16

MQ update

October 2000

In this issue

MQ Update

Published by

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

North American office

Xephon/QNA
Post Office Box 350100
Westminster CO 80035-0100, USA
Telephone: +1 303 410 9344
Fax: +1 303 438 0290

Contributions

Articles published in *MQ Update* are paid for at the rate of £170 ($250) per 1000 words and £90 ($140) per 100 lines of code. For more information about contributing an article, please check Xephon's Web site, where you can download *Notes for Contributors*.

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.

Editor

Harry Lewis

Subscriptions and back-issues

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

An introduction to MQSeries installable services

INSTALLABLE SERVICE TYPES

Most MQSeries administrators should be familiar with the Object Authority Manager (OAM), the MQSeries component that controls access by applications to queue manager resources. The OAM is an example of an installable service – it's effectively a plug-in component of the queue manager and its plug-in interface is the installable services API described in the *Programmable Systems Management Guide*.

There are three types of installable service described by the installable services API:

- An *authorization service*, which mediates application access to MQSeries Objects (queues, processes, and the queue manager itself).
- A *name service*, which provides a look-up mechanism for obtaining the queue manager name on which a particular remote queue is defined, and is used to identity the name of a local transmission queue for routing messages.
- A *userid service*, which provides a user-id and password that are associated with an application (this service applies to MQSeries for OS/2 Warp only).

Of these, the authorization service is probably the one of greatest general interest. To implement a user-written authorization service implies replacing the OAM, and this offers many possibilities, from building a bridge that connects to an existing non-MQSeries authorization service to employing a distributed authorization service, perhaps based on a repository that's accessed via the network.

THE AUTHORIZATION SERVICE INTERFACE

The installable services interface consists of three sets of function definitions, one for each type of installable service (authorization, name, and userid). An installable service implements the set of functions corresponding to its service type, and registers pointers to these functions during initialization in order to make them known to the queue manager. The registration is performed using an additional installable services function, *MQZEP*, which associates a function pointer with the appropriate function identifier defined in the MQSeries include file *cmqz.h*.

Initialization and termination

The identification of a particular module that is to serve as an installable services component is controlled by *Service* and *ServiceComponent* entries in the queue manager configuration file. The queue manager knows the location of installable service functions only after the installable service invokes the function *MQZEP* to register them. *MQZEP* calls are initiated using the function that is declared as the external entry point of the module that implements the service, which must (for an authorization service) conform with the *MQZ_INIT_AUTHORITY* signature. For other service types, *MQZ_INIT_NAME* and *MQZ_INIT_USERID* actually have equivalent signatures.

The service initialization and termination functions are each called in two different modes, where the mode – *primary* or *secondary* – is indicated by the *Options* parameter that's passed to the function. Primary initialization and termination each occur only once during the lifetime of an installable service component – they correspond to the initialization and termination of the queue manager itself.

Secondary initialization and termination may occur many times during the lifetime of the queue manager, and correspond to the initialization and termination of individual processes or threads. The *Hconfig* parameter that's passed during initialization and termination can be used by the service component to correlate a pair of initialization and termination calls.

By including more than one *Service* and *ServiceComponent* entry in the queue manager configuration file, a number of instances of an installable service can be 'chained'. Each function invoked on a particular instance is then able to indicate to the queue manager whether another service in the chain should be invoked by setting the *pContinuation* parameter, if present, in the installable service function call. A trap for the unwary is that the default setting of this parameter, which may be either *MQZCI_CONTINUE* or *MQZCI_STOP*, varies between different functions.

Authorization service operation

For an authorization service, the minimum set of functions that must be implemented is:

```
MQZ_INIT_AUTHORITY

MQZ_TERM_AUTHORITY

MQZ_CHECK_AUTHORITY or MQZ_CHECK_AUTHORITY_2
```

These three functions (counting the two variants of *CHECK_AUTHORITY* as one) are invoked on an authorization service by the queue manager during normal application activity.

The main task of an authorization service is to implement either *MQZ_CHECK_AUTHORITY* or *MQZ_CHECK_AUTHORITY_2*, which determines whether access from a specified entity to a specified MQSeries object is permitted. The definition of the authorization service interface assumes that the entity is a user-id (principal) or group, with an *entityType* value of *MQZAET_PRINCIPAL* or *MQZAET_GROUP* respectively. The implementor of the service is, however, ultimately responsible for determining which of the underlying authorization mechanisms to apply and how to treat user-ids and groups.

A further set of functions defines the interface that's used for administering the permissions managed by the authorization service. These are invoked as a result of administrative activity, such as creating and deleting queues and using the administration commands **dspmqaut** and **setmqaut**. The functions are:

```
MQZ_GET_AUTHORITY or MQZ_GET_AUTHORITY_2

MQZ_GET_EXPLICIT_AUTHORITY or MQZ_GET_EXPLICIT_AUTHORITY_2

MQZ_SET_AUTHORITY or MQZ_SET_AUTHORITY_2

MQZ_COPY_ALL_AUTHORITY

MQZ_DELETE_AUTHORITY
```

If an authorization service is designed to be administered through some mechanism other than calling the MQSeries **dspmqaut** and **setmqaut** commands, then it need not implement this second set of functions.

Authorization service functions

MQZ_INIT_AUTHORITY

This function is called once when the queue manager is started, and thereafter each time an *MQCONN* is performed on the queue manager. The function prototype is shown below.

```
void MQENTRY MQZ_INIT_AUTHORITY (MQHCONFIG Hconfig,
         MQLONG Options, MQCHAR48 pQMgrName,
         MQLONG ComponentDataLength, PMQBYTE pComponentData,
         PMQLONG pVersion, PMQLONG pCompCode, PMQLONG pReason);
```

The first time this function is called, *Options* is set to *MQZIO_PRIMARY*. Thereafter it's set to *MQZIO_SECONDARY*. The *Hconfig* parameter is associated with the connection – it can be used to correlate a later call to *MQZ_TERM_AUTHORITY* with a call to *MQZ_INIT_AUTHORITY*. This would free the resources associated with a particular connection.

The size of the component data area is defined in the *ServiceComponent* entry of the queue manager configuration file. This data area is allocated by the queue manager and made available on every call to the installable service. It provides a mechanism for storing arbitrary data that persists from call to call. If the service does not use the component data area, the area can be configured with a size of zero bytes.

By setting the value of the pointer *pReason* to either *MQRC_SERVICE_NOT_AVAILABLE* or *MQRC_INITIALIZATION_FAILED*, the code that implements the service can indicate to the queue manager that it is unable to execute.

MQZ_TERM_AUTHORITY

This function is called with the *Options* parameter set to *MQZIO_SECONDARY* by each application when the application's connection to the queue manager terminates (a secondary termination), and is called once with the *Options* parameter set to *MQZIO_PRIMARY* when the queue manager itself terminates (a primary termination).

```
void MQENTRY MQZ_TERM_AUTHORITY (MQHCONFIG Hconfig,
    MQLONG Options, PMQCHAR pQMgrName, PMQBYTE pComponentData,
    PMQLONG pCompCode, PMQLONG pReason);
```

MQZ_CHECK_AUTHORITY and MQZ_CHECK_AUTHORITY_2

Either variant of this function is invoked by the queue manager to provide the functionality of an authorization service. The two variants differ only in the way the name of the entity is passed, and only one variant is used in the lifetime of the service. *MQZ_CHECK_AUTHORITY* is passed the name of the entity as a string (*pEntityName*), as shown below.

```
void MQENTRY MQZ_CHECK_AUTHORITY (PMQCHAR pQMgrName,
    PMQCHAR pEntityName, MQLONG EntityType,
    PMQCHAR pObjectName, MQLONG ObjectType, MQLONG Authority,
    PMQBYTE pComponentData, PMQLONG pContinuation,
    PMQLONG pCompCode, PMQLONG pReason);
```

MQZ_CHECK_AUTHORITY_2 takes an *MQZED* structure instead of a string:

```
void MQENTRY MQZ_CHECK_AUTHORITY_2 (PMQCHAR pQMgrName,
    PMQCHAR pEntityName, PMQZED pEntityData,
    PMQCHAR pObjectName, MQLONG ObjectType, MQLONG Authority,
    PMQBYTE pComponentData, PMQLONG pContinuation,
    PMQLONG pCompCode, PMQLONG pReason);
```

The *MQZED* structure contains fields for a domain and security identifier, in addition to the field that stores the entity name. These fields are intended for use in *MQSeries* for Windows NT, though the *MQZ_CHECK_AUTHORITY_2* function and *MQZED* structure are also found on Unix systems, where the domain and security identifier fields are not applicable.

Both variants of this function use the *MQZID_CHECK_AUTHORITY* identifier, which means that only one can be registered with *MQZEP*. *MQZ_INIT_AUTHORITY*'s *pVersion* parameter identifies which variant is used – *MQZAS_VERSION_1* specifies *MQZ_CHECK_AUTHORITY* and *MQZAS_VERSION_2* specifies *MQZ_CHECK_AUTHORITY_2*.

