MQSeries®

Queue Manager Clusters

SC34-5349-01

MQSeries®

Queue Manager Clusters

SC34-5349-01

Note!

Before using this information and the product it supports, be sure to read the general information under "Appendix. Notices" on page 119.

Second edition (March 2000)

This edition applies to the following products:
- MQSeries for AIX® V5.1
- MQSeries for AS/400® V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2® V5.1
- MQSeries for OS/390® V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT® V5.1

and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 1999, 2000. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures

Tables

© Copyright IBM Corp. 1999, 2000

About this book

This book describes how to create and use *clusters* of MQSeries queue managers. It explains the concepts and terminology of clustering and shows how you can benefit by taking advantage of clustering. It details changes to the message queue interface (MQI), and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

Who this book is for

This book is for anyone who needs an understanding of MQSeries clusters. The following readers are specifically addressed:

- Network planners responsible for designing the overall queue manager network
- Application programmers responsible for designing applications that access queues and queue managers within clusters
- Systems administrators responsible for monitoring the local system and implementing some of the planning details
- System programmers with responsibility for designing and programming the user exits

What you need to know to understand this book

This book describes MQSeries clustering in detail, and includes step-by-step examples that you should be able to follow with only limited background knowledge about MQSeries in general. An understanding of the concepts of message queuing, for example the purpose of queues, queue managers, and channels would be an advantage.

To understand fully how to make the best use of clusters, it is useful to be familiar with the MQSeries products for the specific platforms you will be using, and the communications protocols that are used on those platforms. It is also helpful to
| have an understanding of how distributed queue management works. These topics
| are discussed in the *MQSeries Intercommunication* book.

How to use this book

This book contains three parts. The chapters in Part 1. Getting started with queue manager clusters are aimed at users who are new to clusters. Read these chapters first to learn what queue manager clusters are and how to use them. Throughout this part of the book, the use of clusters is compared with more traditional distributed-queuing techniques. If you are not familiar with distributed queuing, you should skip the sections that are not of interest to you. You should still be able to follow the guidance and examples given. The chapters in this part are:

- "Chapter 1. Concepts and terminology" on page 3, which introduces the concepts of queue manager clusters, explains the associated terminology, and highlights the differences between using clusters and using distributed queuing techniques.
- "Chapter 2. Using clusters to ease system administration" on page 11, which shows the benefits of using clusters and shows when and where you might choose to implement them in your existing network.

- “Chapter 3. First tasks” on page 19, which describes some of the first tasks you may need to perform in order to set up and use a cluster. You should be able to accomplish these first tasks without an in-depth understanding of clusters or distributed queuing.

The chapters in Part 2. Using queue manager clusters are aimed at more experienced users who want to understand about clusters in detail. Read these chapters to learn how to use clusters to the best advantage. The chapters in this part are:

- “Chapter 4. How queue manager clusters work” on page 29, which provides more detail about the components of clusters and explains how clustering works.
- “Chapter 5. Using clusters for workload management” on page 41, which describes how to use clusters to achieve workload balancing.
- “Chapter 6. MQSeries commands” on page 53, which introduces commands that are specific to work with MQSeries clusters.
- “Chapter 7. Managing MQSeries clusters” on page 59, which provides administrative information about how to design and maintain a cluster.
- “Chapter 8. Keeping clusters secure” on page 67, which discusses security aspects associated with using clusters.
- “Chapter 9. Advanced tasks” on page 71, which guides you through a series of more advanced tasks.

The chapters in Part 3. Reference information contain reference information about the cluster workload exit. The chapters in this part are:

- “Chapter 10. Cluster workload exit call and data structures” on page 93.
- “Chapter 11. Constants for the cluster workload exit” on page 113.

There is a glossary and a bibliography at the back of the book.

Summary of changes

This section describes changes to this edition of *MQSeries Queue Manager Clusters*. Changes since the previous edition of the book are marked by vertical lines to the left of the changes.

Changes for this edition (SC34-5349-01)

The major change for this edition is the

- Addition of support for clusters on MQSeries for AS/400

© Copyright IBM Corp. 1999, 2000

Changes

Part 1. Getting started with queue manager clusters

© Copyright IBM Corp. 1999, 2000

Chapter 1. Concepts and terminology

This chapter introduces the concepts of queue manager clusters and explains some of the terminology. For the benefit of customers familiar with traditional distributed-queuing techniques, it compares the use of clusters with the use of distributed queuing. If you are not familiar with distributed queuing, you should skip the sections that are not of interest to you.

Concepts

Businesses are increasingly becoming aware of the advantages of establishing an intranet or of connecting processors to a LAN. You might also have connected some OS/390 processors to form a sysplex, or some AIX processors in the form of an SP2®. Processors linked in these ways benefit from support from each other and have access to a far wider range of programs and data.

In the same way, MQSeries queue managers can be connected to form a *cluster*. This facility is available to queue managers on the following platforms:
- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

The queue managers can be connected using any of the communications protocols that are available on your platform. That is, TCP or LU 6.2 on any platform, and in addition, NetBIOS or SPX on OS/2 or Windows NT, and UDP on AIX. Connections on more than one protocol may exist within a cluster. Of course, if you try to make a connection to a queue manager using a protocol that it does not support the channel will not become active.

Comparison with distributed queuing

If you do not use clusters, your queue managers are independent and communicate using distributed queuing. If one queue manager needs to send messages to another it must have defined:
- A transmission queue
- A channel to the remote queue manager
- A remote-queue definition for every queue to which it wants to send messages

Figure 1 on page 4 shows the components required for distributed queuing.

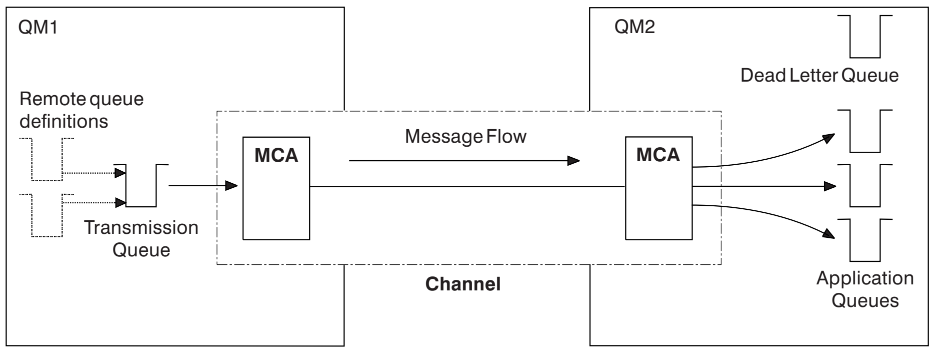

Figure 1. Distributed queuing

If you group queue managers in a cluster, the queue managers can make the queues that they host available to every other queue manager in the cluster. Any queue manager can send a message to any other queue manager in the same cluster without the need for explicit channel definitions, remote-queue definitions, or transmission queues for each destination. Every queue manager in a cluster has a single transmission queue from which it can transmit messages to any other queue manager in the cluster. Each queue manager in a cluster needs to define only:

- One cluster-receiver channel on which to receive messages
- One cluster-sender channel with which it introduces itself and learns about the cluster

Overview of cluster components

Figure 2 on page 5 shows the components of a cluster called CLUSTER.

- In this cluster there are three queue managers, QM1, QM2, and QM3.
- QM1 and QM2 host repositories of information about the queue managers in the cluster. They are referred to as *repository queue managers*. (The repositories are represented in the diagram by the shaded cylinders.)
- QM2 and QM3 host some queues that are accessible to any other queue manager in the cluster. These are called *cluster queues*. (The cluster queues are represented in the diagram by the shaded queues.)

  As with distributed queuing, an application uses the MQPUT call to put a message to a cluster queue at *any* queue manager. An application uses the MQGET call to retrieve messages from a cluster queue on the local queue manager.
- Each queue manager has a definition for the receiving end of a channel called TO.*qmgr* on which it can receive messages. This is a *cluster-receiver channel*. A cluster-receiver channel is similar to a receiver channel used in distributed queuing, but in addition to carrying messages this channel can also carry information about the cluster.
- Each queue manager also has a definition for the sending end of a channel, which connects to the cluster-receiver channel of one of the repository queue managers. This is a *cluster-sender channel*. In Figure 2 on page 5, QM1 and QM3 have cluster-sender channels connecting to TO.QM2. QM2 has a cluster-sender channel connecting to TO.QM1. A cluster-sender channel is similar to a sender channel used in distributed queuing, but in addition to carrying messages this channel can also carry information about the cluster.

  Once both the cluster-receiver end and the cluster-sender end of a channel have been defined, the channel is started automatically.

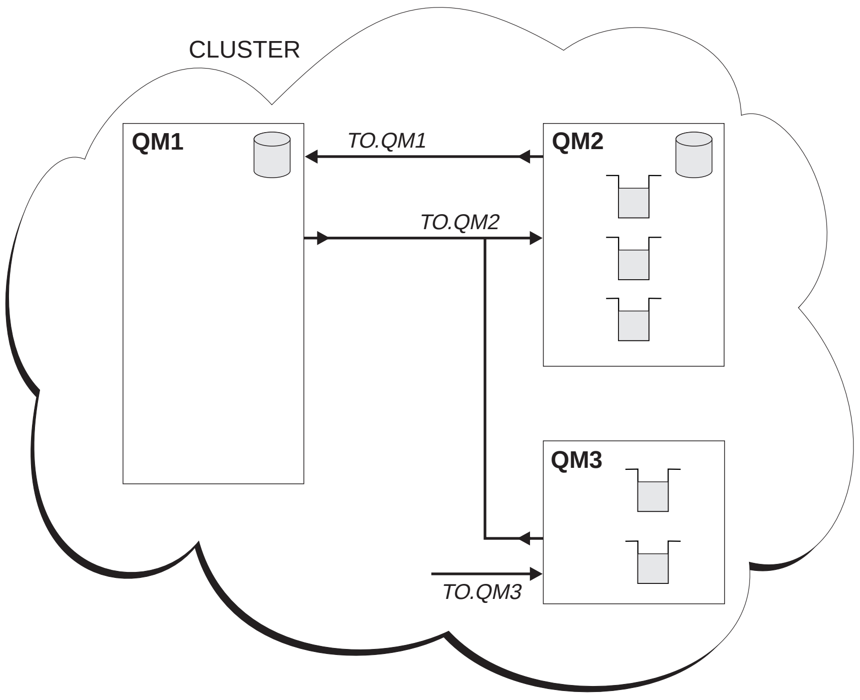

Figure 2. A cluster of queue managers

Terminology

Before proceeding to the next chapter it is useful to understand the following terminology:

Cluster
: A cluster is a network of queue managers that are logically associated in some way. The queue managers in a cluster may be physically remote. For example, they might represent the branches of an international chain store and be physically located in different countries. Each cluster within an enterprise should have a unique name.

Cluster queue manager
: A cluster queue manager is a queue manager that is a member of a cluster. A queue manager may be a member of more than one cluster. (See "Overlapping clusters" on page 61.) Each cluster queue manager must have a name that is unique throughout all the clusters of which it is a member.

A cluster queue manager may host queues, which it *advertises* to the other queue managers in the cluster. A cluster queue manager does not have to host or advertise any queues. It may just feed messages into the cluster and receive only responses that are directed explicitly to it, and not to advertised queues.

Cluster queue managers are autonomous. They have full control over queues and channels that they define. Their definitions cannot be modified by other queue managers. When you make or alter a definition on a cluster queue manager, the information is sent to the repository queue manager and the repositories in the cluster are updated accordingly.

Cluster queue
: A cluster queue is a queue that is hosted by a cluster queue manager and made available to other queue managers in the cluster. The cluster queue

manager makes a local queue definition for the queue specifying the name of the cluster that the queue is to be available in. This definition has the effect of advertising the queue to the other queue managers in the cluster. The other queue managers in the cluster can put messages to a cluster queue without needing a corresponding remote-queue definition. A cluster queue can be advertised in more than one cluster.

Repository

A repository is a collection of information about the queue managers that are members of a cluster. This information includes queue-manager names, their locations, their channels, what queues they host, and so on. The information is stored in the form of messages on a queue called SYSTEM.CLUSTER.REPOSITORY.QUEUE. (This queue is one of the default objects created on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT and is defined as part of queue-manager customization on MQSeries for OS/390.) Typically, two queue managers in a cluster hold a *full repository*. The remaining queue managers all hold a *partial repository*.

Repository queue manager

A repository queue manager is a cluster queue manager that holds a *full* repository. To ensure availability, you are recommended to set up two or more repository queue managers in each cluster. The repository queue managers receive information sent by the other queue managers in the cluster and update their repositories accordingly. The repository queue managers send messages to each other to be sure that they are both kept up to date with new information about the cluster.

Full repository and partial repository

A repository queue manager hosts a *complete* set of information about every queue manager in the cluster. This set of information is called the *repository* or sometimes the *full repository*.

The other queue managers in the cluster inquire on the information in the full repositories and build up their own subsets of this information in partial repositories. A queue manager's partial repository contains information about only those queue managers with which the queue manager needs to exchange messages. The queue managers request updates to the information they need, so that if it changes, the repository queue manager will send them the new information. For much of the time a queue manager's partial repository has all the information it needs to perform within the cluster. When a queue manager needs some additional information it makes inquiries of the full repository and updates its partial repository. The queue managers use a queue called SYSTEM.CLUSTER.COMMAND.QUEUE to request and receive updates to the repositories. This queue is one of the default objects created on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT and is defined as part of queue-manager customization on MQSeries for OS/390.

Cluster-receiver channel

A cluster-receiver (CLUSRCVR) channel definition defines the receiving end of a channel on which a cluster queue manager can receive messages from other queue managers in the cluster. A cluster-receiver channel can also carry information about the cluster—information destined for the repository. The definition of a cluster-receiver channel has the effect of advertising that a queue manager is available to receive messages. You need at least one cluster-receiver channel for each cluster queue manager.

Cluster-sender channel

A cluster-sender (CLUSSDR) channel definition defines the sending end of a channel on which a cluster queue manager can send cluster information to one of the full repositories. The cluster-sender channel is used to notify the repository of any changes to the queue manager's status, for example the addition or removal of a queue. It is also used to transmit messages.

The repository queue managers themselves have cluster-sender channels that point to each other. They use them to communicate cluster status changes to each other.

It is of little importance which repository a queue manager's CLUSSDR channel definition points to. Once the initial contact has been made, further cluster-sender channels are defined automatically as necessary so that the queue manager can send cluster information to every repository, and messages to every queue manager.

Cluster transmission queue

Each cluster queue manager has a cluster transmission queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE. The cluster transmission queue transmits all messages from the queue manager to any other queue manager that is in the same cluster. This queue is one of the default objects created on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT and is defined as part of queue-manager customization on MQSeries for OS/390.

Binding

You may create a cluster in which more than one queue manager hosts an instance of the same cluster queue. This is discussed in "More than one instance of a queue" on page 41. If you do this, you may need to ensure that a sequence of messages are all sent to the **same** instance of the queue. You can bind a series of messages to a particular queue by using the MQOO_BIND_ON_OPEN option on the MQOPEN call (see "MQOPEN" on page 49).

Benefits

There are two quite different reasons for using clusters.

1. Reduced system administration.

   As soon as you start to establish even a small cluster you will benefit from simplified system administration. Establishing a network of queue managers in a cluster involves fewer definitions than establishing a network that is to use distributed queuing. With fewer definitions to make, you can set up or change your network more quickly and easily, and the risk of making an error in your definitions is reduced.

2. Increased availability and workload balancing.

   You may be content to use simple clusters and benefit from the easier system administration. It is not necessary or applicable in every case to consider the availability and workload balancing aspects. If you do choose to use more complicated clusters, you will benefit from scalability of the number of instances of a queue you can define, and therefore from greater availability. Because you can define instances of the same queue on more than one queue manager the workload can be distributed throughout the queue managers in a cluster.

These two objectives are discussed in detail in Chapter 2. Using clusters to ease system administration and Chapter 5. Using clusters for workload management.

Things to consider

- On OS/390 you cannot use clustering if you are using CICS® for distributed queuing.
- To get the most benefit out of clusters, all the queue managers in the network must be on a platform that supports clusters. Until all your systems are migrated to a platform that supports clusters, you may have queue managers outside a cluster that are not able to access your cluster queues without extra manual definitions.
- If two clusters with the same name are merged, it is not possible to separate them again. Therefore it is advisable to give all clusters a unique name.
- If a message arrives at a queue manager but there is no queue there to receive it, the message is put to the dead-letter queue as usual. (If there is no dead-letter queue, the channel fails and retries, as described in the *MQSeries Intercommunication* book.)
- The integrity of persistent messages is maintained. Messages are not duplicated or lost as a result of using clusters.
- Using clusters reduces system administration. Clusters make it easy to connect larger networks with many more queue managers than you would be able to contemplate using distributed queuing. However, as with distributed queuing, there is a risk that you may consume excessive network resources if you attempt to enable communication between *every* queue manager in a cluster.
- If you use the MQSeries Explorer, which presents the queue managers in a tree structure, the view for large clusters may be cumbersome.
- The MQSeries Explorer cannot administer a cluster with repository queue managers on MQSeries for OS/390. You must nominate an additional repository on a system that the MQSeries Explorer can administer.
- The purpose of distribution lists, which are supported on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT, is to use a single MQPUT command to send the same message to multiple destinations. You can use distribution lists in conjunction with queue manager clusters. However, in a clustering environment all the messages are expanded at MQPUT time and so the advantage, in terms of network traffic, is not so great as in a non-clustering environment. The advantage of distribution lists, from the administrator's point of view, is that the numerous channels and transmission queues do not need to be defined manually.
- If you are going to use clusters to achieve workload balancing, you must examine your applications to see whether they require messages to be processed by a particular queue manager or in a particular sequence. Such applications are said to have *message affinities*. **You may need to modify your applications before you can use them in complex clusters**.
- If you use the MQOO_BIND_ON_OPEN option on an MQOPEN call to force messages to be sent to a specific destination, and the destination queue manager is not available, the messages are not delivered. Messages are not routed to another queue manager because of the risk of duplication.
- Take care when using clustering in an environment where IP addresses change on an unpredictable basis, for example on machines where Dynamic Host Configuration Protocol (DHCP) is being used. If you specify the IP address rather than the hostname in your channel definitions, you must update the IP address each time DHCP issues a new one.

Summary of the concepts

If you are familiar with MQSeries and distributed queuing, you may like to think of a cluster as a network of queue managers maintained by a conscientious systems administrator. Whenever you create a receiver channel or define a queue, the systems administrator automatically creates corresponding sender channels and remote-queue definitions on the other queue managers.

You do not need to make transmission queue definitions because MQSeries provides a transmission queue on each queue manager. This single transmission queue can be used to carry messages to any other queue manager.

All the queue managers that join a cluster agree to work in this way. They send out information about themselves and about the queues they host, and they receive information about the other members of the cluster.

This information is stored in repositories. Most queue managers retain only the information that they need, that is, information about queues and queue managers with which they need to communicate. Some queue managers retain a full repository of **all** the information about **all** queue managers in the cluster.

A cluster-receiver channel is a communication channel similar to a receiver channel. When you define a cluster-receiver channel, not only is the object created on your queue manager, but also information about the channel and the queue manager that owns it is stored in the repositories. The definition of a cluster-receiver channel is a queue manager’s initial introduction to a cluster. Once it has been defined, other queue managers can automatically make corresponding definitions for the cluster-sender end of the channel as needed.

A cluster-sender channel is a communication channel similar to a sender channel. You need a cluster-sender channel only if you want to communicate with another cluster queue manager. When another cluster queue manager wants to communicate with you, your cluster-sender channel is created automatically by reference to the appropriate cluster-receiver channel definition. However, each queue manager must have one manually defined cluster-sender channel, through which it makes its initial contact with the cluster.

Queue managers on platforms that support clusters do not have to be part of a cluster. You can continue to use distributed queuing techniques as well as, or instead of, using clusters.

Chapter 2. Using clusters to ease system administration

This chapter describes how you can use clusters to simplify system administration in your environment. It is intended for users who have not used clusters before and who want to learn how they might benefit from setting up and using a simple cluster. This chapter covers:

- "How can I use clusters?"
- "How does the system administrator benefit?" on page 12
- "What about my applications?" on page 14
- "How should I prepare for use of clustering?" on page 14
- "How do I set up a cluster?" on page 15

For information about how to set up a more complex cluster that benefits from workload management, refer to "Chapter 5. Using clusters for workload management" on page 41.

How can I use clusters?

Typically a cluster contains queue managers that are logically related in some way and need to share some data or applications. For example you may have one queue manager for each department in your company, managing data and applications specific to that department. You could group all these queue managers into a cluster so that they all feed into the PAYROLL application. Or you may have one queue manager for each branch of your chain store, managing the stock levels and other information for that branch. If you group these queue managers into a cluster, they are all able to access the same set of SALES and PURCHASES applications, which are held centrally, perhaps on the head-office queue manager.

Once a cluster has been set up, the queue managers within it can communicate with each other without the need for any channel definitions or remote-queue definitions.

You can convert an existing network of queue managers into a cluster or you can establish a cluster straightaway, when setting up a new network.

| An MQSeries client can connect to a queue manager that is part of a cluster, just as
| it can connect to any other queue manager. See the *MQSeries Clients* book for more
| information about clients.

© Copyright IBM Corp. 1999, 2000

How does the system administrator benefit?

Using clusters leads to easier administration of a network. Look at Figure 3, which shows four queue managers each with two queues. Let us consider how many definitions are needed to connect these queue managers using distributed queuing. Then we will see how many definitions are needed to set up the same network as a cluster.

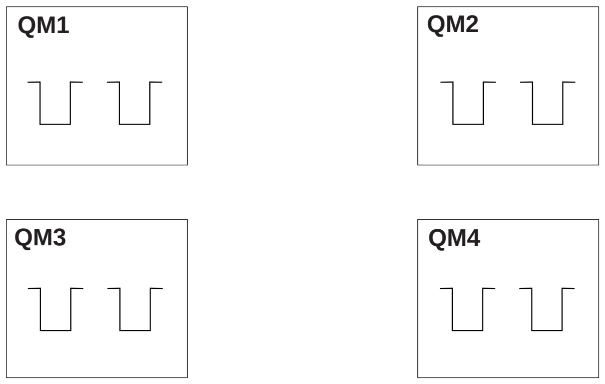

Figure 3. A network of four queue managers

Definitions to set up a network using distributed queuing

To set up the network shown in Figure 3 using distributed queuing, you might have the following definitions:

Table 1. Definitions for distributed queuing

Description	**Number per queue manager**	**Total number**
A sender-channel definition for a channel on which to send messages to every other queue manager	3	12
A receiver-channel definition for a channel on which to receive messages from every other queue manager	3	12
A transmission-queue definition for a transmission queue to every other queue manager	3	12
A local-queue definition for each local queue	2	8
A remote-queue definition for each remote queue to which this queue manager wants to put messages	6	24
Optionally, on OS/390, a process definition specifying trigger data if channels are to be triggered	3	12

While you might reduce this number of definitions by, for example, using generic receiver-channel definitions, the maximum number of definitions could be as many as 20 on each queue manager, which is a total of 80 for this network.

Definitions to set up a network using clusters

When using clusters, you need:
- Just one CLUSSDR and one CLUSRCVR definition at each queue manager
- No separately defined transmission queues
- No remote-queue definitions

Therefore, to set up the network shown in Figure 3 on page 12 using clusters you need the following definitions:

Table 2. Definitions for clustering

Description	Number per queue manager	Total number
A cluster-sender channel definition for a channel on which to send messages to a repository queue manager	1	4
A cluster-receiver channel definition for a channel on which to receive messages from other queue managers in the cluster	1	4
A local-queue definition for each local queue	2	8

To set up this cluster of queue managers (with two full repositories), you would need 4 definitions on each queue manager — a total of 16 definitions all together. You would also need to alter the queue-manager definitions for two of the queue managers, to make them repository queue managers for the cluster.

The CLUSSDR and CLUSRCVR channel definitions need be made only once. When the cluster is in place you can add or remove queue managers (other than the repository queue managers) without any disruption to the other queue managers.

Clearly, this amounts to a significant reduction in the number of definitions required to set up a network containing a large number of queue managers.

With fewer definitions to make there is less risk of error.
- There is no danger of a mismatch of object names, for example the channel name in a sender-receiver pair.
- You do not need to worry that the transmission queue name specified in a channel definition matches the correct transmission queue definition or matches the transmission queue name specified in a remote queue definition.
- There is no danger of a QREMOTE definition pointing to the wrong queue at the remote queue manager.

Furthermore, once a cluster is set up, you can move cluster queues from one queue manager to another within the cluster without having to do any system management work on any other queue manager. There is no danger of forgetting to delete or modify channel, remote-queue, or transmission-queue definitions. You can add new queue managers to a cluster without any disruption to the existing network.

What about my applications?

You need not make any alterations to your applications if you are going to set up a simple MQSeries cluster. The application names the target queue on the MQOPEN call as usual and need not be concerned about the location of the queue manager.

However, if you set up a cluster in which there are multiple definitions for the same queue, as described in "Chapter 5. Using clusters for workload management" on page 41, you must review your applications and modify them as necessary.

How should I prepare for use of clustering?

You can create queue-manager clusters on the following platforms:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

Installation procedures for your platform are described in:

- The *MQSeries for AIX V5.1 Quick Beginnings* book
- The *MQSeries for AS/400 V5.1 Quick Beginnings* book
- The *MQSeries for OS/2 Warp V5.1 Quick Beginnings* book
- The *MQSeries for HP-UX V5.1 Quick Beginnings* book
- The *MQSeries for Sun Solaris V5.1 Quick Beginnings* book
- The *MQSeries for Windows NT V5.1 Quick Beginnings* book
- The *MQSeries for OS/390 Program Directory*

V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT

After installing the product, you have to create queue managers. You use the command crtmqm to create a queue manager with all the default objects that you need. Creation of queue managers is described in the *MQSeries System Administration* book and in the *MQSeries for AS/400 V5.1 System Administration* book.

MQSeries for OS/390

You need to customize your queue managers as described in the *MQSeries for OS/390 System Management Guide*. Ensure that you customize the queue managers to use distributed queuing and clustering.

How do I set up a cluster?

Having decided that you want to create a cluster of certain queue managers, you need to consider which queue managers in the cluster are to hold the full repositories of cluster information. You can choose any number of queue managers for this purpose but the recommended number is two. See “Selecting queue managers to hold repositories” on page 59 for more information.

The smallest possible cluster would contain only two queue managers. In this case both queue managers would contain full repositories. You need only a small number of definitions to set this up, and yet there is a high degree of autonomy at each queue manager.

Figure 4 shows a cluster of two queue managers. You could set up a cluster like this using MQSeries commands (MQSC), or any other type of administration command or utility that is available on your platform. See “Chapter 6. MQSeries commands” on page 53 for more information.

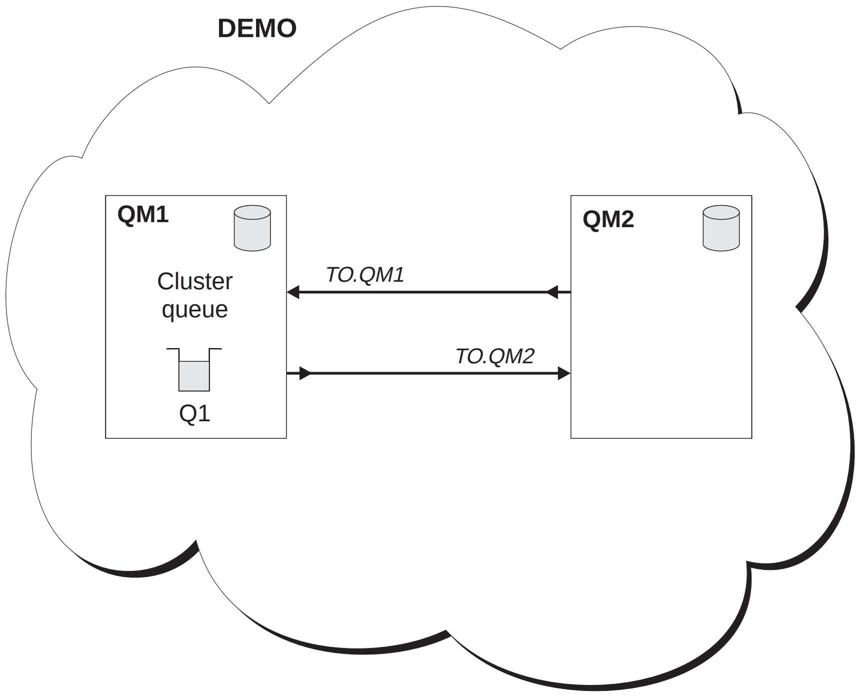

Figure 4. A small cluster of two queue managers

The steps you should take to set up a cluster like this are described in “Task 1: Setting up a new cluster” on page 19.

Establishing communication in a cluster

To establish communication between queue managers in a cluster you need to configure a link using one of the supported communication protocols. The supported protocols are TCP or LU 6.2 on any platform, and in addition NetBIOS or SPX on OS/2 or Windows NT, and UDP on AIX. Configuring communication links is described in detail in the *MQSeries Intercommunication* book. As part of this configuration, you also need channel initiators and channel listeners just as you do with distributed queuing.

Channel initiator

All cluster queue managers need a channel initiator to monitor the system-defined initiation queue SYSTEM.CHANNEL.INITQ. This is the initiation queue for all transmission queues including the cluster transmission queue.

MQSeries for OS/390

There is one channel initiator for each queue manager and it runs as a separate address space. You start it using the MQSeries command START CHINIT, which you would normally issue as part of your queue manager startup.

V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT

When you start a queue manager, a channel initiator is automatically started too.

Channel listener

You need to run a channel listener program on each queue manager. A channel listener program ‘listens’ for incoming network requests and starts the appropriate receiver channel when it is needed.

The implementation of channel listeners is platform specific.

MQSeries for OS/390

Use the channel listener program provided by MQSeries. To start an MQSeries channel listener use the MQSeries command START LISTENER, which you would normally issue as part of your channel initiator startup. For example:

```
START LISTENER PORT(1414) TRPTYPE(TCP)
```

| **MQSeries for AS/400**

| Use the channel listener program provided by MQSeries. To start an
| MQSeries channel listener use the MQSeries for AS/400 CL command
| STRMQMLSR. For example:

```
| STRMQMLSR MQMNAME(QM1) PORT(1414)
```

| Alternatively, you could use the MQSeries command START LISTENER.

MQSeries for OS/2 Warp

Use either the channel listener program provided by MQSeries, or inetd, or the facilities provided by the operating system (for example, Attach manager for LU 6.2 communications).

