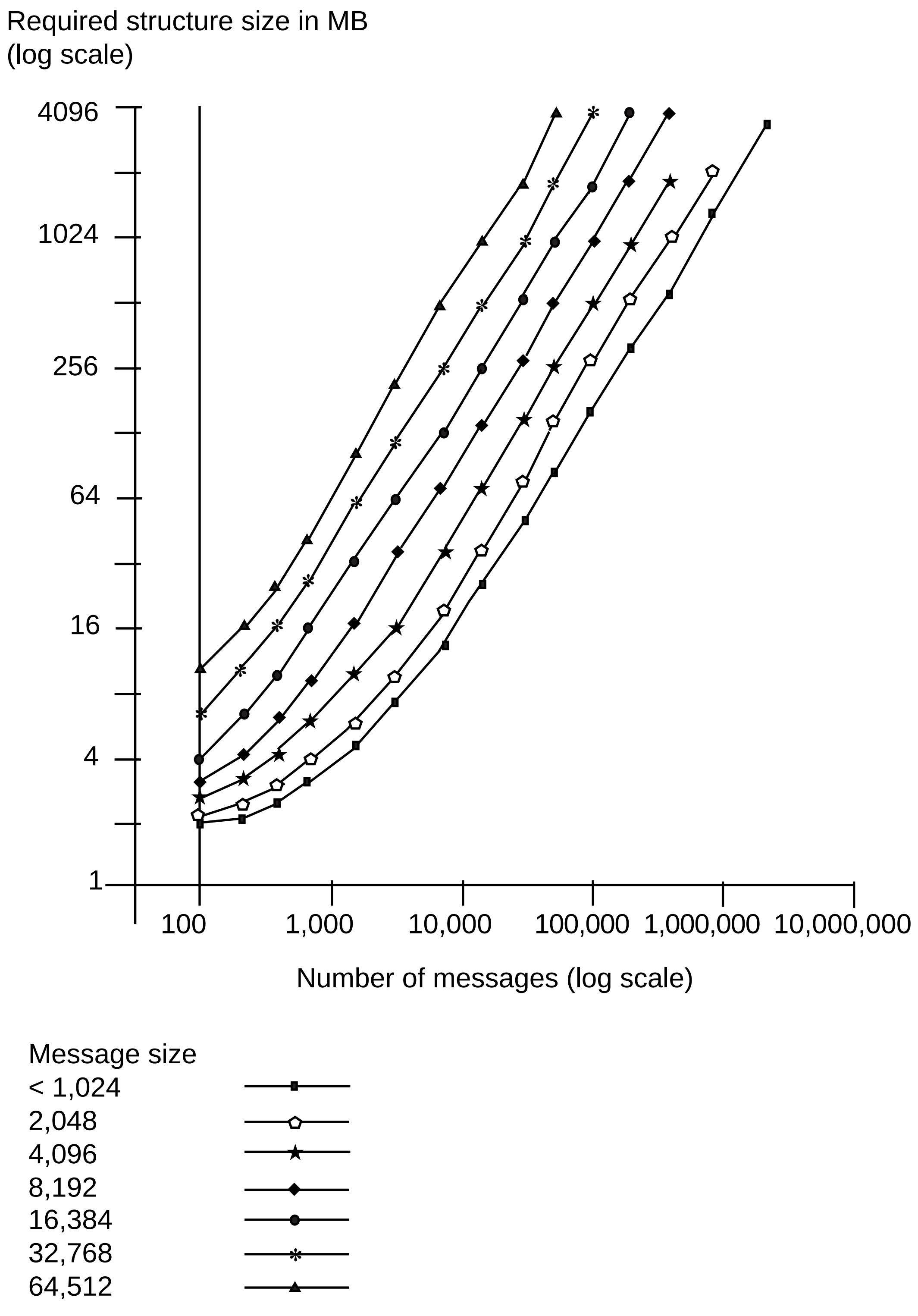

Figure 2: MQSeries CF structure allocation size, OS/390 R9 CFCC level 9

2 Activate the CFRM policy. Issue the following OS/390 command:

```
SETXCF START,POLICY,TYPE=CFRM,POLNAME=policy-name
```

3 Amend security definitions. Grant the queue manager **USERID ALTER** authority to profile:

```
IXLSTR.structure-name in RACF class FACILITY.
```

COUPLING FACILITY COMMANDS

There are, in fact, two sets of display commands: the **D CF** command actually interrogates the hardware, whereas the **D XCF** command looks at the Coupling Facility datasets. It is, therefore, possible to get information using the **D XCF** commands, even if the CF is down.