The check authority function’s task is to check whether an entity identified in *pEntityName* (or *pEntityData->pEntityNamePtr*) is permitted the type of access described by the authority parameter of the MQSeries object *pObjectName*. The result of the check is assigned to both the completion code and reason code variables (pointers to these variables are passed as *pCompCode* and *pReason* respectively). Only three reason codes – *MQRC_NOT_AUTHORIZED*, *MQRC_SERVICE_NOT_AVAILABLE*, and *MQRC_SERVICE_ERROR* – are applicable. In addition, the reason code *MQRC_NONE* indicates that the check was successful.

MQZ_GET_AUTHORITY and MQZ_GET_EXPLICIT_AUTHORITY

These two functions and their variants, *MQZ_GET_AUTHORITY_2* and *MQZ_GET_EXPLICIT_AUTHORITY_2* respectively, which take an input of type *MQZED* instead of a string and are not shown below, are used in administrative operations to discover the authority over a particular object that’s assigned to the entity specified.

```
void MQENTRY MQZ_GET_AUTHORITY (PMQCHAR pQMgrName,
    PMQCHAR pEntityName, MQLONG EntityType,
    PMQCHAR pObjectName, MQLONG ObjectType,
    PMQLONG pAuthority,
    PMQBYTE pComponentData, PMQLONG pContinuation,
    PMQLONG pCompCode, PMQLONG pReason);

void MQENTRY MQZ_GET_EXPLICIT_AUTHORITY (PMQCHAR pQMgrName,
    PMQCHAR pEntityName, MQLONG EntityType,
    PMQCHAR pObjectName, MQLONG ObjectType,
    PMQLONG pAuthority, MQLONG AuthorityMask,
    PMQBYTE pComponentData, PMQLONG pContinuation,
    PMQLONG pCompCode, PMQLONG pReason);
```

Both functions return a collection of *MQZAO* constants (described later) that apply to the entity and the specified MQSeries object. The constants are returned in the location to which the pointer *pAuthority* points. However, the documented behaviour differs subtly between the two functions:

- *MQZ_GET_EXPLICIT_AUTHORITY* masks the returned authority value using the mask in the *AuthorityMask* parameter, which is not present in *MQZ_GET_AUTHORITY*.
- *MQZ_GET_EXPLICIT_AUTHORITY* returns the authority of the principal “without the additional authority of the *nobody*

group”. (In the OAM authorization service implementation, permissions associated with the *nobody* group are used for checking the authority of entities that have no specifically assigned permissions. This may not be appropriate for other implementations.)

MQZ_COPY_ALL_AUTHORITY

This administrative function assigns the authority associated with another object to an MQSeries object.

MQZ_DELETE_AUTHORITY

This function deletes all permissions associated with a specified MQSeries object.

Authority flags

The authority parameter is a combination of *MQZAO** flags, which precisely describe the type of access being requested. There are two

MQZAO MQI group constant	**MQOO constant**
MQZAO_CONNECT	–
MQZAO_BROWSE	MQOO_BROWSE
MQZAO_INPUT	MQOO_INPUT_AS_Q_DEF
	MQOO_INPUT_SHARED
	MQOO_INPUT_EXCLUSIVE
MQZAO_OUTPUT	MQOO_OUTPUT
MQZAO_INQUIRE	MQOO_INQUIRE
MQZAO_SET	MQOO_SET
MQZAO_PASS_IDENTITY_CONTEXT	MQOO_PASS_IDENTITY_CONTEXT
MQZAO_PASS_ALL_CONTEXT	MQOO_PASS_ALL_CONTEXT
MQZAO_SET_IDENTITY_CONTEXT	MQOO_SET_IDENTITY_CONTEXT
MQZAO_SET_ALL_CONTEXT	MQOO_SET_ALL_CONTEXT
MQZAO_ALTERNATE_USER_AUTHORITY	MQOO_ALTERNATE_USER_AUTHORITY

Figure 1: MQZAO constants and their MQOO equivalents

groups of *MQZAO* flags, of which the MQI flags (covered by the mask *MQZAO_ALL_MQI*) correspond closely to the *MQOO* options supplied to the *MQOPEN* call. This is no coincidence – the *MQZ_CHECK_AUTHORITY* function is invoked by the queue manager with a set of *MQZAO* flags each time an *MQOPEN* is attempted. The other *MQZAO* constant in the MQI flags is *MQZAO_CONNECT*, which is specified during an *MQCONN* attempt.

The other group of *MQZAO* flags – the administration flags covered by the *MQZAO_ALL_ADMIN* mask – appear in calls to *MQZ_GET_AUTHORITY*, *MQZ_GET_EXPLICIT_AUTHORITY*, and *MQZ_SET_AUTHORITY* as the result of running the **dspmqaut** and **setmqaut** commands and other administration operations.

A SIMPLE FILE-BASED AUTHORIZATION SERVICE

This section provides an example using a simple file-based authorization service implemented on AIX. The example service uses rules defined in a text file when assigning permissions to MQSeries objects. A rule occupies a single line in the text file and conforms with the following format:

```
object-name [entity:[r][w][x]] [...]
```

The meaning of *r*, *w*, and *x* in the context of the example service is as follows:

- *r* specifies read permission and applies to the constants *MQZAO_BROWSE*, *MQZAO_INPUT*, *MQZAO_INQUIRE*, *MQZAO_DISPLAY*, and *MQZAO_DISPLAY_STATUS*.
- *w* specifies write permission and applies to the constants *MQZAO_OUTPUT*, *MQZAO_SET*, *MQZAO_CREATE*, *MQZAO_DELETE*, *MQZAO_CHANGE*, and *MQZAO_CLEAR*.
- *x* specifies execute permission and applies to the constants *MQZAO_START_STOP*, *MQZAO_RESOLVE_RESET*, and *MQZAO_PING*.

MQZAO_CONNECT authority is provided if any of the three permissions (*r*, *w*, or *x*) are present.

An asterisk (‘*’) can be used as a wildcard to refer to all objects or all entities. Specific permissions take precedence over wildcard permissions, and no distinction is made between users and groups.

Here are three example rules (one on each line):

```
     REQUEST.Q mqm:rwx service:rw bob:rw
     SYSTEM.DEAD.LETTER.Q mqm:rwx *:rw
     * mqm:rwx
```

The first rule sets the permissions for the queue *REQUEST.Q* and assigns permissions *rwx* to the user (or group) *mqm*, permissions *rw* to *service*, and permissions *rw* to *bob*. The second rule assigns *rwx* permission to *mqm* and *rw* to all others for queue *SYSTEM.DEAD.LETTER.QUEUE*. The third rule assigns *rwx* permissions to *mqm* for all objects not named explicitly in a rule.

The example service logs details of all accesses and access attempts (the source code for it, *fileauthservice.c*, is listed below). For simplicity’s sake, it’s presented as a single source file, and there is no corresponding include file. Note the use of the continuation character, ‘➤’, in the code below to indicate a formatting line break that’s not present in either the original source code or the code that can be downloaded from Xephon’s Web site (*http://www.xephon.com/mqupdate.html*).