To start the MQSeries channel listener use the RUNMQLSR command. For example:

```
RUNMQLSR -t tcp -p 1414 -m QM1
```

| Alternatively, you could use the MQSeries command START LISTENER.

To use inetd to start channels, two files must be configured:

1. Edit the file TCPIP\ETC\SERVICES. If you do not have the following line in that file, add it as shown:

   ```
   MQSeries       1414/tcp      # MQSeries channel listener
   ```

   where 1414 is the port number required for MQSeries. You can change this, but it must match the port number specified at the sending end.
2. Edit the file TCPIP\ETC\INETD.LST. If you do not have the following line in that file, add it as shown:

   ```
   MQSeries      tcp C:\MQM\BIN\AMQCRSTA -m queue.manager.name
   ```

   where *queue.manager.name* is the name of your queue manager. If you have MQSeries for OS/2 Warp installed on a different drive, replace the `C:` with the correct drive letter.

MQSeries for Windows NT

Use either the channel listener program provided by MQSeries, or the facilities provided by the operating system.

To start the MQSeries channel listener use the RUNMQLSR command. For example:

```
RUNMQLSR -t tcp -p 1414 -m QM1
```

| Alternatively, you could use the MQSeries command START LISTENER.

MQSeries on UNIX® systems

Use either the channel listener program provided by MQSeries, or the facilities provided by the operating system (for example, inetd for TCP communications).

To start the MQSeries channel listener use the runmqlsr command. For example:

```
runmqlsr -t tcp -p 1414 -m QM1
```

Alternatively, you could use the MQSeries command START LISTENER.

To use inetd to start channels, two files must be configured:

1. Edit the file /etc/services. (To do this you must be logged in as a superuser or root.) If you do not have the following line in that file, add it as shown:

   ```
   MQSeries       1414/tcp      # MQSeries channel listener
   ```

   where 1414 is the port number required by MQSeries. You can change this, but it must match the port number specified at the sending end.
2. Edit the file /etc/inetd.conf. If you do not have the following line in that file, add it as shown:

   For AIX:

   ```
   MQSeries stream tcp nowait root /usr/mqm/bin/amqcrsta amqcrsta
   -m queue.manager.name
   ```

   For Sun Solaris or HP-UX:

   ```
   MQSeries stream tcp nowait mqm /opt/mqm/bin/amqcrsta amqcrsta
   -m queue.manager.name
   ```

The updates become active after inetd has reread the configuration files. Issue the following commands from the root user ID:

On AIX:

```
refresh -s inetd
```

On HP-UX:

```
inetd -c
```

On Sun Solaris:

1. Find the process ID of the inetd with the command:

   ```
   ps -ef | grep inetd
   ```
2. Run the command:

   ```
   kill -1 inetd processid
   ```

Chapter 3. First tasks

This chapter shows how you can perform the following tasks:
- "Task 1: Setting up a new cluster"
- "Task 2: Adding a new queue manager to a cluster" on page 25

Much of the information you need to achieve these tasks is documented elsewhere in the MQSeries library. This chapter gives pointers to that information and fills in details relating specifically to work with clusters.

Notes:

1. Throughout the examples in this chapter and Chapter 9. Advanced tasks the queue managers have illustrative names such as LONDON and NEWYORK. Don't forget that on MQSeries for OS/390, queue-manager names are limited to 4 characters.
2. The names of the queue managers imply that each queue manager is on a separate machine. You could just as easily set up these examples with all the queue managers on the same machine.
3. The examples in these chapters show MQSeries Commands (MQSC) as they would be entered by the system administrator at the command console. For information about other ways of entering commands, refer to "Chapter 6. MQSeries commands" on page 53.

Task 1: Setting up a new cluster

Scenario:

- You are setting up a new MQSeries network for a chain store. The store has two branches, one in London and one in New York. The data and applications for each store are hosted by systems running separate queue managers. The two queue managers are called LONDON and NEWYORK.
- The inventory application runs on the system in New York, connected to queue manager NEWYORK. The application is driven by the arrival of messages on the INVENTQ queue, hosted by NEWYORK.
- The two queue managers, LONDON and NEWYORK, are to be linked in a cluster called 'INVENTORY' so that they can both put messages to the INVENTQ.
- The network protocol is TCP.

Figure 5 on page 22 shows what this cluster is to look like.

Note: On MQSeries for Windows NT you can use one of the wizards supplied with MQSeries Explorer to create a new cluster similar to the one accomplished by this task.

The steps required to complete task 1

To set up this cluster, follow these steps.

1. Prepare the queue managers

Preparation of queue managers is described in “How should I prepare for use of clustering?” on page 14.

2. Decide on the organization of the cluster and its name

You have decided to link the two queue managers, LONDON and NEWYORK, into a cluster. A cluster with only two queue managers offers only marginal benefit over a network that is to use distributed queuing, but is a good way to start and does provide scope for future expansion. When you open new branches of your store, you will be able to add the new queue managers to the cluster easily and without any disruption to the existing network. “Task 2: Adding a new queue manager to a cluster” on page 25 describes how to do this.

For the time being the only application you are running is the inventory application. The cluster name is INVENTORY.

3. Determine which queue managers should hold full repositories

In any cluster you need to nominate at least one queue manager, or preferably two, to hold full repositories. See “Selecting queue managers to hold repositories” on page 59 for more information. In this example there are only two queue managers, LONDON and NEWYORK. You are recommended to make both of these queue managers hold full repositories.

Notes:

1. The remaining steps may be performed in any order. The sequence shown is logical but need not be adhered to.
2. As you proceed through the steps, you may notice that warning messages are written to the queue-manager log or the OS/390 system console if some expected definitions have yet to be made.

```
Examples of the responses to the commands are shown in a box
like this after each step in this task.
These examples show the responses returned by MQSeries for AIX.
The responses vary on other platforms.
```

3. Before proceeding with these steps make sure that the queue managers are started.

4. Alter the queue-manager definitions to add repository definitions

On each queue manager that is to hold a full repository, you need to make an alteration to the queue-manager definition. Do this using the ALTER QMGR command and specifying the REPOS attribute:

```
ALTER QMGR REPOS(INVENTORY)
```

```
     1 : ALTER QMGR REPOS(INVENTORY)
AMQ8005: MQSeries queue manager changed.
```

5. Define the CLUSRCVR channels

On every queue manager in a cluster you need to define a cluster-receiver channel on which the queue manager can receive messages. This definition defines the queue manager's network address and has the effect of advertising the queue manager's availability to receive messages from other queue managers in the cluster.

On the LONDON queue manager, define:

```
DEFINE CHANNEL(TO.LONDON) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(LONDON.CHSTORE.COM) CLUSTER(INVENTORY)
```

```
     1 : DEFINE CHANNEL(TO.LONDON) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
         CONNAME(LONDON.CHSTORE.COM) CLUSTER(INVENTORY)
AMQ8014: MQSeries channel created.
07/09/98  12:56:35  No repositories for cluster 'INVENTORY'
```

In this example the channel name is TO.LONDON, the transport protocol is TCP, and the network address (CONNAME) of the machine the queue manager resides on is LONDON.CHSTORE.COM. The CLUSTER keyword causes the definition to be advertised to the other queue managers in the cluster. The machine's network address is stored in the repositories, where it can be referenced by other queue managers. (The network address could be entered either as an alphanumeric DNS hostname, or as a dotted-decimal IP address.)

On the NEWYORK queue manager, define:

```
DEFINE CHANNEL(TO.NEWYORK) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(NEWYORK.CHSTORE.COM) CLUSTER(INVENTORY)
```

6. Define the CLUSSDR channels

On every queue manager in a cluster you need to define one cluster-sender channel on which the queue manager can send messages to one of the repository queue managers. In this case there are only two queue managers, both of which hold repositories. They must each have a CLUSSDR definition that points to the CLUSRCVR channel defined at the other queue manager. Note that the channel names given on the CLUSSDR definitions must match those on the corresponding CLUSRCVR definitions.

On the LONDON queue manager, define:

```
DEFINE CHANNEL(TO.NEWYORK) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(NEWYORK.CHSTORE.COM) CLUSTER(INVENTORY)
```

```
     1 : DEFINE CHANNEL(TO.NEWYORK) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
         CONNAME(NEWYORK.CHSTORE.COM) CLUSTER(INVENTORY)
AMQ8014: MQSeries channel created.
07/09/98  13:00:18    Channel program started.
```

On the NEWYORK queue manager, define:

```
DEFINE CHANNEL(TO.LONDON) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(LONDON.CHSTORE.COM) CLUSTER(INVENTORY)
```

Once a queue manager has definitions for both a cluster-receiver channel and a cluster-sender channel in the same cluster, the cluster-sender channel is started.

7. Define the cluster queue INVENTQ

Define the INVENTQ queue on the NEWYORK queue manager, specifying the CLUSTER keyword.

```
DEFINE QLOCAL(INVENTQ) CLUSTER(INVENTORY)
```

```
     1 : DEFINE QLOCAL(INVENTQ) CLUSTER(INVENTORY)
AMQ8006: MQSeries queue created.
```

| The CLUSTER keyword causes the queue to be advertised to the cluster. As soon
| as the queue is defined it becomes available to the other queue managers in the
| cluster. They can send messages to it without having to make a remote-queue
| definition for it.

Now that you have completed all the definitions, if you have not already done so you should start the channel initiator on MQSeries for OS/390 and, on all platforms, start a listener program on each queue manager. The listener program listens for incoming network requests and starts the cluster-receiver channel when it is needed. See "Establishing communication in a cluster" on page 16 for more information.

The cluster achieved by task 1

The cluster set up by this task looks like this:

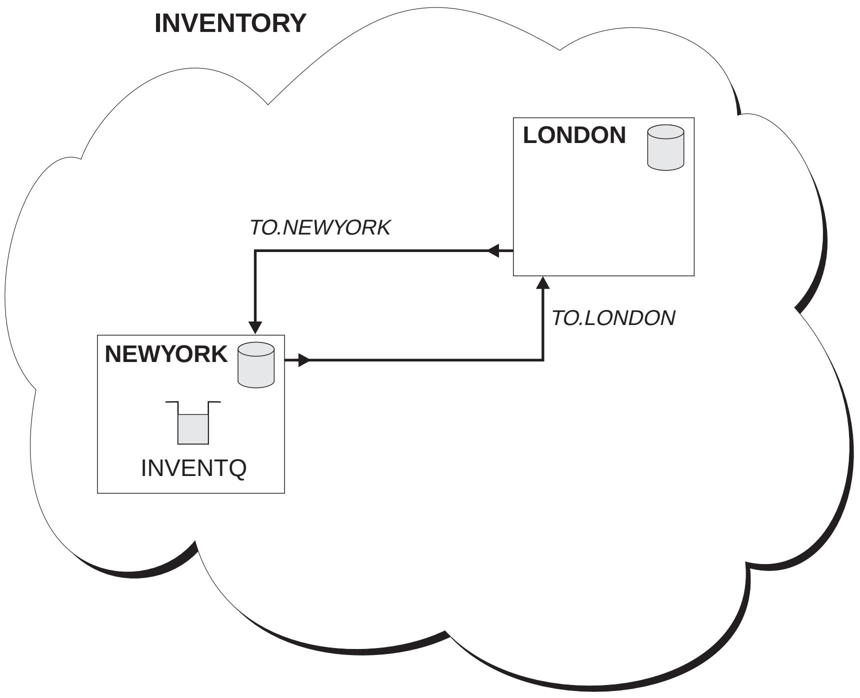

Figure 5. The INVENTORY cluster with two queue managers

Clearly, this is a very small cluster. However, it is useful as a proof of concept. The important thing to understand about this cluster is the scope it offers for future enhancement.

Verifying task 1

You could issue some DISPLAY commands to verify the cluster that you have set up. The responses you see should be similar to those shown in the examples that follow.

From the NEWYORK queue manager, issue the command:

```
dis clusqmgr(*)
```

```
     1 : dis clusqmgr(*)
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(NEWYORK)             CLUSTER(INVENTORY)
   CHANNEL(TO.NEWYORK)
AMQ8441: Display Cluster Queue Manager details.
   CLUSQMGR(LONDON)              CLUSTER(INVENTORY)
   CHANNEL(TO.LONDON)
```

Now issue the corresponding DISPLAY CHANNEL STATUS command:

```
dis chstatus(*)
```

```
     1 : dis chstatus(*)
AMQ8417: Display Channel Status details.
   CHANNEL(TO.NEWYORK)           XMITQ( )
   CONNAME(9.20.40.24)           CURRENT
   CHLTYPE(CLUSRCVR)             STATUS(RUNNING)
AMQ8417: Display Channel Status details.
   CHANNEL(TO.LONDON)            XMITQ(SYSTEM.CLUSTER.TRANSMIT.QUEUE)
   CONNAME(9.20.51.25)           CURRENT
   CHLTYPE(CLUSSDR)              STATUS(RUNNING)
```

Using the cluster set up in task 1

Because the INVENTQ queue has been advertised to the cluster there is no need for remote-queue definitions. Applications running on NEWYORK and applications running on LONDON can put messages to the INVENTQ queue. They can receive responses to their messages by providing a reply-to queue and specifying its name when they put messages.

Test your setup by sending some messages between the two queue managers. In the following example LONDON puts a message to the INVENTQ at NEWYORK and receives a reply on its queue LONDON_reply.

```
On LONDON:

Define a local queue called LONDON_reply
Set the MQOPEN options to MQOO_OUTPUT
Issue the MQOPEN call to open the queue INVENTQ
Set the ReplyToQ name in the message descriptor to LONDON_reply
Issue the MQPUT call to put the message

On NEWYORK:

Set the MQOPEN options to MQOO_BROWSE
Issue the MQOPEN call to open the queue INVENTQ
Issue the MQGET call to get the message from INVENTQ
Retrieve the ReplyToQ name from the message descriptor
Put the ReplyToQ name in the ObjectName field of the object descriptor
Set the MQOPEN options to MQOO_OUTPUT
Issue the MQOPEN call to open LONDON_reply at queue manager LONDON
Issue the MQPUT call to put the message to LONDON_reply

On LONDON:
Set the MQOPEN options to MQOO_BROWSE
Issue the MQOPEN call to open the queue LONDON_reply
Issue the MQGET call to get the message from LONDON_reply
```

Note: The definition for the local queue LONDON_reply does not need the CLUSTER attribute. NEWYORK replies to this queue by explicitly specifying the queue manager name. Another way of doing this would be to use a temporary dynamic queue. See the *MQSeries Application Programming Guide* for more information.

| You could also test the setup of this cluster using the amqsput sample application.

Converting an existing network into a cluster

If you were converting an existing network into a cluster like this, you might not need to follow step 1, depending on what version of MQSeries the existing network was on. In step 7, you would need to alter the existing queue definition. You would also need to delete the remote queue definition at LONDON for the INVENTQ queue. See “Task 7: Converting an existing network into a cluster” on page 81 for an example of this.

Task 2: Adding a new queue manager to a cluster

Scenario:

- The INVENTORY cluster has been set up as described in “Task 1: Setting up a new cluster” on page 19. It contains two queue managers, LONDON and NEWYORK, which both hold full repositories.
- A new branch of the chain store is being set up in Paris and you want to add a queue manager called PARIS to the cluster.
- Queue manager PARIS will send inventory updates to the application running on the system in New York by putting messages on the INVENTQ queue.
- Network connectivity exists between all three systems.
- The network protocol is TCP.

The steps required to complete task 2

To achieve this, follow these steps:

1. Prepare the PARIS queue manager

Preparation of queue managers is described in “How should I prepare for use of clustering?” on page 14.

2. Determine which full repository PARIS should refer to first

Every queue manager in a cluster must refer to one or other of the full repositories in order to gather information about the cluster and so build up its own partial repository. It is of no particular significance which repository you choose because as soon as a new queue manager is added to the cluster it immediately learns about the other repository as well. Information about changes to a queue manager is sent directly to two repositories. In this example we choose to link PARIS to the queue manager LONDON, purely for geographical reasons.

Note: The remaining steps may be performed in any order. Before proceeding with these steps make sure that queue manager PARIS is started.

3. Define a CLUSRCVR channel on queue manager PARIS

Every queue manager in a cluster needs to define a cluster-receiver channel on which it can receive messages. On PARIS, define:

```
DEFINE CHANNEL(TO.PARIS) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(PARIS.CHSTORE.COM) CLUSTER(INVENTORY)
```

This advertises the queue manager’s availability to receive messages from other queue managers in the cluster INVENTORY. There is no need to make definitions on other queue managers for a sending end to the cluster-receiver channel TO.PARIS. These will be made automatically when needed.

4. Define a CLUSSDR channel on queue manager PARIS

Every queue manager in a cluster needs to define one cluster-sender channel on which it can send messages to its initial repository. On PARIS, make the following definition for a channel called TO.LONDON to the queue manager whose network address is LONDON.CHSTORE.COM.

```
DEFINE CHANNEL(TO.LONDON) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(LONDON.CHSTORE.COM) CLUSTER(INVENTORY)
```

Adding a queue manager

Now that you have completed all the definitions, if you have not already done so you should start the channel initiator on MQSeries for OS/390 and, on all platforms, start a listener program on queue manager PARIS. The listener program listens for incoming network requests and starts the cluster-receiver channel when it is needed. See "Establishing communication in a cluster" on page 16 for more information.

The cluster achieved by task 2

The cluster set up by this task looks like this:

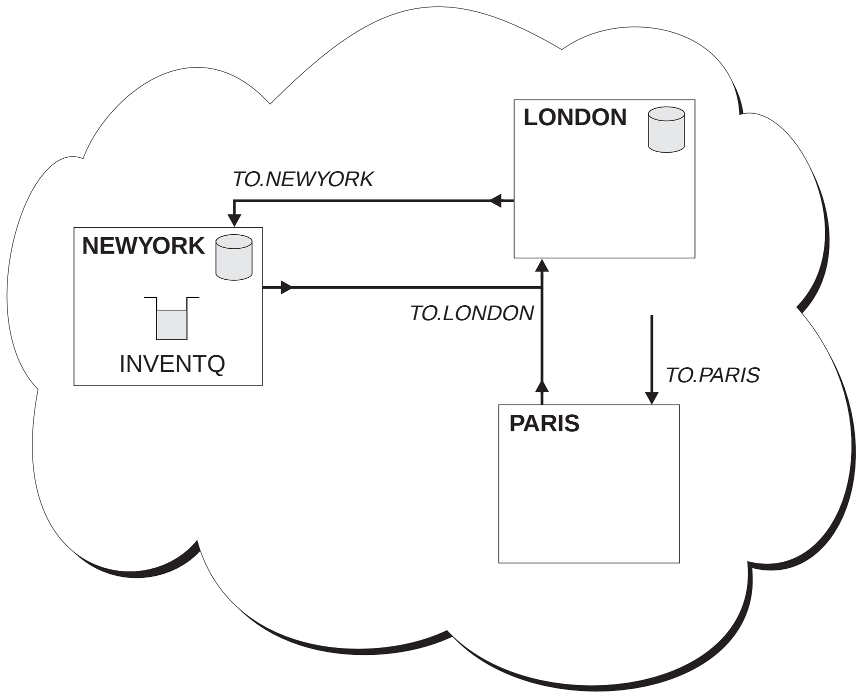

Figure 6. The INVENTORY cluster with three queue managers

By making only two definitions, a CLUSRCVR definition and a CLUSSDR definition, we have added the queue manager PARIS to the cluster.

Now the PARIS queue manager learns, from the repository at LONDON, that the INVENTQ queue is hosted by queue manager NEWYORK. When an application hosted by the system in Paris tries to put messages to the INVENTQ, PARIS automatically defines a cluster-sender channel to connect to the cluster-receiver channel TO.NEWYORK. The application can receive responses when its queue-manager name is specified as the target queue manager and a reply-to queue is provided.

Part 2. Using queue manager clusters

© Copyright IBM Corp. 1999, 2000

Chapter 4. How queue manager clusters work

This chapter provides more detailed information about clusters and how they work. It discusses:

- "Components of a cluster"
- "What makes clustering work?" on page 32
- "Using aliases and remote-queue definitions with clusters" on page 33

Components of a cluster

Let us now consider how the components of a cluster work together and look at some more of the components and features of MQSeries clusters.

Queue managers and repositories

As discussed, every cluster has at least one (preferably two) queue managers holding full repositories of information about the queue managers, queues, and channels in a cluster. These repositories also contain requests from the other queue managers in the cluster for updates to the information.

The other queue managers each hold a partial repository, containing information about the subset of queues and queue managers with which they need to communicate. The queue managers build up their partial repositories by making inquiries when they first need to access another queue or queue manager, and by requesting that thereafter they be notified of any new information concerning that queue or queue manager.

Each queue manager stores its repository information in messages on a queue called SYSTEM.CLUSTER.REPOSITORY.QUEUE. The queue managers exchange repository information in messages on a queue called SYSTEM.CLUSTER.COMMAND.QUEUE.

Each queue manager that joins a cluster defines a cluster-sender (CLUSSDR) channel to one of the repositories. When it does this, it immediately 'learns' which other queue managers in the cluster hold full repositories. From then on the queue manager can request information from any of the repositories. When the queue manager sends out any information about itself, for example when it creates a new queue definition, this information is sent to the chosen repository and also to one other repository (if there is one).

A full repository is updated when the queue manager hosting it receives new information from one of the queue managers that is linked to it. The new information is in fact sent to another repository as well, to reduce the risk of it being delayed if a repository queue manager is out of service. Because all the information is sent twice, the repositories have to discard duplicates. Each item of information carries a sequence number, which the repositories use to identify duplicates. All repositories are kept in step with each other by exchanges of messages between them.

Queues

A queue manager that hosts cluster queues must advertise its queues to the cluster. It does this using the DEFINE QLOCAL command with the CLUSTER option, for example:

© Copyright IBM Corp. 1999, 2000

```
DEFINE QLOCAL(Q1) CLUSTER(SALES)
```

Once a queue has been advertised, any queue manager in the cluster can put messages to it. To put a message, the queue manager has to find out, from the full repositories, where the queue is hosted. Then it adds some routing information to the message and puts the message on its cluster transmission queue.

Cluster transmission queue

Each cluster queue manager has a cluster transmission queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE. A definition for this queue (and others required for clustering) is created by default on every queue manager on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT. On OS/390 it can be defined using the supplied sample CSQ4INSX.

A queue manager that is part of a cluster can send messages on the cluster transmission queue to any other queue manager that is in the same cluster. **Applications must not write directly to the cluster transmission queue. They should write to named queues and not be aware that the queue names resolve to the cluster transmission queue**.

In addition, queue managers may communicate with other queue managers that are not part of a cluster. To do this, a queue manager must define channels and a transmission queue to the other queue manager, in the same way as in a distributed-queuing environment.

During name resolution, the cluster transmission queue takes precedence over the default transmission queue. When a queue manager that is not part of a cluster puts a message to a remote queue, the default action, if there is no transmission queue with the same name as the destination queue manager, is to use the default transmission queue. When the sending queue manager is part of a cluster, the default action is to use SYSTEM.CLUSTER.TRANSMIT.QUEUE, except when the destination queue is not part of the cluster. In short, if the normal resolution takes place, the normal transmission queue is used, however, if the queue is resolved via the repository, SYSTEM.CLUSTER.TRANSMIT.QUEUE is used.

Cluster channels

The *MQSeries Intercommunication* book describes how message channels are used in distributed queuing. Within clusters, messages are distributed between cluster queue managers on a special type of channel for which you need cluster-receiver channel definitions and cluster-sender channel definitions.

A cluster-receiver channel definition defines a channel on which a queue manager can receive messages. A queue manager’s CLUSRCVR definition enables other queue managers to auto-define their corresponding CLUSSDR channel definitions to that queue manager. First, however, each queue manager must manually define a cluster-sender channel. This definition enables the queue manager to make its initial contact with the cluster. It names the repository queue manager to which the queue manager preferentially chooses to send cluster information.

The CLUSSDR definitions made on the repository queue managers are special. All the updates exchanged by the repositories flow exclusively on these channels. This means that the administrator can control the network of repositories explicitly.

Auto-definition of remote queues

When you use distributed queuing, a queue manager that wants to put a message to a queue on a remote queue manager must have a remote-queue definition for that queue.

A queue manager in a cluster does not need a remote-queue definition for remote queues in the cluster. The cluster queue manager finds out the location of a remote queue (from the repository) and then adds routing information to the message and puts it on the cluster transmission queue. MQSeries automatically creates a definition equivalent to a remote-queue definition so that the message can be sent.

You cannot alter or delete an automatically-created remote-queue definition. However, you can look at it using the DISPLAY QUEUE command with the CLUSINFO attribute. For example:

```
DISPLAY QUEUE(*) CLUSINFO
```

See the *MQSeries Command Reference* book for more information about the DISPLAY QUEUE command.

Auto-definition of channels

When you use distributed queuing, a queue manager must have a definition for a sender channel before it can send a message to a remote destination.

However, once a queue manager has joined a cluster by making its initial CLUSSDR and CLUSRCVR definitions, it does not need to make any other definitions for channels to other queue managers in the cluster. MQSeries automatically makes cluster-sender channel definitions when they are needed. Auto-defined cluster-sender channels take their attributes from those specified in the corresponding cluster-receiver channel definition on the receiving queue manager. (Even if there is a manually-defined cluster-sender channel, its attributes may be automatically modified to ensure that they match those on the corresponding cluster-receiver definition.) When both the cluster-sender end and the cluster-receiver end of a channel are defined, the channel is started. An auto-defined channel remains active until it is no longer needed and is shut down using the normal disconnect-interval rules.

You cannot see automatically-defined channels using the DISPLAY CHANNEL command. To see the auto-defined channels use the command DISPLAY CLUSQMGR(qmname) to see all channels defined on the named queue manager. You can also use the command DISPLAY CHSTATUS(channelname) to display the status of the auto-defined CLUSSDR channel corresponding to the CLUSRCVR channel definition you created. For more information about these commands, refer to the *MQSeries Command Reference* book.

You can enable the MQSeries channel auto-definition exit if you wish to write a user exit program to customize a cluster-sender channel or cluster-receiver channel. You might, for example, do this in a cluster environment to:
- Tailor communications definitions, that is, SNA LU 6.2 names
- Add or remove other exits, for example, security exits

See the *MQSeries Intercommunication* book for information about the channel auto-definition exit and how to use it.

What makes clustering work?

The definition of a cluster-sender channel has the effect of introducing a queue manager to one of the repository queue managers. The repository queue manager updates the information in its repository accordingly. Then it automatically creates a cluster-sender channel back to the queue manager, and sends the queue manager information about the cluster. Thus a queue manager learns about a cluster and a cluster learns about a queue manager.

Look again at Figure 2 on page 5. Suppose that queue manager QM1 wants to send some messages to the queues at QM2. It knows what queues are available at QM2, because QM2 has defined a cluster-sender channel to it and so introduced itself. QM1 has defined a cluster-sender channel to QM2, on which it can send messages.

QM3 has introduced itself to QM2. Because QM1 also holds a repository, QM2 has passed on all the information about QM3 to QM1. Therefore, QM1 knows what queues are available at QM3, and what cluster-receiver channel QM3 has defined. If QM1 wants to send some messages to queues at QM3, it automatically creates a cluster-sender channel connecting to the cluster-receiver channel at QM3.

Figure 7 shows the same cluster, and, in addition, shows the two cluster-sender channels that have been created automatically. (These are represented by the two dashed lines that join with the cluster-receiver channel TO.QM3.) It also shows the cluster transmission queue, SYSTEM.CLUSTER.TRANSMIT.QUEUE, which QM1 uses to send its messages. All queue managers in the cluster have a cluster transmission queue, from which they can send messages to any other queue manager in the same cluster.

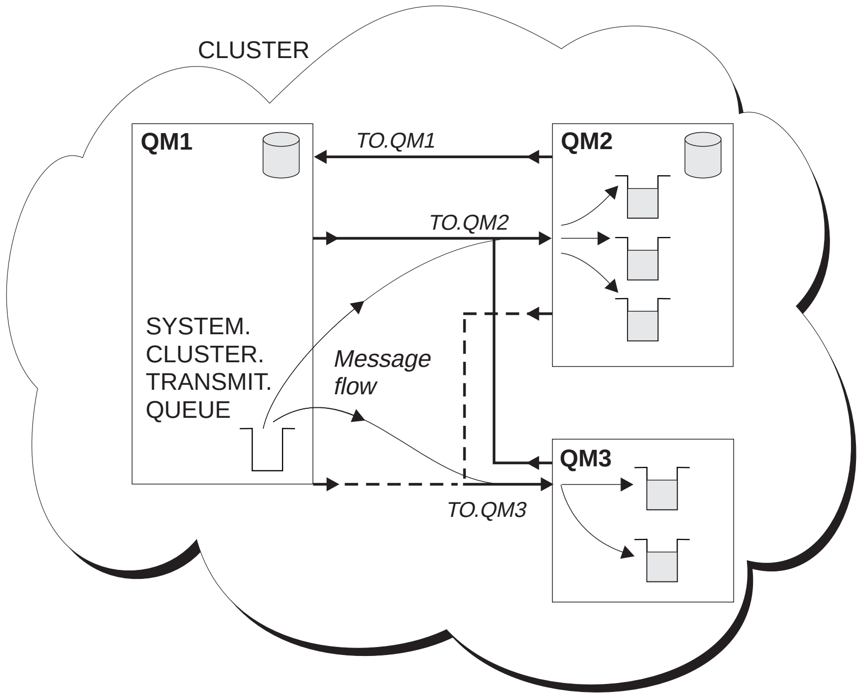

Figure 7. A cluster of queue managers, showing auto-defined channels

The auto-definition of cluster-sender channels—defined automatically when needed—is crucial to the function and efficiency of clusters. However, so that you can see only what **you** need to define to make clusters work, the diagrams throughout this book show only channels (or the receiving ends of channels) for which you make manual definitions.

Using aliases and remote-queue definitions with clusters

There are three types of alias; queue-manager aliases, reply-to queue aliases, and queue aliases. These apply in a clustered environment just as well as in a distributed-queuing environment. This section describes how aliases can be used in conjunction with clusters. When reading this section, it may be useful to refer to the *MQSeries Application Programming Guide*.

Queue-manager aliases

The concept of queue-manager aliasing is described in detail in the *MQSeries Intercommunication* book.