1 Display name of the CF (and its back-up, if available):

```
d xcf,cf
```

2 Display the couple datasets:

```
d xcf,couple
```

3 Display the CF policy:

```
d xcf,policy
```

4 Display all the structure names:

```
d xcf,structure
```

5 Display the names of the LPARs in the sysplex:

```
d xcf,sysplex
```

6 Display a specific structure in detail:

```
d xcf,structure,strname=QSG1APPLIC1    (Application structure)
STRNAME         : QSG1APPLIC1
CFNAME          : CFØ1B           ﬂ back-up CF
COUPLING FACILITY: SIMDEV.IBM.EN.ØØØØØØØCFØ1B
        PARTITION: Ø   CPCID: ØØ
ACTUAL SIZE     : 1Ø24Ø K
STORAGE INCREMENT SIZE: 256 K
PHYSICAL VERSION: B4895D61 9B54A984
LOGICAL  VERSION: B4895D61 9B54A984
SYSTEM-MANAGED PROCESS LEVEL: 9
XCF GRPNAME     : IXCLOØ29
DISPOSITION     : KEEP
ACCESS TIME     : NOLIMIT
```

```
MAX CONNECTIONS: 32
# CONNECTIONS  : 2
CONNECTION NAME  ID VERSION  SYSNAME  JOBNAME  ASID STATE
________ _ ____ ____ ____ __ ___
CSQELPR1MQT1Ø1   Ø2 ØØØ2ØØØ6 LPR1     MQT1MSTR ØØ66 ACTIVE
CSQELPR1MQT2Ø2   Ø1 ØØØ1ØØØ8 LPR1     MQT2MSTR Ø1F6 ACTIVE
d xcf,structure,strname=QSG1CSQ_ADMIN (System Admin structure)
STRNAME     : QSG1CSQ_ADMIN
CFNAME         : CFØ1            ﬂ main CF
COUPLING FACILITY: SIMDEV.IBM.EN.ØØØØØØØØCFØ1
        PARTITION: Ø   CPCID: ØØ
ACTUAL SIZE    : 1Ø24Ø K
STORAGE INCREMENT SIZE: 256 K
PHYSICAL VERSION: B4895D23 64712F82
LOGICAL  VERSION: B4895D23 64712F82
SYSTEM-MANAGED PROCESS LEVEL: 9
XCF GRPNAME    : IXCLOØ28
DISPOSITION    : KEEP
ACCESS TIME    : NOLIMIT
MAX CONNECTIONS: 32
# CONNECTIONS  : 2
CONNECTION NAME  ID VERSION  SYSNAME  JOBNAME  ASID STATE
________ _ ____ ____ ____ __ ___
CSQELPR1MQT1Ø1   Ø2 ØØØ2ØØØ5 LPR1     MQT1MSTR ØØ66 ACTIVE
CSQELPR1MQT2Ø2   Ø1 ØØØ1ØØØE LPR1     MQT2MSTR Ø1F6 ACTIVE
```

7 To delete a structure, issue both these commands in order. (Obtain advice first, as this will delete your shared queue data.)

```
setxcf force,con,strnm=QSG1APPLIC1,connm=all
setxcf force,structure,strnm=QSG1APPLIC1
```

CONCLUSION

DB2 and the Coupling Facility are essential components to support the new shared queues facility. It does, indeed, further improve the availability of MQSeries by decoupling shared queues from a single queue manager. Shared queue definitions, as well as data, are available to all queue managers in the queue sharing group (QSG), providing ‘pull’ workload balancing.

IT departments wishing to use the shared queues must be very familiar with both DB2 and the Coupling Facility, and must adjust their operational procedures to take account of the new inter-dependencies.

Ruud van Zundert,
Independent Consultant (UK)

Transaction integrity in a message-oriented world

The subject of transaction integrity is a complex one. If you take a room full of IT people and ask them to write down what is meant by a transaction you would end up with a collection of different answers.

The 'classic' definition is that a transaction is a single 'unit of work'. Again, this term is often unfamiliar to those not brought up on CICS or IMS systems. IBM coined the term 'ACID' for the properties of a true transaction, meaning it has to have:

- Atomicity.
- Consistency.
- Isolation.
- Durability.

But first let's look at why transaction integrity is so important.

Take, for example, a banking transaction which involves the movement of money from account 'A' to account 'B'. What happens if the transaction fails during the update? A classic mainframe transaction is designed so that either the money stays in account 'A' or arrives successfully in account 'B', and is never just lost in 'cyberspace' if the first debit worked and the second credit failed. This is achieved by performing both the debit and credit database updates under a single unit of work, so they either both get completed or both are backed-out to a point of consistency.

That's 'old hat' to CICS/DB2 programmers, who simply allow CICS to act as the transaction manager to coordinate the database updates. If MQSeries messages are used, then they can become part of the same unit of work (make sure they are persistent messages). But is it that simple to include MQSeries in this scenario? What happens if the transaction to update the other database (initiated by the MQSeries message) fails to execute successfully when it gets there?

The problem with MQSeries-initiated updates is that you are only ensuring the commitment of messages onto the MQSeries queue in the original unit of work, and not ensuring the update actually works, because it might run on another platform at another time. We are now in the scary world of distributed transactions! One way to deal with this is to perform remote database updates using a distributed transaction coordinator. MQSeries itself (not the OS/390 version) can act as an XA-compliant transaction coordinator.

This allows distributed databases to participate in a single unit of work, so that your single transaction can update these databases and still provide the traditional commit and roll-back logic that we are used to with CICS/DB2. You have to make sure that all the databases involved support the use of an XA transaction coordinator. Personally, I have not seen this done often, and I don’t think it is particularly well understood in the industry. One requirement, for example, is that all the systems have some awareness of each other, and that’s unusual with the rapid development needed for e-business projects.

If we assume that XA coordination is not the easy option, where does that leave us in the modern world of multiple platforms interconnected by MQSeries? You may start to hear from IT designers the term ‘compensating transaction’. This means that, if a business update spans two or more units of work and the second or subsequent fails, you issue a compensating transaction to effectively ‘roll-back’ the earlier one by reversing the update on the database.

This means a lot of extra work to code the compensating transactions and don’t dare ask what happens if they fail! If you have several levels of updates it gets even more difficult to control.

Designing mission-critical systems to perform updates of vital data over multiple units of work is something to be avoided if at all possible. If it has to happen, then consider how you are going to keep track of the state of the business updates. Daisy-chaining systems together is not a good idea if the topology changes, so we are now naturally heading towards an integration hub.