FILEAUTHSERVICE.C

```
#pragma options CPLUSCMT

#define _THREAD_SAFE

#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <time.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/mode.h>
#include <search.h>
#include <cmqc.h>
#include <cmqzc.h>

#define MQCHARS "abcdefghijklmnopqrstuvwxyz" \
   "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
   "0123456789" \
   "./_%"
#define CONFIG_FILE "/var/mqm/fileauthservice.conf"
#define LOG_FILE "/var/mqm/fileauthservice.log"

// Prototypes for installable service functions

void mqz_init_authority (MQHCONFIG config, MQLONG options, MQCHAR48 qm,
   MQLONG dataLength, PMQBYTE data, PMQLONG version,
   PMQLONG cc, PMQLONG rc);
void mqz_term_authority (MQHCONFIG config, MQLONG options, PMQCHAR qm,
   PMQBYTE data, PMQLONG cc, PMQLONG rc);
void mqz_check_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_check_authority_2 (PMQCHAR qm,
   PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);

void mqz_get_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, PMQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_get_authority_2 (PMQCHAR qm,
   PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, PMQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_get_explicit_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType,
   PMQLONG authority, MQLONG authorityMask,
   PMQBYTE componentData,
   PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_get_explicit_authority_2 (PMQCHAR qm,
   PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType,
   PMQLONG authority, MQLONG authorityMask,
   PMQBYTE componentData,
   PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_set_authority (PMQCHAR qm, PMQCHAR entityName, MQLONG
➤   entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_set_authority_2 (PMQCHAR qm, PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_copy_all_authority (PMQCHAR qm, PMQCHAR refObjectName,
   PMQCHAR objectName, MQLONG objectType,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc);
void mqz_delete_authority (PMQCHAR qm, PMQCHAR objectName, MQLONG
➤   objectType,
   PMQBYTE data, PMQLONG continuation, PMQLONG cc, PMQLONG rc);

// Prototypes for other functions

int init_file_authority (char *filename);
void check_file_authority (PMQCHAR entityName,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQLONG cc, PMQLONG rc);
void check_auth (char * permissions, MQLONG authority,
   PMQLONG cc, PMQLONG rc);

// Prototypes for utility functions

size_t mqstrlen (const char *string);
char *mqstrncpy (char *dest, const char *src, size_t number);
void replace (char *string, char a, char b);
int log (char *message, ...);

// Hashtable used to store authorization entries for look-up

struct hsearch_data hashtable = { NULL, 0, 0, 0, 0 };

// Buffer contains configuration data from file

char *buffer = NULL;

// File descriptor for log file

int logfd;

// Entry point of the installable service function MQZ_INIT_AUTHORITY

void mqz_init_authority (MQHCONFIG config, MQLONG options, MQCHAR48 qm,
   MQLONG dataLength, PMQBYTE data, PMQLONG version,
   PMQLONG cc, PMQLONG rc)
{
   if (options == MQZIO_PRIMARY)
```

```
    {
       logfd = open (LOG_FILE, O_WRONLY | O_CREAT | O_APPEND,
          S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
       log ("fileauthservice initialized (%s last modified %s)\n",
          __FILE__, __TIMESTAMP__);
    }
    else
    {
       logfd = open (LOG_FILE, O_WRONLY | O_APPEND, 0);
    }

    MQZEP (config, MQZID_INIT_AUTHORITY,
       (PMQFUNC) mqz_init_authority, cc, rc);
    MQZEP (config, MQZID_TERM_AUTHORITY,
       (PMQFUNC) mqz_term_authority, cc, rc);
    MQZEP (config, MQZID_COPY_ALL_AUTHORITY,
       (PMQFUNC) mqz_copy_all_authority, cc, rc);
    MQZEP (config, MQZID_DELETE_AUTHORITY,
       (PMQFUNC) mqz_delete_authority, cc, rc);

    if (*version == MQZAS_VERSION_2)
    {
       MQZEP (config, MQZID_CHECK_AUTHORITY,
          (PMQFUNC) mqz_check_authority_2, cc, rc);
       MQZEP (config, MQZID_GET_AUTHORITY,
          (PMQFUNC) mqz_get_authority_2, cc, rc);
       MQZEP (config, MQZID_GET_EXPLICIT_AUTHORITY,
          (PMQFUNC) mqz_get_explicit_authority_2, cc, rc);
       MQZEP (config, MQZID_SET_AUTHORITY,
          (PMQFUNC) mqz_set_authority_2, cc, rc);
    }
    else
    {
       MQZEP (config, MQZID_CHECK_AUTHORITY,
          (PMQFUNC) mqz_check_authority, cc, rc);
       MQZEP (config, MQZID_GET_AUTHORITY,
         (PMQFUNC) mqz_get_authority, cc, rc);
       MQZEP (config, MQZID_GET_EXPLICIT_AUTHORITY,
          (PMQFUNC) mqz_get_explicit_authority, cc, rc);
       MQZEP (config, MQZID_SET_AUTHORITY,
          (PMQFUNC) mqz_set_authority, cc, rc);
    }

    if (init_file_authority (CONFIG_FILE) < 0)
    {
       *cc = MQCC_FAILED;
       *rc = MQRC_SERVICE_ERROR;
    }
}
```

```
// Installable service function MQZ_TERM_AUTHORITY

void mqz_term_authority (MQHCONFIG config, MQLONG options, PMQCHAR qm,
   PMQBYTE data, PMQLONG cc, PMQLONG rc)
{
   if (buffer)
   {
      free (buffer);
   }

   if (options == MQZIO_PRIMARY)
   {
      log ("fileauthservice terminated.\n");
   }

   close (logfd);
}

// Installable service function MQZ_CHECK_AUTHORITY

void mqz_check_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   check_file_authority (entityName, objectName, objectType,
      authority, cc, rc);
}

// Installable service function MQZ_CHECK_AUTHORITY_2

void mqz_check_authority_2 (PMQCHAR qm,
   PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   check_file_authority (entity->EntityNamePtr, objectName, objectType,
      authority, cc, rc);
}

// Installable service function MQZ_GET_AUTHORITY
// (not implemented in this example)

void mqz_get_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, PMQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
```

```
{
   return;
}

// Installable service function MQZ_GET_AUTHORITY
// (not implemented in this example)

void mqz_get_authority_2 (PMQCHAR qm,
   PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, PMQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   return;
}

// Installable service function MQZ_GET_EXPLICIT AUTHORITY
// (not implemented in this example)

void mqz_get_explicit_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType,
   PMQLONG authority, MQLONG authorityMask,
   PMQBYTE componentData,
   PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   return;
}

// Installable service function MQZ_GET_EXPLICIT_AUTHORITY_2
// (not implemented in this example)

void mqz_get_explicit_authority_2 (PMQCHAR qm,
   PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType,
   PMQLONG authority, MQLONG authorityMask,
   PMQBYTE componentData,
   PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   return;
}

// Installable service function MQZ_SET_AUTHORITY
// (not implemented in this example)

void mqz_set_authority (PMQCHAR qm,
   PMQCHAR entityName, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   return;
}

// Installable service function MQZ_SET_AUTHORITY_2
// (not implemented in this example)

void mqz_set_authority_2 (PMQCHAR qm, PMQZED entity, MQLONG entityType,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   return;
}

// Installable service function MQZ_COPY_ALL_AUTHORITY
// (not implemented in this example)

void mqz_copy_all_authority (PMQCHAR qm, PMQCHAR refObjectName,
   PMQCHAR objectName, MQLONG objectType,
   PMQBYTE componentData, PMQLONG continuation,
   PMQLONG cc, PMQLONG rc)
{
   return;
}

// Installable service function MQZ_DELETE_AUTHORITY
// (not implemented in this example)

void mqz_delete_authority (PMQCHAR qm,
   PMQCHAR objectName, MQLONG objectType,
   PMQBYTE data, PMQLONG continuation, PMQLONG cc, PMQLONG rc)
{
   return;
}

// This function builds a hashtable of authorization entries
// from a configuration file that contains lines in the format:
//
// object-name [entity":"["r"]["w"]["x"]] [...]
//
// Examples:
// REQUEST.Q alice:rw bob:rw carol:r
// SYSTEM.DEAD.LETTER.QUEUE mqm:rwx *:rw
// * mqm:rwx

int init_file_authority (char *filename)
{
```

```
static char *buffer;

struct stat file_info;
char *start;
char *next_line;
char *comment;
int fd;
int size;
int entries = 0;

// Open the configuration file and find its size.

if ((fd = open (filename, O_RDONLY, 0)) == -1 ||
   fstat (fd, &file_info) == -1)
{
   return -1;
}

// Allocate a buffer for the file contents and read them

if ((buffer = (char *) malloc (file_info.st_size + 1)) &&
   (size = read (fd, buffer, file_info.st_size)) ==
   ➤  file_info.st_size)
{
   *(buffer + size) = '\0';
   start = buffer;
   close (fd);
}
else
{
   close (fd);

   return -2;
}

// Create a hashtable. As a rough estimate, we'll guess there are
// size/20 entries (average 20 chars per line). The size we request
// doesn't have to be exactly right -- it will grow if necessary.

if (!hcreate_r ((size / 20) + 1, &hashtable))
{
   return -3;
}

// Create a hashtable entry for each valid authorization line
// in the file.

while (start)
{
   // Find the end of the line and terminate it.
```

```
      if (next_line = strchr (start, '\n'))
      {
         *next_line++ = '\0';
      }
      else
      {
         next_line = NULL;
      }

      // If there's a comment, terminate the line there.

      if ((comment = strchr (start, '#')))
      {
         *comment = '\0';
      }

      // Split the line into { key value } where key is the first token
      // and value is separated from key by whitespace.

      if ((start = strtok (start, " \t")))
      {
         ENTRY entry, *result;

         entry.key = start;
         entry.data = strtok (NULL, "\0");

         if (!hsearch_r (entry, ENTER, &result, &hashtable))
         {
            entries++;
         }
         else
         {
            // Insert into hashtable failed.

            return -4;
         }
      }

      start = next_line;
   }

   // Return the number of entries added to the hashtable (for info).

   return entries;
}

// This function checks whether entityName has the specified authority
// over objectName. It looks up objectName in the hashtable and
// searches for permissions flags associated with entityName.
```

```
void check_file_authority (PMQCHAR entityName,
   PMQCHAR objectName, MQLONG objectType, MQLONG authority,
   PMQLONG cc, PMQLONG rc)
{
   char target [MQ_Q_NAME_LENGTH + 1]; // + 1: trailing null
   char entity [MQ_USER_ID_LENGTH + 2]; // + 2: trailing colon and null
   char *permissions = NULL;
   ENTRY named = { target, NULL };
   ENTRY unnamed = { "*", NULL };
   ENTRY *result;

   // Get null-terminated copies of the objectName and entityName.

   mqstrncpy (target, objectName, MQ_Q_NAME_LENGTH);
   mqstrncpy (entity, entityName, MQ_USER_ID_LENGTH);
   strcat (entity, ":");

   // We do a hashtable look-up using the target (the object name) as
   // the key. If the object has no specific named entry, we try "*"
   // instead.

   if (!hsearch_r (named, FIND, &result, &hashtable) ||
   !hsearch_r (unnamed, FIND, &result, &hashtable))
   {
      // We look for a token entity: in the result.

      permissions = strstr (result->data, entity);

      // If this entity does not appear, check for wildcard
      // permissions.

      if (!permissions)
      {
         permissions = strstr (result->data, "*:");
      }
   }

   // Perform the check with whatever flags we found (if any) and log
   // the result

   check_auth (permissions, authority, cc, rc);
   log ("%s %s authority 0x%08lX on target %s\n",
      entity, (*rc == MQRC_NONE ? "granted" : "denied"),
      authority, target);
}

// Check whether the specified permissions flags allow the requested
// authority

void check_auth (char * permissions, MQLONG authority,
   PMQLONG cc, PMQLONG rc)
{
   // These definitions group MQZAO constants under broader read/write/
   // execute/connect permissions flags. Connections are allowed if any
   // read/write/execute permissions are present.

   #define READ_OPTIONS (MQZAO_BROWSE | MQZAO_INPUT | \
      MQZAO_INQUIRE | MQZAO_DISPLAY | MQZAO_DISPLAY_STATUS)
   #define WRITE_OPTIONS (MQZAO_OUTPUT | MQZAO_SET | \
      MQZAO_CREATE | MQZAO_DELETE | MQZAO_CHANGE | MQZAO_CLEAR)
   #define EXECUTE_OPTIONS (MQZAO_START_STOP | MQZAO_RESOLVE_RESET | \
      MQZAO_PING)
   #define CONNECT_OPTIONS (MQZAO_CONNECT)

   if (permissions)
   {
      char *r = strchr (permissions, 'r');
      char *w = strchr (permissions, 'w');
      char *x = strchr (permissions, 'x');

      if (((authority & CONNECT_OPTIONS) && (!r && !w && !x)) ||
         ((authority & READ_OPTIONS) && !r) ||
         ((authority & WRITE_OPTIONS) && !w) ||
         ((authority & EXECUTE_OPTIONS) && !x))
      {
         *cc = MQCC_FAILED;
         *rc = MQRC_NOT_AUTHORIZED;
      }
   }
   else
   {
      *cc = MQCC_FAILED;
      *rc = MQRC_NOT_AUTHORIZED;
   }
}

// Internal convenience functions

size_t mqstrlen (const char *string)
{
   return (string ? strspn (string, MQCHARS) : 0);
}

char *mqstrncpy (char *dest, const char *src, size_t number)
{
   // dest must be (size + 1) bytes or larger to accommodate the
   // terminating byte

   strncpy (dest, src, number);
   dest [number] = '\0';
   dest [strspn (dest, MQCHARS)] = '\0';

   return dest;
}

void replace (char *string, char a, char b)
{
   register char    *c;

   for (c = string; *c; c++)
   {
      if (*c == a) *c = b;
   }
}

// This function writes log file entries with a date/timestamp and
// process identifier

int log (char *message, ...)
{
   #define BUFFER_SIZE 4096

   static char buffer [BUFFER_SIZE];
   int bytes = 0;

   // Write the log message after a timestamp

   if (logfd)
   {
      time_t timenow;
      struct tm *tm;
      va_list arg_list;

      timenow = time (NULL);
      tm = localtime (&timenow);
      bytes = sprintf (buffer, "[%ld] %02d/%02d/%04d %02d:%02d:%02d ",
         getpid (),
         tm->tm_mon + 1, tm->tm_mday, tm->tm_year + 1900,
         tm->tm_hour, tm->tm_min, tm->tm_sec);

      va_start (arg_list, message);
      bytes += vsprintf (&buffer [bytes], message, arg_list);
      va_end (arg_list);

      write (logfd, buffer, bytes);
   }

   return bytes;
}
```