Queue-manager aliases, which are created using a remote-queue definition with a blank RNAME, have three uses:

Remapping the queue-manager name when sending messages

A queue-manager alias can be used to remap the queue-manager name specified in an MQOPEN call to another queue-manager. This may be a cluster queue manager. For example, a queue manager might have the queue-manager alias definition:

```
DEFINE QREMOTE(YORK) RNAME(' ') RQMNAME(CLUSQM)
```

This defines YORK as a queue-manager name that can be used as an alias for the queue manager called CLUSQM. When an application on the queue manager that made this definition puts a message to queue manager YORK, the local queue manager resolves the name to CLUSQM. If the local queue manager is not called CLUSQM, it puts the message on the cluster transmission queue, to be moved to CLUSQM, and changes the transmission header to say CLUSQM instead of YORK.

Note: This does not mean that all queue managers in the cluster will resolve the name YORK to CLUSQM. The definition applies only on the queue manager that makes it. To advertise the alias to the whole cluster, you would need to add the CLUSTER attribute to the remote-queue definition. Then messages from other queue managers that were destined for YORK would be sent to the queue manager with the alias definition and the alias would be resolved.

Aliasing and remote-queue definitions

Altering or specifying the transmission queue when sending messages

This method of aliasing can be used to join a cluster to a non-cluster system. For example, for queue managers in the cluster ITALY to communicate with the queue manager called PALERMO, which is outside the cluster, one of the queue managers in the cluster must act as a gateway. From this queue manager, issue the command:

```
DEFINE QREMOTE(ROME) RNAME(' ') RQMNAME(PALERMO) XMITQ(X) CLUSTER(ITALY)
```

This is a queue-manager alias definition, which defines and advertises ROME as a queue manager over which messages from any queue manager in the cluster ITALY can multi-hop to reach their destination at PALERMO. Any message put to a queue opened with the queue-manager name set to ROME in the open handle, is sent to the gateway queue manager, where the queue manager alias definition was made. Once there, it is put on the transmission queue X and moved by conventional, non-cluster channels to the queue manager PALERMO.

The choice of the name ROME in this example is not significant. The values for QREMOTE and RQMNAME could both be the same.

Determining the destination when receiving messages

When a queue manager receives a message it looks in the transmission header to see the name of the destination queue and queue manager. If it has a queue-manager alias definition with the same name as the queue manager referenced, it substitutes the RQMNAME from its definition in place of the queue manager name in the transmission header.

There are two reasons for using a queue-manager alias in this way:

- To direct messages to another queue manager
- To alter the queue manager name to be the same as the local queue manager

Reply-to queue aliases

A reply-to queue alias definition is used to specify alternative names for reply information. Reply-to queue alias definitions can be used in conjunction with clusters just the same as in a distributed queuing environment. For example:

- Queue manager VENICE sends a message to queue manager PISA using the MQPUT call and specifying the following in the message descriptor:
  ```
  ReplyToQ='QUEUE'
  ReplyToQMgr=''
  ```
- So that replies sent to QUEUE can actually be received on OTHERQ at PISA, you create a reply-to queue alias definition on VENICE:
  ```
  DEFINE QREMOTE(QUEUE) RNAME(OTHERQ) RQMNAME(PISA)
  ```

  This form of the remote-queue definition creates a reply-to alias. This alias is effective only on the system on which it was created.

It is possible for RQMNAME and QREMOTE to specify the same names even if RQMNAME is itself a cluster queue manager.

See the *MQSeries Intercommunication* book for more information.

Queue aliases

A QALIAS definition is used to create an ALIAS by which a queue is to be known. You might do this if, for example, you want to start using a different queue but you do not want to change your applications. You might also do this if for some reason you do not want your applications to know the real name of the queue to which they are putting messages, or because you have a naming convention that differs from the one where the queue is defined. Another reason might be security; your applications may not be authorized to access the queue by its real name but only by its alias.

You create a QALIAS definition on a queue manager using the DEFINE QALIAS command. For example, the command:

```
DEFINE QALIAS(PUBLIC) TARGQ(LOCAL) CLUSTER(C)
```

advertises a queue called PUBLIC to the queue managers in cluster C. PUBLIC is an alias that resolves to the queue that is really called LOCAL. Messages sent to PUBLIC are routed to the queue called LOCAL.

You can also use a queue alias definition to resolve a queue name to a cluster queue. For example the command:

```
DEFINE QALIAS(PRIVATE) TARGQ(PUBLIC)
```

enables a queue manager to use the name PRIVATE to access a queue advertised elsewhere in the cluster by the name PUBLIC. Because this definition does not include the CLUSTER attribute it applies only to the queue manager that makes it.

Examples of using aliases within clusters

Figure 8 and Figure 9 on page 38 show a queue manager called QM3 that is outside the cluster called DEMO. (QM3 could be a queue manager on an MQSeries product that does not support clusters.) QM3 hosts a queue called Q3, which is defined as follows:

```
DEFINE QLOCAL(Q3)
```

Inside the cluster, are two queue managers called QM1 and QM2. QM2 hosts a cluster queue called Q2, which is defined as follows:

```
DEFINE QLOCAL(Q2) CLUSTER(DEMO)
```

To communicate with a queue manager outside the cluster, one or more queue managers inside the cluster must act as a gateway. The gateway in this example is QM1.

Putting from a queue manager outside a cluster

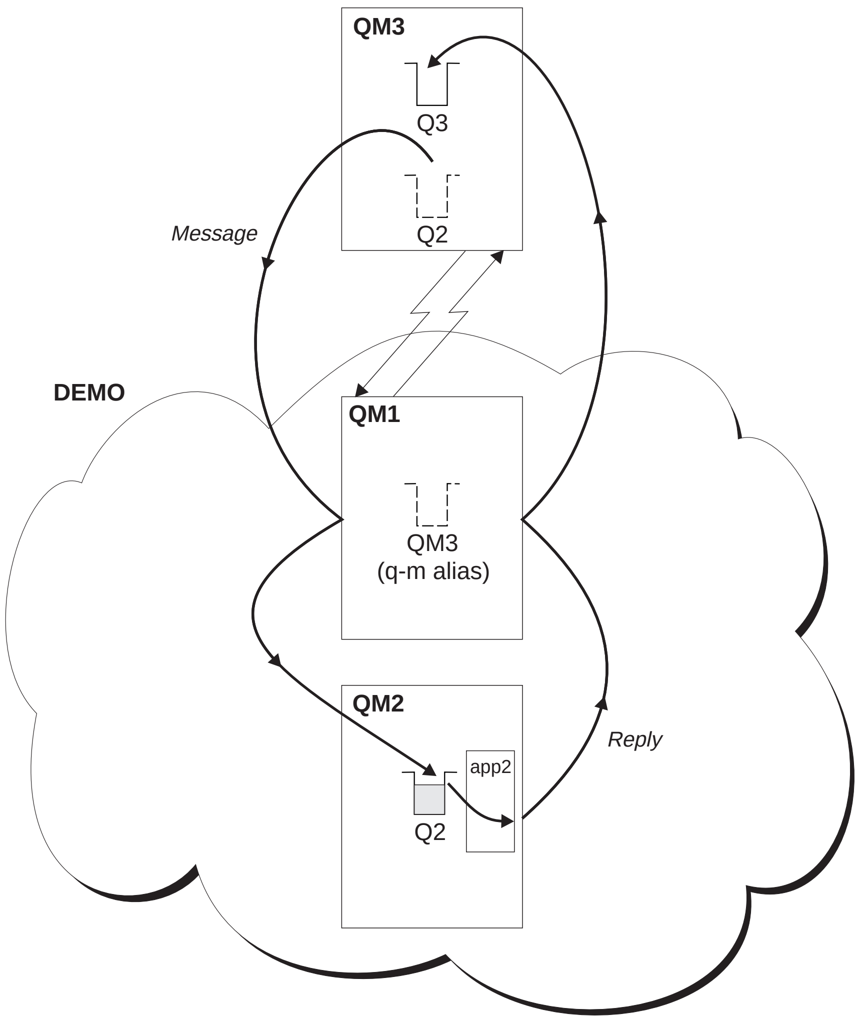

Figure 8. Putting from a queue manager outside the cluster

Let us consider how the queue manager that is outside the cluster can put a message to the queue Q2 at QM2, which is inside the cluster.

The queue manager outside the cluster (QM3 in Figure 8 on page 36) must have a QREMOTE definition for each queue in the cluster that it wants to put messages to. For example:

```
DEFINE QREMOTE(Q2) RNAME(Q2) RQMNAME(QM2) XMITQ(QM1)
```

You can see this remote queue on QM3 in Figure 8 on page 36.

Because QM3 is not part of a cluster, it must communicate using distributed queuing techniques. Therefore, it must also have a sender channel to QM1 and a transmission queue to QM1. QM1 needs a corresponding receiver channel. The channels and transmission queues are not shown explicitly in Figure 8 on page 36.

When an application at QM3 issues an MQPUT call to put a message to Q2, the QREMOTE definition causes the message to be routed through the gateway queue manager QM1.

Replying to a queue manager outside the cluster

To form a return path for replies, the gateway (QM1) advertises a queue-manager alias for the queue manager outside the cluster. It advertises this alias to the whole cluster by adding the cluster attribute to its queue-manager alias definition. (Remember that a queue-manager alias definition is like a remote queue definition, but with a blank RNAME.) For example:

```
DEFINE QREMOTE(QM3) RNAME(' ') RQMNAME(QM3) CLUSTER(DEMO)
```

Again, because QM3 is not part of a cluster, it must communicate using distributed queuing techniques. Therefore, QM1 must also have a sender channel to QM3 and a transmission queue to QM3. QM3 needs a corresponding receiver channel. The channels and transmission queues are not shown explicitly in Figure 8 on page 36.

When the application (app2) on QM2 issues an MQPUT call to send a reply to Q3 at QM3, the reply is sent to the gateway, which uses its queue-manager alias to resolve the destination-queue and queue-manager name.

Note: You may define more than one route out of a cluster.

Putting from a queue manager outside the cluster - alternative

There is another technique for putting from a queue manager outside a cluster.

On the gateway queue manager define a queue-manager alias called, for example, ANY.CLUSTER:

```
DEFINE QREMOTE(ANY.CLUSTER) RNAME(' ') RQMNAME(' ')
```

This maps any response to the queue manager ANY.CLUSTER to ‘null’, which means the QREMOTE definition in the queue manager outside the cluster can use the queue manager name ANY.CLUSTER instead of having to use the exact queue manager name. Therefore, on the queue manager outside the cluster, the definition:

```
DEFINE QREMOTE(Q2) RNAME(Q2) RQMNAME(ANY.CLUSTER) XMITQ(QM1)
```

would cause messages to go to QM1 initially, and from there be routed to any queue manager in the cluster that hosts the cluster queue Q2.

Putting to a queue manager outside the cluster

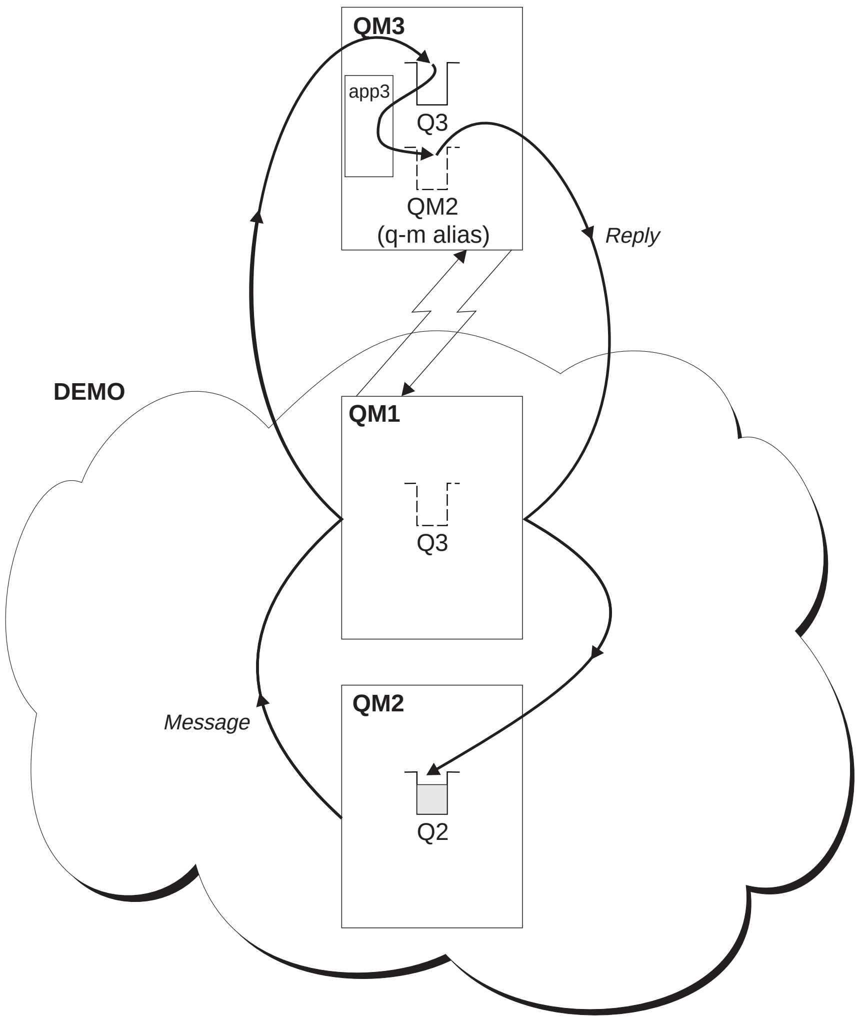

Figure 9. Putting to a queue manager outside the cluster

Now let us consider how to put a message from QM2, which is inside the cluster, to the queue Q3 at QM3, which is outside the cluster.

The gateway, in this example QM1, has a QREMOTE definition that advertises the remote queue (Q3) to the cluster:

```
DEFINE QREMOTE(Q3) RNAME(Q3) RQMNAME(QM3) CLUSTER(DEMO)
```

It also has a sender channel and a transmission queue to the queue manager that is outside the cluster. QM3 has a corresponding receiver channel. These are not shown in Figure 9.

To put a message, an application on QM2 issues an MQPUT call specifying the target queue name (Q3) and specifying the name of the queue to which replies are to be sent (Q2). The message is sent to QM1, which uses its remote-queue definition to resolve the queue name to Q3 at QM3.

Note: You may define more than one route out of a cluster.

Replying from a queue manager outside the cluster

So that QM3 can send replies to the queue managers inside the cluster, it must have a queue-manager alias for each queue manager in the cluster with which it needs to communicate. This queue-manager alias must specify the name of the gateway through which messages are to be routed, that is, the name of the transmission queue to the gateway queue manager. In this example, QM3 needs a queue manager alias definition for QM2:

```
DEFINE QREMOTE(QM2) RNAME(' ') RQMNAME(QM2) XMITQ(QM1)
```

QM3 also needs a sender channel and transmission queue to QM1 and QM1 needs a corresponding receiver channel.

The application (app3) on QM3 can then send replies to QM2, by issuing an MQPUT call and specifying the queue name (Q2) and the queue manager name (QM2).

Putting across clusters

Instead of grouping all your queue managers together in one big cluster, you may have many smaller clusters with one or more queue managers in each acting as a bridge. The advantage of this is that you can restrict the visibility of queue and queue-manager names across the clusters. (See “Overlapping clusters” on page 61.) You can use aliases to change the names of queues and queue managers to avoid name conflicts or to comply with local naming conventions.

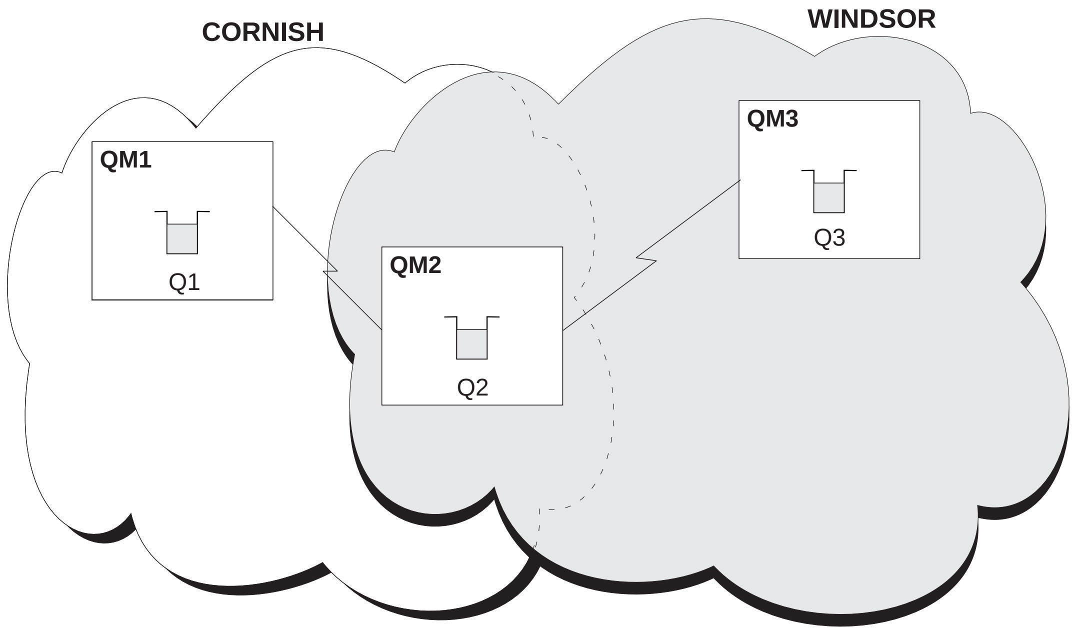

Figure 10. Bridging across clusters

Figure 10 shows two clusters with a bridge between them. (There could be more than one bridge.) QM1 has defined a cluster queue Q1, as follows:

```
DEFINE QLOCAL(Q1) CLUSTER(CORNISH)
```

QM3 has defined a cluster queue Q3, as follows:

```
DEFINE QLOCAL(Q3) CLUSTER(WINDSOR)
```

Aliasing and remote-queue definitions

QM2 has created a namelist called CORNISHWINDSOR, containing the names of both clusters:

```
DEFINE NAMELIST(CORNISHWINDSOR)
 DESCR('CornishWindsor namelist')
 NAMES(CORNISH, WINDSOR)
```

QM2 has also defined a cluster queue Q2, as follows:

```
DEFINE QLOCAL(Q2) CLUSNL(CORNISHWINDSOR)
```

QM2 is a member of both clusters and is the bridge between them. For each queue that you want to make visible across the bridge, you need a QALIAS definition on the bridge. For example in Figure 10 on page 39, on QM2, you would have:

```
DEFINE QALIAS(MYQ3) TARGQ(Q3) CLUSTER(CORNISH)
```

This would mean that an application connected to a queue manager in CORNISH (for example QM1), could put a message to a queue, which it refers to as MYQ3, and this message would be routed to Q3 at QM3.

For each queue manager that you want to make visible, you have a queue-manager alias definition. For example on QM2 you would have:

```
DEFINE QREMOTE(QM1) RNAME(' ') RQMNAME(QM1) CLUSTER(WINDSOR)
```

This would mean that an application connected to any queue manager in WINDSOR (for example QM3), could put a message to any queue on QM1, by naming QM1 explicitly on the MQOPEN call.

Chapter 5. Using clusters for workload management

This chapter describes the advanced method of using MQSeries clusters. It describes how you can set up a cluster that has more than one definition for the same queue, and can therefore benefit from increased availability and workload balancing in your network. This chapter discusses workload management and the implications of using clusters in this way.

More than one instance of a queue

As well as setting up clusters to reduce system administration, (as described in "Chapter 2. Using clusters to ease system administration" on page 11) there is another way of using them. You can create clusters in which more than one queue manager hosts an instance of the same queue.

You may organize your cluster such that the queue managers in it are clones of each other. This means that they are able to run the same applications and have local definitions of the same queues. For example, in a System/390® parallel sysplex the cloned applications might access and update data in a shared DB2® database or a shared Virtual Storage Access Method (VSAM) database. Because you can have more than one instance of an application each receiving messages and running independently of each other, the workload can be spread between your queue managers.

The advantages of using clusters in this way are:
- Increased availability of your queues and applications
- Faster throughput of messages
- More even distribution of workload in your network

Any one of the queue managers that hosts an instance of a particular queue can handle messages destined for that queue. This means that applications need not explicitly name the queue manager when sending messages. A workload management algorithm determines which queue manager should handle the message.

Figure 11 on page 42 shows a cluster in which there is more than one definition for the queue Q3. When an application at QM1 puts a message to Q3, it does not necessarily know which particular instance of Q3 will process its message. (Note, however, that when an application on QM2 or an application on QM4 puts a message to Q3 the local instance of the queue is used.)

Because more than one queue manager is able to handle the same message, the risk of delayed delivery when a queue manager or communications link is unavailable is greatly reduced. The workload management algorithm tries one queue manager after another, if an initial attempt to deliver a message should fail.

© Copyright IBM Corp. 1999, 2000

Multiple queue definitions

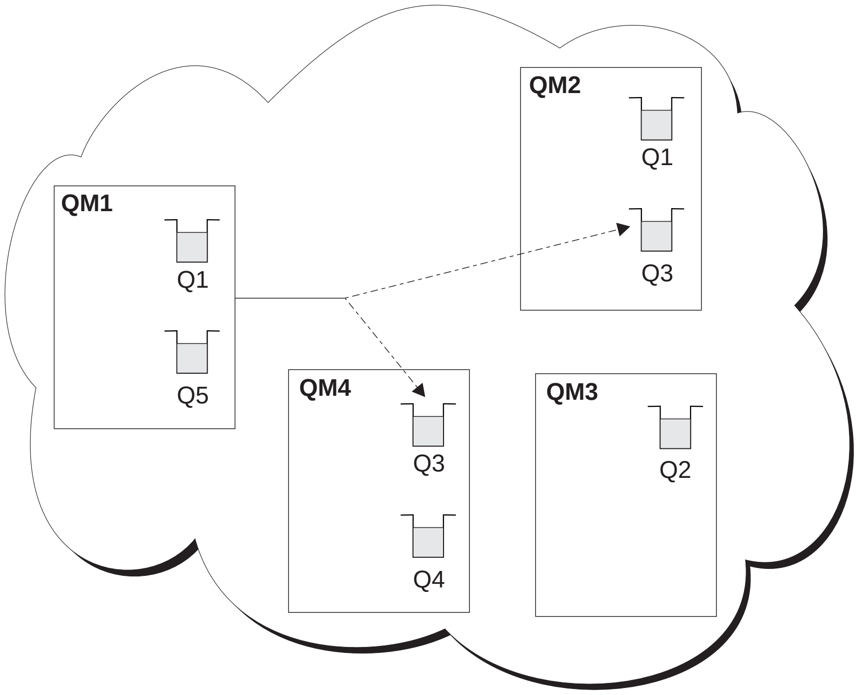

Figure 11. A cluster with multiple instances of the same queue

If the destination queue manager goes out of service while there is still a message on the transmission queue for it, the system attempts to reroute the message. However, in so doing it does not affect the integrity of the message by running the risk of losing it or by creating a duplicate. If a queue manager fails and leaves a message in doubt, that message is not rerouted.

Notes:

1. Before setting up a cluster that has multiple instances of the same queue, it is important to ensure that your messages do not have dependencies on each other, for example needing to be processed in a specific sequence or by the same queue manager. See “Programming considerations” on page 46.
2. It is advisable to make the definitions for different instances of the same queue identical. Otherwise you will get different results from different MQINQ calls.

| “Task 3: Adding a new queue manager that hosts a queue” on page 71 shows how
| to set up a cluster with more than one instance of a queue.

Workload balancing

When you have clusters containing more than one instance of the same queue, MQSeries uses a workload management algorithm to determine the best queue manager to route a message to. The workload management algorithm selects the local queue manager as the destination whenever possible. If there is no instance of the queue on the local queue manager, the algorithm determines which destinations are suitable. Suitability is based on the state of the channel (including any priority you might have assigned to the channel), and also the availability of the queue manager and queue. The algorithm uses a round-robin approach to finalize its choice between the suitable queue managers.

Cluster workload user exit

In most cases the workload management algorithm will be sufficient for your needs. However, so that you can provide your own user-exit program to tailor workload management, MQSeries includes a user exit, the cluster workload exit.

If you have some particular information about your network or messages that you could use to influence workload balancing, you may decide to write a cluster workload exit program or use one supplied by a third party. For example, you may know which are the high-capacity channels or the cheap network routes, or you might decide that you want to route messages depending upon their content.

The cluster workload exit is called when a cluster queue is opened using the MQOPEN or MQPUT1 call, and when a message is put to a queue using the MQPUT call. The target queue manager selected at MQOPEN time is fixed if MQOO_BIND_ON_OPEN is specified (see "MQOPEN" on page 49). In this case, even if the target queue manager fails, the exit is not run again. In cases where the target queue manager is not fixed, if the target queue manager chosen at the time of an MQPUT call is unavailable, or fails while the message is still on the transmission queue, the exit is called again to select a new target queue manager.

You name cluster workload exits in the queue-manager definition. Do this by specifying the CLWLEXIT attribute on the ALTER QMGR command. For example:

```
ALTER QMGR CLWLEXIT(myexit)
```

If the queue-manager definition does not contain a workload-exit program name, the workload exit is not called.

Cluster workload exits are called with an exit parameter structure (MQWXP), a message definition structure (MQMD), a message length parameter, and a copy of the message (or part of the message). See "Chapter 10. Cluster workload exit call and data structures" on page 93 for reference information about the cluster workload exit and the associated data structures.

Writing and compiling cluster workload exit programs

| The following guidance applies specifically to cluster workload exit programs.
| Read it in conjunction with the general application-programming guidance given
| in the *MQSeries Application Programming Guide*.

MQSeries for OS/390

Cluster workload exits are invoked as if by an OS/390 LINK, in:
- Non-authorized problem program state
- Primary address space control mode
- Non-cross-memory mode
- Non-access register mode
- 31-bit addressing mode
- Storage key 8
- Program Key Mask 8
- TCB key 8

The link-edited modules must be placed in the data set that is specified by the CSQXLIB DD statement of the queue manager address space procedure. The names of the load modules are specified as the workload exit names in the queue-manager definition.

Workload balancing

When writing workload exits for MQSeries for OS/390, the following rules apply:

- Exits must be written in assembler or C. If C is used, it must conform to the C systems programming environment for system exits, described in the *OS/390 C/C++ Programming Guide*, SC09-2362.
- Exits are loaded from the non-authorized libraries that are defined by a CSQXLIB DD statement. Providing CSQXLIB has DISP=SHR, exits can be updated while the queue manager is running, with the new version used when the next MQCONN thread is started by the queue manager.
- Exits must be reentrant, and capable of running anywhere in virtual storage.
- Exits must reset the environment, on return, to that at entry.
- Exits must free any storage obtained, or ensure that it will be freed by a subsequent exit invocation.
- No MQI calls are allowed.
- Exits should not use any system services that could cause a wait, because this would severely degrade the performance of the queue manager. In general, therefore, SVCs, PCs, and I/O should be avoided.
- Exits should not issue ESTAEs or SPIEs, apart from within any subtasks they attach.

Note that there are no absolute restrictions on what you can do in an exit. However, most SVCs involve waits, so you should avoid them, except for the following:
- GETMAIN/FREEMAIN
- LOAD/DELETE

You should not use ESTAEs and ESPIEs because their error handling might interfere with the error handling performed by MQSeries. This means that MQSeries might not be able to recover from an error, or that your exit program might not receive all the error information.

The system parameter EXITLIM, which is described in the *MQSeries for OS/390 System Management Guide*, limits the amount of time an exit may run for. The default value for EXITLIM is 30 seconds. If you see the return code MQRC_CLUSTER_EXIT_ERROR (2266 X'8DA') your exit may be looping. If you think the exit does need more than 30 seconds to complete, you should increase the value of EXITLIM.

For information about building your application see the *MQSeries Application Programming Guide* book and the *MQSeries Intercommunication* book.

V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT

Cluster workload exits must not use MQI calls.

In other respects, the rules for writing and compiling cluster workload exit programs on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT, are similar to the rules that apply to channel exit programs. These are described in detail in the *MQSeries Intercommunication* book.

Sample cluster workload exit

MQSeries includes a sample cluster workload exit program. You can copy this sample and use it as a basis for your own programs.

MQSeries for OS/390
The sample cluster workload exit program is supplied in Assembler and in C. The Assembler version is called CSQ4BAF1 and can be found in the library `thlqual`.SCSQASMS. The C version is called CSQ4BCF1 and can be found in the library `thlqual`.SCSQC37S. (`thlqual` is the target library high-level qualifier for MQSeries data sets in your installation.)

V5.1 of MQSeries for AIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT
The sample cluster workload exit program is supplied in C and is called amqswlm0.c. It can be found in:
- /usr/mqm/samp on AIX
- /opt/mqm/samp on Sun Solaris and HP-UX
- C:\mqm\tools\c\samples on OS/2 Warp
- C:\Program Files\MQSeries\Tools\c\Samples on Windows NT

(Where C is the drive on which you have installed the product.)

| **MQSeries for AS/400 V5.1**
| The sample cluster workload exit program is supplied in C and is called
| AMQSWLM4. It can be found in file QCSRC of library QMQMSAMP.

The purpose of this sample exit is to route all messages to a particular queue manager, except if that queue manager becomes unavailable. It reacts to the failure of the queue manager by routing messages to another queue manager instead.

You indicate which queue manager you want messages to be sent to by supplying the name of its cluster-receiver channel in the CLWLDATA attribute on the queue-manager definition. For example:

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

To enable the exit, supply its full path and name in the CLWLEXIT attribute:

On UNIX systems:

```
ALTER QMGR CLWLEXIT('path/amqswlm(clwlFunction)')
```

On OS/2 Warp and Windows NT:

```
ALTER QMGR CLWLEXIT('path\amqswlm(clwlFunction)')
```

On OS/390:

```
ALTER QMGR CLWLEXIT(CSQ4BxF1)
```

where `x` is either ‘A’ or ‘C’, depending on the programming language of the version you are using.

| On OS/400®:

| Either enter the MQSC command:

```
ALTER QMGR CLWLEXIT('AMQSWLM   library    ')
```

| (both the program name and the library name occupy 10 characters and are
| blank-padded to the right if necessary). Alternatively, use the CL command:

```
CHGMQM MQMNAME(qmgrname) CLWLEXIT('library/AMQSWLM')
```

| Now, instead of using the supplied workload management algorithm, MQSeries calls this exit to route all messages to your chosen queue manager.

Programming considerations

As described in the *MQSeries Application Programming Reference* book, applications can open a queue using the MQOPEN call. Applications use the MQPUT call to put messages onto a queue that is open. Applications can put a single message onto a queue that is not already open, using the MQPUT1 call.

If you set up clusters that do not have multiple instances of the same queue, there are no specific application programming considerations.