MQSeries Systems Integrator

Enter MQSeries Systems Integrator Version 2 (MQSI). I won’t explain what MQSI is, since you can find the sales material on the IBM Web site. But does it solve all these transactional problems?

Well, not really. MQSI uses MQSeries for all its inputs and most outputs, so it can use MQSeries units of work for internal consistency of MQSI ‘process flows’. So what’s the transaction issue now?

Consider system ‘A’ sending an MQSeries message to the MQSI hub, which in turn, sends updates to systems ‘B’ and ‘C’. Is this one unit of work? No, unfortunately, it is four. The first UOW must be committed to allow the input message to appear on the MQSI input queue. MQSI reads it under another UOW and decides to generate two output messages. This second UOW must commit for the two output messages to appear on the MQSI output queues.

Then, systems ‘B’ and ‘C’ each deal with their own MQSeries input messages under their own UOWs. But if the database update on system ‘C’ fails there is no automatic way to back-out the corresponding update on system ‘B’, let alone all the way back to the originating system ‘A’.

If you question designers at this point they tend to respond by saying that the failing system will back-out its MQSeries input messages and they will then be manually detected and corrected.

This sounds easier than it really is to implement in a high-volume production environment. The skills required to determine the state of an update become incredibly high, and systems management is hard to do. You might get past the development stage of a project and then find your actual customers expose the inadequacies of this design in a big way.

What’s really needed is a way to maintain the state of the overall business update in the hub, so that corrective processes can be automated centrally. MQSI can update DB/2 directly from its process flows and this is probably the best way to deal with this requirement, although you don’t need MQSI to build this logic yourself if your integration requirements are relatively simple. System management tools could be interfaced to this central state repository.

One way to implement this would be to record the status of system ‘B’ and ‘C’ updates as ‘pending’ in the state repository, and await a confirmation message from each system before marking them as ‘complete’. When all updated systems have completed, the state record might be discarded. IBM is developing (but has not yet released) an ‘aggregation’ node for MQSI V2, which might be ideal for implementing this without writing your own code.

You might be asking, if we are recording the ‘state’ of a business update, then is this workflow? I would say no, if the updates involve disparate IT systems and occur over a relatively short time interval. However, if you want to include manual procedures (ie people) or long time intervals, then you are starting to design a workflow and should consider a workflow product to help you.

SUMMING UP

This article has described the need for update process flow coordination when distributed systems are interconnected by MQSeries messages and the updates are not performed under a single unit of work. To decide how this affects your systems design you should take a look at your business data updates and draw the transactional unit of work boundaries. These are the boundaries within which the updates are automatically backed-out if the transaction fails, without the use of additional ‘compensatory’ transactions.

When you find (as you surely will) that you no longer have the single unit of work transactional integrity that most of us (and most business analysts) take for granted, start asking hard questions about what happens when (and not if) things go wrong at each point. Any reliance on manual problem detection and correction should be exposed as a risk that could cause you major problems. Make sure that any design has the ability to scale without requiring massive manual intervention effort, or causing you system performance problems.

MQSI certainly has a potential role to play in solving this issue and its internal transactional integrity is welcome (and lacking in some EAI competitors). But don’t assume that the implementation of MQSI will miraculously make all of your distributed systems act as one. That silver bullet has yet to be invented. In the meantime, let your CICS or DB systems programmer take a look at your e-business designs and ask the necessary questions before you go live!

Peter Toogood ©Xephon

MQSeries quick start

INTRODUCTION

When MQSeries is first used, there are several key areas that require fairly specialized expertise in order to implement message-based systems:

- MQ installation, sizing and configuration.
- MQ operational support and troubleshooting.
- MQ application development (client and server components).

All of these key areas attract various packaged solutions or practices that can ease initial or ongoing pressures. This article will examine the third of these challenges, exploring the early complexities of MQ application development as well as common practices that can help reduce ongoing skills costs.

This article does not attempt to provide an alternative API with which to access MQSeries, as technologies such as AMI and JMS already exist in this arena. Instead, it examines useful strategies and short cuts that can be employed to boost the productivity of developers in the difficult early days of MQSeries development.

Included are Java code samples that can be used as a basis for initial proof of concept or prototyping exercises. They are implemented on top of the basic MQI (in light of personal experience). Many readers may prefer to implement helper facilities on top of one of the higher-level APIs available. The code is illustrative rather than a serious implementation. Although it will run (on most Windows and Unix platforms), its primary goal is to illustrate some of the practices described herein. Additionally, the Java implementation is purely a personal preference – most of the techniques described are applicable to other languages (although some areas do have an object-oriented slant).

HIGH-LEVEL STEPS

As with many other technologies, MQSeries development can be simplified in a number of ways, for example by:

- Wrapping the MQSeries API to make it behave in a way more familiar to developers (eg the JMS).
- Providing default values for MQ API calls, thereby reducing the depth to which developers need to understand all options.
- Allowing parameterization of variable MQSeries attributes (eg queue names), reducing ‘hard-coding’ and hence, the amount of maintenance that must be carried out on MQ applications.
- Providing shared components to carry out supporting functions for applications, such as error recovery and logging.

All of these techniques can be used in conjunction with each other, leading to a situation where less MQ-specific application code needs to be written, maintenance tasks are required less frequently, and fewer developers will require in-depth knowledge of MQSeries.

This lends itself to a scenario in which a shared MQSeries support team can be assembled to provide application development support and utilities. This might also overlap with operational support and configuration capability, all contained within a single MQSeries ‘centre of excellence’.