BUILDING THE EXAMPLE SERVICE

The following command can be used (or included in a makefile) to build the example installable service from the source file *fileauthservice.c* as object *fileauthservice*:

```
cc -o fileauthservice fileauthservice.c -lc_r -lpthreads
➤  -lmqmzf_r -e mqz_init_authority -bM:SRE
```

The threaded versions of the C, ‘pthreads’, and MQSeries installable service libraries are used. The initialization function *mqz_init_authority* is defined as the external entry point.

CONFIGURING THE EXAMPLE SERVICE

The queue manager’s authorization service is specified in the *qm.ini* configuration file. It’s advisable to use a newly created and dedicated queue manager rather than an existing one that was previously configured to use the OAM. This avoids any difficulty of reinstating the OAM later in the presence of a previously defined set of access control definitions.

The *Module* line in the *ServiceComponent* entry must identify the path of the object file that contains the installable service module (here, it is assumed to be the directory */var/mqm*).

```
Service:
   Name=AuthorizationService
   EntryPoints=9
ServiceComponent:
   Service=AuthorizationService
   Name=fileauthservice
   Module=/var/mqm/fileauthservice
   ComponentDataSize=0
```

THE EXAMPLE SERVICE IN OPERATION

The installable service creates a log file (specified in the source as */var/mqm/fileauthservice.log*), in which an entry is written for every *MQZ_CHECK_AUTHORITY* request. Here is an example collection of entries that occur during the initialization of the queue manager. Each line begins with the process identifier in square brackets, followed by the date and timestamp of the entry. The *entityName* for whom authority is requested appears next, followed by a colon, the set of authority bits (*MQZAO* constants) requested, and the name of the target object.

```
[10468] 08/24/2000 22:08:02 fileauthservice initialized
➤  (fileauthservice.c last modified Sun Aug 24 22:01:23 2000)
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000001 on
➤  target SERVICE.QM
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000010 on
➤  target
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000010 on
➤  target SYSTEM.DEFAULT.NAMELIST
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000034 on
➤  target SYSTEM.CLUSTER.COMMAND.QUEUE
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x0000000E on
➤  target SYSTEM.CLUSTER.TRANSMIT.QUEUE
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000200 on
➤  target SERVICE.QM
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000200 on
➤  target SYSTEM.CLUSTER.TRANSMIT.QUEUE
[9454] 08/24/2000 22:08:08 mqm: granted authority 0x00000002 on
➤  target SYSTEM.CLUSTER.REPOSITORY.QUEUE
[9712] 08/24/2000 22:08:09 mqm: granted authority 0x00000001 on
➤  target SERVICE.QM
[12274] 08/24/2000 22:08:09 mqm: granted authority 0x00000001 on
➤  target SERVICE.QM
[12274] 08/24/2000 22:08:09 mqm: granted authority 0x00000024 on
➤  target SYSTEM.CHANNEL.INITQ
[9454] 08/24/2000 22:08:09 mqm: granted authority 0x0000000E on
➤  target SYSTEM.CLUSTER.REPOSITORY.QUEUE
[9454] 08/24/2000 22:08:09 mqm: granted authority 0x00000006 on
➤  target SYSTEM.CLUSTER.TRANSMIT.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00000010 on
➤  target
[9454] 08/24/2000 22:08:10 mqm: denied authority 0x00040000 on
➤  target APP_TEST.Q
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target REPLY.Q
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target REQUEST.Q
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.ADMIN.CHANNEL.EVENT
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.ADMIN.COMMAND.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.ADMIN.PERFM.EVENT
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.ADMIN.QMGR.EVENT
```

```
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.CHANNEL.INITQ
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.CHANNEL.SYNCQ
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.CICS.INITIATION.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.CLUSTER.COMMAND.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.CLUSTER.REPOSITORY.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.CLUSTER.TRANSMIT.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.DEAD.LETTER.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.DEFAULT.ALIAS.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.DEFAULT.INITIATION.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.DEFAULT.LOCAL.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.DEFAULT.MODEL.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.DEFAULT.REMOTE.QUEUE
[9454] 08/24/2000 22:08:10 mqm: granted authority 0x00040000 on
➤  target SYSTEM.MQSC.REPLY.QUEUE
```

INSTALLABLE SERVICES AS A WINDOW TO QUEUE MANAGER ACTIVITY

As installable services are exposed to details of the queue manager's internal operation, they offer interesting possibilities for monitoring the activity of the queue manager. In the log excerpt above, the process identifiers correspond to the execution controller (*10468*) and two agents (*9454* and *12274*). This information can be obtained from the **ps** command. From the queues and authorities requested, we can infer that *9454* is acting for the cluster controller and *12274* for the channel initiator, and we can observe the subsequent activity of each of them.

Chris Markes
HCI Architect
IBM UK Laboratories Ltd (UK)

MQSeries clusters: a hands-on view (part 2)

This is the second part of this two-part article on MQSeries clustering (the first part appeared in last month's issue). This article illustrates MQSeries clustering using the example set-up show in Figure 1. Note that this figure also appeared in last month's instalment, though the boxes around the queue managers were omitted in that version.