However, if you set up a network in which there are multiple definitions of the same queue you must review your applications for message affinities as described in “Reviewing applications for message affinities”.

If an application opens a target queue for input or for set, that is, so that it can read messages from it or set its attributes, the MQOPEN call operates only on the local version of the queue.

To benefit from the workload management aspects of clustering, you may need to modify your applications.

If you have an instance of a cluster queue on the local queue manager and also on another queue manager, and you open the queue for output, the MQOPEN call operates only on the local instance of the queue. This may limit the ability of your applications to exploit clustering.

Reviewing applications for message affinities

Before starting to use clusters with multiple definitions for the same queue, you must examine your applications to see whether there are any that have message affinities, that is, a requirement for an exchange of related messages. With clusters, a message may be routed to *any* queue manager that hosts a copy of the correct queue. Therefore, the logic of applications with message affinities may be upset.

For example, you may have two applications that rely on a series of messages flowing between them in the form of questions and answers. It may be important that all the questions are sent to the same queue manager and that all the answers are sent back to the other queue manager. In this situation, it is important that the workload management routine does not send the messages to any queue manager that just happens to host a copy of the correct queue.

Similarly, you may have applications that require messages to be processed in sequence, for example a file transfer application or database replication application that sends batches of messages that must be retrieved in sequence.

You may find other circumstances in which a series of messages must be processed by a particular queue manager or in a particular sequence.

Note: The use of segmented messages may also result in an affinity problem.

Handling message affinities

If you find that you have applications with message affinities, you should attempt, where possible, to remove the affinities before starting to use clusters.

Removing message affinities improves the availability of applications. If an application that has message affinities sends a batch of messages to a queue manager and the queue manager fails after receiving only part of the batch, the sending queue manager must wait for it to recover before it can send any more messages.

Removing messages affinities also improves the scalability of applications. A batch of messages with affinities may cause resources at the destination queue manager to become locked while waiting for subsequent messages. These resources may remain locked for long periods of time, and this might prevent other applications from doing their work.

Furthermore, message affinities prevent the cluster workload management routines from making the best choice of queue manager.

To remove affinities, consider the following possibilities:

- Carrying state information in the messages
- Maintaining state information in nonvolatile storage that is accessible to any queue manager, for example in a DB2 database
- Replicating read-only data so that it is accessible to more than one queue manager

If it is not appropriate to modify your applications to remove message affinities, there are a number of other possible solutions to the problem. For example, you can:

Name a specific destination on the MQOPEN call

One solution is to specify the remote-queue name and the queue manager name on each MQOPEN call. If you do this, all messages put to the queue using that object handle go to the same queue manager. (This may be the local queue manager.)

The disadvantages to this technique are:

- No workload management is carried out. This prevents you from taking advantage of the benefits of cluster workload management.
- If the target queue manager is remote and there is more than one channel to it, the messages may take different routes and so the sequence of messages is still not preserved.
- If your queue manager has a definition for a transmission queue with the same name as the destination queue manager, messages go on that transmission queue rather than on the cluster transmission queue.

Return the queue-manager name in the reply-to queue manager field

A variation on the first solution is to allow the queue manager that receives the first message in a batch to return its name in its response. It does this using the ReplyToQMgr field of the message descriptor. The queue manager at the sending end can then extract this queue manager name and specify it on all subsequent messages.

The advantage of this method over the previous one is that some workload balancing is carried out in order to deliver the first message.

The disadvantage of this method is that the first queue manager must wait for a response to its first message and must be prepared to find and use the ReplyToQMgr information before sending subsequent messages. As with the previous method, if there is more than one route to the queue manager, the sequence of the messages may not be preserved.

Use the MQOO_BIND_ON_OPEN option on the MQOPEN call

Another solution to the message affinities problem is to force all your messages to be put to the same destination. Do this using the MQOO_BIND_ON_OPEN option on the MQOPEN call. By opening a queue and specifying MQOO_BIND_ON_OPEN you force all messages that are sent to this queue to be sent to the same instance of the queue. MQOO_BIND_ON_OPEN binds all messages to the same queue manager and also to the same route. Therefore, if there is, for example, an IP route and a NetBIOS route to the same destination, one of these will be selected when the queue is opened and this selection will be honored for all messages put to the same queue using the object handle obtained.

By specifying MQOO_BIND_ON_OPEN you force all messages to be routed to the same destination. Therefore applications with message affinities are not disrupted. If the destination is not available, the messages remain on the transmission queue until it becomes available again.

MQOO_BIND_ON_OPEN also applies when the queue manager name is specified in the object descriptor when you open a queue. There may be more than one route to the named queue manager (for example, there may be multiple network paths or another queue manager may have defined an alias). If you specify MQOO_BIND_ON_OPEN, a route is selected when the queue is opened.

Note: This is the recommended technique. However, it does not work in a multi-hop configuration in which a queue manager advertises an alias for a cluster queue. Nor does it help in situations in which applications specifically use different queues on the same queue manager for different groups of messages.

As an alternative to specifying MQOO_BIND_ON_OPEN on the MQOPEN call, you can achieve the same effect by modifying your queue definitions. On your queue definitions, specify DEFBIND(OPEN), and allow the MQOO_BIND option on the MQOPEN call to default to MQOO_BIND_AS_Q_DEF. See “Queue definition commands” on page 56 for more information about using the DEFBIND attribute in queue definitions.

Use the cluster workload exit

Instead of modifying your applications you could circumvent the message affinities problem by writing a cluster workload exit program. This would not be easy and is not a recommended solution. This program would have to be designed to recognize the affinity by inspecting the content of messages. Having recognized the affinity, the program would have to force the workload management utility to route all related messages to the same queue manager.

MQI and clusters

Some changes have been made to the MQSeries application programming interface to facilitate the use of clusters. The affected calls are:
- MQOPEN
- MQPUT and MQPUT1
- MQINQ
- MQSET

| The options on these calls that relate specifically to clustering are described here.
| This information should be read in conjunction with the *MQSeries Application*
| *Programming Reference* book, which describes each call in full.

The remainder of this chapter contains general-use programming interface information.

MQOPEN

An option on the MQOPEN call, the MQOO_BIND_ON_OPEN option, allows you to specify that when there are multiple instances of the same queue within a cluster, the target queue manager needs to be fixed. That is, all messages put to the queue specifying the object handle returned from the MQOPEN call must be directed to the **same** queue manager via the same route.

You might use this option when you have messages with affinities. For example, if you need a batch of messages all to be processed by the same queue manager, specify MQOO_BIND_ON_OPEN when you open the queue. This causes MQSeries to fix the queue manager and also the route to be taken by all messages put to that queue.

If you do not want to force all your messages to be written to the same destination, specify MQOO_BIND_NOT_FIXED on the MQOPEN call. This causes a destination to be selected at MQPUT time, that is, on a message-by-message basis.

Note: Do not specify MQOO_BIND_NOT_FIXED and MQMF_SEGMENTATION_ALLOWED at the same time. If you do, the segments of your message may all be delivered to different queue managers, scattered throughout the cluster.

If you do not specify either MQOO_BIND_ON_OPEN or MQOO_BIND_NOT_FIXED, the default option is MQOO_BIND_AS_Q_DEF. Using MQOO_BIND_AS_Q_DEF causes the binding that is used for the queue handle to be taken from the DefBind queue attribute. See “Queue definition commands” on page 56.

You can also cause a destination queue manager to be chosen, by specifying its name in the object descriptor on the MQOPEN call. In this way, you can select any queue manager, including the local one.

If you specify one or more of the options MQOO_BROWSE, MQOO_INPUT_*, or MQOO_SET on the MQOPEN call, there must be a local instance of the cluster queue in order for the open to succeed. If you specify one or more of the options MQOO_OUTPUT, MQOO_BIND_*, or MQOO_INQUIRE on the MQOPEN call, and none of the options MQOO_BROWSE, MQOO_INPUT_*, or MQOO_SET (which always cause the local instance to be selected) the instance opened is either:
- The instance on the local queue manager, if there is one, or
- An instance elsewhere in the cluster, if there is no local queue-manager instance

For full details about the MQOPEN call and how to use it, see the *MQSeries Application Programming Reference* book.

Resolved queue manager name

When a queue manager name is resolved at MQOPEN time, the resolved name is returned to the application. If the application tries to use this name on a subsequent MQOPEN call, it may find that is it not authorized to access the name.

MQPUT and MQPUT1

If MQOO_BIND_NOT_FIXED is specified on an MQOPEN call, each subsequent MQPUT call invokes the workload management routine to determine which queue manager to send the message to. Therefore, the destination and route to be taken are selected on a message-by-message basis. The destination and route may be changed after the message has been put if conditions in the network change. The MQPUT1 call always operates as though MQOO_BIND_NOT_FIXED were in effect, that is, it always invokes the workload management routine.

When the workload management routine has selected a queue manager, the local queue manager completes the put operation. If the target queue manager is a member of the same cluster as the local queue manager, the local queue manager puts the message on the cluster transmission queue, SYSTEM.CLUSTER.TRANSMIT.QUEUE, for transmission to the destination. If the target queue manager is outside the cluster, and the local queue manager has a transmission queue with the same name as the target queue manager, it puts the message on that transmission queue.

If MQOO_BIND_ON_OPEN is specified on the MQOPEN call, MQPUT calls do not need to invoke the workload management routine because the destination and route have already been selected.

MQINQ

Before you can inquire on a queue, you must open it using the MQOPEN call and specifying MQOO_INQUIRE.

If you have clusters in which there are multiple instances of the same queue, you should be aware that the attributes that can be inquired depend on whether there is a local instance of the cluster queue, and on how the queue is opened.

If, on the MQOPEN call, in addition to specifying MQOO_INQUIRE, you also specify one of the options MQOO_BROWSE, MQOO_INPUT_*, or MQOO_SET, then there must be a local instance of the cluster queue for the open to succeed. In this case you can inquire on all the attributes that are valid for local queues.

If, on the MQOPEN call, you specify only MQOO_INQUIRE, or MQOO_INQUIRE and MQOO_OUTPUT (but specify none of the options MQOO_BROWSE, MQOO_INPUT_*, or MQOO_SET, which always cause the local instance of a cluster queue to be selected) the instance opened is either:

- The instance on the local queue manager, if there is one, or
- An instance elsewhere in the cluster, if there is no local queue-manager instance

If the queue that is opened is not a local queue, only the attributes listed below can be inquired. The QType attribute has the value MQQT_CLUSTER in this case.

- DefBind
- DefPersistence
- DefPriority
- InhibitPut
- QDesc
- QName
- QType

To inquire on the DefBind attribute of a cluster queue, use the MQINQ call with the selector MQIA_DEF_BIND. The value returned is either MQBND_BIND_ON_OPEN or MQBND_BIND_NOT_FIXED. To inquire on the CLUSTER and CLUSNL attributes of the local instance of a queue, use the MQINQ call with the selector MQCA_CLUSTER_NAME or the selector MQCA_CLUSTER_NAMELIST respectively.

Note: If you open a cluster queue with no fixed binding (that is, specifying MQOO_BIND_NOT_FIXED on the MQOPEN call, or specifying MQOO_BIND_AS_Q_DEF when the DefBind attribute of the queue has the value MQBND_BIND_NOT_FIXED) successive MQINQ calls may inquire different instances of the cluster queue.

MQSET

If you open a cluster queue to set its attributes (specifying the MQOO_SET option), there must be a local instance of the cluster queue for the open to succeed. Therefore you cannot use the MQSET call to set the attributes of a queue elsewhere in the cluster. However, if you open an alias queue or a remote queue that was defined with the cluster attribute you can use the MQSET call to set attributes of the alias queue or remote queue even if the target queue or remote queue it resolves to is a cluster queue.

Return codes

These are the return codes specific to work with clusters.

MQRC_CLUSTER_EXIT_ERROR (2266 X'8DA')

Occurs when an MQOPEN, MQPUT, or MQPUT1 call was issued to open or put a message on a cluster queue, but the cluster workload exit defined by the queue-manager’s ClusterWorkloadExit attribute failed unexpectedly or did not respond in time.

On MQSeries for OS/390 a message is written to the system log giving more information about this error.

Subsequent MQOPEN, MQPUT, and MQPUT1 calls for this queue handle are processed as though the ClusterWorkloadExit attribute were blank.

Return codes

MQRC_CLUSTER_EXIT_LOAD_ERROR (2267 X'8DB')

| Occurs on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT when an MQCONN or MQCONNX call was issued to connect to a queue manager, but the call failed because the cluster workload exit defined by the queue-manager's ClusterWorkloadExit attribute could not be loaded.

On OS/390, if the cluster workload exit cannot be loaded, a message is written to the system log and processing continues as though the ClusterWorkloadExit attribute had been blank.

MQRC_CLUSTER_PUT_INHIBITED (2268 X'8DC')

Occurs when an MQOPEN call with the MQOO_OUTPUT and MQOO_BIND_ON_OPEN options in effect is issued for a cluster queue, but all of the instances of the queue in the cluster are currently put-inhibited, that is, all of the queue instances have the "InhibitPut" attribute set to MQQA_PUT_INHIBITED. Because there are no queue instances available to receive messages, the MQOPEN call fails.

This reason code occurs only when both of the following are true:

- There is no local instance of the queue. (If there is a local instance, the MQOPEN call succeeds, even if the local instance is put-inhibited.)
- There is no cluster workload exit for the queue, or there is a cluster workload exit but it did not choose a queue instance. (If the cluster workload exit does choose a queue instance, the MQOPEN call succeeds, even if that instance is put-inhibited.)

If the MQOO_BIND_NOT_FIXED option is specified on the MQOPEN call, the call can succeed even if all of the queues in the cluster are put-inhibited. However, a subsequent MQPUT call may fail if all of the queues are still put-inhibited at the time of that call.

MQRC_CLUSTER_RESOLUTION_ERROR (2189 X'88D')

Occurs when an MQOPEN, MQPUT, or MQPUT1 call was issued to open or put a message on a cluster queue, but the queue definition could not be resolved correctly because a response was required from the repository queue manager but none was available.

MQRC_NO_DESTINATIONS_AVAILABLE (2270 X'8DE')

Occurs when an MQPUT or MQPUT1 call was issued to put a message on a cluster queue, but at the time of the call there were no longer any instances of the queue in the cluster. The PUT fails and the message is not sent.

This situation can occur when MQOO_BIND_NOT_FIXED is specified on the MQOPEN call that opens the queue, or MQPUT1 is used to put the message.

MQRC_STOPPED_BY_CLUSTER_EXIT (2188 X'88C')

Occurs when an MQOPEN, MQPUT, or MQPUT1 call was issued to open or put a message on a cluster queue, but the cluster workload exit rejected the call.

Chapter 6. MQSeries commands

This chapter gives an overview of all the MQSeries commands (MQSC), attributes, and parameters that apply specifically to the use of clusters. This information should help you to follow the examples in this book. It should be read in conjunction with the *MQSeries Command Reference* book, which provides details about all the MQSeries commands, their syntax, attributes, and parameters.

Note that the attributes used in commands shown in the *MQSeries Command Reference* book differ from the full-length attribute names that are shown in the *MQSeries Application Programming Reference* book. See Table 3 and Table 4 on page 93 for some examples.

| For each MQSC described here, there is an equivalent Programmable Command
| Format (PCF) command that you can use on V5.1 of MQSeries for AIX, AS/400,
| HP-UX, OS/2 Warp, Sun Solaris, and Windows NT. For details about PCFs, refer to
| the *MQSeries Programmable System Management* book.

Throughout this book, MQSeries commands are shown as they would be entered by the system administrator at the command console. Remember that you do not have to issue the commands in this way. There are a number of other methods, depending on your platform. For example:

- On V5.1 of MQSeries for AIX, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT you can store the commands in a file and use runmqsc, as described in the *MQSeries System Administration* book.
- | On MQSeries for AS/400 you can use CL commands or you can store MQSC
  | commands in a file and use the STRMQMMQSC CL command. See the *MQSeries*
  | *for AS/400 V5.1 System Administration* book for more information.
- On OS/390 you can use the COMMAND function of the CSQUTIL utility, or you can use the operations and control panels. These are described in the *MQSeries for OS/390 System Management Guide*.
- On MQSeries for Windows NT you can use Web Administration, which is a web-based application that allows you to administer your MQSeries network from a Windows NT workstation.

For a complete description of the different methods of issuing MQSC, refer to the *MQSeries Planning Guide* book.

On MQSeries for Windows NT you can also use MQSeries Explorer to work with clusters. For example you can view cluster queues and inquire about the status of cluster-sender and cluster-receiver channels. MQSeries Explorer includes three wizards, which you can use to guide you through the following tasks:
- Creating a new cluster
- Joining an independent queue manager to a cluster
- Joining a cluster queue manager to another cluster

See the *MQSeries System Administration* book for more information about using MQSeries Explorer.

MQSeries command attributes

A number of MQSeries commands (MQSC) have attributes that relate specifically to work with clusters. These attributes are introduced here, under the following headings:

- Queue-manager definition commands
- Channel definition commands
- Queue definition commands

In the commands, a cluster name, specified using the CLUSTER attribute, may be up to 48 characters long. Cluster names must conform to the rules described in the *MQSeries Command Reference* book.

A list of cluster names, specified using the CLUSNL attribute, may contain up to 256 names. To create a cluster namelist, use the command DEFINE NAMELIST described in the *MQSeries Command Reference* book.

Queue-manager definition commands

The ALTER QMGR and DISPLAY QMGR commands have attributes associated with clusters and cluster workload management. To specify that a queue manager holds a repository for a cluster, use the ALTER QMGR command specifying the attribute REPOS(*clustername*). To specify a list of several cluster names, define a cluster namelist and then use the attribute REPOSNL(*namelist*) on the ALTER QMGR command:

```
DEFINE NAMELIST(CLUSTERLIST)
       DESCR('List of clusters whose repositories I host')
       NAMES(CLUS1, CLUS2, CLUS3)

ALTER QMGR REPOSNL(CLUSTERLIST)
```

Use the CLWLEXIT(*name*) attribute to specify the name of a user exit to be called when a message is put to a cluster queue. Use the CLWLDATA(*data*) attribute to specify data to be passed to the cluster workload user exit. Use the CLWLLEN(*length*) attribute to specify the maximum amount of message data to be passed to the cluster workload user exit.

The attributes on the ALTER QMGR command also apply to the DISPLAY QMGR command.

For full details of the attributes and syntax of the ALTER QMGR command and the DISPLAY QMGR command, refer to the *MQSeries Command Reference* book.

The equivalent PCFs are MQCMD_CHANGE_Q_MGR and MQCMD_INQUIRE_Q_MGR. These are described in the *MQSeries Programmable System Management* book.

Channel definition commands

The DEFINE CHANNEL, ALTER CHANNEL, and DISPLAY CHANNEL commands have parameters and attributes specifically applicable to clusters. There are two specific CHLTYPE parameters; CLUSRCVR and CLUSSDR. To define a cluster-receiver channel you use the DEFINE CHANNEL command, specifying CHLTYPE(CLUSRCVR). Many of the other attributes needed on a cluster-receiver channel definition are the same as those that apply to a receiver-channel or a sender-channel definition. To define a cluster-sender channel you use the DEFINE CHANNEL command, specifying CHLTYPE(CLUSSDR), and many of the same attributes as you use to define a sender channel.

The attributes on the DEFINE CHANNEL and ALTER CHANNEL commands that are specific to clusters are:
- CLUSTER
- CLUSNL
- NETPRTY

The CLUSTER and CLUSNL attributes are applicable only if you specify CHLTYPE(CLUSRCVR) or CHLTYPE(CLUSSDR). The NETPRTY attribute is applicable only to cluster-receiver channels.

Use the CLUSTER attribute to specify the name of the cluster with which this channel is associated. Alternatively, you can use the CLUSNL attribute to specify a namelist of cluster names.

Use the NETPRTY attribute to specify a priority for the channel. This is to assist the workload management routines. If there is more than one possible route to a destination, the workload management routine selects the one with the highest priority.

These attributes on the DEFINE CHANNEL command and ALTER CHANNEL command also apply to the DISPLAY CHANNEL command.

Note that the DISPLAY CHANNEL command does not display auto-defined channels. However, you can use the DISPLAY CLUSQMGR command, introduced in “DISPLAY CLUSQMGR” on page 57 to examine the attributes of auto-defined cluster-sender channels.

You can use the DISPLAY CHSTATUS command to display the status of a cluster-sender or cluster-receiver channel. This command gives the status of all channels, not just manually defined channels. Therefore, it will return the status of auto-defined channels.

For full details of the attributes and syntax of the DEFINE CHANNEL, ALTER CHANNEL, DISPLAY CHANNEL, and DISPLAY CHSTATUS commands, refer to the *MQSeries Command Reference* book.

The equivalent PCFs are MQCMD_CHANGE_CHANNEL, MQCMD_COPY_CHANNEL, MQCMD_CREATE_CHANNEL, and MQCMD_INQUIRE_CHANNEL. For information about these PCFs, refer to the *MQSeries Programmable System Management* book.

Queue definition commands

The DEFINE QLOCAL, DEFINE QREMOTE, and DEFINE QALIAS commands, and the three equivalent ALTER commands, have attributes specifically associated with clusters.

- Use the CLUSTER attribute to specify the name of the cluster to which the queue belongs.
- Alternatively, use the CLUSNL attribute to specify a namelist of cluster names.
- Use the DEFBIND attribute to specify the binding to be used when an application specifies MQOO_BIND_AS_Q_DEF on the OPEN call. The default for this attribute is DEFBIND(OPEN), which specifies that the queue handle is to be bound to a specific instance of the cluster queue when the queue is opened. The alternative is to specify DEFBIND(NOTFIXED) so that the queue handle is not bound to any particular instance of the cluster queue. When you specify DEFBIND on a queue definition, the queue is defined with one of the following attributes, MQBND_BIND_ON_OPEN or MQBND_BIND_NOT_FIXED.

  You are recommended to set the DEFBIND attribute to the same value on all instances of the same cluster queue.

These attributes on the DEFINE QLOCAL, DEFINE QREMOTE, and DEFINE QALIAS commands also apply to the DISPLAY QUEUE command. To display information about cluster queues, specify a queue type of QCLUSTER or specify the keyword CLUSINFO on the DISPLAY QUEUE command, or alternatively, use the command DISPLAY QCLUSTER.

The DISPLAY QUEUE or DISPLAY QCLUSTER command returns the name of the queue manager that hosts the queue (or the names of all queue managers if there is more than one instance of the queue). It also returns the system name for each queue manager that hosts the queue, the queue type represented, and the date and time at which the definition became available to the local queue manager. This information is returned using the CLUSQMGR, QMID, CLUSQT, CLUSDATE, and CLUSTIME attributes respectively.

The system name for the queue manager (QMID), is a unique, system-generated name for the queue manager.

For full details of the parameters and syntax of the QUEUE definition commands, refer to the *MQSeries Command Reference* book.

The equivalent PCFs are MQCMD_CHANGE_Q, MQCMD_COPY_Q, MQCMD_CREATE_Q, and MQCMD_INQUIRE_Q. For information about these PCFs, refer to the *MQSeries Programmable System Management* book.

MQSeries commands for work with clusters

This section introduces MQSeries commands (MQSC) that apply specifically to work with MQSeries clusters:
- DISPLAY CLUSQMGR
- SUSPEND QMGR
- RESUME QMGR
- REFRESH CLUSTER
- RESET CLUSTER

The PCF equivalents to these commands are:
- MQCMD_INQUIRE_CLUSTER_Q_MGR
- MQCMD_SUSPEND_Q_MGR_CLUSTER
- MQCMD_RESUME_Q_MGR_CLUSTER
- MQCMD_REFRESH_CLUSTER
- MQCMD_RESET_CLUSTER

These are described in the *MQSeries Programmable System Management* book.

DISPLAY CLUSQMGR

Use the DISPLAY CLUSQMGR command to display cluster information about queue managers in a cluster. If you issue this command from a queue manager with a full repository, the information returned pertains to every queue manager in the cluster. If you issue this command from a queue manager that does not have a full repository, the information returned pertains only to the queue managers in which you have an interest. That is, every queue manager to which you have tried to send a message and every queue manager that holds a full repository.

The information includes:
- How the queue manager was defined (DEFTYPE)
- Whether it holds a repository (QMTYPE)
- The date and time at which the definition became available to the local queue manager (CLUSDATE and CLUSTIME)
- The current status of the cluster-sender channel for this queue manager (STATUS)
- Whether the queue manager is suspended (SUSPEND)
- What clusters the queue manager is in (CLUSTER)
- The cluster-receiver channel name for the queue manager (CHANNEL)

and most other channel attributes that apply to cluster-sender and cluster-receiver channels.

SUSPEND QMGR and RESUME QMGR

The SUSPEND QMGR command and RESUME QMGR command are used to remove a queue manager from a cluster temporarily, for example for maintenance, and then to reinstate it. Use of these commands is discussed in “Maintaining a queue manager” on page 63.

REFRESH CLUSTER

You can issue the REFRESH CLUSTER command from a queue manager to discard all locally held information about a cluster. It is unlikely that you will need to use this command during normal circumstances. Use it only if you want your queue manager to make a fresh start in a cluster. For example you might use it if you think your repository may not be up-to-date, perhaps because you have accidentally restored an out-of-date backup. The format of the command is:

```
REFRESH CLUSTER(clustername)
```

| The queue manager from which you issue this command loses all the information in its repository concerning the named cluster. **It also loses any auto-defined channels that are in doubt and which are not attached to a repository queue manager**. The queue manager has to make a cold-start in that cluster. It must reissue all information about itself and must renew its requests for updates to other information that it is interested in. (It does this automatically.)

Since you are unlikely to need to use this command, except in exceptional circumstances, you may choose to avoid the danger of issuing it accidentally. On MQSeries for OS/390 you can use a security profile to protect the command and prevent it from being issued.

RESET CLUSTER

You may, under certain circumstances, need to forcibly remove a queue manager from a cluster. You can do this from a repository queue manager by issuing the command:

```
RESET CLUSTER(clustername) QMNAME(qmname) ACTION(FORCEREMOVE)
```

You might do this if, for example, a queue manager has been deleted but still has cluster-receiver channels defined to the cluster. Instead of waiting for MQSeries to remove these definitions (which it does automatically) you could issue the RESET CLUSTER command to tidy up sooner. All other queue managers in the cluster are then informed that the queue manager is no longer available.

| Using the RESET CLUSTER command is the only way to delete auto-defined cluster-sender channels. You are unlikely to need this command in normal circumstances but may be advised by your IBM® Support Center to issue the command to tidy up the cluster information held by cluster queue managers. Do not use this command as a short cut to removing a queue manager from a cluster. The correct way to do this is described in “Task 5: Removing a queue manager from a cluster” on page 77.

You can issue the RESET CLUSTER command only from repository queue managers.

The queue manager that was forcibly removed can reconnect to the cluster later. It does this automatically (unless it has been deleted). If you wish to prevent a queue manager from rejoining a cluster it is your responsibility to take appropriate security measures. See “Preventing queue managers joining a cluster” on page 68.

Chapter 7. Managing MQSeries clusters

This chapter provides information for the system administrator, including:
- "Cluster-design considerations"
- "Cluster-administration considerations" on page 63

Cluster-design considerations

Let us now look at how you might design a new cluster.

Selecting queue managers to hold repositories

In each cluster you must select at least one, preferably two, or possibly more of the queue managers to hold repositories. A cluster could work quite adequately with only one repository but using two improves availability.

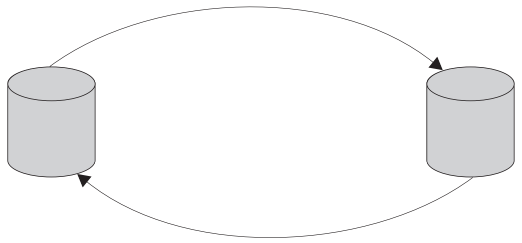

Figure 12. A typical 2-repository topology

Figure 12 shows a typical 2-repository topology. This is the topology used in the cluster shown in Figure 2 on page 5.

The repository queue managers must be fully interconnected with each other and positioned in the network so as to give a high level of availability.
- The most important consideration is that the queue managers chosen need to be reliable and well managed. For example, it would be far better to choose queue managers on a stable OS/390 system than queue managers on a portable personal computer that is frequently disconnected from the network.
- You might also consider the location of the queue managers and choose ones that are in a central position geographically or perhaps ones that are located on the same system as a number of other queue managers in the cluster.
- Another consideration might be whether a queue manager already holds the repositories for other clusters. Having made the decision once, and made the necessary definitions to set up a queue manager as a repository for one cluster, you may well choose to rely on the same queue manager to hold the repositories for other clusters of which it is a member.

When a queue manager sends out some information about itself or requests some information about another queue manager, the information or request is sent to two repositories. A repository named on a CLUSSDR definition handles the request whenever possible but if the chosen repository is not available another repository is used. When the first repository becomes available again it collects the latest new and changed information from the others so that they keep in step.

Design considerations

In very large clusters, containing thousands of queue managers, you may want to have more than two repositories. Then you might, for example, have one of the following topologies.