The four options outlined above will now be examined in more detail.

DETAILED QUICK START TECHNIQUES

Classes of service

The central premise behind providing quick start development shortcuts with MQSeries is the understanding that, often, there are only four basic tasks that MQSeries is used to fulfil:

- Non-persistent client.
- Non-persistent server.
- Persistent client.
- Persistent server.

Non-persistent client

In this scenario, MQSeries functionality is used to make requests that are serviced by a partner system. Typically, messages are put onto queues as soon as all requisite data is available, and reply messages are waited for (in a multi-user environment, reply messages are tied back to requests by propagation of the request message id to the correlation id or message id of the reply message).

The retrieval of replies will often not occur immediately after request generation, but may be delayed until a critical point has been reached and no further processing can be undertaken without the reply (hence, hiding the response time of a partner system).

Non-persistent server

In this scenario, a long-running task dedicates one or more processing threads to requesting messages from a queue. Messages are processed (in whatever manner is appropriate) and replies are placed onto an appropriate queue. The message id of the request is propagated to the message id or correlation id of the reply. While messages are present on the request queue, no server ‘worker’ threads will remain idle; ie messages are replied to as soon as possible.

A variation on the server theme involves use of MQSeries application triggering. Triggering on the arrival of every message can be expensive in many environments (in terms of initializing the server application) and should be avoided in all but the most infrequently used servers. Triggering on the arrival of the first message (ie every time the request queue depth changes from 0 to 1) is less expensive, as the server will generally run until a queue is emptied. This article assumes triggering is used only on the arrival of the first message, or not at all.

Persistent client

The persistent client is usually used to instigate functionality that is performed offline in ‘fire-and-forget’ mode. While there are some scenarios in which replies are generated in response to persistent client messages, this is uncommon. If message replies are required, it is often more appropriate to use non-persistent messaging and cater for message loss as a foreseen exception condition.

Persistent server

The persistent server is similar to the non-persistent server in that it dedicates one or more activity thread to the retrieval of messages from a queue. It may also be triggered or submitted by some external mechanism (eg a task scheduler).

The persistent server differs from the non-persistent server in two key areas:

- All messages are accepted under syncpoint.
- No replies are generated.

Some systems require further messages to be generated by a persistent server. Often, this can be viewed more clearly as a server that also includes a client component. (As in the case of multi-step chains of fire-and-forget persistent server components.)

External syncpoint coordinators (eg CICS) are often used to execute commit or roll-back functionality. Messages that have been backed-out several times (ie have failed and been re-delivered to the server) must be side-lined in a safe manner so as to avoid clogging-up the server. This necessities manual interaction to remedy such cases. One effective means of achieving this is for the persistent server to act as a persistent client to a side-lined 'poisoned letter' queue.

Using these scenarios, various 'classes of service' can be defined and generic solutions can be written around them.

Wrapping the MQI

There are a number of benefits that can be realized by wrapping the standard MQI. This is not to suggest that the API provided is inherently complex, rather that, as a generic tool, it can benefit from customization. The AMI is an example of an MQSeries wrapper that has been provided to simplify some of the laborious tasks involved in MQI utilization. High-level APIs such as this are, however, still generic in nature, and although less development is required to wrap them, benefit can still be drawn from a thin abstraction and defaulting layer.

The primary goal of any abstraction layer must be simplification of the development process. In MQSeries, this can be achieved by minimizing the number of steps that are required in the implementation of a messaging system. To understand this more fully, consider the following processes.

Initialization

Responsibilities:

- Set up various MQ client environmental parameters.
- Establish connection to a queue manager.
- Open requisite queue(s).

Operation

Responsibilities:

- Set up various put or get message options.
- Execute MQ puts or gets.
- Handle commit or rollback.
- Propagate message and correlation ids.

Error conditions

Responsibilities:

- Attempt to re-establish communications with a queue manager.
- Side-line messages with an unacceptable back-out count.

Termination

Responsibilities:

- Close all queues.
- Disconnect from the queue manager.

In a high level MQ wrapper, the four steps (above) can be exposed to developers without the need to understand the underlying MQ tasks (note that these four steps differ between the various classes of service). An object-oriented wrapper might even include initialization and termination tasks in its constructor and destructor. As a result, only a handful of functions/methods need to be called by the developer. The following Java interface could be used to describe an extremely simple MQ wrapper:

```
public interface IMQWrapper {
     public void write(String message) throws MessagingException;
     public String read() throws MessagingException;
}
Note: The MessagingException class is a wrapper for
com.ibm.mq.MQException.
```

This approach is useful, as it relies on clearly understood programming techniques, such as reading and writing data. Alternatively, an existing object or interface (in Java) could be extended to provide MQ functionality. Such approaches could make use of well-known abstract classes such as *java.io.Reader* or *java.io.OutputStream*.

Another benefit associated with a high-level wrapper such as this (or indeed JMS), is the encapsulation of the actual underlying messaging system responsible for transmission of messages. In a diverse messaging environment, MQSeries could sit alongside many other protocols and share a common interface. Applications need not explicitly cater for MQSeries, hence increasing their re-usability.

Basic wrapper implementation

The MQAdapter class (see sample code) is capable of performing basic messaging functions to support the non-persistent client and server classes of service.

Defaulting common parameters

MQSeries is, undoubtedly, a functionally-rich product, and as such, the array of options presented to a new developer can be somewhat overwhelming. The MQSeries constants interface, for example, (MQC.class in Java) contains well over 300 options for use in various MQI functions.