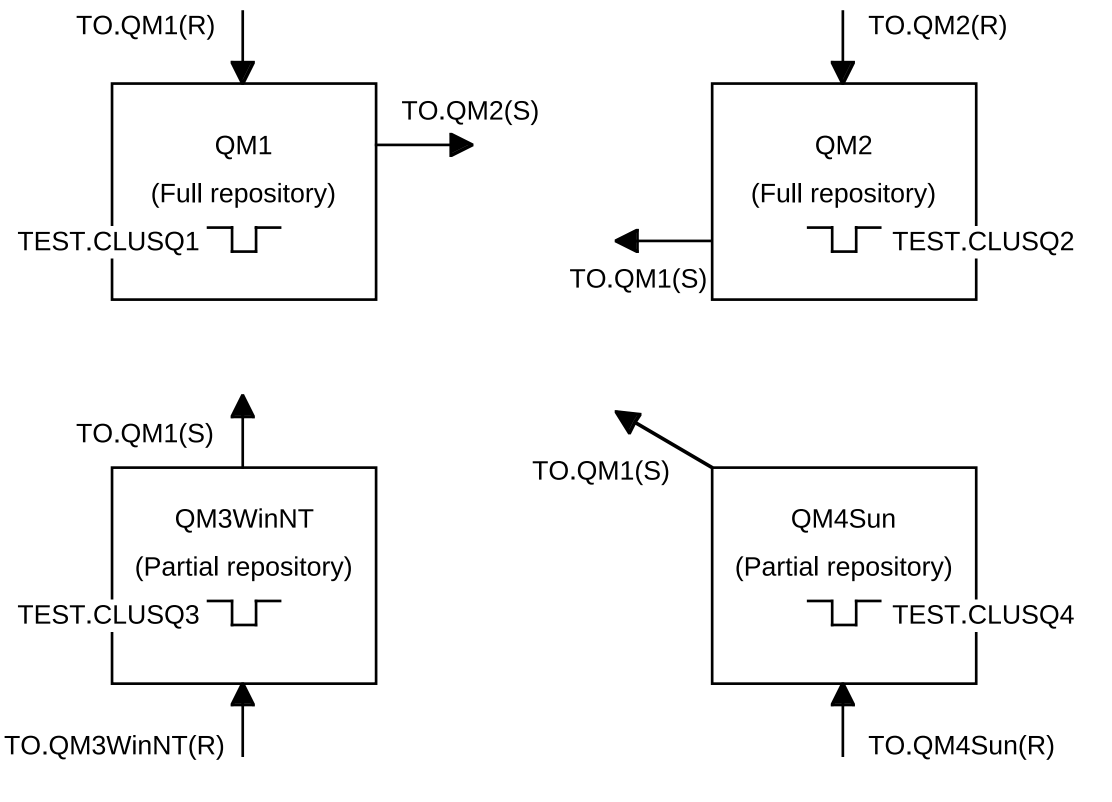

Figure 1: The sample MQSeries cluster

COMPARISON OF THE NUMBER OF DEFINITIONS

At this point, we have defined a cluster with four queue managers, so let's check how many definitions were made:

```
                                                   Number of
                                                  definitions
One cluster sender channel per queue manager            4
One cluster receiver channel per queue manager          4
One cluster queue per queue manager                     4

Total                                                  12
```

So, using just 12 definitions (ignoring the required 'system' definitions), I am able to connect four queue managers together, and each is able to write to any of the cluster queues.

If the same network of queue managers was created using 'traditional' queuing definitions, we would need:

	Number of definitions
Three sender channels per queue manager	Total 12
Three receiver channels per queue manager	Total 12
One local queue per queue manager	Total 4
Three remote queues per queue manager	Total 12
Three XmitQs per queue manager	Total 12
Three process definitions per queue manager	Total 12
Total	Total 64

This clearly demonstrates the significant saving in administration that clustering yields.

DATA CONVERSION

As would be the case with traditional distributed queuing methods, messages going from one platform to another will probably need conversion.

Even though MQSeries will happily convert data at the channel level, it is recommended that conversion is managed by the application when it gets messages – this is easily achieved by using *MQGET* with the *CONVERT* option. This is particularly important when a message is sent to several platforms (so-called 'multi-hopping'). You want to avoid having to convert the message at each stage – instead convert at the last stage using your program. Note that changing from *MQGET* alone to *MQGET* plus *CONVERT* means that you'll need to handle situations where MQSeries is unable to convert the data.

A close inspection of the new cluster channels shows that the receiver channel has a new field 'convert by sender'. It is this field – and this field only – that determines whether MQSeries will automatically convert the data. This can lead to some confusing situations, as demonstrated by the table overleaf.

```
                 CONVERT = ?
      Cluster sender       Cluster receiver          Does conversion
         TO.QM1              TO.QM3WinNT               take place?
           Y                      Y                        Yes
           Y                      N                        No
           N                      Y                        Yes
           N                      N                        No
```

In the second row, the ‘sender’ indicates that conversion is to take place, and yet the overall result is that it does not. Rather than display the channel attributes to check whether conversion takes place, it is better – in a clustering environment – to display the channels from the cluster’s point of view using the *DIS CLUSQMGR* command and check the status of the *CONVERT* field.

Another issue you may encounter when using clustering concerns the use of channel message exits in traditional distributed queuing environments. I myself don’t like using exits – the system should cope with my needs – though there are situations when exits have to be introduced. One such situation is when you use ‘MERVA’ on the mainframe (MERVA is a financial messaging system that allows access to the SWIFT network), whose internal settings expect data to be in codepage 500. MERVA uses a set of control characters to recognize the start and end of data. The control characters are ‘{’ and ‘}’, and MERVA expects them as hex characters xC0 and xD0. The problem arises if the mainframe queue manager uses, say, codepage 273 (German) instead of codepage 500. If the MQSeries channels convert messages (in other words, you set *CONVERT=Y* and the message format as *MQSTR*), the control characters will be converted to x43 and xDC respectively. Many years ago it was decided that a message exit should be used to convert the data to code page 500, even though the queue manager uses codepage 273. This leads to the horrible situation where the *CCSID* in the MQSeries message header reads ‘273’, though the data is, in reality, in *CCSID* ‘500’!

This ‘works’ in a traditional channel set-up, but fails in a clustered environment. Remember that the receiving queue manager has only one cluster receiver channel, and so all data sent to this channel would be converted to codepage 500 if the message exit is used there! The solution, as mentioned earlier, is to ensure that the receiving program, and not the channel, does the conversion by using the *CONVERT* option with *MQGET* and setting the codepage to 500.

Another solution to the MERVA problem is to set the *CCSID* of the queue manager to 500, though this may lead to even bigger problems if your language's alphabet includes special characters.

OPERATIONS

Once clustering is implemented, the *Master* address space will show these new messages at start-up:

```
CSQV452I MQV1 CSQVXLDR Cluster workload exits not available
```

This is fine (if you are not using the exit specified).

```
CSQI007I MQV1 CSQIERS3 BUILDING IN-STORAGE INDEX FOR
QUEUE SYSTEM.CLUSTER.REPOSITORY.QUEUE
```

The above message results from the fact that MQSeries builds indices for every queue that has an index.

When the OS/390 channel initiator (CHIN) address space starts, you should see the Repository Manager starting (see the message below). This is an essential component of the repository.

```
CSQX131I MQV1 CSQXADPI 8 adapter subtasks started, 0 failed
CSQX410I MQV1 CSQXREPO Repository Manager started
```

If the Repository Manager isn't running, cluster queues cannot be seen and new cluster definitions cannot be propagated to other queue managers.

Under normal circumstances, you should see cluster sender and receiver channels starting and stopping in exactly the same way as standard channels, as indicated by the following messages:

```
CSQX500I MQV2 CSQXRESP Channel TO.MQV2 started
CSQX500I MQV2 CSQXRCTL Channel TO.MQV1 started
```

PROBLEMS AND HOW TO DEAL WITH THEM

As with any other type of system, problems do occur in clustered environments. The following is a subset of things that can happen:

1 The Repository Manager stops.

This is potentially a nasty one. The first thing to do is ensure you've got all the latest maintenance patches applied. This problem can happen when an internal MQSeries error has occurred, for instance:

```
CSQX038E MQXX CSQXREPO Unable to put message to
SYSTEM.CLUSTER.TRANSMIT.QUEUE, MQCC=2 MQRC=2013
CSQX448E MQXX CSQXREPO Repository Manager stopping because
➤  of errors. Restart in 600 seconds
```

As you can see, the Repository Manager will attempt a restart in ten minutes' time, but if the message states 'Repository Manager stopped', the only course of action is a *CHIN* restart. By the way, you cannot change the restart time of 600 seconds.

2 You get a return code 2085 (unknown object) after an *MQPUT* to a cluster queue.

This is a beauty! There are many reasons for getting this return code, but you would have thought it wouldn't happen in a clustered environment. Provided the Repository Manager hasn't stopped, and you've given the correct cluster queue name, then the only reason that remains is that your local queue manager can't 'see' the cluster queue, even though it is part of the cluster! The first thing to do is to issue some display commands:

```
DIS QCLUSTER(*)
```

This should display all instances of the queue, and this command:

```
DIS CLUSQMGR(*)
```

should display information on all relevant queue managers and cluster channels.