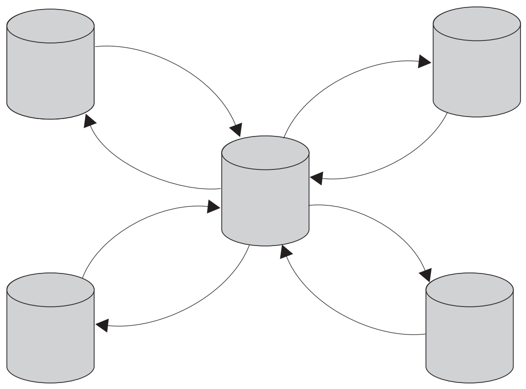

Figure 13. A hub and spoke arrangement of repositories

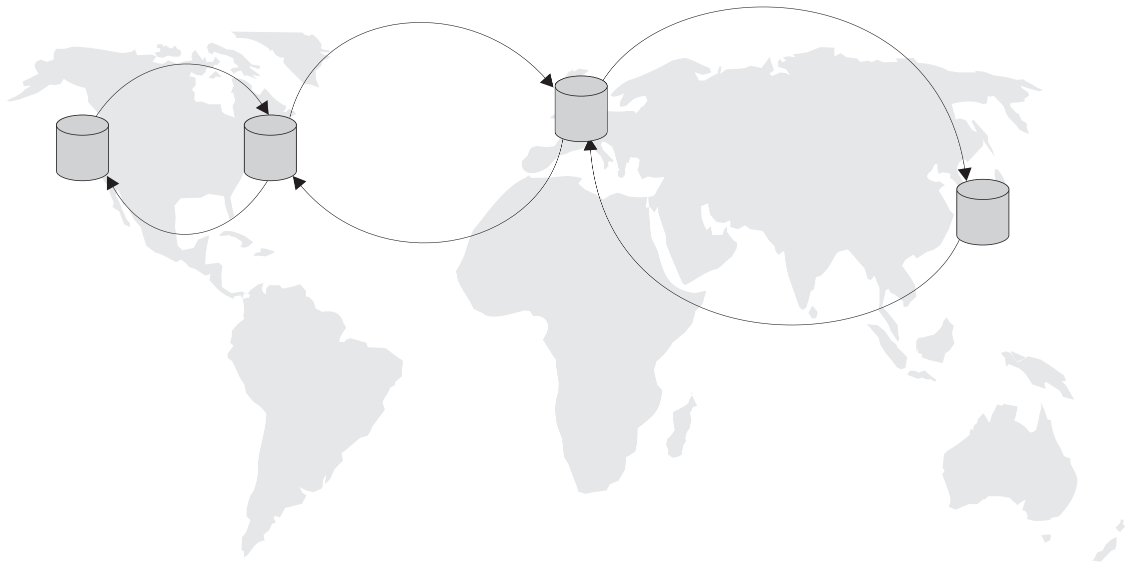

Figure 14. A complex repository topology

If all the repository queue managers go out of service at the same time, queue managers continue to work using the information they have in their partial repositories. Clearly they are limited to using the information that they have. New information and requests for updates cannot be processed. When the repository queue managers reconnect to the network, messages are exchanged to bring all repositories (both full and partial) back up to date.

Organizing a cluster

Having selected some queue managers to hold repositories, you need to decide which queue managers should link to which repository. The CLUSSDR channel definition links a queue manager to a repository from which it finds out about the other repositories in the cluster. From then on, the queue manager sends messages to any two repositories, but it always tries to use the one to which it has a CLUSSDR channel definition first. It is not significant which repository you choose. However, you should consider the topology of your configuration, and perhaps the physical or geographical location of the queue managers as shown in Figure 12 through Figure 14.

It is not advisable to use a repository queue manager on an OS/390 system as the repository queue manager through which the MQSeries Explorer connects to a cluster. This is because there is no command server on MQSeries for OS/390. To ensure that a particular repository queue manager is not used by the MQSeries Explorer, include the string '%NOREPOS%' in the description field of its cluster-receiver channel definition. When the explorer is choosing which repository to link to, it ignores those whose channel description contains '%NOREPOS%', and treats them as though they did not hold a repository for the cluster.

Because all cluster information is sent to two repositories, there may be situations in which you want to make a second CLUSSDR channel definition. You might do this in a cluster that has a large number of repositories, spread over a wide area, to control which repositories your information is sent to.

Choosing names

When setting up a new cluster, you may want to consider a naming convention for the queue managers. Every queue manager must have a different name, but it may help you to remember which queue managers are grouped where if you give them a set of similar names.

Also, every cluster-receiver channel must have a unique name. One possibility is to use the queue-manager name preceded by the preposition 'to'. For example `TO.QM1`, `TO.QM2`, and so on. If you have more than one channel to the same queue manager, each with different priorities or using different protocols, for example, you might extend this convention to use names such as `TO.QM1.A1`, `TO.QM1.N3`, and `TO.QM1.T4`. `A1` might be the first APPC channel, `N3` might be the NetBIOS channel with a network priority of 3, and so on.

Remember that all cluster-sender channels have the same name as their corresponding cluster-receiver channel.

Overlapping clusters

You may create clusters that overlap, as illustrated in Figure 15 on page 62.

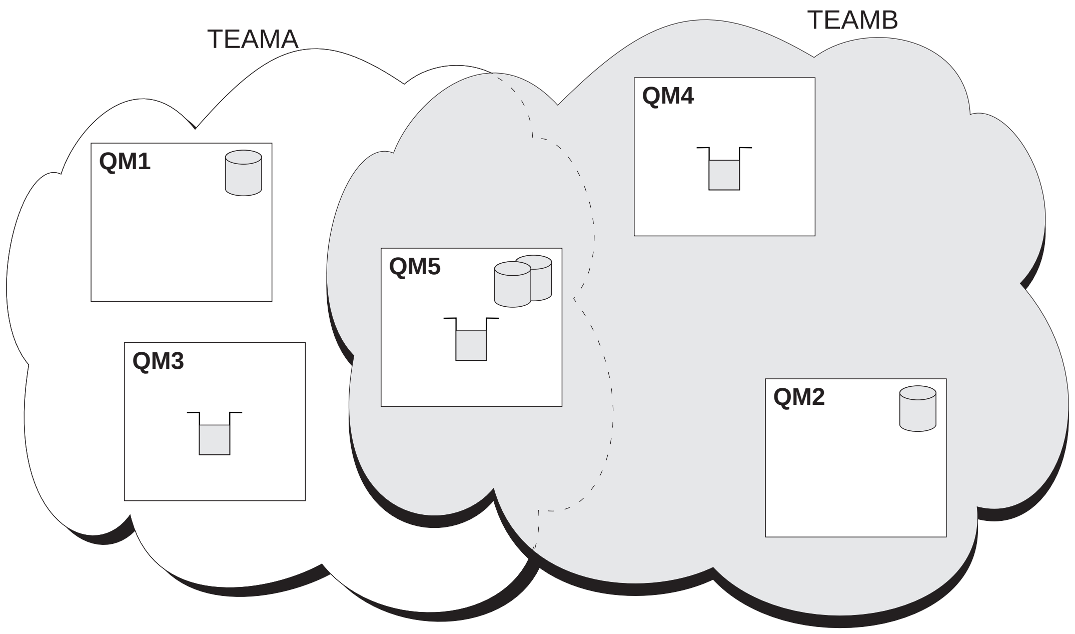

Figure 15. Overlapping clusters

There are a number of reasons you might do this, for example:

- To allow different organizations to have their own administration.
- To allow independent applications to be administered separately.
- To create a class of service. For example you could have a cluster called STAFF that is a subset of the cluster called STUDENTS. When you put a message to a queue advertised in the STAFF cluster, a restricted channel is used. When you put a message to a queue advertised in the STUDENTS cluster, either a general channel or a restricted channel may be used.
- To create test and production environments.

If you do have more than one cluster in your network it is essential to give them different names. **If two clusters with the same name are ever merged, it will not be possible to separate them again**. It is also a good idea to give the clusters, queue managers, and channels different names so that they are more easy to distinguish when you look at the output from DISPLAY commands.

In Figure 15 the queue manager QM5 is a member of both the clusters illustrated. When a queue manager is a member of more than one cluster, you can take advantage of *namelists* to reduce the number of definitions you need. A namelist can contain a list of names, for example, cluster names. Therefore, you could create a namelist naming the clusters TEAMA and TEAMB, and then specify this namelist on the ALTER QMGR command for QM5 to make QM5 a repository queue manager for both of the clusters. See “Task 8: Adding a new, interconnected cluster” on page 85 for some examples of how to use namelists.

Objects

| The following objects are needed when using MQSeries clusters. They are included
| in the set of default objects defined when you create a queue manager on V5.1 of
| MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT,
| and in the customization samples for MQSeries for OS/390.

Do **not** alter the default queue definitions. You could alter the default channel definitions in the same way as any other channel definition, using MQSC or PCF commands.

SYSTEM.CLUSTER.REPOSITORY.QUEUE
Each queue manager in a cluster has a local queue called SYSTEM.CLUSTER.REPOSITORY.QUEUE. This queue is used to store all the repository information.

SYSTEM.CLUSTER.COMMAND.QUEUE
Each queue manager in a cluster has a local queue called SYSTEM.CLUSTER.COMMAND.QUEUE. This queue is used to carry messages to the repository. The queue manager uses this queue to send any new or changed information about itself to the repository queue manager and to send any requests for information about other queue managers.

SYSTEM.CLUSTER.TRANSMIT.QUEUE
Each queue manager has a definition for a local queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE. This is the transmission queue for all messages to all queues and queue managers that are within clusters.

SYSTEM.DEF.CLUSSDR
Each cluster has a default CLUSSDR channel definition called SYSTEM.DEF.CLUSSDR. This is used to supply default values for any attributes that you do not specify when you create a cluster-sender channel on a queue manager in the cluster.

SYSTEM.DEF.CLUSRCVR
Each cluster has a default CLUSRCVR channel definition called SYSTEM.DEF.CLUSRCVR. This is used to supply default values for any attributes that you do not specify when you create a cluster-receiver channel on a queue manager in the cluster.

Cluster-administration considerations

Let us now look at some considerations affecting the system administrator.

Maintaining a queue manager

From time to time, you may need to perform maintenance on a queue manager that is part of a cluster. For example, you may need to take backups of the data in its queues, or apply fixes to the software. If the queue manager hosts any queues, its activities must be suspended. When the maintenance is complete, its activities can be resumed.

To suspend a queue manager, issue the SUSPEND QMGR command, for example:

```
SUSPEND QMGR CLUSTER(SALES)
```

This sends a notification to the queue managers in the cluster SALES advising them that this queue manager has been suspended. The purpose of the SUSPEND QMGR command is only to **advise** other queue managers that they should avoid sending messages to this queue manager if possible. It does not mean that the queue manager is disabled. While the queue manager is suspended the workload management routines avoid sending messages to it, other than messages that **have** to be handled by that queue manager. Messages that **have** to be handled by that queue manager include messages sent by the local queue manager. The workload management routines choose the local queue manager whenever possible - even if it is suspended.

When the maintenance is complete the queue manager can resume its position in the cluster. It should issue the command RESUME QMGR, for example:

```
RESUME QMGR CLUSTER(SALES)
```

This sends a notification to the repositories advising them that the queue manager is available again. The repository queue managers disseminate this information to other queue managers that have requested updates to information concerning this queue manager.

It is possible to enforce the suspension of a queue manager by using the FORCE option on the SUSPEND QMGR command, for example:

```
SUSPEND QMGR CLUSTER(SALES) MODE(FORCE)
```

This forcibly stops all inbound channels to other queue managers in the cluster. If you do not specify MODE(FORCE), the default MODE(QUIESCE) applies.

Refreshing a queue manager

| A queue manager can make a fresh start in a cluster. This is unlikely to be
| necessary in normal circumstances but you may be asked to do this by your IBM
| Support Center. You can issue the REFRESH CLUSTER command from a queue
| manager to remove all cluster queue-manager objects and all cluster queue objects
| relating to queue managers other than the local one, from the local repository. The
| command also removes any auto-defined channels that do not have messages on
| the cluster transmission queue and which are not attached to a repository queue
| manager. Effectively, the REFRESH CLUSTER command allows a queue manager
| to be “cold started” with respect to its repository content. (MQSeries ensures that
| no data is lost from your queues.)

Maintaining the cluster transmission queue

The availability and performance of the cluster transmission queue are essential to the performance of clusters. Make sure that it does not become full, and take care not to accidentally issue an ALTER command to set it either get-disabled or put-disabled. Also make sure that the medium the cluster transmission queue is stored on (for example OS/390 page sets) does not become full. For performance reasons, on OS/390 the INDXTYPE of the cluster transmission queue should be set to CORRELID.

What happens when a queue manager fails?

If a message-batch is sent to a particular queue manager and that queue manager becomes unavailable there are several courses of action:

- With the exception of non-persistent messages on a fast channel (which might be lost) the undelivered batch of messages is backed out to the cluster transmission queue on the sending queue manager.
  - If the backed-out batch of messages is not in doubt and the messages are not bound to the particular queue manager, the workload management routine is called. The workload management routine selects a suitable alternative queue manager and the messages are sent there.
  - Messages that have already been delivered to the queue manager, or are in doubt, or have no suitable alternative, must wait until the original queue manager becomes available again.

The restart can be automated using Automatic Restart Management (ARM) on OS/390, HACMP on AIX, or any other restart mechanism available on the platform.

What happens when a repository fails?

Cluster information is carried to repositories (whether full or partial) on a local queue called SYSTEM.CLUSTER.COMMAND.QUEUE. If this queue should fill up, perhaps because the queue manager has stopped working, the cluster-information messages are routed to the dead-letter queue. If you observe that this is happening, from the messages on your queue-manager log or OS/390 system console, you will need to run an application to retrieve the messages from the dead-letter queue and reroute them to the correct destination.

If errors occur on a repository queue manager you will see messages telling you what error has occurred and how long the queue manager will wait before trying to restart. On MQSeries for OS/390 the SYSTEM.CLUSTER.COMMAND.QUEUE is get-disabled. When you have identified and resolved the error, you must get-enable the SYSTEM.CLUSTER.COMMAND.QUEUE so that the queue manager will be able to restart successfully.

In the unlikely event of a queue manager’s repository running out of storage, you will see storage allocation errors appearing on your queue-manager log or OS/390 system console. If this happens, stop and then restart the queue manager. When the queue manager is restarted, more storage is automatically allocated to hold all the repository information.

What happens if I put-disable a cluster queue?

When a cluster queue is put-disabled, this situation is reflected in the repository of each queue manager that is interested in that queue. The workload management algorithm attempts when possible to send messages to destinations that are put-enabled. If there are no put-enabled destinations and no local instance of a queue, an MQOPEN call that specified MQOO_BIND_ON_OPEN returns a return code of MQRC_CLUSTER_PUT_INHIBITED to the application. If MQOO_BIND_NOT_FIXED is specified, or there is a local instance of the queue, an MQOPEN call succeeds but subsequent MQPUT calls fail with return code MQRC_PUT_INHIBITED.

You may write a user exit program to modify the workload management routines so that messages can be routed to a destination that is put-disabled. If a message arrives at a destination that is put-disabled (because it was in flight at the time the queue became disabled or because a workload exit chose the destination explicitly), then the workload management routine at the queue manager may choose another appropriate destination if there is one, or may place the message on the dead-letter queue, or if there is no dead-letter queue, return the message to the originator.

How long do the repositories retain information?

When a queue manager sends out some information about itself, for example to advertise the creation of a new queue, the repository queue managers store the information for 30 days. To prevent information in the repositories from expiring, queue managers automatically resend all information about themselves after 27 days. If no update is received within 90 days of the expiry date, the information is removed from the repositories. The period of 90 days is to allow for the fact that a queue manager may have been temporarily out of service. If a queue manager becomes disconnected from a cluster for more than 90 days it will cease to be part of the cluster at all. However, if it reconnects to the network it will become part of the cluster again. Note that repositories do not use information that has expired to satisfy new requests from other queue managers.

Similarly, when a queue manager sends a request for up-to-date information from a repository, the request lasts for 30 days. After 27 days MQSeries checks the request. If it has been referenced during the 27 days, it is remade automatically. If not, it is left to expire and is remade by the queue manager if it is needed again. This is to prevent a build up of requests for information about dormant queue managers.

Cluster channels

Although using clusters relieves you of the need to define channels (because MQSeries defines them for you) the same channel technology used in distributed queuing is used for communication between queue managers in a cluster. You need to be familiar with matters such as:
- How channels operate
- How to find their status
- How to use channel exits

These topics are all discussed in the *MQSeries Intercommunication* book.

| Do not set the disconnect interval on your cluster-sender channels and
| cluster-receiver channels too low (less than about 10 seconds). If you do, the
| channel may close down between sending a request to a repository queue manager
| and receiving the response.

| If the cluster-sender end of a channel fails and subsequently tries to restart, the
| restart is rejected if the cluster-receiver end of the channel has remained active. To
| avoid this problem on V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun
| Solaris, and Windows NT, you can arrange for the cluster-receiver channel to be
| terminated and restarted, when a cluster-sender channel attempts to restart. To
| control this, use the AdoptNewMCA, AdoptNewMCATimeout, and
| AdoptNewMCACheck attributes in the qm.ini file or the Windows NT Registry.
| See the *MQSeries System Administration* book for more information.

Chapter 8. Keeping clusters secure

This chapter discusses the following security considerations:

- "Stopping unauthorized queue managers sending messages to your queue manager"
- "Stopping unauthorized queue managers putting messages to your queues"
- "Stopping your queue manager putting messages to remote queues" on page 68
- "Preventing queue managers joining a cluster" on page 68
- "Forcing unwanted queue managers to leave a cluster" on page 69

Stopping unauthorized queue managers sending messages to your queue manager

You may want to prevent certain queue managers from sending messages to your queue manager. You can do this by defining a channel security exit program on the CLUSRCVR channel definition. Write a program that authenticates queue managers trying to send messages on your cluster-receiver channel and denies them access if they are not authorized. Channel security exit programs are called at MCA initiation and termination. See the *MQSeries Intercommunication* book for more information.

Clustering has no effect on the way security exits work. You can restrict access to a queue manager in the same way as you would in a distributed queuing environment.

Stopping unauthorized queue managers putting messages to your queues

You may want to prevent certain queue managers from putting messages to a queue. All MQSeries resources are protected by security facilities available on the platform. For example:

- RACF® or other external security managers on MQSeries for OS/390
- | The Object Authority Manager (OAM) on MQSeries for AS/400, MQSeries on UNIX systems, and MQSeries for Windows NT
- User-written procedures on MQSeries for OS/2 Warp

You can take advantage of these facilities to protect your queues.

In addition, you can use the PUT authority (PUTAUT) attribute on the CLUSRCVR channel definition. The PUTAUT attribute allows you to specify what user IDs are to be used to establish authority to put a message to a queue. The options on the PUTAUT attribute are:

DEF The default user ID is used. On OS/390 this may involve using both the user ID received from the network and that derived from MCAUSER.

CTX The user ID in the context information associated with the message is used. On OS/390 this may involve using either the user ID received from the network, or that derived from MCAUSER, or both. Use this option if the link is trusted and authenticated.

ONLYMCA
As DEF, but any user ID received from the network will not be used. This option is supported on MQSeries for OS/390 only. Use this option if the link is not trusted and you want to allow only a specific set of actions on it, which are defined for the MCAUSER.

ALTMCA
As CTX, but any user ID received from the network will not be used.

For more information about using the PUTAUT attribute on a channel definition, see the *MQSeries Intercommunication* book or see the *MQSeries Command Reference* book.

Note: As with any other transmission queue, it is not possible for applications to put messages directly to SYSTEM.CLUSTER.TRANSMIT.QUEUE without special authorization.

Stopping your queue manager putting messages to remote queues

MQSeries for OS/390
You can use RACF to prevent your queue manager putting messages to a remote queue. With RACF you can set up permissions for a named queue regardless of whether that queue exists on your system. The authorization required is MQOO_OUTPUT.

V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT
On these platforms you cannot restrict access to individual queues that do not exist on your queue manager. However, you can restrict access to **all** the queues in a cluster. On queue manager CORK, to grant the user MYUSER access to the queues in a cluster, issue the following setmqaut commands:

```
setmqaut -m CORK -t qmgr -p MYUSER +connect
setmqaut -m CORK -t qmgr -p MYUSER +setall
setmqaut -m CORK -n SYSTEM.CLUSTER.TRANSMIT.QUEUE
        -t queue -p MYUSER +setall
```

On OS/400, the equivalent CL commands are:

```
GRTMQAUT OBJ(CORK) OBJYTPE(*MQM) USER(MYUSER) AUT(*CONNECT)
GRTMQAUT OBJ(CORK) OBJYTPE(*MQM) USER(MYUSER) AUT(*SETALL)
GRTMQAUT OBJ(SYSTEM.CLUSTER.TRANSMIT.QUEUE) OBJYTPE(*Q) +
        USER(MYUSER) AUT(*SETALL) MQMNAME(CORK)
```

Setting access in this way allows the user MYUSER to put messages to any queue in the cluster.

Preventing queue managers joining a cluster

If you want to ensure that only certain authorized queue managers attempt to join a cluster you must either use a security exit program on the cluster-receiver channel, or you must write an exit program to prevent unauthorized queue managers from writing to SYSTEM.CLUSTER.COMMAND.QUEUE. You must not restrict access to SYSTEM.CLUSTER.COMMAND.QUEUE such that no queue manager can write to it. If you did, you would prevent any queue manager from being able to join the cluster.

Preventing queue managers joining

It is difficult to stop a queue manager that is a member of a cluster from defining a queue. Therefore, there is a danger that a rogue queue manager could join a cluster, learn what queues are in it, define its own instance of one of those queues, and so receive messages that it should not be authorized to receive.

To prevent a queue manager receiving messages that it should not, you could write:

- A channel exit program on each cluster-sender channel, which uses the connection name to determine the suitability of the destination queue manager to be sent the messages
- A cluster workload exit program, which uses the destination records to determine the suitability of the destination queue and queue manager to be sent the messages
- A channel auto-definition exit program, which uses the connection name to determine the suitability of defining channels to the destination queue manager

Forcing unwanted queue managers to leave a cluster

You can force an unwanted queue manager to leave a cluster. You may need to do this to tidy up, if for example, a queue manager was deleted but its cluster-receiver channels are still defined to the cluster. You may have other reasons for wanting to eject a queue manager.

Only repository queue managers are authorized to eject a queue manager from a cluster. For example, to eject the queue manager OSLO from the cluster NORWAY, the repository queue manager would issue the command:

```
RESET CLUSTER(NORWAY) QMNAME(OSLO) ACTION(FORCEREMOVE)
```

Forcing queue managers to leave

Chapter 9. Advanced tasks

This chapter shows some advanced tasks that you can perform to extend the cluster created in "Task 1: Setting up a new cluster" on page 19 and "Task 2: Adding a new queue manager to a cluster" on page 25. These tasks are:
- "Task 3: Adding a new queue manager that hosts a queue"
- "Task 4: Removing a cluster queue from a queue manager" on page 75
- "Task 5: Removing a queue manager from a cluster" on page 77
- "Task 6: Moving a repository to another queue manager" on page 79

The chapter then goes on to demonstrate two further tasks:
- "Task 7: Converting an existing network into a cluster" on page 81
- "Task 8: Adding a new, interconnected cluster" on page 85

You can perform these tasks, and the two described in "Chapter 3. First tasks" on page 19 without stopping your existing cluster queue managers or disrupting your existing network in any way.

Much of the information you need to achieve these tasks is documented elsewhere in the MQSeries library. This chapter gives pointers to that information and fills in details relating specifically to work with clusters.

Notes:
1. Throughout the examples in this chapter and Chapter 3. First tasks the queue managers have illustrative names such as LONDON and NEWYORK. Don't forget that on MQSeries for OS/390, queue-manager names are limited to 4 characters.
2. | The names of the queue managers imply that each queue manager is on a separate machine. You could just as easily set up these examples with all the queue managers on the same machine.
3. | The examples in these chapters show MQSeries Commands (MQSC) as they would be entered by the system administrator at the command console. For information about other ways of entering commands, refer to "Chapter 6. MQSeries commands" on page 53.

Task 3: Adding a new queue manager that hosts a queue

Scenario:
- The INVENTORY cluster has been set up as described in "Task 2: Adding a new queue manager to a cluster" on page 25. It contains three queue managers; LONDON and NEWYORK both hold full repositories, PARIS holds a partial repository. The inventory application runs on the system in New York, connected to the NEWYORK queue manager. The application is driven by the arrival of messages on the INVENTQ queue.
- A new store is being set up in Toronto. To provide additional capacity you want to run the inventory application on the system in Toronto as well as New York.
- Network connectivity exists between all four systems.
- The network protocol is TCP.

The steps required to complete task 3

To achieve this, follow these steps:

1. Prepare the TORONTO queue manager

Preparation of queue managers is described in "How should I prepare for use of clustering?" on page 14.

2. Determine which full repository TORONTO should refer to first

Every queue manager in a cluster must refer to one or other of the repositories in order to gather information about the cluster and so build up its own partial repository. It is of no particular significance which repository you choose. In this example we choose NEWYORK. Once the new queue manager has joined the cluster it will communicate with both of the repositories.

3. Define the CLUSRCVR channel

Every queue manager in a cluster needs to define a cluster-receiver channel on which it can receive messages. On TORONTO, define:

```
DEFINE CHANNEL(TO.TORONTO) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(TORONTO.CHSTORE.COM) CLUSTER(INVENTORY)
```

This advertises the queue manager's availability to receive messages from other queue managers in the cluster, INVENTORY.

4. Define a CLUSSDR channel on queue manager TORONTO

Every queue manager in a cluster needs to define one cluster-sender channel on which it can send messages to its first repository. In this case we have chosen NEWYORK, so TORONTO needs the following definition:

```
DEFINE CHANNEL(TO.NEWYORK) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(NEWYORK.CHSTORE.COM) CLUSTER(INVENTORY)
```

5. Review the inventory application for message affinities

Before proceeding, you must ensure that the inventory application does not have any dependencies on the sequence of processing of messages. See "Reviewing applications for message affinities" on page 46 for more information.

6. Install the inventory application on the system in Toronto

See the *MQSeries Application Programming Guide* for information about how to do this.

7. Define the cluster queue INVENTQ

The INVENTQ queue, which is already hosted by the NEWYORK queue manager, is also to be hosted by TORONTO. Define it on the TORONTO queue manager as follows:

```
DEFINE QLOCAL(INVENTQ) CLUSTER(INVENTORY)
```

Now that you have completed all the definitions, if you have not already done so you should start the channel initiator on MQSeries for OS/390 and, on all platforms, start a listener program on queue manager TORONTO. The listener program listens for incoming network requests and starts the cluster-receiver channel when it is needed. See "Establishing communication in a cluster" on page 16 for more information.

The cluster achieved by task 3

The cluster set up by this task looks like this:

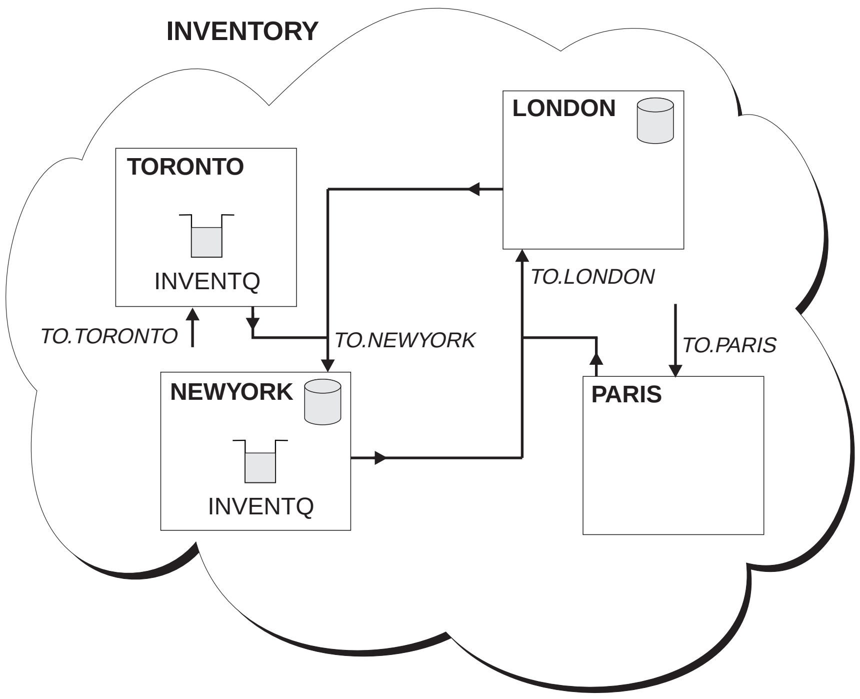

Figure 16. The INVENTORY cluster with four queue managers

The INVENTQ queue and the inventory application are now hosted on two queue managers in the cluster. This increases their availability, speeds up throughput of messages, and allows the workload to be distributed between the two queue managers. Messages put to INVENTQ by either TORONTO or NEWYORK are handled by the instance on the local queue manager whenever possible. Messages put by LONDON or PARIS are routed alternately to TORONTO or NEWYORK, so that the workload is balanced.

This modification to the cluster was accomplished without you having to make any alterations to the queue managers NEWYORK, LONDON, and PARIS. The repositories in these queue managers are updated automatically with the information they need to be able to send messages to INVENTQ at TORONTO.

Assuming that the inventory application is designed appropriately and that there is sufficient processing capacity on the systems in New York and Toronto, the inventory application will continue to function if either the NEWYORK or the TORONTO queue manager becomes unavailable.

Extensions to this task

As you can see from the result of this task, it is possible to have the same application running on more than one queue manager. You can use the facility to allow even distribution of your workload, or you may decide to control the distribution yourself by using a *data partitioning* technique.

For example, suppose that you decide to add a customer-account query and update application running in LONDON and NEWYORK. Account information can only be held in one place, but you could arrange for half the records, for example for account numbers 00000 to 49999, to be held in LONDON, and the other half, in the range 50000 to 99999, to be held in NEWYORK. You would write a cluster workload exit program to examine the account field in all messages, and route the messages to the appropriate queue manager.

This is just one example of data partitioning. There are, of course, other ways you could do this.

Task 4: Removing a cluster queue from a queue manager

Scenario:

- The INVENTORY cluster has been set up as described in "Task 3: Adding a new queue manager that hosts a queue" on page 71. It contains four queue managers. LONDON and NEWYORK hold full repositories. PARIS and TORONTO hold partial repositories. The inventory application runs on the systems in New York and Toronto and is driven by the arrival of messages on the INVENTQ queue.
- Because of reduced workload, you no longer want to run the inventory application in Toronto. You want to disable the INVENTQ queue hosted by the queue manager TORONTO, and have TORONTO feed messages to the INVENTQ queue in NEWYORK.
- Network connectivity exists between all four systems.
- The network protocol is TCP.

The steps required to complete task 4

To achieve this, you need to perform the following tasks:

1. Indicate that the queue is no longer available

To remove a queue from a cluster, you need to remove the cluster name from the local queue definition. Do this from queue manager TORONTO, using the ALTER QLOCAL command and specifying a blank cluster name, like this:

```
ALTER QLOCAL(INVENTQ) CLUSTER(' ')
```

2. Disable the queue

Disable the INVENTQ queue at TORONTO so that no further messages can be written to it:

```
ALTER QLOCAL(INVENTQ) PUT(DISABLED)
```

Now messages in transit to this queue using MQOO_BIND_ON_OPEN will go to the dead-letter queue. You need to stop all applications from putting messages explicitly to the queue on this queue manager.

3. Monitor the queue until it is empty

Monitor the queue using the DISPLAY QUEUE command and specifying the attributes IPPROCS, OPPROCS, and CURDEPTH. When the number of input processes, the number of output processes, and the current depth of the queue are all zero, you can be sure that the queue is empty.