It is likely that most organizations will use fewer than 20 of these constants, so providing defaults values for key MQI calls should save significant development time. Taking the non-persistent client as an example class of service, the ‘get’ message options could comprise the following parameters:

```
MQGMO_NO_SYNCPOINT
MQGMO_CONVERT
MQGMO_FAIL_IF_QUIESCING
MQGMO_WAIT
```

These can be combined and defined as a constant, eg in a C header file with **#define** instructions, or in Java, by declaring an interface with final static members to hold all default values.

Defaulting implementation

The MQConstants interface in the enclosed Java example contains examples of default 'get' and 'put' message options for a non-persistent client and server.

Parameterization of variables

As in most development environments, the 'hard-coding' of parameters is undesirable as it leads to brittle applications that require frequent maintenance (including recompilation and testing) to cater for the most trivial of environmental changes. This is especially true in an environment such as MQSeries, where there is an extremely large number of messaging options available.

A good approach to developing flexible MQSeries applications (or wrappers) is to understand which parameters are likely to change, and which are not. By appreciating the distinction between these areas, parameter files (or similar mechanisms/repositories) needn't be bogged down with excessive detail. Another good approach is to provide a hard-coded default operation and allow the overriding of this, when required, via a parameter file. An obvious example of this behaviour is the specification of TCP/IP port number for a queue manager; the default (1414) is used so frequently that it can be excluded from most parameter files. It need only be specified in the unusual case that an alternative is required.

Frequently changed parameters

The following items will probably change often in the life of an MQSeries application or wrapper (as a result of it being redeployed with alternative queue managers or queues):

- Queue manager name.

- Queue manager hostname.
- Client channel name.
- Request queue name.
- Reply queue name.

The following items may change less frequently in light of specific networking or queue manager configuration:

- Queue manager port.
- Message expiry interval.
- Get (with wait) timeout.

Other items are usually less likely to change during the lifetime of an application. They could be omitted from the parameterization process at first, and only included as and when their use becomes necessary.

Parameterization implementation

The MQConstants interface (enclosed) provides key names, which can be entered into a parameter file, eg:

```
public final static String CHANNEL_NAME = "ChannelName";
```

The MQAdapter class defines default values, eg:

```
private String iChannelName  = "SYSTEM.DEF.SVRCONN";
```

The **overrideDefaults** method of MQAdapter shows how default parameters may be replaced by those supplied in a parameter file (or a *java.util.Properties* object, in this case). It should be noted that this method is somewhat tedious to implement, hence the exclusion of properties unlikely to change.

Provision of shared utilities and supporting functionality

The final area of assistance that can be provided for the inexperienced MQ developer is the development (or procurement) of supporting functionality, such as error handlers and simple logging mechanisms.

Logging

The simplest (and perhaps most useful) of these utilities is a well-implemented logging strategy. Features of this should include:

- Varied output media (eg to file or system console).
- A simple interface that can be readily used throughout applications.
- The time-stamping of all log entries.
- The ability to class the criticality of log messages (eg from trace information, through warning, to severe error messages).
- The ability to exclude classes of log messages (eg exclude all but severe errors) so as not to impact the performance of live systems.
- The ability to readily vary the level of logging, ideally, without restarting applications.

Exception handling

Focusing, as before, on the Java MQ client, all MQSeries messaging errors are reported to the controlling application by throwing MQExceptions. This class contains various items of useful information, most notably the reason code to describe the exception. If a utility is supplied to intercept MQException objects (or to simply interpret reason codes arising from failures), several useful facilities can be provided.

Firstly, MQExceptions fall into several categories, including:

- Parameter errors: ie errors encountered during development that occur because of improper or malformed calls to the API in use.
- Recoverable errors: ie errors that pertain to individual messages or errors that indicate no message has arrived within an acceptable time period.
- Fatal errors: ie errors that affect the clients' ability to carry out messaging, for example a failure of the network connection to the queue manager.

An MQException wrapper could be provided to intercept common MQ reason codes and categorize them according to the classes above. It could also provide an English translation for common reason codes to ease debugging effort (eg by interfacing with an IBM error message support pack). Finally, the exception wrapper could automatically write log records on encountering errors, thus ensuring that all MQExceptions are correctly logged for subsequent investigation.

Recovery

One final supporting process that can be implemented is the automatic recovery of MQ resources from failure conditions. The objective of this is to simplify the provision of a high-availability, MQ-dependent system. Recovery strategies can be proactive or reactive, but generally include the ability to terminate MQSeries connectivity and attempt re-initialization.

Proactive recovery strategies could include a dedicated 'heartbeat' function. This could include the exchange of a dummy request-reply message pair, possibly running periodically in a separate thread (although thread affinity of MQ resources may complicate this in some environments). Failure in the heartbeat process could trigger recovery activity in the main application. A reactive recovery strategy is generally simpler than a proactive one. It merely triggers recovery mechanisms on encountering fatal MQExceptions or reason codes.

Supporting functions implemented

Few supporting functions have been implemented in the Java example. This is largely because of the monotonous nature of classifying large numbers of MQ reason codes and the abundance of third-party or freeware logging facilities. The only recovery mechanism that has been implemented is the basic ability to discard and recreate an MQAdapter object on encountering failures. This is not particularly efficient or proactive, but will cater for the common scenario in which networking failures or queue manager maintenance necessitates the interruption of messaging services.