If these are correct, then you may have a problem similar to one I discovered a little while ago. Remember the set-up of a cluster where the third queue manager was added? This works fine, provided the third queue manager has a partial repository. If, however, the third queue manager has a full repository, then don't rely on dynamically generated cluster channels – for a full repository queue manager to join a cluster and ensure that all updates are reflected in its repository, its cluster channels must be defined manually.

In our case, when a new cluster queue was defined, it was not propagated to the newly added queue manager, so it got an RC2085 (unknown object) when it then performed an *MQPUT*.

You can temporarily get the repository back 'in line' by issuing the command *REFRESH CLUSTER(TEST_REPOS)*, though this is not a command to be used lightly. This command basically deletes all the cluster information and rebuilds it.

3 When you try to write to a cluster queue, the message does not arrive, but when the operation is carried out using a remote queue, it works fine.

The scenario here is that you've got an existing working application using traditional channels. The application does an *MQPUT* specifying the destination queue name and queue manager. A standard transmission queue with the same name as the remote queue manager exists. You then prepare for clustering by going to the remote queue manager, defining the cluster channels, and changing the queue to a cluster queue. The standard receiver channel is then stopped.

The application does an *MQPUT*, which succeeds, but the message doesn't land on the cluster queue. You may imagine that the message instead landed on the cluster transmission queue, but this is not the case. The message was finally tracked down to the old transmission queue – the reason for it being sent there is that the application specified the destination queue manager, and, because a transmission queue with the same name as the remote queue manager exists, MQSeries duly placed the message there, as expected! To remedy this, you either remove the transmission queue or, if you intend to continue using it, change the program by not specifying the destination queue manager.

4 You write to a cluster queue that is defined in two places but the message only arrives in one place.

Check the queue's *Default Bind Option*. For equal distribution across all available queue managers, this should be set to 'N' (Not fixed). If the option is correct, then check the status of the queue manager itself, then check the channel, and finally verify that the queue is put-enabled.

5 Beware of the DLQ!

If you write messages to a clustered queue on another platform, and that queue becomes full, the sending application will not be informed of the problem (however, with a local queue, you'd get an 'RC=2053'). Any messages that cannot be delivered are written to the DLQ. If the DLQ is not there or is 'put-disabled', messages remain on the sending system's *SYSTEM.CLUSTER.TRANSMIT.QUEUE*. Obviously, You also need a program to process messages on the DLQ.

6 The effect of the *Default Bind Option*.

Imagine you have a cluster with three queue managers. An application *MQPUT*s on the first queue manager, while on the other two you have defined the same cluster queue, each with *Default Bind Option=N* (Not fixed). If no errors occur, then messages are distributed equally among them in a round-robin fashion.

Now imagine that, while writing one hundred messages, the channel to one of the queue managers becomes unavailable. What happens?

- On one cluster queue, where the channel is unavailable, no messages are delivered.
- On the other cluster queue we see messages 2, 4, 6, 8, …, 100 followed by 1, 3, 5, 7, …, 99 (originally bound for the other queue).

The same would happen if the queue is made 'put disabled'.

Now suppose that both cluster queues are changed to use *Default Bind Option=O* (Open) and another hundred messages are written to them. All messages stay on the sending system's *SYSTEM.CLUSTER.TRANSMIT.QUEUE*.

Why does this happen? The answer is that MQSeries selects one of the queue managers and starts writing to it. Option 'O' means 'write all messages to the queue manager selected', so when its channel becomes unavailable, there is no other place to send messages, and messages remain at the source queue manager. When you browse the cluster *xmitq*, you see that its *CORRELID* is the name of the cluster channel that became unavailable.

You must, therefore, realize that these messages will remain unprocessed until the channel becomes available again. The same happens if the queue manager is forcibly stopped.

Finally, imagine a scenario where a cluster queue exists only in one place and the queue is 'put disabled'. This results in an 'RC=2051'. If the cluster queue exists in more than one place, the message is redirected to the next available queue manager.

7 How do I get rid of old cluster definitions?

If you want permanently to remove a queue manager from a cluster, follow the instructions in *Apar PQ35026*. If you want to remove the queue manager temporarily, use the *SUSPEND* and *RESUME* commands. There are, however, situations where 'orphan' cluster items are left around that cannot be deleted using either TSO panels or commands. This can happen if, for instance, a machine crashes and has to be rebuilt. This can lead to confusing situations where a cluster channel appears to reside in more than one cluster when you list it via the *DIS CLUSQMGR* command. If the channel is started, then both entries are started. This is something that I would like IBM to address.

VARIOUS CLUSTER COMMANDS

The existing commands, like *DEFINE*, *ALTER*, and *DISPLAY*, have been updated with the new cluster attributes. The new commands are:

```
REFRESH CLUSTER(xxxxxxxx)
```

This discards locally held information, and rebuilds the cluster. This command should not be used as a matter of course, though experience has shown that this is sometimes the only way!

```
RESET CLUSTER(xxxxxxxx) ACTION(FORCEREMOVE) QMNAME(qqqq)
```

This command forcibly removes a queue manager 'qqqq' from the cluster.

```
SUSPEND QMGR CLUSTER(xxxxxxxx) MODE(mmmm)
```

This command temporarily suspends a queue manager’s membership of a cluster. The mode (*QUIESCE* or *FORCE*) determines how this is achieved.

```
RESUME QMGR CLUSTER(xxxxxxxx)
```

This command is the reverse of *SUSPEND QMGR*.

```
DIS CLUSQMGR
```

This command displays list of queue managers in a cluster along with their cluster channels.

```
DIS QCLUSTER
```

This command displays a list of cluster queues and default bindings and tells you where they’re defined.

One thing that’s worth mentioning is that you can define a *QALIAS* as part of a cluster. The result is that the queue to which the alias points gets ‘advertised’ in the cluster.

CLUSTER WORKLOAD EXIT

Most installations will probably be happy with the default settings of this feature. It’s also worth pointing out that the name ‘workload exit’ is a little misleading, as the exit only distributes messages to ‘available’ destinations. The exit checks the queue manager, the status of the channels, and whether the queue is ‘put enabled’. It doesn’t check how ‘busy’ a queue manager is or the general workload on the machine. Also note that, if MQSeries finds a cluster queue that is defined locally, then messages are written locally and are not subject to cluster distribution.

You can, of course, code your own exit, and IBM has provided samples to help you do this. However, the samples do not correspond with the ‘default’ way MQSeries distributes messages. The assembler sample (CSQ4BAF1 for OS/390) works with the ‘Exit Data’ supplied (see below).

There are several things you need to do to activate your own exit:

- Assemble/link the source.
- Add a *DD* statement *//CSQXLIB* to the *Master* address space pointing to an exit load library. (You probably already have a *DD* statement allocated to *CHIN* for channel exits.)
- Alter the queue manager by setting the *Exit Name* and *Exit Data* – for example:

```
ALTER QMGR CLWLEXIT(CSQ4BAF1)

ALTER QMGR CLWLDATA('TO.QMGR')
```

TO.QMGR is the name of the cluster sender channel to which you want to send messages.

SECURITY

You should have all your MQSeries resources protected before you introduce clustering.

Although clustering doesn't change your security requirements, if your security is not fully in place, then you should be aware of the following security problem: in a traditional distributed queuing environment, channels have to be defined at both ends. In a lot of firms, the mainframe definitions are made by one department, and the definitions for other platforms by another. If only one set of definitions is in place, you won't be able to send messages, and this is quite a good way to stop 'unauthorized' queue managers from connecting to the host.

However, with clustering, if one end is defined, MQSeries will define the return cluster channels dynamically. So, equipped with the host's IP address, MQSeries port, and the name of the cluster, any Tom, Dick, or Harry can connect his PC using, say, MQSeries for Windows NT to the host, and ….

Just a thought: you can code a channel security exit to stop this happening, and there are security exits available from third-party vendors.

USING A QUEUE MANAGER ALIAS

In order fully to enjoy the benefits of being in a cluster, a queue manager needs to be a ‘member’ of it. There may be circumstances in which you cannot do this for either technical reasons or as a result of the local set-up. In such cases, a special set of definitions could allow an ‘external’ queue manager to make use of a cluster.

In order to send messages from a queue manager outside a cluster to a queue hosted by a queue manager inside a cluster (or vice versa) you have to build a so-called ‘gateway’. This handles messages transferred into and out of clusters. The gateway itself has to be a member of the cluster. Use the following procedure to implement a gateway (see Figure 2):

1 Define the cluster *TEST.CLUSTER*, as above.

2 Define a cluster queue (*Q.GENERAL* in Figure 2) as a local queue on *QM2*.

3 Decide which queue manager is to be the gateway within the cluster (*QM1* in Figure 2).

4 Define a queue manager alias on *QM1* that looks like a remote queue but isn’t:

```
DEFINE QREMOTE(Qmgr.Alias) RNAME(' ') RQMNAME(' ')
```

This entry maps any message destined for queue manager *Qmgr.Alias* to null, meaning that the *QREMOTE* definition in the queue manager outside the cluster can use *Qmgr.Alias* as the queue manager name, instead of using the actual queue manager’s name.