4. Monitor the channel to ensure there are no in-doubt messages

To be sure that there are no in-doubt messages on the channel TO.TORONTO, monitor the cluster-sender channel called TO.TORONTO on each of the other queue managers. To do this, issue the DISPLAY CHSTATUS command specifying the INDOUBT parameter from each queue manager:

```
DISPLAY CHSTATUS(TO.TORONTO) INDOUBT
```

If there are any in-doubt messages, you must take action to resolve them before proceeding. For example, you might try issuing the RESOLVE channel command or stopping and restarting the channel.

5. Delete the local queue

When you are satisfied that there are no more messages to be delivered to the inventory application at TORONTO, you can delete the queue:

```
DELETE QLOCAL(INVENTQ)
```

The cluster achieved by task 4

The cluster set up by this task is similar to that set up by the previous task, except that the INVENTQ queue is no longer available at queue manager TORONTO.

When you took the queue out of service (step 1), the TORONTO queue manager sent a message to the two repository queue managers notifying them of the change in status. The repository queue managers pass on this information to other queue managers in the cluster that have requested updates to information concerning the INVENTQ.

Now when a queue manager wants to put a message to the INVENTQ, it sees, from its updated partial repository, that the INVENTQ is available only at NEWYORK, and so sends its message there.

You may now choose to remove the inventory application from the system in Toronto, to avoid duplication and save space on the system.

Extensions to this task

In this task description there is only one queue to remove and only one cluster to remove it from.

Suppose that there were many queues referring to a namelist containing many cluster names. For example, the TORONTO queue manager might host not only the INVENTQ, but also the PAYROLLQ, SALESQ, and PURCHASESQ. TORONTO would make these queues available in all the appropriate clusters, INVENTORY, PAYROLL, SALES, and PURCHASES. To do this, TORONTO would define a namelist of the cluster names:

```
DEFINE NAMELIST(TOROLIST)
       DESCR('List of clusters TORONTO is in')
       NAMES(INVENTORY, PAYROLL, SALES, PURCHASES)
```

and specify this namelist on each queue definition, like this:

```
DEFINE QLOCAL(INVENTQ) CLUSNL(TOROLIST)
DEFINE QLOCAL(PAYROLLQ) CLUSNL(TOROLIST)
DEFINE QLOCAL(SALESQ) CLUSNL(TOROLIST)
DEFINE QLOCAL(PURCHASESQ) CLUSNL(TOROLIST)
```

Now suppose that you want to remove all those queues from the SALES cluster, because the SALES operation is to be taken over by the PURCHASES operation. All you would need to do is alter the TOROLIST namelist to remove the name of the SALES cluster from it.

If you wanted to remove a single queue from one of the clusters in the namelist, you would need to create a new namelist, containing the remaining list of cluster names, and then alter the queue definition to use the new namelist. To remove the PAYROLLQ from the INVENTORY cluster:

1. Create a new namelist:

   ```
   DEFINE NAMELIST(TOROSHORTLIST)
          DESCR('List of clusters TORONTO is in other than INVENTORY')
          NAMES(PAYROLL, SALES, PURCHASES)
   ```

2. Alter the PAYROLLQ queue definition:

   ```
   ALTER QLOCAL(PAYROLLQ) CLUSNL(TOROSHORTLIST)
   ```

Task 5: Removing a queue manager from a cluster

Scenario:

- The INVENTORY cluster has been set up as described in “Task 3: Adding a new queue manager that hosts a queue” on page 71 and modified as described in “Task 4: Removing a cluster queue from a queue manager” on page 75.
- For business reasons you no longer wish to carry out any inventory work at Toronto and so you wish to remove the TORONTO queue manager from the cluster.

The steps required to complete task 5

To achieve this, you need to perform the following tasks at the TORONTO queue manager.

1. Suspend queue manager TORONTO

Issue the SUSPEND QMGR command to suspend availability of the queue manager to the INVENTORY cluster:

```
SUSPEND QMGR CLUSTER(INVENTORY)
```

When you issue this command, other queue managers are advised that they should refrain from sending messages to TORONTO.

2. Stop the CLUSRCVR channel at TORONTO

Issue the STOP CHANNEL command to stop the cluster-receiver channel:

```
STOP CHANNEL(TO.TORONTO)
```

Once the channel is stopped, no more messages can be sent to TORONTO.

3. Remove the CLUSRCVR channel definition

You should now remove the CLUSRCVR definition from the cluster:

```
ALTER CHANNEL(TO.TORONTO) CHLTYPE(CLUSRCVR) CLUSTER(' ')
```

This command causes the repository queue managers to remove all information about that channel from their repositories, so that queue managers will no longer try to send messages to it.

Later, to tidy up, you will probably want to delete the channel:

```
DELETE CHANNEL(TO.TORONTO)
```

4. Delete the CLUSSDR channel definition

The CLUSSDR channel definition points to the repository at queue manager NEWYORK. Stop this channel as follows:

```
STOP CHANNEL(TO.NEWYORK)
```

and then delete it:

```
DELETE CHANNEL(TO.NEWYORK)
```

The cluster achieved by task 5

The cluster set up by this task looks like this:

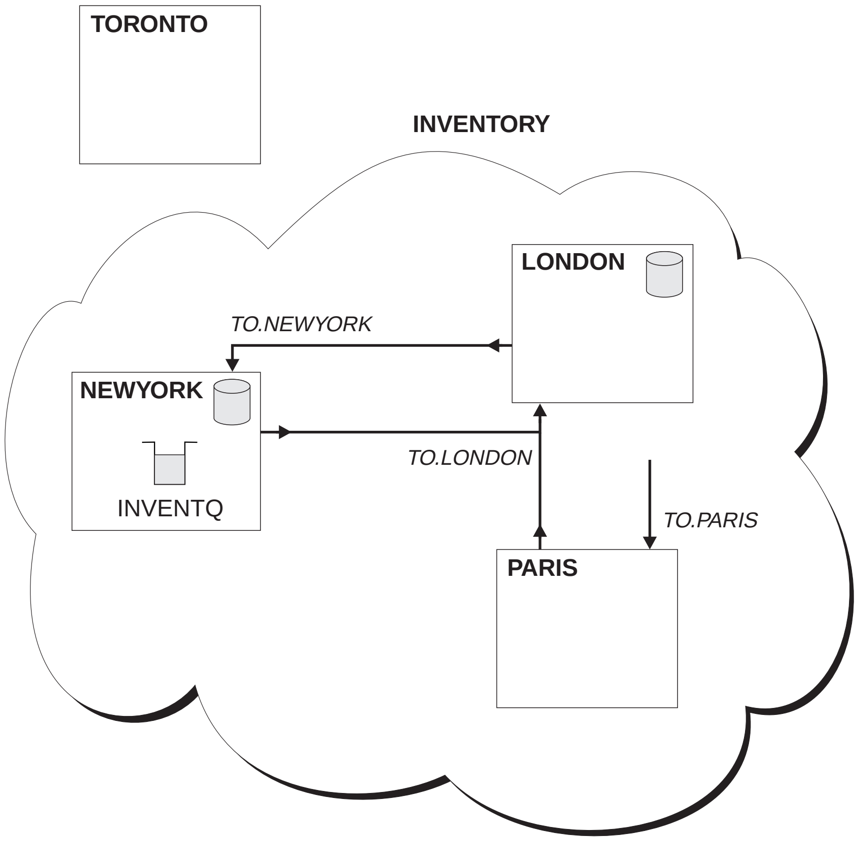

Figure 17. The INVENTORY cluster, with TORONTO outside the cluster

The queue manager TORONTO is no longer part of the cluster. However, it can still function as an independent queue manager.

This modification to the cluster was accomplished without you having to make any alterations to the queue managers NEWYORK, LONDON, and PARIS.

Note: Before you can remove a repository queue manager from a cluster you must perform an additional step. You must alter the queue manager definition to set the REPOS and REPOSNL attributes to blank. This sends a notification to other queue managers advising them that they must stop sending cluster information to this queue manager.

Task 6: Moving a repository to another queue manager

Scenario:

- The INVENTORY cluster has been set up as described in “Task 3: Adding a new queue manager that hosts a queue” on page 71 and modified as described in “Task 4: Removing a cluster queue from a queue manager” on page 75 and “Task 5: Removing a queue manager from a cluster” on page 77.
- For business reasons you now want to remove the repository from queue manager LONDON, and replace it with a repository at queue manager PARIS. The NEWYORK queue manager is to continue holding a full repository.

The steps required to complete task 6

To achieve this, you need to perform the following tasks.

1. Alter PARIS to make it a repository queue manager

On PARIS, issue the following command:

```
ALTER QMGR REPOS(INVENTORY)
```

2. Add a CLUSSDR channel on PARIS

PARIS currently has a cluster-sender channel pointing to LONDON. Now that LONDON is no longer to hold a full repository for the cluster, PARIS must have a new cluster-sender channel that points to NEWYORK, where the other repository is held.

```
DEFINE CHANNEL(TO.NEWYORK) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(NEWYORK.CHSTORE.COM) CLUSTER(INVENTORY)
```

3. Define a CLUSSDR channel on NEWYORK that points to PARIS

Currently NEWYORK has a cluster-sender channel pointing to LONDON. Now that the other repository has moved to PARIS, you need to add a new cluster-sender channel at NEWYORK that points to PARIS.

```
DEFINE CHANNEL(TO.PARIS) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(PARIS.CHSTORE.COM) CLUSTER(INVENTORY)
```

When you do this, the PARIS queue manager immediately learns all about the cluster, from NEWYORK, and builds up its own repository.

4. Alter the queue-manager definition on LONDON

The final step is to alter the queue manager at LONDON so that it no longer holds a full repository for the cluster. On LONDON, issue the command:

```
ALTER QMGR REPOS(' ')
```

The queue manager no longer receives any cluster information. After 30 days the information that is stored in its repository expires. The queue manager LONDON now builds up its own partial repository.

5. Remove or change any outstanding definitions

When you are sure that the new arrangement of your cluster is working as expected, you may remove or change any outstanding definitions that are no longer up-to-date. It is not essential that you do this, but you may choose to in order to tidy up.

- On the PARIS queue manager, delete the cluster-sender channel to LONDON.
  ```
  DELETE CHANNEL(TO.LONDON)
  ```
- On the NEWYORK queue manager, delete the cluster-sender channel to LONDON.
  ```
  DELETE CHANNEL(TO.LONDON)
  ```
- Similarly, replace all other cluster-sender channels in the cluster that point to LONDON with channels that point to either NEWYORK or PARIS. (In this small example there are no others.) To check whether there were any others that you had forgotten about, you could issue the DISPLAY CHANNEL command from each queue manager, specifying TYPE(CLUSSDR). For example:
  ```
  DISPLAY CHANNEL(*) TYPE(CLUSSDR)
  ```

The cluster achieved by task 6

The cluster set up by this task looks like this:

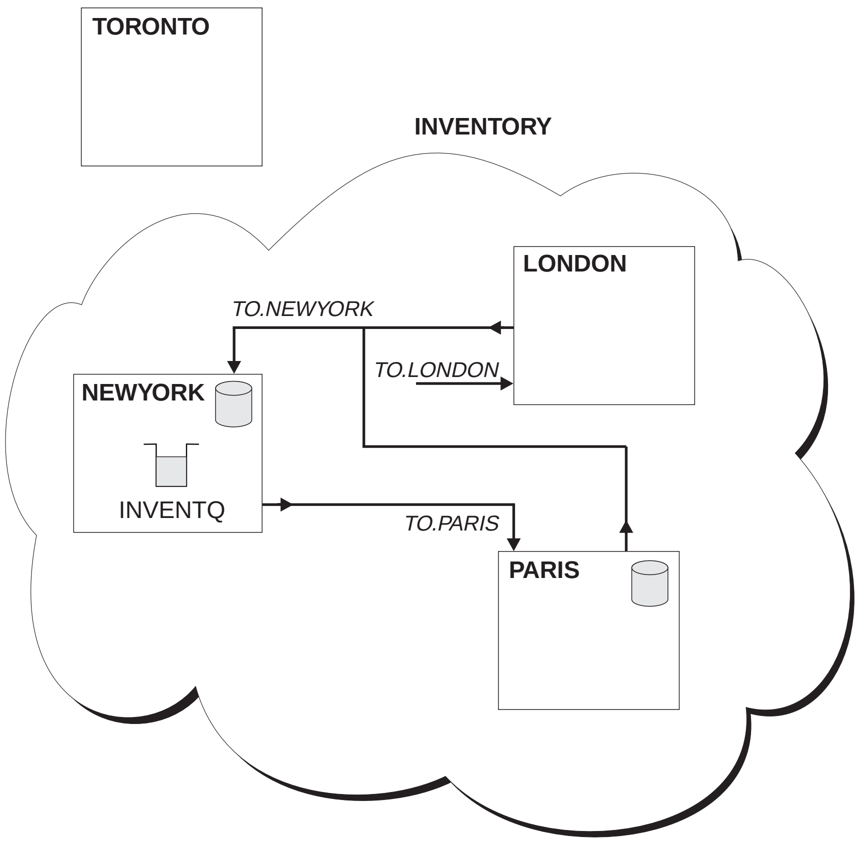

Figure 18. The INVENTORY cluster with the repository moved to PARIS

Task 7: Converting an existing network into a cluster

"Task 1: Setting up a new cluster" on page 19 through "Task 6: Moving a repository to another queue manager" on page 79 have been concerned with setting up and then extending a new cluster. The remaining two tasks explore a different approach: that of converting an existing network of queue managers into a cluster.

Scenario:

- An MQSeries network is already in place, connecting the nation-wide branches of a chain store. It has a hub and spoke structure: all the queue managers are connected to one central queue manager. The central queue manager is on the system on which the inventory application runs. The application is driven by the arrival of messages on the INVENTQ queue, for which each queue manager has a remote-queue definition.

  This network is illustrated in Figure 19.

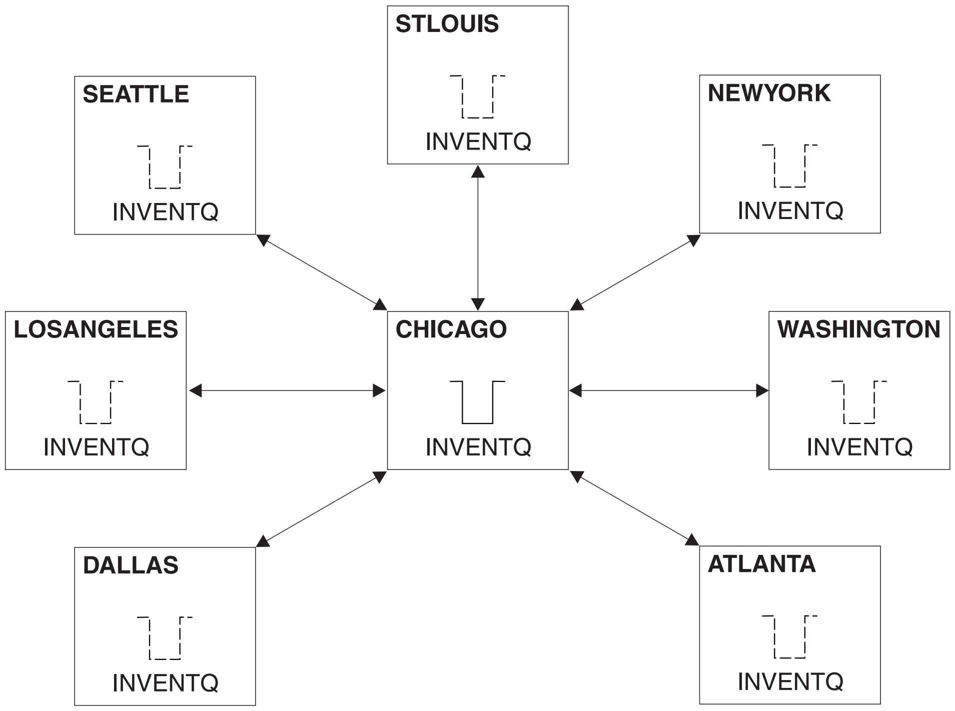

Figure 19. A hub and spoke network

- To ease administration you are going to convert this network into a cluster and create another queue manager at the central site to share the workload.
- Both the central queue managers are to host full repositories and be accessible to the inventory application.
- The inventory application is to be driven by the arrival of messages on the INVENTQ queue hosted by either of the central queue managers.
- The inventory application is to be the only application running in parallel and accessible by more than one queue manager. All other applications will continue to run as before.
- All the branches have network connectivity to the two central queue managers.
- The network protocol is TCP.

The steps required to complete task 7

To achieve this, you need to perform the following tasks.

Note: You do not need to convert your entire network all at once. This task could be completed in stages.

1. Upgrade MQSeries on your system

To use clusters, you must install one of the following products, V5.1 of MQSeries for AIX, AS/400, HP-UX, OS/2 Warp, Sun Solaris, and Windows NT, or MQSeries for OS/390.

2. Review the inventory application for message affinities

Before proceeding it is important to ensure that the application will not have a problem with message affinities. See “Reviewing applications for message affinities” on page 46 for more information.

3. Prepare the new queue manager at the central site

Preparation of queue managers is described in “How should I prepare for use of clustering?” on page 14.

4. Alter the two central queue managers to make them repository queue managers

The two queue managers CHICAGO and CHICAGO2 are at the hub of this network. You have decided to concentrate all activity associated with the chainstore cluster on to those two queue managers. As well as the inventory application and the definitions for the INVENTQ queue, you want these queue managers to host the two full repositories for the cluster. At each of the two queue managers, issue the following command:

```
ALTER QMGR REPOS(CHAINSTORE)
```

5. Define a CLUSRCVR channel on each queue manager

At each queue manager in the cluster, you need to define a cluster-receiver channel and a cluster-sender channel. It does not matter which of these you define first.

Make a CLUSRCVR definition to advertise each queue manager, its network address, and so on, to the cluster. For example, on queue manager ATLANTA:

```
DEFINE CHANNEL(TO.ATLANTA) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(ATLANTA.CHSTORE.COM) CLUSTER(CHAINSTORE)
```

6. Define a CLUSSDR channel on each queue manager

Make a CLUSSDR definition at each queue manager to link that queue manager to one or other of the repository queue managers. For example, you might link ATLANTA to CHICAGO2:

```
DEFINE CHANNEL(TO.CHICAGO2) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(CHICAGO2.CHSTORE.COM) CLUSTER(CHAINSTORE)
```

7. Install the inventory application on CHICAGO2

You already have the inventory application on queue manager CHICAGO. Now you need to make a copy of this application on queue manager CHICAGO2. Refer to the *MQSeries Application Programming Guide* and install the inventory application on CHICAGO2.

8. Define the INVENTQ queue on the central queue managers

On CHICAGO you need to modify the local queue definition for the queue INVENTQ to make the queue available to the cluster. Issue the command:

```
ALTER QLOCAL(INVENTQ) CLUSTER(CHAINSTORE)
```

On CHICAGO2 you need to make a definition for the same queue:

```
DEFINE QLOCAL(INVENTQ) CLUSTER(CHAINSTORE)
```

(On OS/390 you could use the MAKEDEF option of the COMMAND function of CSQUTIL to make an exact copy on CHICAGO2 of the INVENTQ on CHICAGO. See the *MQSeries for OS/390 System Management Guide* for details.)

When you make these definitions, a message is sent to the repositories at CHICAGO and CHICAGO2 and the information in them is updated. From then on, when a queue manager wants to put a message to the INVENTQ, it will find out from the repositories that there is a choice of destinations for messages sent to that queue.

9. Delete all remote-queue definitions for the INVENTQ

Now that you have linked all your queue managers together in the CHAINSTORE cluster, and have defined the INVENTQ to the cluster, the queue managers no longer need remote-queue definitions for the INVENTQ. At every queue manager, issue the command:

```
DELETE QREMOTE(INVENTQ)
```

Until you do this, the remote-queue definitions will continue to be used and you will not get the benefit of using clusters.

10. Implement the cluster workload exit (optional step)

Because there is more than one destination for messages sent to the INVENTQ, the workload management algorithm will determine which destination each message will be sent to.

If you wish to implement your own workload management routine, you can write a cluster workload exit program. See “Workload balancing” on page 42 for more information.

Now that you have completed all the definitions, if you have not already done so you should start the channel initiator on MQSeries for OS/390 and, on all platforms, start a listener program on each queue manager. The listener program listens for incoming network requests and starts the cluster-receiver channel when it is needed. See “Establishing communication in a cluster” on page 16 for more information.

The cluster achieved by task 7

The cluster set up by this task looks like this:

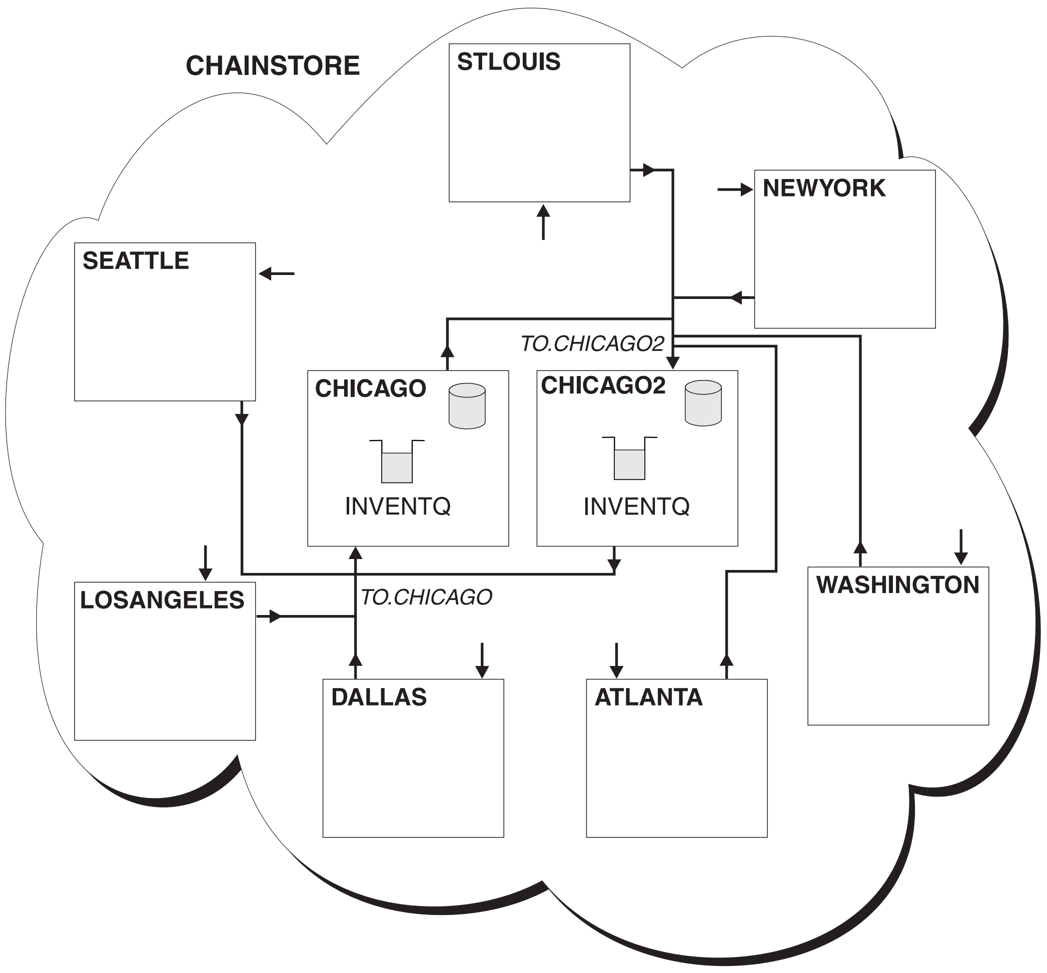

Figure 20. A cluster with a hub and spokes

Remember, that as with the other diagrams in this book, this diagram shows only the channels that you have to define manually. Cluster-sender channels are defined automatically when needed so that ultimately all queue managers can receive cluster information from the two repository queue managers and also messages from the two applications.

Once again, this is a very small example - little more than a proof of concept. In your enterprise it is unlikely that you would have a cluster of this size with only one queue.

You could easily add more queues to the cluster environment by adding the CLUSTER parameter to your queue definitions, and then removing all corresponding remote-queue definitions from the other queue managers.

Task 8: Adding a new, interconnected cluster

Scenario:

- An MQSeries cluster has been set up as described in “Task 7: Converting an existing network into a cluster” on page 81.
- A new cluster called MAILORDER is to be implemented. This cluster will comprise four of the queue managers that are in the CHAINSTORE cluster; CHICAGO, CHIGACO2, SEATTLE, and ATLANTA, and two additional queue managers; HARTFORD and OMAHA. The MAILORDER application will run on the system at Omaha, connected to queue manager OMAHA. It will be driven by the other queue managers in the cluster putting messages on the MORDERQ queue.
- The repositories for the MAILORDER cluster will be maintained on the two queue managers CHICAGO and CHICAGO2.
- The network protocol is TCP.

The steps required to complete task 8

To achieve this, you need to perform the following tasks.

1. Create a namelist of the cluster names

The repository queue managers at CHICAGO and CHICAGO2 are now going to hold the repositories for both of the clusters CHAINSTORE and MAILORDER. You need to alter the queue-manager definitions to add the new cluster name, but before you can do this you must create a namelist containing the names of the clusters. Define the namelist on CHICAGO and CHICAGO2, as follows:

```
DEFINE NAMELIST(CHAINMAIL)
       DESCR('List of cluster names')
       NAMES(CHAINSTORE, MAILORDER)
```

2. Alter the two queue-manager definitions

Now you can alter the two queue-manager definitions, at CHICAGO and CHICAGO2. Currently these definitions show that the queue managers hold repositories for the cluster CHAINSTORE. You need to change that definition to show that the queue managers hold repositories for all clusters listed in the CHAINMAIL namelist. To do this, at both CHICAGO and CHICAGO2, specify:

```
ALTER QMGR REPOS(' ') REPOSNL(CHAINMAIL)
```

3. Alter the CLUSRCVR channels on CHICAGO and CHICAGO2

The CLUSRCVR channel definitions at CHICAGO and CHICAGO2 show that the channels are available in the cluster CHAINSTORE. You need to change this to show that the channels are available to all clusters listed in the CHAINMAIL namelist. To do this, at CHICAGO, enter the command:

```
ALTER CHANNEL(TO.CHICAGO) CHLTYPE(CLUSRCVR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

At CHICAGO2, enter the command:

```
ALTER CHANNEL(TO.CHICAGO2) CHLTYPE(CLUSRCVR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

4. Alter the CLUSSDR channels on CHICAGO and CHICAGO2

Similarly, you need to change the two CLUSSDR channel definitions to add the namelist. At CHICAGO, enter the command:

```
ALTER CHANNEL(TO.CHICAGO2) CHLTYPE(CLUSSDR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

At CHICAGO2, enter the command:

```
ALTER CHANNEL(TO.CHICAGO) CHLTYPE(CLUSSDR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

5. Create a namelist on SEATTLE and ATLANTA

Because SEATTLE and ATLANTA are going to be members of more than one cluster, you must create a namelist containing the names of the clusters. Define the namelist on SEATTLE and ATLANTA, as follows:

```
DEFINE NAMELIST(CHAINMAIL)
       DESCR('List of cluster names')
       NAMES(CHAINSTORE, MAILORDER)
```

6. Alter the CLUSRCVR channels on SEATTLE and ATLANTA

The CLUSRCVR channel definitions at SEATTLE and ATLANTA show that the channels are available in the cluster CHAINSTORE. You need to change this to show that the channels are available to all clusters listed in the CHAINMAIL namelist. To do this, at SEATTLE, enter the command:

```
ALTER CHANNEL(TO.SEATTLE) CHLTYPE(CLUSRCVR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

At ATLANTA, enter the command:

```
ALTER CHANNEL(TO.ATLANTA) CHLTYPE(CLUSRCVR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

7. Alter the CLUSSDR channels on SEATTLE and ATLANTA

Similarly, you need to change the two CLUSSDR channel definitions to add the namelist. At SEATTLE, enter the command:

```
ALTER CHANNEL(TO.CHICAGO) CHLTYPE(CLUSSDR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

At ATLANTA, enter the command:

```
ALTER CHANNEL(TO.CHICAGO2) CHLTYPE(CLUSSDR)
      CLUSTER(' ') CLUSNL(CHAINMAIL)
```

8. Prepare the queue managers HARTFORD and OMAHA

Preparation of queue managers is described in “How should I prepare for use of clustering?” on page 14.

9. Define CLUSRCVR and CLUSSDR channels on HARTFORD and OMAHA

On the two new queue managers HARTFORD and OMAHA, you need to define cluster-receiver and cluster-sender channels. It doesn’t matter in which sequence you do this. At HARTFORD, enter:

```
DEFINE CHANNEL(TO.HARTFORD) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(HARTFORD.CHSTORE.COM) CLUSTER(MAILORDER)

DEFINE CHANNEL(TO.CHICAGO) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(CHICAGO.CHSTORE.COM) CLUSTER(MAILORDER)
```

At OMAHA, enter:

```
DEFINE CHANNEL(TO.OMAHA) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
CONNAME(OMAHA.CHSTORE.COM) CLUSTER(MAILORDER)

DEFINE CHANNEL(TO.CHICAGO) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
CONNAME(CHICAGO.CHSTORE.COM) CLUSTER(MAILORDER)
```

10. Define the MORDERQ queue on OMAHA

The final step to complete this task is to define the queue MORDERQ on the queue manager OMAHA. To do this, enter:

```
DEFINE QLOCAL(MORDERQ) CLUSTER(MAILORDER)
```

The cluster achieved by task 8

The cluster set up by this task looks like this:

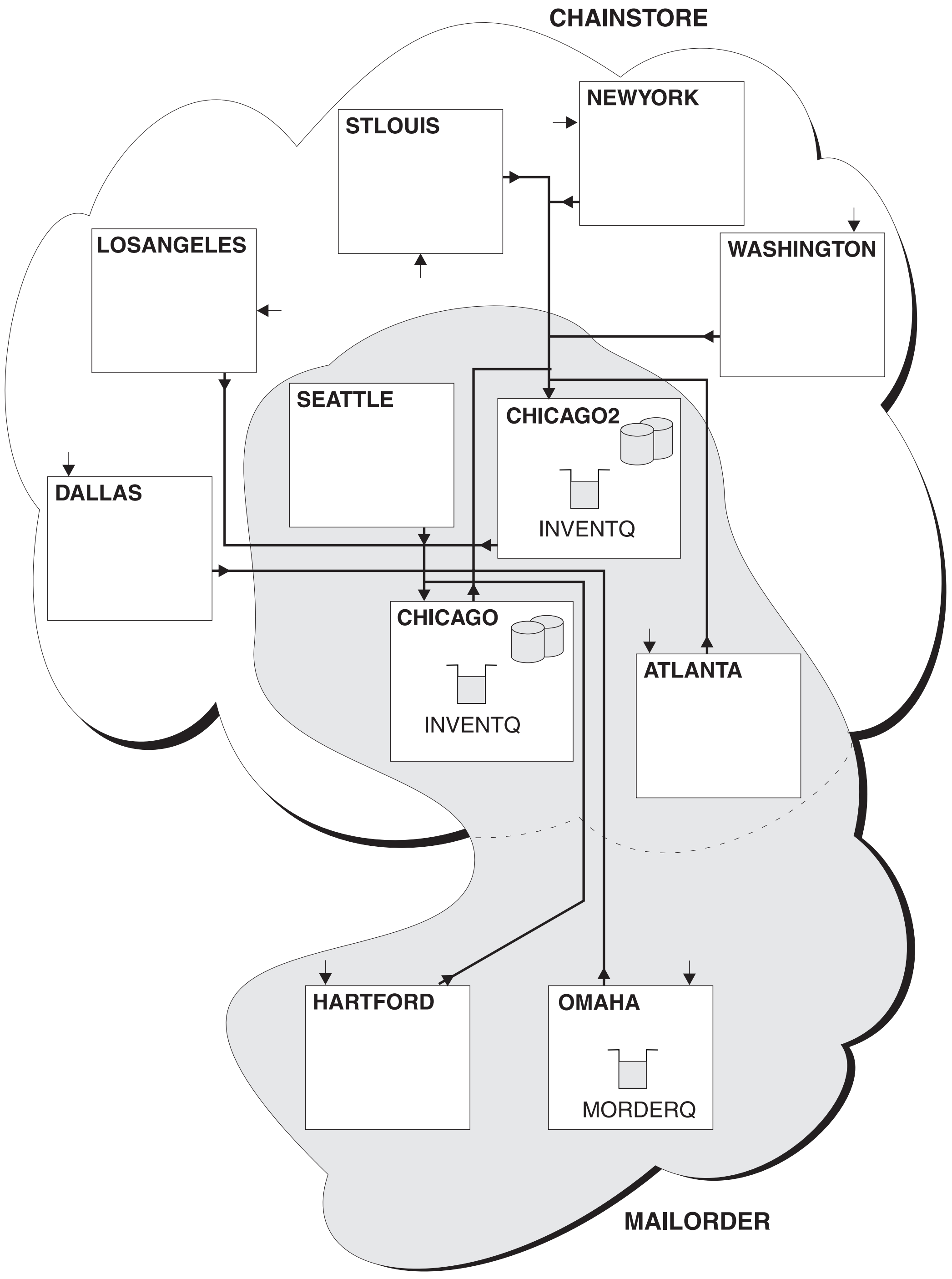

Figure 21. Interconnected clusters

Now we have two overlapping clusters. The repositories for both clusters are held at CHICAGO and CHICAGO2. The mailorder application that runs on OMAHA is independent of the inventory application that runs at CHICAGO. However, some of the queue managers that are in the CHAINSTORE cluster are also in the MAILORDER cluster, and so they can send messages to either application. Before carrying out this task to overlap two clusters, you should be aware of the possibility of queue-name clashes. Suppose that on NEWYORK in cluster CHAINSTORE and on OMAHA in cluster MAILORDER, there was a queue called ACCOUNTQ. If you overlap the clusters and then an application on a queue manager that is a member of both clusters, for example SEATTLE, puts a message to the queue ACCOUNTQ, the message could go to either instance of the ACCOUNTQ. This may or may not be what you intended.

Before starting this task the system administrators of the two clusters must check for queue-name clashes and be sure that they understand the consequences. You may need to rename a queue, or perhaps set up queue aliases before you can proceed.

Extensions to this task

One day in the future you might decide to merge the MAILORDER cluster with the CHAINSTORE cluster to form one large cluster called CHAINSTORE.

To merge the MAILORDER cluster with the CHAINSTORE cluster, such that CHICAGO and CHICAGO2 hold the repositories, you need to:

- Alter the queue manager definitions for CHICAGO and CHICAGO2, removing the REPOSNL attribute, which specifies the namelist (CHAINMAIL), and replacing it with a REPOS attribute specifying the cluster name (CHAINSTORE). For example:

  ```
  ALTER QMGR(CHICAGO) REPOSNL(' ') REPOS(CHAINSTORE)
  ```
- On each queue manager in the MAILORDER cluster, alter all the channel definitions and queue definitions to change the value of the CLUSTER attribute from MAILORDER to CHAINSTORE. For example, at HARTFORD, enter:

  ```
  ALTER CHANNEL(TO.HARTFORD) CLUSTER(CHAINSTORE)
  ```

  At OMAHA enter:

  ```
  ALTER QLOCAL(MORDERQ) CLUSTER(CHAINSTORE)
  ```
- Alter all definitions that specify the cluster namelist CHAINMAIL, that is, the CLUSRCVR and CLUSSDR channel definitions at CHICAGO, CHICAGO2, SEATTLE, and ATLANTA, to specify instead the cluster CHAINSTORE.

From this example, you can see the advantage of using namelists. Instead of altering the queue manager definitions for CHICAGO and CHICAGO2 you could just alter the value of the namelist CHAINMAIL. Similarly, instead of altering the CLUSRCVR and CLUSSDR channel definitions at CHICAGO, CHICAGO2, SEATTLE, and ATLANTA, you could achieve the required result by altering the namelist.

Part 3. Reference information

© Copyright IBM Corp. 1999, 2000

Chapter 10. Cluster workload exit call and data structures

This chapter provides reference information concerning the cluster workload exit and the data structures it uses. This is general-use programming interface information.

| You can write cluster workload exits in the following programming languages:
| • C
| • System / 390 assembler (MQSeries for OS / 390)

| The call is described in:

- "MQ_CLUSTER_WORKLOAD_EXIT - Cluster workload exit" on page 94

The structure data types used by the exit are described in:

- "MQWXP - Cluster workload exit parameter structure" on page 95
- "MQWDR - Cluster workload destination-record structure" on page 102
- "MQWQR - Cluster workload queue-record structure" on page 106
- "MQWCR - Cluster workload cluster-record structure" on page 110

Constants relating to the cluster workload exit are listed with their values in "Chapter 11. Constants for the cluster workload exit" on page 113.

Throughout the following pages queue-manager attributes and queue attributes are shown in full, as defined in the *MQSeries Application Programming Reference* book. The equivalent names that are used in the MQSC commands described in the *MQSeries Command Reference* book are shown in Table 3 and Table 4.

Table 3. Queue-manager attributes

Full name	**Name used in MQSC**
ClusterWorkloadData	CLWLDATA
ClusterWorkloadExit	CLWLEXIT
ClusterWorkloadLength	CLWLLEN

Table 4. Queue attributes

Full name	**Name used in MQSC**
DefBind	DEFBIND
DefPersistence	DEFPSIST
DefPriority	DEFPRTY
InhibitPut	PUT
QDesc	DESCR

MQ_CLUSTER_WORKLOAD_EXIT - Cluster workload exit

This call definition describes the parameters that are passed to the cluster workload exit called by the queue manager.

Note: No entry point called MQ_CLUSTER_WORKLOAD_EXIT is actually provided by the queue manager. This is because the name of the cluster workload exit is defined by the *ClusterWorkloadExit* queue-manager attribute.

| This exit is supported in the following environments: AIX, HP-UX, OS/2, OS/390,
| OS/400, Sun Solaris, and Windows NT.

Syntax

MQ_CLUSTER_WORKLOAD_EXIT *(ExitParms)*

Parameters

ExitParms (MQWXP) – input/output
Exit parameter block.

This structure contains information relating to the invocation of the exit. The exit sets information in this structure to indicate how the workload should be managed.

Usage notes

1. The function performed by the cluster workload exit is defined by the provider of the exit. The exit, however, must conform to the rules defined in the associated control block MQWXP.
2. No entry point called MQ_CLUSTER_WORKLOAD_EXIT is actually provided by the queue manager. However, a **typedef** is provided for the name MQ_CLUSTER_WORKLOAD_EXIT in the C programming language, and this can be used to declare the user-written exit, to ensure that the parameters are correct.

C invocation

```
exitname (&ExitParms);
```

Declare the parameters as follows:

```
MQWXP  ExitParms;  /* Exit parameter block */
```

System/390 assembler invocation

```
         CALL EXITNAME,(EXITPARMS)
```

Declare the parameters as follows:

```
EXITPARMS          CMQWXPA             Exit parameter block
```

MQWXP - Cluster workload exit parameter structure

The following table summarizes the fields in the structure.

Table 5. Fields in MQWXP

Field	Description	Page
StrucId	Structure identifier	95
Version	Structure version number	96
ExitId	Type of exit	96
ExitReason	Reason for invoking exit	96
ExitResponse	Response from exit	97
ExitResponse2	Secondary response from exit	97
Feedback	Feedback code	97
ExitUserArea	Exit user area	98
ExitData	Exit data	98
MsgDescPtr	Address of message descriptor (MQMD)	98
MsgBufferPtr	Address of buffer containing some or all of the message data	98
MsgBufferLength	Length of buffer containing message data	99
MsgLength	Length of complete message	99
QName	Name of queue	99
QMgrName	Name of local queue manager	99
DestinationCount	Number of possible destinations	99
DestinationChosen	Destination chosen	99
DestinationArrayPtr	Address of an array of pointers to destination records (MQWDR)	100
QArrayPtr	Address of an array of pointers to queue records (MQWQR)	100

The MQWXP structure describes the information that is passed to the cluster workload exit.

| This structure is supported in the following environments: AIX, HP-UX, OS/390,
| OS/400, OS/2, Sun Solaris, and Windows NT.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value is:

MQWXP_STRUC_ID
Identifier for cluster workload exit parameter structure.

For the C programming language, the constant MQWXP_STRUC_ID_ARRAY is also defined; this has the same value as MQWXP_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)
Structure version number.

The value is:

MQWXP_VERSION_1
Version-1 cluster workload exit parameter structure.

The following constant specifies the version number of the current version:

MQWXP_CURRENT_VERSION
Current version of cluster workload exit parameter structure.

This is an input field to the exit.

ExitId (MQLONG)
Type of exit.

This indicates the type of exit being called. The value is:

MQXT_CLUSTER_WORKLOAD_EXIT
Cluster workload exit.

| This type of exit is supported in the following environments: AIX,
| HP-UX, OS/2, OS/390, OS/400, Sun Solaris, and Windows NT.

This is an input field to the exit.

ExitReason (MQLONG)
Reason for invoking exit.

This indicates the reason why the exit is being called. Possible values are:

MQXR_INIT
Exit initialization.

This indicates that the exit is being invoked for the first time. It allows the exit to acquire and initialize any resources that it may need (for example: main storage).

MQXR_TERM
Exit termination.

This indicates that the exit is about to be terminated. The exit should free any resources that it may have acquired since it was initialized (for example: main storage).

MQXR_CLWL_OPEN
Called from MQOPEN processing.

MQXR_CLWL_PUT
Called from MQPUT or MQPUT1 processing.

MQXR_CLWL_MOVE
Called from MCA when the message state has changed.

MQXR_CLWL_REPOS
Called from MQPUT or MQPUT1 processing for a repository-manager PCF message.

MQXR_CLWL_REPOS_MOVE
: Called from MCA for a repository-manager PCF message when the message state has changed.

This is an input field to the exit.

ExitResponse (MQLONG)
: Response from exit.

This is set by the exit to indicate whether processing of the message should continue. It must be one of the following:

MQXCC_OK
: Continue normally.

This indicates that processing of the message should continue normally. *DestinationChosen* identifies the destination to which the message should be sent.

MQXCC_SUPPRESS_FUNCTION
: Suppress function.

This indicates that processing of the message should be discontinued:

- For MQXR_CLWL_OPEN, MQXR_CLWL_PUT, and MQXR_CLWL_REPOS invocations, the MQOPEN, MQPUT, or MQPUT1 call fails with completion code MQCC_FAILED and reason code MQRC_STOPPED_BY_CLUSTER_EXIT.
- For MQXR_CLWL_MOVE and MQXR_CLWL_REPOS_MOVE invocations, the message is placed on the dead-letter queue.

MQXCC_SUPPRESS_EXIT
: Suppress exit.

This indicates that processing of the current message should continue normally, but that the exit should not be invoked again until termination of the queue manager. The queue manager processes subsequent messages as if the *ClusterWorkloadExit* queue-manager attribute were blank. *DestinationChosen* identifies the destination to which the current message should be sent.

If any other value is returned by the exit, the queue manager processes the message as if MQXCC_SUPPRESS_FUNCTION had been specified.

This is an output field from the exit.

ExitResponse2 (MQLONG)
: Reserved.

This is a reserved field. The value is zero.

Feedback (MQLONG)
: Reserved.

This is a reserved field. The value is zero.

Reserved (MQLONG)
: Reserved.

This is a reserved field. The value is zero.

ExitUserArea (MQBYTE16)
Exit user area.

This is a field that is available for the exit to use. It is initialized to MQXUA_NONE (binary zero) before the first invocation of the exit, and thereafter any changes made to this field by the exit are preserved across the invocations of the exit that occur between the MQCONN call and the matching MQDISC call. The field is reset to MQXUA_NONE when the MQDISC call occurs. The first invocation of the exit is indicated by the *ExitReason* field having the value MQXR_INIT.

The following value is defined:

MQXUA_NONE
No user information.

The value is binary zero for the length of the field.

For the C programming language, the constant MQXUA_NONE_ARRAY is also defined; this has the same value as MQXUA_NONE, but is an array of characters instead of a string.

The length of this field is given by MQ_EXIT_USER_AREA_LENGTH. This is an input/output field to the exit.

ExitData (MQCHAR32)
Exit data.

This is set on input to the exit routine to the value of the *ClusterWorkloadData* queue-manager attribute. If no value has been defined for that attribute, this field is all blanks.

The length of this field is given by MQ_EXIT_DATA_LENGTH. This is an input field to the exit.

MsgDescPtr (PMQMD)
Address of message descriptor.

This is the address of a copy of the message descriptor (MQMD) for the message being processed. Any changes made to the message descriptor by the exit are ignored by the queue manager.

No message descriptor is passed to the exit if *ExitReason* has one of the following values:
MQXR_INIT
MQXR_TERM
MQXR_CLWL_OPEN

In these cases, *MsgDescPtr* is the null pointer.

This is an input field to the exit.

MsgBufferPtr (PMQVOID)
Address of buffer containing some or all of the message data.

This is the address of a buffer containing a copy of the first *MsgBufferLength* bytes of the message data. Any changes made to the message data by the exit are ignored by the queue manager.

No message data is passed to the exit in the following cases:
- When *MsgDescPtr* is the null pointer.
- When the message has no data.
- When the *ClusterWorkloadLength* queue-manager attribute is zero.

In these cases, *MsgBufferPtr* is the null pointer.

This is an input field to the exit.

MsgBufferLength (MQLONG)

Length of buffer containing message data.

This is the length of the message data passed to the exit. This length is controlled by the *ClusterWorkloadLength* queue-manager attribute, and may be less than the length of the complete message (see *MsgLength*).

This is an input field to the exit.

MsgLength (MQLONG)

Length of complete message.

Be aware that the length of the message data passed to the exit (*MsgBufferLength*) may be less than the length of the complete message. *MsgLength* is zero if *ExitReason* is MQXR_INIT, MQXR_TERM, or MQXR_CLWL_OPEN.

This is an input field to the exit.

QName (MQCHAR48)

Queue name.

This is the name of the destination queue; this queue is a cluster queue.

The length of this field is given by MQ_Q_NAME_LENGTH. This is an input field to the exit.

QMgrName (MQCHAR48)

Name of local queue manager.

This is the name of the queue manager that has invoked the cluster workload exit.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. This is an input field to the exit.

DestinationCount (MQLONG)

Number of possible destinations.

This specifies the number of destination records (MQWDR) that describe instances of the destination queue. There is one MQWDR structure for each possible route to each instance of the queue. The MQWDR structures are addressed by an array of pointers (see *DestinationArrayPtr*).

This is an input field to the exit.

DestinationChosen (MQLONG)

Destination chosen.

This is the number of the MQWDR structure that identifies the route and queue instance to which the message should be sent. The value is in the range 1 through *DestinationCount*.

On input to the exit, *DestinationChosen* indicates the route and queue instance that the queue manager has selected. The exit can accept this choice, or choose a different route and queue instance. However, the value returned by the exit must be in the range 1 through *DestinationCount*. If any other value is returned, the queue manager uses the value that *DestinationChosen* had on input to the exit.

This is an input/output field to the exit.

DestinationArrayPtr (PPMQWDR)
Address of an array of pointers to destination records.

This is the address of an array of pointers to destination records (MQWDR). There are *DestinationCount* destination records.

This is an input field to the exit.

QArrayPtr (PPMQWQR)
Address of an array of pointers to queue records.

This is the address of an array of pointers to queue records (MQWQR). If queue records are available, there are *DestinationCount* of them. If no queue records are available, *QArrayPtr* is the null pointer.

Note: *QArrayPtr* can be the null pointer even when *DestinationCount* is greater than zero.

This is an input field to the exit.

C declaration

```
typedef struct tagMQWXP {
  MQCHAR4   StrucId;             /* Structure identifier */
  MQLONG    Version;             /* Structure version number */
  MQLONG    ExitId;              /* Type of exit */
  MQLONG    ExitReason;          /* Reason for invoking exit */
  MQLONG    ExitResponse;        /* Response from exit */
  MQLONG    ExitResponse2;       /* Reserved */
  MQLONG    Feedback;            /* Reserved */
  MQLONG    Reserved;            /* Reserved */
  MQBYTE16  ExitUserArea;        /* Exit user area */
  MQCHAR32  ExitData;            /* Exit data */
  PMQMD     MsgDescPtr;          /* Address of message descriptor */
  PMQVOID   MsgBufferPtr;        /* Address of buffer containing some
                                    or all of the message data */
  MQLONG    MsgBufferLength;     /* Length of buffer containing message
                                    data */
  MQLONG    MsgLength;           /* Length of complete message */
  MQCHAR48  QName;               /* Queue name */
  MQCHAR48  QMgrName;            /* Name of local queue manager */
  MQLONG    DestinationCount;    /* Number of possible destinations */
  MQLONG    DestinationChosen;   /* Destination chosen */
  PPMQWDR   DestinationArrayPtr; /* Address of an array of pointers to
                                    destination records */
  PPMQWQR   QArrayPtr;           /* Address of an array of pointers to
                                    queue records */
 } MQWXP;
```

System/390 assembler declaration

```
MQWXP                         DSECT
MQWXP_STRUCID                 DS   CL4      Structure identifier
MQWXP_VERSION                 DS   F        Structure version number
MQWXP_EXITID                  DS   F        Type of exit
MQWXP_EXITREASON              DS   F        Reason for invoking exit
MQWXP_EXITRESPONSE            DS   F        Response from exit
MQWXP_EXITRESPONSE2           DS   F        Reserved
MQWXP_FEEDBACK                DS   F        Reserved
MQWXP_RESERVED                DS   F        Reserved
MQWXP_EXITUSERAREA            DS   XL16     Exit user area
MQWXP_EXITDATA                DS   CL32     Exit data
MQWXP_MSGDESCPTR              DS   F        Address of message
*                                           descriptor
MQWXP_MSGBUFFERPTR            DS   F        Address of buffer containing
*                                           some or all of the message
*                                           data
MQWXP_MSGBUFFERLENGTH         DS   F        Length of buffer containing
*                                           message data
MQWXP_MSGLENGTH               DS   F        Length of complete message
MQWXP_QNAME                   DS   CL48     Queue name
MQWXP_QMGRNAME                DS   CL48     Name of local queue manager
MQWXP_DESTINATIONCOUNT        DS   F        Number of possible
*                                           destinations
MQWXP_DESTINATIONCHOSEN       DS   F        Destination chosen
MQWXP_DESTINATIONARRAYPTR     DS   F        Address of an array of
*                                           pointers to destination
*                                           records
MQWXP_QARRAYPTR               DS   F        Address of an array of
*                                           pointers to queue records
MQWXP_LENGTH                  EQU  *-MQWXP  Length of structure
                              ORG  MQWXP
MQWXP_AREA                    DS   CL(MQWXP_LENGTH)
```

MQWDR - Cluster workload destination-record structure

The following table summarizes the fields in the structure.

Table 6. Fields in MQWDR

Field	Description	Page
StrucId	Structure identifier	102
Version	Structure version number	102
StrucLength	Length of MQWDR structure	103
QMgrFlags	Queue-manager flags	103
QMgrIdentifier	Queue-manager identifier	103
QMgrName	Queue-manager name	103
ClusterRecOffset	Offset of first cluster record (MQWCR)	104
ChannelState	Channel state	104
ChannelDefOffset	Offset of channel-definition structure (MQCD)	104

The MQWDR structure contains information relating to one of the possible destinations for the message. There is one MQWDR structure for each instance of the destination queue.

| This structure is supported in the following environments: AIX, HP-UX, OS/2,
| OS/390, OS/400, Sun Solaris, and Windows NT.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value is:

MQWDR_STRUC_ID
Identifier for cluster workload destination record.

For the C programming language, the constant MQWDR_STRUC_ID_ARRAY is also defined; this has the same value as MQWDR_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)
Structure version number.

The value is:

MQWDR_VERSION_1
Version-1 cluster workload destination record.

The following constant specifies the version number of the current version:

MQWDR_CURRENT_VERSION
Current version of cluster workload destination record.

This is an input field to the exit.

StrucLength (MQLONG)
Length of MQWDR structure.

The value is:

MQWDR_LENGTH_1
Length of version-1 cluster workload destination record.

The following constant specifies the length of the current version:

MQWDR_CURRENT_LENGTH
Length of current version of cluster workload destination record.

This is an input field to the exit.

QMgrFlags (MQLONG)
Queue-manager flags

These are bit flags that indicate various properties of the queue manager that hosts the instance of the destination queue described by this MQWDR structure. The following flags are defined:

MQQMF_REPOSITORY_Q_MGR
Destination is a repository queue manager.

MQQMF_CLUSSDR_USER_DEFINED
Cluster sender channel was defined manually.

MQQMF_CLUSSDR_AUTO_DEFINED
Cluster sender channel was defined automatically.

MQQMF_AVAILABLE
Destination queue manager is available to receive messages.

Note: Other flags in the field may be set by the queue manager for internal purposes.

This is an input field to the exit.

QMgrIdentifier (MQCHAR48)
Queue-manager identifier.

This is a string that acts as a unique identifier for the queue manager. It is generated by the queue manager.

The length of this field is given by MQ_Q_MGR_IDENTIFIER_LENGTH. This is an input field to the exit.

QMgrName (MQCHAR48)
Queue-manager name.

This is the name of the queue manager that hosts the instance of the destination queue described by this MQWDR structure. This can be the name of the local queue manager.

The length of this field is given by MQ_Q_MGR_NAME_LENGTH. This is an input field to the exit.

ClusterRecOffset (MQLONG)
Offset of first cluster record.

This is the offset of the first MQWCR structure that belongs to this MQWDR structure. The offset is measured in bytes from the start of the MQWDR structure.

This is an input field to the exit.

ChannelState (MQLONG)
Channel state.

This indicates the state of the channel that links the local queue manager to the queue manager identified by this MQWDR structure. The following values are possible:

MQCHS_INACTIVE
Channel is not active.

MQCHS_BINDING
Channel is negotiating with the partner.

MQCHS_STARTING
Channel is waiting to become active.

MQCHS_RUNNING
Channel is transferring or waiting for messages.

MQCHS_STOPPING
Channel is in the process of stopping.

MQCHS_RETRYING
Channel is reattempting to establish connection.

MQCHS_STOPPED
Channel is stopped.

MQCHS_REQUESTING
Requester channel is requesting connection.

MQCHS_PAUSED
Channel is paused.

MQCHS_INITIALIZING
Channel is initializing.

This is an input field to the exit.

ChannelDefOffset (MQLONG)
Offset of channel definition structure.

This is the offset of the channel definition (MQCD) for the channel that links the local queue manager to the queue manager identified by this MQWDR structure. The offset is measured in bytes from the start of the MQWDR structure.

This is an input field to the exit.

C declaration

```
typedef struct tagMQWDR {
  MQCHAR4   StrucId;          /* Structure identifier */
  MQLONG    Version;          /* Structure version number */
  MQLONG    StrucLength;      /* Length of MQWDR structure */
  MQLONG    QMgrFlags;        /* Queue-manager flags */
  MQCHAR48  QMgrIdentifier;   /* Queue-manager identifier */
  MQCHAR48  QMgrName;         /* Queue-manager name */
  MQLONG    ClusterRecOffset; /* Offset of first cluster record */
  MQLONG    ChannelState;     /* Channel state */
  MQLONG    ChannelDefOffset; /* Offset of channel definition
                                 structure */
 } MQWDR;
```

System/390 assembler declaration

```
MQWDR                        DSECT
MQWDR_STRUCID                DS   CL4      Structure identifier
MQWDR_VERSION                DS   F        Structure version number
MQWDR_STRUCLENGTH            DS   F        Length of MQWDR structure
MQWDR_QMGRFLAGS              DS   F        Queue-manager flags
MQWDR_QMGRIDENTIFIER         DS   CL48     Queue-manager identifier
MQWDR_QMGRNAME               DS   CL48     Queue-manager name
MQWDR_CLUSTERRECOFFSET       DS   F        Offset of first cluster
*                                          record
MQWDR_CHANNELSTATE           DS   F        Channel state
MQWDR_CHANNELDEFOFFSET       DS   F        Offset of channel definition
*                                          structure
MQWDR_LENGTH                 EQU  *-MQWDR  Length of structure
                             ORG  MQWDR
MQWDR_AREA                   DS   CL(MQWDR_LENGTH)
```

MQWQR - Cluster workload queue-record structure

The following table summarizes the fields in the structure.

Table 7. Fields in MQWQR

Field	Description	Page
StrucId	Structure identifier	106
Version	Structure version number	106
StrucLength	Length of MQWQR structure	107
QFlags	Queue flags	107
QName	Queue name	107
QMgrIdentifier	Queue-manager identifier	107
ClusterRecOffset	Offset of first cluster record (MQWCR)	107
QType	Queue type	108
QDesc	Queue description	108
DefBind	Default binding	108
DefPersistence	Default message persistence	108
DefPriority	Default message priority	108
InhibitPut	Whether put operations on the queue are allowed	109

The MQWQR structure contains information relating to one of the possible destinations for the message. There is one MQWQR structure for each instance of the destination queue.

| This structure is supported in the following environments: AIX, HP-UX, OS/2,
| OS/390, OS/400, Sun Solaris, and Windows NT.

Fields

StrucId (MQCHAR4)
Structure identifier.

The value is:

MQWQR_STRUC_ID
Identifier for cluster workload queue record.

For the C programming language, the constant MQWQR_STRUC_ID_ARRAY is also defined; this has the same value as MQWQR_STRUC_ID, but is an array of characters instead of a string.

This is an input field to the exit.

Version (MQLONG)
Structure version number.

The value is:

MQWQR_VERSION_1
Version-1 cluster workload queue record.

The following constant specifies the version number of the current version:

MQWQR_CURRENT_VERSION
Current version of cluster workload queue record.

This is an input field to the exit.

StrucLength (MQLONG)
Length of MQWQR structure.

The value is:

MQWQR_LENGTH_1
Length of version-1 cluster workload queue record.

The following constant specifies the length of the current version:

MQWQR_CURRENT_LENGTH
Length of current version of cluster workload queue record.

This is an input field to the exit.

QFlags (MQLONG)
Queue flags.

These are bit flags that indicate various properties of the queue. The following flag is defined:

MQQF_LOCAL_Q
Destination is a local queue.

Note: Other flags in the field may be set by the queue manager for internal purposes.

This is an input field to the exit.

QName (MQCHAR48)
Queue name.

The length of this field is given by MQ_Q_NAME_LENGTH. This is an input field to the exit.

QMgrIdentifier (MQCHAR48)
Queue-manager identifier.

This is a string that acts as a unique identifier for the queue manager that hosts the instance of the queue described by this MQWQR structure. The identifier is generated by the queue manager.

The length of this field is given by MQ_Q_MGR_IDENTIFIER_LENGTH. This is an input field to the exit.

ClusterRecOffset (MQLONG)
Offset of first cluster record.

This is the offset of the first MQWCR structure that belongs to this MQWQR structure. The offset is measured in bytes from the start of the MQWQR structure.

This is an input field to the exit.

QType (MQLONG)
Queue type.

The following values are possible:

MQCQT_LOCAL_Q
Local queue.

MQCQT_ALIAS_Q
Alias queue.

MQCQT_REMOTE_Q
Remote queue.

MQCQT_Q_MGR_ALIAS
Queue-manager alias.

This is an input field to the exit.

QDesc (MQCHAR64)
Queue description.

This is the value of the *QDesc* queue attribute as defined on the queue manager that hosts the instance of the destination queue described by this MQWQR structure.

The length of this field is given by MQ_Q_DESC_LENGTH. This is an input field to the exit.

DefBind (MQLONG)
Default binding.

This is the value of the *DefBind* queue attribute as defined on the queue manager that hosts the instance of the destination queue described by this MQWQR structure. The following values are possible:

MQBND_BIND_ON_OPEN
Binding fixed by MQOPEN call.

MQBND_BIND_NOT_FIXED
Binding not fixed.

This is an input field to the exit.

DefPersistence (MQLONG)
Default message persistence.

This is the value of the *DefPersistence* queue attribute as defined on the queue manager that hosts the instance of the destination queue described by this MQWQR structure. The following values are possible:

MQPER_PERSISTENT
Message is persistent.

MQPER_NOT_PERSISTENT
Message is not persistent.

This is an input field to the exit.

DefPriority (MQLONG)
Default message priority.

This is the value of the *DefPriority* queue attribute as defined on the queue manager that hosts the instance of the destination queue described by this MQWQR structure. Priorities are in the range zero (lowest) through

MaxPriority (highest), where *MaxPriority* is the queue-manager attribute of the queue manager that hosts this instance of the destination queue.

This is an input field to the exit.

InhibitPut (MQLONG)
Whether put operations on the queue are allowed.

This is the value of the *InhibitPut* queue attribute as defined on the queue manager that hosts the instance of the destination queue described by this MQWQR structure. The following values are possible:

MQQA_PUT_INHIBITED
Put operations are inhibited.

MQQA_PUT_ALLOWED
Put operations are allowed.

This is an input field to the exit.

C declaration