DESCRIPTION OF EXAMPLE JAVA SOURCE SUPPLIED

Adaptor classes

The adaptor classes, MQAdapter, and its supporting MQAdapterException class and MQConstants interface, comprise a primitive MQSeries Java client wrapper. The key behaviour of these objects is controlled by MQAdapter, and includes the following steps:

- Instantiation using a properties object.

- Set-up of default MQ options, as defined in the MQConstants interface.
- The overriding of any default parameters as specified in the properties object.
- Connection to an MQSeries queue manager.
- Opening of a request and reply queue pair.
- Servicing of write and read calls, as requested by the calling application.
- Graceful closing of MQSeries objects when the adaptor is destroyed.

Example server

The example server demonstrates how quickly an MQSeries server can be developed for prototyping purposes. It creates an MQAdapter object on instantiation, and then uses it from the **doServerProcessing** method.

Little processing is carried out here, except the reading and writing of messages using the MQAdapter. The server will run until an exception is encountered (most commonly, 'no message available' on the request queue). It prints all messages that it receives (from clients) and generates a simple reply message containing an incremented counter.

Example client

The example client is as simple as the example server, as it also carries out very little processing apart from the utilization of the MQAdapter 'read' and 'write' methods (in the **doClientMessagePair** method).

The client generates messages, hence driving the example server. Client messages contain another incremented counter. Reply messages are printed to the system console. It should be noted that multiple clients can connect to one server. The server will accept messages from both clients, and replies will be sent back as usual. Each client will, however, display non-consecutive reply messages, as their requests are interwoven with their partner's messages.

Properties files used

The example supplied will run with only the queue manager name/ location properties and queue names supplied. To experiment with multiple clients communicating with a single server, it is beneficial to increase the 'WaitInterval' parameter for the server. This prevents **MQRC_NO_MSG_AVAILABLE** exceptions being thrown, and, hence, the server terminating.

Runtime requirements

All sample code was developed in Java 1.2 and should run on all platforms compatible with this, as well as Java 1.1.6 through 1.1.8. The ExampleClient and ExampleServer classes have main methods that can be run via the **java.exe launcher**, eg:

```
Java com.dmitri.firstmq.ExampleServer
```

Note that the properties files *server.prop* and *client.prop* must be present in the Java classpath in order to override the default MQSeries options:

- *Server.prop*

  ```
  OutputQueueName=EXAMPLE.REPLY
  InputQueueName=EXAMPLE.REQUEST
  ```

- *Client.prop*

  ```
  InputQueueName=EXAMPLE.REPLY
  OutputQueueName=EXAMPLE.REQUEST
  ```

CONCLUSIONS

This article has outlined several simple but useful steps that can be taken to ease an organization's first encounter with MQSeries. Its focus has been on minimizing the development effort (and indepth MQSeries expertise) that is required of developers. The cost of this is an initial investment in various helper facilities that may subsequently be shared by many developments.

It is hoped that the basic MQAdapter class (supplied) demonstrates that the initial investment in MQSeries does not have to be particularly large, and that a productive MQSeries development environment is achievable in a short period of time.

EXAMPLECLIENT.JAVA

```
package com.dmitri.firstmq;

/**
 * A very basic implementation of an MQ client using the MQAdapter
 * helper. This client sends a very simple message to a server and waits
 * for a reply.
 */
public class ExampleClient {
private MQAdapter iAdaptor = MQAdapter.newAdapter("client.prop", false);
       private int iCounter = Ø;
/**
 * ExampleClient constructor.
 */
public ExampleClient() {
       super();
}
/**
 * This method calls the MQAdapter to send a message to a server, then
 * wait for a response.
 */
public void doClientMessagePair() {
       try {
              iAdaptor.write(generateMessage());
              processReply(iAdaptor.read());
       } catch (MQAdapterException mqaEx) {
              // Not Implemented
       } catch (java.io.IOException ioEx) {
              // Not Implemented
       }
}
/**
 * Create an example message to send to the server.
 * @return java.lang.String
 */
private String generateMessage() {
       return "Example request message number : " + iCounter++;
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {
       ExampleClient cli = new ExampleClient();
       for (int i = Ø; i < 1ØØ; i++)
              cli.doClientMessagePair();
}
/**
 * Take some action based on the contents of a reply message. In this
 * case, simply log it.
 * @param reply java.lang.String
 */
private void processReply(String reply) {
       System.out.println("Message received : " + reply);
}
}
```

EXAMPLESERVER.JAVA

```
package com.dmitri.firstmq;
/**
 * A very basic implementation of an MQ server using the MQAdapter
 * helper. This server receives MQ messages and replies to them with a
 * simple count message.
 */
public class ExampleServer {
 private MQAdapter iAdaptor = MQAdapter.newAdapter("server.prop", true);
 private int iCounter = 0;
/**
 * ExampleServer constructor.
 */
public ExampleServer() {
       super();
}
/**
 * This method is used to process incoming request messages. It will
 * loop until no further messages remain to be processed. Since this is
 * a server, it will require a longer wait interval than the client.
 * This enables it to remain operational for use by multiple clients.
 */
public void doServerProcessing() {
       try {
              for (;;) {
                     String request = iAdaptor.read();
                     processRequest(request);
                     iAdaptor.write(generateMessage());
              }
       } catch (MQAdapterException mqaEx) {
              // Not Implemented
       } catch (java.io.IOException ioEx) {
              // Not Implemented
       }
}
/**
 * Create an example message to return to the client.
 * @return java.lang.String
 */
private String generateMessage() {
       return "Example reply message number : " + iCounter++;
}
/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(java.lang.String[] args) {

       new ExampleServer().doServerProcessing();
}
/**
 * Take some action based on the contents of a request message. In this
 * case, simply log it.
 * @param reply java.lang.String
 */
private void processRequest(String request) {
       System.out.println("Message received : " + request);
}
}
```