5 On the queue manager outside the cluster (*QM5*), define the remote queue:

```
DEFINE QREMOTE(Q.GENERAL) RNAME(Q.GENERAL)
RQMNAME(Qmgr.Alias) XMITQ(QM1)
```

6 Define standard sender and receiver channels between the gateway queue manager and the queue manager outside the cluster (plus, of course, a standard transmission queue).

If an application like AMQSPUT on *QM5* issues an *MQPUT* call to put a message on *Q.GENERAL*, the remote definition causes the message to be routed to the gateway queue manager (*QM1*) first, and from there to any queue manager in the cluster hosting cluster queue *Q.GENERAL* (such as *QM2*), regardless of the target queue's location.

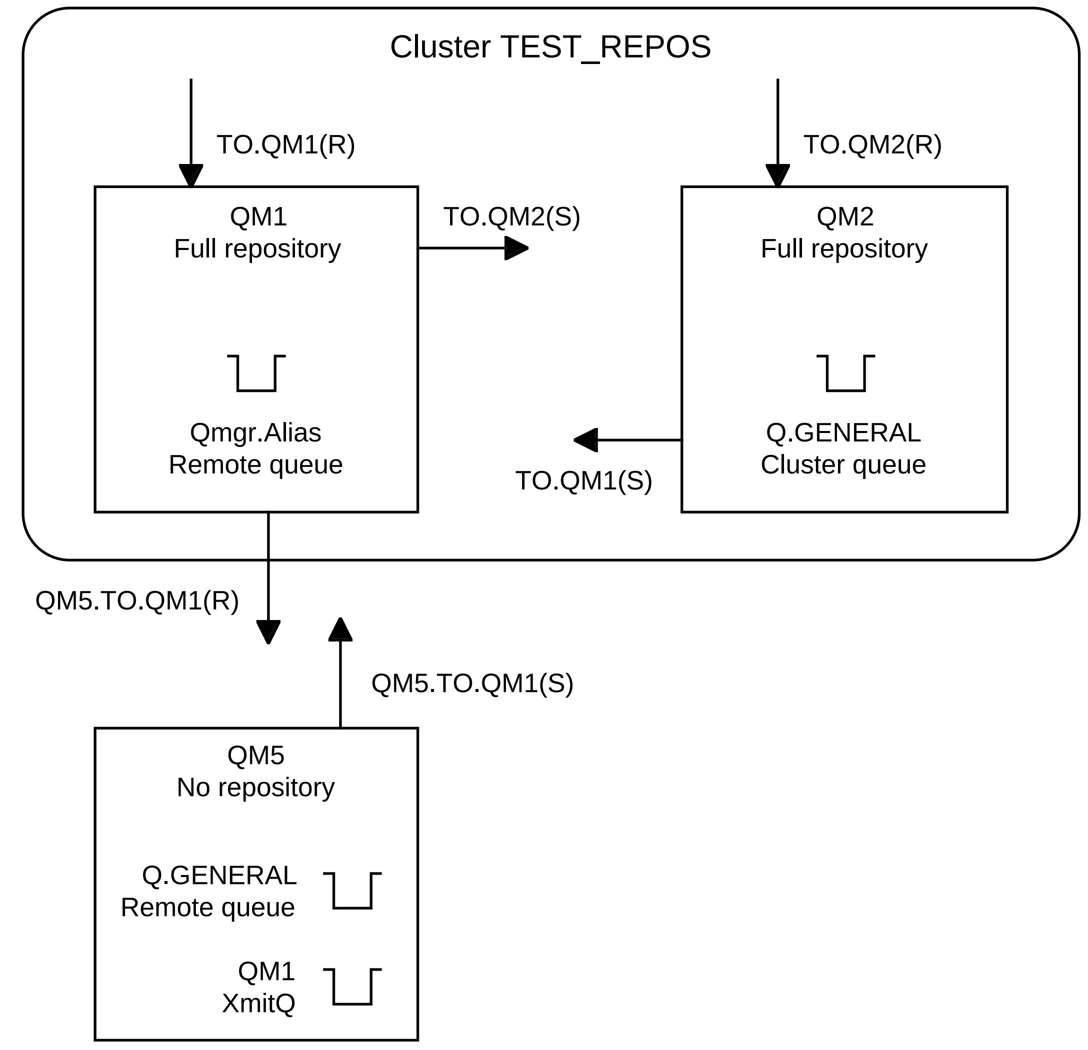

Figure 2: Using a queue manager alias in a cluster

VARIOUS APARS/PTFS RELATED TO CLUSTERING

When examining the list below, remember that it's not exhaustive and some PTFs may be superseded.

APAR	PTF	Description
PQ36946	UQ41981	Fix for RC2087 when using a Queue Manager Alias.
PQ35026		Method for removing clustering data from a queue manager.
PQ36747	UQ41650	Fix for RC2189 when putting messages on SYSTEM.CLUSTER.COMMAND.QUEUE.
PQ37956	UQ43192	Fix for RC2013 after REFRESH CLUSTER
PQ38133	UQ44637	New messages for clusters on OS/390s.
PQ34433	UQ41413	Undeliverable repository command after REFRESH.

FUTURES

On April 4, 2000 IBM made a preview announcement of MQSeries for OS/390 V2.2.

This new release introduces the concept of shared queues held in a coupling facility. Two or more queue managers on OS/390 can participate in this environment, and all can write to and read from the same shared queue. As shared queues aren't owned by a particular queue manager, if one queue manager fails, the others continue to work with the shared queue.

Shared queues are restricted to non-persistent messages with a maximum size of 63 KB.

This is a further improvement to MQSeries availability. Shared queues can, of course, be part of a cluster, which means that any application (including non-OS/390 applications) can write to a shared queue, provided the queue manager is part of the cluster.

CONCLUSION

It is clear that MQSeries clusters reduce administration and improve availability. However, while the benefits of reduced administration are obvious, the side-effects of clustering can at times be surprising. In order to implement a cluster, you will need a detailed knowledge of the applications that read and write MQSeries messages, any affinities that might exist, and any data conversion requirements.

It is also important to make sure that operations staff are aware of the changes to TSO ISPF panels and of modified and new commands.

Applications that have special requirements with regards to affinity may need to be changed, though you may be able to handle them by changing the *Default Bind Option* to ‘Open’. Systems that use channel message exits should ideally have them removed and their applications changed to issue *MQGET*s with the *CONVERT* option set to ‘Y’.

To make full use of clustering, applications should not have any affinities that tie them to one particular environment or link them to one particular system. To maximize availability, you need two or more queue managers that are configured in the same way (cloned), that are all part of a cluster, and are all served by – and trigger – applications that can run in more than one place.

Ruud van Zundert (ruud@tesco.net)
Independent Consultant (UK)

Customizing CSQ4ZPRM

This article and the next one (Starting the channel initiator and command server) offer further examples of installing and customizing MQSeries on the mainframe. This article deals with *CSQZMQxx*, which is an example of how the file *CSQ4ZPRM* supplied in the *MQM.SCSQPROC* dataset can be customized to meet an installation’s specific requirements. Here you can define the Hlq that’s to be used with your archive dataset and activate and de-activate archiving, SMF tracing, and so on. This job is used to create the Queue Manager Options module. Edit the parameters for the *CSQ6LOGP*, *CSQ6ARVP*, and *CSQ6SYSP* macros to determine your system parameter’s relink module, *CSQZPARM*.

An example of a customized *CSQZPARM(CSQZMQxx)* module is shown below. If you search the file for the string ‘MQxx’, you’ll find references that you need to set.