MQADAPTEREXCEPTION.JAVA

```
package com.dmitri.firstmq;

import com.ibm.mq.*;/**
 * A simple exception class to wrap around all MQ exceptions. This class
 * could be extended to provide further advice to calling applications,
 * such as:
 *             Whether the exception is retryable
 *             An English translation of the error
 *             Automatic logging.
 *
 */
public class MQAdapterException extends Exception {
       private int iReasonCode;/**
 * This constructor stores MQ reason code for subsequent (unimplemented)
 * analysis.
 * @param mqEx com.ibm.mq.MQException
 */
public MQAdapterException(MQException mqEx) {
     super(mqEx.toString());
     iReasonCode = mqEx.reasonCode;
     // This could be a good place to log the exception.
}/**
 * @return int
 */
public int getReasonCode() {
     return iReasonCode;
}
}
```

MQADAPTOR.JAVA

```
package com.dmitri.firstmq;

import com.ibm.mq.*;
import java.util.*;
import java.io.*;
/**
 * This class provides a high-level abstraction layer for non-persistent
 * MQSeries messaging. It aims to default all options (as appropriate)
 * and provide a quick introduction to MQSeries messaging. This is aimed
 * at initial prototyping excercises with the intention of reducing
 * early MQSeries learning curves for otherwise experienced developers.
 */
public class MQAdapter implements MQConstants {
     private MQQueueManager iQmgr;
     private MQQueue iInputQ;
     private MQQueue iOutputQ;
     // Some default values that may be overridden in the property
     // object supplied at instantiation
     private String iQmgrname         = "";
     private String iHostname         = "localhost";
     private int iHostPort            = 1414;
     private String iChannelName      = "SYSTEM.DEF.SVRCONN";
     private String iInputQName       = "SYSTEM.DEFAULT.LOCAL.QUEUE";
     private String iOutputQName      = "SYSTEM.DEFAULT.LOCAL.QUEUE";
     private int iWaitInterval        = 30000;    // milliseconds
     private int      iExpiryInterval = 30000;
     // note, MQ uses tenths of a second, so this figure (currently
     // in milliseconds) is divided by 100 before use. For this
     // example, message Id will always be propagated to correlationId.
     private byte iMessageIdStore[];
     // Server mode or client mode?
     private boolean iServerMode;
     private MQPutMessageOptions iPMO = new MQPutMessageOptions();
     private MQGetMessageOptions iGMO = new MQGetMessageOptions();
/**
 * This constructor reads default parameters for the MQSeries session,
 * then connects to a queue manager and opens a pair of queues (request
 * and reply). The simple Boolean flag determines whether the connection
 * is in server more (true) or client mode (false).
 * @param overrideProps java.util.Properties
 * @param serverMode boolean
 */
public MQAdapter(Properties overrideProps, boolean serverMode) {
     super();
     iServerMode = serverMode;
     overrideDefaults(overrideProps);
     // Define MQ sonnectivity parameters
     MQEnvironment.hostname = iHostname;
     MQEnvironment.port = iHostPort;
     MQEnvironment.channel = iChannelName;
     // Set up basic put and get message options
     iPMO.options = DEFAULT_PUT_OPTIONS;
     iGMO.options = DEFAULT_GET_OPTIONS;
     iGMO.waitInterval = iWaitInterval;
     // Establish MQ connection and open queues
     try {
      iQmgr = new MQQueueManager(iQmgrname);
      iInputQ = iQmgr.accessQueue(iInputQName, MQC.MQOO_INPUT_AS_Q_DEF);
      iOutputQ = iQmgr.accessQueue(iOutputQName, MQC.MQOO_OUTPUT);
     } catch (MQException mqEx) {
            mqEx.printStackTrace();
     }
}
/**
 * Closes all MQ resources.
 */
public void finalize() {
     try {
            iOutputQ.close();
            iInputQ.close();
            iQmgr.disconnect();
     // Nothing can be done about exceptions here.
     } catch (MQException ignored) {}
}
/**
 * Factory method that loads a property object before delegating all
 * initialization to the main constructor.
 *
 * @param propName java.lang.String
 * @param serverMode boolean
 * @return MQAdapter a newly instantiated adapter object.
 */
public static MQAdapter newAdapter(String propName, boolean serverMode)
{
    Properties prop = new Properties();
      try {
           // Assumes that the properties file is somewhere
           //in the current classpath.
      InputStream str = ClassLoader.getSystemResourceAsStream(propName);
             prop.load(str);
             str.close();
      } catch (IOException ioEx) {
           // No override properties available.
      } catch (NullPointerException npEx) {}
           // No override properties available.
      return new MQAdapter(prop, serverMode);
}
/**
 * This method extracts user-defined properties from the supplied
 * properties object to replace the default (hard-coded) MQ options.
 * @param fromProps java.util.Properties
 */
private void overrideDefaults(Properties fromProps) {
 if (fromProps == null || fromProps.size() == 0) return;
 // nothing overridden
 if (fromProps.getProperty(CHANNEL_NAME) != null)
       iChannelName = fromProps.getProperty(CHANNEL_NAME);
 if (fromProps.getProperty(EXPIRY_INTERVAL) != null)
 iExpiryInterval =
 //Integer.parseInt(fromProps.getProperty(EXPIRY_INTERVAL));
 if (fromProps.getProperty(HOST_NAME) != null)
        iHostname = fromProps.getProperty(HOST_NAME);
 if (fromProps.getProperty(HOST_PORT) != null)
        iHostPort = Integer.parseInt(fromProps.getProperty(HOST_PORT));
 if (fromProps.getProperty(INPUT_QUEUE_NAME) != null)
        iInputQName = fromProps.getProperty(INPUT_QUEUE_NAME);
 if (fromProps.getProperty(OUTPUT_QUEUE_NAME) != null)
        iOutputQName = fromProps.getProperty(OUTPUT_QUEUE_NAME);
 if (fromProps.getProperty(QUEUE_MGR_NAME) != null)
        iQmgrname = fromProps.getProperty(QUEUE_MGR_NAME);
 if (fromProps.getProperty(WAIT_INTERVAL) != null)
 iWaitInterval = Integer.parseInt(fromProps.getProperty(WAIT_INTERVAL));
}
/**
 * This method reads messages from the current queue. Messages are
 * returned as simple strings.
 * @return java.lang.String
 * @exception com.dmitri.firstmq.MQAdapterException a wrapped
 * MQException
 * @exception java.io.IOException - doesn't usually occur
 */
public String read() throws MQAdapterException, IOException {
       MQMessage mess = new MQMessage();
       if (!iServerMode) mess.correlationId = iMessageIdStore;
       try {
              // On some older queue managers, it is necessary to
              // specify max message
              // size to ensure correct translation of large messages:
              // iInputQ.get(mess, iGMO, MAX_MESSAGE_LENGTH);
              iInputQ.get(mess, iGMO);
              if (iServerMode) iMessageIdStore = mess.messageId;
              return mess.readUTF();
       } catch (MQException mqEx) {
              throw new MQAdapterException(mqEx);
       }
}
/**
 * Writes a message to the output queue.
 *
 * @param message java.lang.String
 * @exception com.dmitri.firstmq.MQAdapterException
 * The wrapped MQException
 * @exception java.io.IOException - doesn't usually occur
 */
public void write(String message) throws MQAdapterException, IOException
{
       MQMessage mess = new MQMessage();
       if (iServerMode) mess.correlationId = iMessageIdStore;
       mess.expiry = iExpiryInterval / 100;
       try {
              mess.writeUTF(message);
              iOutputQ.put(mess, iPMO);
              if (!iServerMode) iMessageIdStore = mess.messageId;
       } catch (MQException mqEx) {
              throw new MQAdapterException(mqEx);
       }
}
}
```

MQCONSTANTS.JAVA

```
package com.dmitri.firstmq;

import com.ibm.mq.*;
public interface MQConstants {

  public final static String CHANNEL_NAME = "ChannelName";
  public final static String EXPIRY_INTERVAL = "ExpiryInterval";
  public final static String HOST_NAME = "HostName";
  public final static String HOST_PORT = "HostPort";
  public final static String INPUT_QUEUE_NAME = "InputQueueName";
  public final static String OUTPUT_QUEUE_NAME = "OutputQueueName";
  public final static String QUEUE_MGR_NAME = "QueueManagerName";
  public final static String WAIT_INTERVAL = "WaitInterval";
  public final static int DEFAULT_PUT_OPTIONS = MQC.MQPMO_NO_SYNCPOINT |
       MQC.MQPMO_FAIL_IF_QUIESCING;
  public final static int DEFAULT_GET_OPTIONS = MQC.MQGMO_NO_SYNCPOINT |
       MQC.MQGMO_CONVERT | MQC.MQGMO_FAIL_IF_QUIESCING | MQC.MQGMO_WAIT;
}
```

Dominic Blatchford (UK) © Xephon 2001

MQ news

IBM Business Partner Evolutionary Technologies International (ETI) has recently introduced ETI•EXTRACT, which is designed to enable companies to automate the process of data collection, transformation, and migration between incompatible systems in heterogeneous computing environments. It is claimed that, in comparison with hand-coding, savings in resources expended can exceed a ratio of 40-to-1.

ETI•EXTRACT keeps a metadata audit trail of work carried out, and provides users with a means for impact analysis so they can quickly see what interfaces and adapters are affected when something changes.

ETI•EXTRACT adds to the existing stable of EAI (enterprise application integration) tools, which also includes the recently enhanced ETI•Accelerator for MQSeries. This is claimed to reduce significantly the time and cost required to build an EAI infrastructure for MQSeries users, by enabling the process of writing adapters to be automated.

For further information contact:
ETI, 816 Congress Ave, Suite 1300, Frost Bank Plaza, Austin, TX 78701, USA.
Tel: +1 512 383 3000
Fax: +1 512 383 3300
Web: http://www.ETI.com

* * *

Willow Technology's MQSeries products are now being sold by IBM as part of its reseller programme. Specifically, the family now includes MQSeries for DYNIX/ptx and MQSeries Clients for SCO OpenServer, UnixWare, SGI IRIX, MPE/iX, and Mac OS.

The client version of selected products extends use to individual workstations not dependent on a server environment.

Each product is compatible with MQSeries V2 level, while the version for DYNIX/ptx is a compatible MQSeries V5 level server. Also, they interoperate with all MQSeries V1, V2, and V5 client and server products available from IBM and Willow.

For further information contact:
Willow Technologies Inc, PO Box 320005, Los Gatos, CA 95032, USA
Tel: +1 408 377 7292
Fax: +1 408 377 7293
Web: http://www.willowtech.com

* * *