JOB CSQZ4PRM

```
//JOBCARD
//*************************************************************************
//* Customize MQP1 Options module
//*************************************************************************
//* The Macros in this deck are tracked by OS/390 SMPE usermod
//*************************************************************************
//* CUSTOMIZE THIS JOB HERE FOR YOUR INSTALLATION
//* YOU MUST DO GLOBAL CHANGES ON THESE PARAMETERS USING YOUR EDITOR
//*************************************************************************
//*
//*              IBM MQSeries for MVS/ESA
//*    This job assembles and links a new system parameter module.
//*    Edit the parameters for the CSQ6LOGP,
//*    CSQ6ARVP, and CSQ6SYSP macros to determine your
//*    system parameters.
//*    See "MQSeries for MVS/ESA System Management Guide"
//*    for a full description of the parameters.
//*
//*    This member replaces CSQZPARM
//*
//*************************************************************************
//*
//*            Assemble step for CSQ6LOGP
//*
//LOGP   EXEC PGM=ASMA90,PARM='DECK,NOOBJECT,LIST,XREF(SHORT)'
//SYSLIB   DD DSN=MQM.SCSQMACS,DISP=SHR,UNIT=3390,VOL=SER=SYS001     <=VOL?
//         DD DSN=SYS1.MACLIB,DISP=SHR,UNIT=3390,VOL=SER=SYS001      <=VOL?
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPUNCH DD DSN=&&LOGP,
//       UNIT=SYSDA,DISP=(,PASS),
//       SPACE=(400,(100,100,1))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
         CSQ6LOGP INBUFF=48,          LOG INPUT BUFFER SIZE KB               X
               MAXALLC=3,             MAX ALLOCATED ARCHLOG VOLS             X
               MAXARCH=5,             MAX ARCH LOG VOLUMES MQxx              X
               OFFLOAD=YES,           ARCHIVING ACTIVE                       X
               OUTBUFF=4000,          LOG OUTPUT BUFFER SIZE                 X
               TWOACTV=YES,           DUAL ACTIVE LOGGING                    X
               TWOARCH=NO,            DUAL ARCHIVE LOGGING                   X
               TWOBSDS=YES,           DUAL BSDS                              X
               WRTHRSH=32             ACTIVE LOG BUFFERS
         END
/*
//*
//*            Assemble step for CSQ6ARVP
//*
//ARVP   EXEC PGM=ASMA90,COND=(0,NE),
//            PARM='DECK,NOOBJECT,LIST,XREF(SHORT)'
//SYSLIB   DD DSN=MQM.SCSQMACS,DISP=SHR,UNIT=3390,VOL=SER=SYS001     <=vol?
//         DD DSN=SYS1.MACLIB,DISP=SHR,UNIT=3390,VOL=SER=SYS001      <=vol?
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPUNCH DD DSN=&&ARVP,
//         UNIT=SYSDA,DISP=(,PASS),
//         SPACE=(400,(100,100,1))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
         CSQ6ARVP ALCUNIT=CYL,               ARCLOG ALLOCATION UNIT            X
               ARCPFX1=MQARCH.MQxx.LOG1,     DSN PREFIX FOR ARCLOG1 MQxx       X
               ARCPFX2=MQARCH.MQxx.LOG2,     DSN PREFIX FOR ARCLOG2 MQxx       X
               ARCRETN=3,                    ARCLOG RETENION PERIOD DAYS       X
               ARCWRTC=(1,3,4),              ARCHIVE WTO ROUTE CODE            X
               ARCWTOR=NO,                   PROMPT BEFORE ARCLOG MOUNT        X
               BLKSIZE=24576,                ARCLOG BLOCKSIZE                  X
               CATALOG=YES,                  CATALOG ARCLOG DATASETS           X
               COMPACT=NO,                   ARCHIVE LOGS COMPACTED            X
               PRIQTY=800,                   PRIMARY SPACE ALLOCATION          X
               PROTECT=NO,                   NO DISCRETE PROFILES CREATED      X
               QUIESCE=80,                   MAX QUIESCE TIME IN SECONDS       X
               SECQTY=80,                    SECONDARY SPACE ALLOCATION        X
               TSTAMP=YES,                   NO TIMESTAMP SUFFIX IN DSN        X
               UNIT=DISK                     ARCLOG ALLOCATION UNIT MQxx
         END
/*
//*
//*            Assemble step for CSQ6SYSP
//*
//SYSP   EXEC PGM=ASMA90,COND=(0,NE),
//            PARM='DECK,NOOBJECT,LIST,XREF(SHORT)'
//SYSLIB   DD DSN=MQM.SCSQMACS,DISP=SHR,UNIT=3390,VOL=SER=SYS001    <=vol?
//         DD DSN=SYS1.MACLIB,DISP=SHR,UNIT=3390,VOL=SER=SYS001     <=vol?
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPUNCH DD DSN=&&SYSP,
//        UNIT=SYSDA,DISP=(,PASS),
//        SPACE=(400,(100,100,1))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
         CSQ6SYSP CTHREAD=330,           TOTAL NUMBER OF CONNECTIONS           X
               CMDUSER=CSQOPR,           DEFAULT USERID FOR COMMANDS           X
               IDBACK=20,                NUMBER OF NON-TSO CONNECTIONS         X
               IDFORE=100,               NUMBER OF TSO CONNECTIONS             X
               LOGLOAD=100000,          LOG RECORD CHECKPOINT NUMBER          X
               OTMACON=(,,DFSYDRU0,2147483647,CSQ),   OTMA PARAMETERS         X
               QMCCSID=0,               QMGR CCSID                            X
               ROUTCDE=1,               DEFAULT WTO ROUTECODE                 X
               SMFACCT=YES,             GATHER SMF ACCOUNTING  MQxx           X
               SMFSTAT=YES,             GATHER SMF STATS       MQxx           X
               STATIME=15,              STATISTICS RECORD INTERVAL            X
               TRACSTR=NO,              TRACING AUTO START                    X
               TRACTBL=16               GLOBAL TRACE TABLE SIZE X4K
         END
/*
//*
//*  LINKEDIT CSQARVP, CSQLOGP and CSQSYSP into a system parameter module.
//*
//LKED   EXEC PGM=IEWL,COND=(0,NE),
//      PARM='SIZE=(900K,124K),RENT,NCAL,LIST,AMODE=31,RMODE=ANY'
//*
//*   OUPUT AUTHORIZED APF LIBRARY FOR THE NEW SYSTEM
//*   PARAMETER MODULE.
//*
//*YSLMOD  DD DSN=MQM.SCSQAUTH,DISP=SHR
//SYSLMOD  DD DSN=MQM.MQxx.PARMODS,DISP=SHR   <= MQxx Options loadlib
//SYSUT1   DD UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=*
//ARVP     DD DSN=&&ARVP,DISP=(OLD,DELETE)
//LOGP     DD DSN=&&LOGP,DISP=(OLD,DELETE)
//SYSP     DD DSN=&&SYSP,DISP=(OLD,DELETE)
//*
//*   LOAD LIBRARY containing the default system parameter module (CSQZPARM).
//*
//OLDLOAD  DD DSN=MQM.SCSQAUTH,DISP=SHR
//SYSLIN   DD *
   INCLUDE SYSP
   INCLUDE ARVP
   INCLUDE LOGP
   INCLUDE OLDLOAD(CSQZPARM)
 ENTRY CSQZMSTR
 NAME CSQZMQxx(R)             MQxx system parameter module name
/*
```

Saida Davies
IBM (UK)

Starting the channel initiator and command server

CSQ4CHNA is a brief code sample that contains the channel initiator and command server start commands.

These can be added at the beginning of the *CSQ4CHNL* member file for each queue manager. The reason to put these files in a separate member is to accommodate situations where a queue manager is designed to be shared between systems in a parallel Sysplex. In such instances, all channels that are to be shared are isolated in one member and the channel initiator is defined in another, which is started via a unique Options module (*CSQ4XPARM*).

AN EXAMPLE OF CUSTOMIZED CSQ4INP2 (CSQ4CHNA)

```
*
* Start MQA1 Channel Initiator with LPARA1 options module
START CHINIT PARM(CSQXMQA1)
*
* Start command server
START CMDSERV
*
```

Saida Davies
IBM (UK)

Contributing to *MQ Update*

Contributions to *MQ Update* may be sent to the editor, Harry Lewis, at: *MQ Update, Xephon, 27-35 London Road, Newbury, Berkshire RG14 1JL, UK*. You may also e-mail articles to *harryl@xephon.com*. For more information about contributing, please download a copy of *Notes for contributors* from Xephon’s Web site at *www.xephon.com/nfc.pdf*.

MQ news

CommerceQuest has entered into an OEM agreement with Tivoli, whereby its e-Adapter Suite, a bulk data movement and integration package based on MQSeries, is to be combined with Tivoli's system management suite under the name 'Tivoli Data Exchange'. Tivoli will sell and promote the product. The e-Adapter Suite adds bulk data movement, conversion, and integration capabilities to MQSeries and its status management subsystem provides an audit trail of information as it flows between systems.

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

CommerceQuest, 3550 West Waters Avenue, Tampa, FL 33614, USA
Tel: +1 813 903 3000
Fax: +1 813 903 3095
Web: http://www.commercequest.com

* * *

WRQ has announced a new suite of integration tools, including WRQ VeraStream, based on the recently-acquired SuperNova technology. VeraStream combines rapid visual development tools, native database and application adapters, and an integration engine. Also included are visual tools for implementation connectivity options including MQSeries, XML, CORBA/IIOP, CICS, and Java.

VeraStream is out now and prices start at US$10,000.

For further information contact:
WRQ, 1500 Dexter Avenue North, Seattle, WA 98109, USA
Tel: +1 206 217 7100
Fax: +1 206 217 0293
Web: http://www.wrq.com

WRQ, 40 West Street, Marlow, Buckinghamshire SL7 2NB, UK
Tel: +44 1628 400 800
Fax: +44 1628 400 801

* * *

IBM has announced VisualAge for Java Version 3.5 Professional and Enterprise Editions. Both support incremental RAD and server-side programming, and come with a WebSphere test environment and Tool Integrator API. Also new is better support for change management tools, such as VisualAge TeamConnection, Merant PVCS, Microsoft SourceSafe, and Rational ClearCase. The Enterprise Edition also comes with Enterprise Access Builders for MQSeries.

Out now, the Professional Edition costs US$150 and the Enterprise Edition US$3,000.

For further information, contact your local IBM representative.

xephon