```
typedef struct tagMQWQR {
  MQCHAR4   StrucId;          /* Structure identifier */
  MQLONG    Version;          /* Structure version number */
  MQLONG    StrucLength;      /* Length of MQWQR structure */
  MQLONG    QFlags;           /* Queue flags */
  MQCHAR48  QName;            /* Queue name */
  MQCHAR48  QMgrIdentifier;   /* Queue-manager identifier */
  MQLONG    ClusterRecOffset; /* Offset of first cluster record */
  MQLONG    QType;            /* Queue type */
  MQCHAR64  QDesc;            /* Queue description */
  MQLONG    DefBind;          /* Default binding */
  MQLONG    DefPersistence;   /* Default message persistence */
  MQLONG    DefPriority;      /* Default message priority */
  MQLONG    InhibitPut;       /* Whether put operations on the queue
                                 are allowed */
 } MQWQR;
```

System/390 assembler declaration

```
MQWQR                         DSECT
MQWQR_STRUCID                 DS   CL4      Structure identifier
MQWQR_VERSION                 DS   F        Structure version number
MQWQR_STRUCLENGTH             DS   F        Length of MQWQR structure
MQWQR_QFLAGS                  DS   F        Queue flags
MQWQR_QNAME                   DS   CL48     Queue name
MQWQR_QMGRIDENTIFIER          DS   CL48     Queue-manager identifier
MQWQR_CLUSTERRECOFFSET        DS   F        Offset of first cluster
*                                           record
MQWQR_QTYPE                   DS   F        Queue type
MQWQR_QDESC                   DS   CL64     Queue description
MQWQR_DEFBIND                 DS   F        Default binding
MQWQR_DEFPERSISTENCE          DS   F        Default message persistence
MQWQR_DEFPRIORITY             DS   F        Default message priority
MQWQR_INHIBITPUT              DS   F        Whether put operations on
*                                           the queue are allowed
MQWQR_LENGTH                  EQU  *-MQWQR  Length of structure
                              ORG  MQWQR
MQWQR_AREA                    DS   CL(MQWQR_LENGTH)
```

MQWCR - Cluster workload cluster-record structure

The following table summarizes the fields in the structure.

Table 8. Fields in MQWCR

Field	Description	Page
ClusterName	Name of cluster	110
ClusterRecOffset	Offset of next cluster record (MQWCR)	110
ClusterFlags	Cluster flags	110

The MQWCR structure contains information relating to a cluster to which an instance of the destination queue belongs. There is one MQWCR for each such cluster.

| This structure is supported in the following environments: AIX, HP-UX, OS/2,
| OS/390, OS/400, Sun Solaris, and Windows NT.

Fields

ClusterName (MQCHAR48)
Cluster name.

This is the name of a cluster to which the instance of the destination queue that owns this MQWCR structure belongs. The destination queue instance is described by an MQWDR structure.

The length of this field is given by MQ_CLUSTER_NAME_LENGTH. This is an input field to the exit.

ClusterRecOffset (MQLONG)
Offset of next cluster record.

This is the offset of the next MQWCR structure. The offset is measured in bytes from the start of the current MQWCR structure. If there are no more MQWCR structures, *ClusterRecOffset* is zero.

This is an input field to the exit.

ClusterFlags (MQLONG)
Cluster flags.

These are bit flags that indicate various properties of the queue manager identified by this MQWCR structure. The following flags are defined:

MQQMF_REPOSITORY_Q_MGR
Destination is a repository queue manager.

MQQMF_CLUSSDR_USER_DEFINED
Cluster sender channel was defined manually.

MQQMF_CLUSSDR_AUTO_DEFINED
Cluster sender channel was defined automatically.

MQQMF_AVAILABLE
Destination queue manager is available to receive messages.

Note: Other flags in the field may be set by the queue manager for internal purposes.

This is an input field to the exit.

C declaration

```
typedef struct tagMQWCR {
  MQCHAR48  ClusterName;       /* Cluster name */
  MQLONG    ClusterRecOffset;  /* Offset of next cluster record */
  MQLONG    ClusterFlags;      /* Cluster flags */
 } MQWCR;
```

System/390 assembler declaration

```
MQWCR                          DSECT
MQWCR_CLUSTERNAME              DS   CL48     Cluster name
MQWCR_CLUSTERRECOFFSET         DS   F        Offset of next cluster
*                                            record
MQWCR_CLUSTERFLAGS             DS   F        Cluster flags
MQWCR_LENGTH                   EQU  *-MQWCR  Length of structure
                               ORG  MQWCR
MQWCR_AREA                     DS   CL(MQWCR_LENGTH)
```

Chapter 11. Constants for the cluster workload exit

This chapter specifies the values of the named constants that apply to the cluster workload exit. This information is general-use programming interface information.

The constants are grouped according to the parameter or field to which they relate. All of the names of the constants in a group begin with a common prefix of the form "MQ*xxxx*_", where *xxxx* represents a string of 0 through 4 characters that indicates the parameter or field to which the values relate. The constants are ordered alphabetically by this prefix.

Notes:

1. For constants with numeric values, the values are shown in both decimal and hexadecimal forms.
2. Hexadecimal values are represented using the notation `X'hhhh'`, where each "h" denotes a single hexadecimal digit.
3. Character values are shown delimited by single quotation marks; the quotation marks are not part of the value.
4. Blanks in character values are represented by one or more occurrences of the symbol "b".

List of constants

The following sections list all of the named constants mentioned in this book, and show their values.

MQ_* (Lengths of character string and byte fields)

MQ_CLUSTER_NAME_LENGTH	48	X'00000030'
MQ_EXIT_DATA_LENGTH	32	X'00000020'
MQ_EXIT_USER_AREA_LENGTH	16	X'00000010'
MQ_Q_DESC_LENGTH	64	X'00000040'
MQ_Q_MGR_IDENTIFIER_LENGTH	48	X'00000030'
MQ_Q_MGR_NAME_LENGTH	48	X'00000030'
MQ_Q_NAME_LENGTH	48	X'00000030'

|
MQBND_* (Binding)

| See the *DefBind* field described in "MQWQR - Cluster workload queue-record
| structure" on page 106.

MQBND_BIND_ON_OPEN	0	X'00000000'
MQBND_BIND_NOT_FIXED	1	X'00000001'

|

|
MQCHS_* (Channel status)

See the *ChannelState* field described in "MQWDR - Cluster workload destination-record structure" on page 102.

MQCHS_INACTIVE	0	X'00000000'
MQCHS_BINDING	1	X'00000001'
MQCHS_STARTING	2	X'00000002'

MQCHS_RUNNING	3	X'00000003'
MQCHS_STOPPING	4	X'00000004'
MQCHS_RETRYING	5	X'00000005'
MQCHS_STOPPED	6	X'00000006'
MQCHS_REQUESTING	7	X'00000007'
MQCHS_PAUSED	8	X'00000008'
MQCHS_INITIALIZING	13	X'0000000D'

MQCQT_* (Cluster queue type)

See the *QType* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQCQT_LOCAL_Q	1	X'00000001'
MQCQT_ALIAS_Q	2	X'00000002'
MQCQT_REMOTE_Q	3	X'00000003'
MQCQT_Q_MGR_ALIAS	4	X'00000004'

MQPER_* (Persistence)

See the *DefPersistence* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQPER_NOT_PERSISTENT	0	X'00000000'
MQPER_PERSISTENT	1	X'00000001'

MQQA_* (Inhibit put)

See the *InhibitPut* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQQA_PUT_ALLOWED	0	X'00000000'
MQQA_PUT_INHIBITED	1	X'00000001'

MQQF_* (Queue flags)

See the *QFlags* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQQF_LOCAL_Q	1	X'00000001'

MQQMF_* (Queue-manager flags)

See the *QMgrFlags* field described in "MQWDR - Cluster workload destination-record structure" on page 102.

MQQMF_REPOSITORY_Q_MGR	2	X'00000002'
MQQMF_CLUSSDR_USER_DEFINED	8	X'00000008'
MQQMF_CLUSSDR_AUTO_DEFINED	16	X'00000010'
MQQMF_AVAILABLE	32	X'00000020'

MQWDR_* (Cluster workload exit destination-record length)

See the *StrucLength* field described in "MQWDR - Cluster workload destination-record structure" on page 102.

MQWDR_LENGTH_1	124	X'0000007C'
MQWDR_CURRENT_LENGTH	124	X'0000007C'

MQWDR_* (Cluster workload exit destination-record structure identifier)

See the *StrucId* field described in "MQWDR - Cluster workload destination-record structure" on page 102.

MQWDR_STRUC_ID	'WDRb'

For the C programming language, the following array version is also defined:

MQWDR_STRUC_ID_ARRAY	'W','D','R','b'

MQWDR_* (Cluster workload exit destination-record version)

See the *Version* field described in "MQWDR - Cluster workload destination-record structure" on page 102.

MQWDR_VERSION_1	1	X'00000001'
MQWDR_CURRENT_VERSION	1	X'00000001'

MQWQR_* (Cluster workload exit queue-record length)

See the *StrucLength* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQWQR_LENGTH_1	200	X'000000C8'
MQWQR_CURRENT_LENGTH	200	X'000000C8'

MQWQR_* (Cluster workload exit queue-record structure identifier)

See the *StrucId* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQWQR_STRUC_ID	'WQRb'

For the C programming language, the following array version is also defined:

MQWQR_STRUC_ID_ARRAY	'W','Q','R','b'

MQWQR_* (Cluster workload exit queue-record version)

See the *Version* field described in "MQWQR - Cluster workload queue-record structure" on page 106.

MQWQR_VERSION_1	1	X'00000001'
MQWQR_CURRENT_VERSION	1	X'00000001'

MQWXP_* (Cluster workload exit structure identifier)

See the *StrucId* field described in "MQWXP - Cluster workload exit parameter structure" on page 95.

MQWXP_STRUC_ID	'WXPb'

For the C programming language, the following array version is also defined:

MQWXP_STRUC_ID_ARRAY	'W','X','P','b'

MQWXP_* (Cluster workload exit version)

See the *Version* field described in "MQWXP - Cluster workload exit parameter structure" on page 95.

MQWXP_VERSION_1	1	X'00000001'
MQWXP_CURRENT_VERSION	1	X'00000001'

MQXCC_* (Exit response)

See the *ExitResponse* field described in "MQWXP - Cluster workload exit parameter structure" on page 95.

MQXR_* (Exit reason)

See the *ExitReason* field described in "MQWXP - Cluster workload exit parameter structure" on page 95.

MQXT_* (Exit identifier)

See the *ExitId* field described in "MQWXP - Cluster workload exit parameter structure" on page 95.

MQXUA_* (Exit user area)

See the *ExitUserArea* field described in "MQWXP - Cluster workload exit parameter structure" on page 95.

MQXUA_NONE	X'00...00' (16 nulls)

For the C programming language, the following array version is also defined:

MQXUA_NONE_ARRAY	'\0','\0',...'\0','\0'

Part 4. Appendixes

Appendix. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Programming interface information

This book is intended to help you understand and control clusters of MQSeries queue managers.

This book also documents General-use Programming Interface and Associated Guidance Information provided by:

MQSeries for AIX V5.1
MQSeries for AS/400 V5.1
MQSeries for HP-UX V5.1
MQSeries for OS/2 Warp V5.1
MQSeries for OS/390 V2.1
MQSeries for Sun Solaris V5.1
MQSeries for Windows NT V5.1

General-use programming interfaces allow the customer to write programs that obtain the services of these products.

General-use Programming Interface and Associated Guidance Information is identified where it occurs, by an introductory statement to a chapter or section.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	AS/400	BookManager
CICS	DB2	IBM
IBMLink	IMS	MQSeries
OS/2	OS/390	OS/400
RACF	SP2	System/390
VSE/ESA	400	

Lotus and LotusScript are trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names may be trademarks or service marks of others.

Glossary of terms and abbreviations

This glossary defines MQSeries terms and abbreviations used in this book. If you do not find the term you are looking for, see the Index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 11 West 42 Street, New York, New York 10036. Definitions are identified by the symbol (A) after the definition.

A

administrator commands. MQSeries commands used to manage MQSeries objects, such as queues, processes, and namelists.

Advanced Program-to-Program Communication (APPC). The general facility characterizing the LU 6.2 architecture and its various implementations in products.

affinity. An association between objects that have some relationship or dependency upon each other.

alias queue object. An MQSeries object, the name of which is an alias for a base queue defined to the local queue manager. When an application or a queue manager uses an alias queue, the alias name is resolved and the requested operation is performed on the associated base queue.

alternate user security. A security feature in which the authority of one user ID can be used by another user ID; for example, to open an MQSeries object.

APPC. Advanced Program-to-Program Communication.

application log. In Windows NT, a log that records significant application events.

application queue. A queue used by an application.

asynchronous messaging. A method of communication between programs in which programs place messages on message queues. With asynchronous messaging, the sending program proceeds with its own processing without waiting for a reply to its message. Contrast with *synchronous messaging*.

attribute. One of a set of properties that defines the characteristics of an MQSeries object.

authorization checks. Security checks that are performed when a user tries to issue administration commands against an object, for example to open a queue or connect to a queue manager.

authorization file. In MQSeries on UNIX systems, a file that provides security definitions for an object, a class of objects, or all classes of objects.

B

browse. In message queuing, to use the MQGET call to copy a message without removing it from the queue. See also *get*.

C

channel. See *message channel*.

channel exit program. A user-written program that can be entered from one of a defined number of places during channel operation.

channel initiator. A component of MQSeries distributed queuing, which monitors the initiation queue to see when triggering criteria have been met and then starts the sender channel.

channel listener. A component of MQSeries distributed queuing, which monitors the network for a startup request and then starts the receiving channel.

CL. Control Language.

CLUSRCVR. Cluster-receiver channel definition.

CLUSSDR. Cluster-sender channel definition.

cluster. A network of queue managers that are logically associated in some way.

cluster queue. A queue that is hosted by a cluster queue manager and made available to other queue managers in the cluster.

cluster queue manager. A queue manager that is a member of a cluster. A queue manager may be a member of more than one cluster.

cluster-receiver channel (CLUSRCVR). A channel on which a cluster queue manager can receive messages from other queue managers in the cluster and cluster information from the repository queue managers.

© Copyright IBM Corp. 1999, 2000

cluster-sender channel (CLUSSDR). A channel on which a cluster queue manager can send messages to other queue managers in the cluster and cluster information to the repository queue managers.

cluster transmission queue. A transmission queue that transmits all messages from a queue manager to any other queue manager that is in the same cluster. The queue is called SYSTEM.CLUSTER.TRANSMIT.QUEUE.

command. In MQSeries, an administration instruction that can be carried out by the queue manager.

completion code. A return code indicating how an MQI call has ended.

connection handle. The identifier or token by which a program accesses the queue manager to which it is connected.

context security. In MQSeries, a method of allowing security to be handled such that messages are obliged to carry details of their origins in the message descriptor.

control command. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a command that can be entered interactively from the operating system command line. Such a command requires only that the MQSeries product be installed; it does not require a special utility or program to run it.

Control Language (CL). In MQSeries for AS/400, a language that can be used to issue commands, either at the command line or by writing a CL program.

D

dead-letter queue (DLQ). A queue to which a queue manager or application sends messages that it cannot deliver to their correct destination.

default object. A definition of an object (for example, a queue) with all attributes defined. If a user defines an object but does not specify all possible attributes for that object, the queue manager uses default attributes in place of any that were not specified.

distributed application. In message queuing, a set of application programs that can each be connected to a different queue manager, but that collectively constitute a single application.

distributed queue management (DQM). In message queuing, the setup and control of message channels to queue managers on other systems.

distribution list. A list of queues to which a message can be put using a single MQPUT or MQPUT1 statement.

DLQ. Dead-letter queue.

DQM. Distributed queue management.

E

ESM. External security manager.

external security manager (ESM). A security product that is invoked by the OS/390 System Authorization Facility. RACF is an example of an ESM.

F

full repository. A *complete* set of information about every queue manager in a cluster. This set of information is called the repository or sometimes the full repository and is held usually by two of the queue managers in the cluster. Contrast with *partial repository*.

G

get. In message queuing, to use the MQGET call to remove a message from a queue. See also *browse*.

H

handle. See *connection handle* and *object handle*.

I

in-doubt unit of recovery. In MQSeries, the status of a unit of recovery for which a syncpoint has been requested but not yet confirmed.

Internet Protocol (IP). A protocol used to route data from its source to its destination in an Internet environment. This is the base layer, on which other protocol layers, such as TCP and UDP are built.

IP. Internet Protocol.

L

listener. In MQSeries distributed queuing, a program that monitors for incoming network connections.

local definition. An MQSeries object belonging to a local queue manager.

local definition of a remote queue. An MQSeries object belonging to a local queue manager. This object defines the attributes of a queue that is owned by another queue manager. In addition, it is used for queue-manager aliasing and reply-to-queue aliasing.

local queue. A queue that belongs to the local queue manager. A local queue can contain a list of messages waiting to be processed. Contrast with *remote queue*.

local queue manager. The queue manager to which a program is connected and that provides message queuing services to the program. Queue managers to which a program is not connected are called *remote queue managers*, even if they are running on the same system as the program.

log. In MQSeries, a file recording the work done by queue managers while they receive, transmit, and deliver messages, to enable them to recover in the event of failure.

log file. In MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT, a file in which all significant changes to the data controlled by a queue manager are recorded. If the primary log files become full, MQSeries allocates secondary log files.

logical unit of work (LUW). See *unit of work*.

LU 6.2. A type of logical unit (LU) that supports general communication between programs in a distributed processing environment.

M

MCA. Message channel agent.

message. In message queuing applications, a communication sent between programs. In system programming, information intended for the terminal operator or system administrator.

message channel. In distributed message queuing, a mechanism for moving messages from one queue manager to another. A message channel comprises two message channel agents (a sender at one end and a receiver at the other end) and a communication link. Contrast with *MQI channel*.

message channel agent (MCA). A program that transmits prepared messages from a transmission queue to a communication link, or from a communication link to a destination queue. See also *message queue interface*.

message queue. Synonym for *queue*.

message queue interface (MQI). The programming interface provided by the MQSeries queue managers. This programming interface allows application programs to access message queuing services.

message queuing. A programming technique in which each program within an application communicates with the other programs by putting messages on queues.

message-retry. An option available to an MCA that is unable to deliver a message. The MCA can wait for a predefined amount of time and then try to send the message again.

messaging. See *synchronous messaging* and *asynchronous messaging*.

MQI. Message queue interface.

MQI channel. Connects an MQSeries client to a queue manager on a server system, and transfers only MQI calls and responses in a bidirectional manner. Contrast with *message channel*.

MQSC. MQSeries commands.

MQSeries. A family of IBM licensed programs that provides message queuing services.

MQSeries commands (MQSC). Human readable commands, uniform across all platforms, that are used to manipulate MQSeries objects. Contrast with *programmable command format (PCF)*.

multi-hop. To pass through one or more intermediate queue managers when there is no direct communication link between a source queue manager and the target queue manager.

N

namelist. An MQSeries object that contains a list of names, for example, queue names.

NetBIOS. Network Basic Input/Output System. An operating system interface for application programs used on IBM personal computers that are attached to the IBM Token-Ring Network.

null character. The character that is represented by X'00'.

O

object. In MQSeries, an object is a queue manager, a queue, a process definition, a channel, a namelist, or a storage class (OS/390 only).

object handle. The identifier or token by which a program accesses the MQSeries object with which it is working.

P

partial repository. A *partial* set of information about queue managers in a cluster. A partial repository is maintained by all cluster queue managers that do not host a *full repository*.

PCF. Programmable command format.

PCF command. See *programmable command format*.

platform. In MQSeries, the operating system under which a queue manager is running.

process definition object. An MQSeries object that contains the definition of an MQSeries application. For example, a queue manager uses the definition when it works with trigger messages.

programmable command format (PCF). A type of MQSeries message used by:

- User administration applications, to put PCF commands onto the system command input queue of a specified queue manager
- User administration applications, to get the results of a PCF command from a specified queue manager
- A queue manager, as a notification that an event has occurred

Contrast with *MQSC*.

Q

queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages—they point to other queues, or can be used as models for dynamic queues.

queue manager. A system program that provides queuing services to applications. It provides an application programming interface so that programs can access messages on the queues that the queue manager owns. See also *local queue manager* and *remote queue manager*. An MQSeries object that defines the attributes of a particular queue manager.

queuing. See *message queuing*.

R

reason code. A return code that describes the reason for the failure or partial success of an MQI call.

receiver channel. In message queuing, a channel that responds to a sender channel, takes messages from a communication link, and puts them on a local queue.

remote queue. A queue belonging to a remote queue manager. Programs can put messages on remote queues, but they cannot get messages from remote queues. Contrast with *local queue*.

remote queue manager. To a program, a queue manager that is not the one to which the program is connected.

remote queue object. See *local definition of a remote queue*.

remote queuing. In message queuing, the provision of services to enable applications to put messages on queues belonging to other queue managers.

reply message. A type of message used for replies to request messages. Contrast with *request message* and *report message*.

reply-to queue. The name of a queue to which the program that issued an MQPUT call wants a reply message or report message sent.

report message. A type of message that gives information about another message. A report message can indicate that a message has been delivered, has arrived at its destination, has expired, or could not be processed for some reason. Contrast with *reply message* and *request message*.

repository. A collection of information about the queue managers that are members of a cluster. This information includes queue manager names, their locations, their channels, what queues they host, and so on. See also *full repository* and *partial repository*.

repository queue manager. A queue manager that hosts the *full repository* of information about a cluster.

requester channel. In message queuing, a channel that may be started remotely by a sender channel. The requester channel accepts messages from the sender channel over a communication link and puts the messages on the local queue designated in the message. See also *server channel*.

request message. A type of message used to request a reply from another program. Contrast with *reply message* and *report message*.

resource. Any facility of the computing system or operating system required by a job or task. In MQSeries for OS/390, examples of resources are buffer pools, page sets, log data sets, queues, and messages.

resource manager. An application, program, or transaction that manages and controls access to shared resources such as memory buffers and data sets. MQSeries, CICS, and IMS™ are resource managers.

return codes. The collective name for completion codes and reason codes.

return-to-sender. An option available to an MCA that is unable to deliver a message. The MCA can send the message back to the originator.

S

Scalable Parallel 2 (SP2). IBM's parallel UNIX system — effectively parallel AIX systems on a high-speed network.

sender channel. In message queuing, a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel.

sequential delivery. In MQSeries, a method of transmitting messages with a sequence number so that the receiving channel can reestablish the message sequence when storing the messages. This is required where messages must be delivered only once, and in the correct order.

server channel. In message queuing, a channel that responds to a requester channel, removes messages from a transmission queue, and moves them over a communication link to the requester channel.

server connection channel type. The type of MQI channel definition associated with the server that runs a queue manager. See also *client connection channel type*.

SNA. Systems Network Architecture.

source queue manager. See *local queue manager*.

SPX. Sequenced Packet Exchange transmission protocol.

SP2. Scalable Parallel 2

store and forward. The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

synchronous messaging. A method of communication between programs in which programs place messages on message queues. With synchronous messaging, the sending program waits for a reply to its message before resuming its own processing. Contrast with *asynchronous messaging*.

sysplex. A multiple OS/390-system environment that allows multiple-console support (MCS) consoles to receive console messages and send operator commands across systems.

system control commands. Commands used to manipulate platform-specific entities such as buffer pools, storage classes, and page sets.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

T

target queue manager. See *remote queue manager*.

TCP. Transmission Control Protocol.

TCP/IP. Transmission Control Protocol/Internet Protocol.

trace. In MQSeries, a facility for recording MQSeries activity. The destinations for trace entries can include GTF and the system management facility (SMF).

Transmission Control Protocol (TCP). Part of the TCP/IP protocol suite. A host-to-host protocol between hosts in packet-switched communications networks. TCP provides connection-oriented data stream delivery. Delivery is reliable and orderly.

Transmission Control Protocol/Internet Protocol (TCP/IP). A suite of communication protocols that support peer-to-peer connectivity functions for both local and wide area networks.

transmission queue. A local queue on which prepared messages destined for a remote queue manager are temporarily stored.

U

UDP. User Datagram Protocol.

undelivered-message queue. See *dead-letter queue*.

unit of work. A recoverable sequence of operations performed by an application between two points of consistency. A unit of work begins when a transaction starts or after a user-requested syncpoint. It ends either at a user-requested syncpoint or at the end of a transaction.

User Datagram Protocol (UDP). Part of the TCP/IP protocol suite. A packet-level protocol built directly on the Internet Protocol layer. UDP is a connectionless and less reliable alternative to TCP. It is used for application-to-application programs between TCP/IP host systems.

utility. In MQSeries, a supplied set of programs that provide the system operator or system administrator with facilities in addition to those provided by the MQSeries commands. Some utilities invoke more than one function.

Bibliography

This section describes the documentation available for all current MQSeries products.

MQSeries cross-platform publications

Most of these publications, which are sometimes referred to as the MQSeries "family" books, apply to all MQSeries Level 2 products. The latest MQSeries Level 2 products are:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for AT&T GIS UNIX V2.2
- MQSeries for Compaq (DIGITAL) OpenVMS V2.2.1.1
- MQSeries for DIGITAL UNIX (Compaq Tru64 UNIX) V2.2.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for SINIX and DC/OSx V2.2
- MQSeries for Sun Solaris V5.1
- MQSeries for Tandem NonStop Kernel V2.2.0.1
- MQSeries for VSE/ESA V2.1
- MQSeries for Windows® V2.0
- MQSeries for Windows V2.1
- MQSeries for Windows NT V5.1

Any exceptions to this general rule are indicated.

MQSeries Brochure
The *MQSeries Brochure*, G511-1908, gives a brief introduction to the benefits of MQSeries. It is intended to support the purchasing decision, and describes some authentic customer use of MQSeries.

MQSeries: An Introduction to Messaging and Queuing
An Introduction to Messaging and Queuing, GC33-0805, describes briefly what MQSeries is, how it works, and how it can solve some classic interoperability problems. This book is intended for a more technical audience than the *MQSeries Brochure*.

MQSeries Planning Guide
The *MQSeries Planning Guide*, GC33-1349, describes some key MQSeries concepts, identifies items that need to be considered before MQSeries is installed, including storage requirements, backup and recovery, security, and migration from earlier releases, and specifies hardware and software requirements for every MQSeries platform.

MQSeries Intercommunication
The *MQSeries Intercommunication* book, SC33-1872, defines the concepts of distributed queuing and explains how to set up a distributed queuing network in a variety of MQSeries environments. In particular, it demonstrates how to (1) configure communications to and from a representative sample of MQSeries products, (2) create required MQSeries objects, and (3) create and configure MQSeries channels. The use of channel exits is also described.

MQSeries Queue Manager Clusters
MQSeries Queue Manager Clusters, SC34-5349, describes MQSeries clustering. It explains the concepts and terminology and shows how you can benefit by taking advantage of clustering. It details changes to the MQI, and summarizes the syntax of new and changed MQSeries commands. It shows a number of examples of tasks you can perform to set up and maintain clusters of queue managers.

This book applies to the following MQSeries products only:

- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Clients
The *MQSeries Clients* book, GC33-1632, describes how to install, configure, use, and manage MQSeries client systems.

MQSeries System Administration
The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem

determination, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

This book applies to the following MQSeries products only:
- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Command Reference

The *MQSeries Command Reference*, SC33-1369, contains the syntax of the MQSC commands, which are used by MQSeries system operators and administrators to manage MQSeries objects.

MQSeries Programmable System Management

The *MQSeries Programmable System Management* book, SC33-1482, provides both reference and guidance information for users of MQSeries events, Programmable Command Format (PCF) messages, and installable services.

MQSeries Administration Interface Programming Guide and Reference

The *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390, provides information for users of the MQAI. The MQAI is a programming interface that simplifies the way in which applications manipulate Programmable Command Format (PCF) messages and their associated data structures.

This book applies to the following MQSeries products only:
- MQSeries for AIX V5.1
- MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries Messages

The *MQSeries Messages* book, GC33-1876, which describes "AMQ" messages issued by MQSeries, applies to these MQSeries products only:
- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries for Windows V2.0
- MQSeries for Windows V2.1

This book is available in softcopy only.

For other MQSeries platforms, the messages are supplied with the system. They do not appear in softcopy manual form.

MQSeries Application Programming Guide

The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference

The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Application Programming Reference Summary

The *MQSeries Application Programming Reference Summary*, SX33-6095, summarizes the information in the *MQSeries Application Programming Reference* manual.

MQSeries Using C++

MQSeries Using C++, SC33-1877, provides both guidance and reference information for users of the MQSeries C++ programming-language binding to the MQI. MQSeries C++ is supported by these MQSeries products:
- MQSeries for AIX V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for AS/400 V5.1
- MQSeries for OS/390 V2.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

MQSeries C++ is also supported by MQSeries clients supplied with these products and installed in the following environments:
- AIX
- HP-UX

- OS/2
- Sun Solaris
- Windows NT
- Windows 3.1
- Windows 95 and Windows 98

MQSeries Using Java™

MQSeries Using Java, SC34-5456, provides both guidance and reference information for users of the MQSeries Bindings for Java and the MQSeries Client for Java. MQSeries classes for Java are supported by these MQSeries products:
- MQSeries for AIX V5.1
- | MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- | MQSeries for MVS/ESA V1.2
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1

| This book is available in softcopy only.

MQSeries platform-specific publications

Each MQSeries product is documented in at least one platform-specific publication, in addition to the MQSeries family books.

MQSeries for AIX

MQSeries for AIX V5.1 Quick Beginnings, GC33-1867

MQSeries for AS/400

| *MQSeries for AS/400 V5.1 Quick*
| *Beginnings*, GC34-5557

| *MQSeries for AS/400 V5.1 System*
| *Administration*, SC34-5558

| *MQSeries for AS/400 V5.1 Application*
| *Programming Reference (ILE RPG)*,
| SC34-5559

MQSeries for AT&T GIS UNIX

MQSeries for AT&T GIS UNIX System Management Guide, SC33-1642

| **MQSeries for Compaq (DIGITAL) OpenVMS**

| *MQSeries for Digital OpenVMS System*
| *Management Guide*, GC33-1791

| **MQSeries for Digital UNIX (Compaq Tru64**
| **UNIX)**

| *MQSeries for Digital UNIX System*
| *Management Guide*, GC34-5483

| **MQSeries for HP-UX**

MQSeries for HP-UX V5.1 Quick Beginnings, GC33-1869

MQSeries for OS/2 Warp

MQSeries for OS/2 Warp V5.1 Quick Beginnings, GC33-1868

MQSeries for OS/390

MQSeries for OS/390 Version 2 Release 1 Licensed Program Specifications, GC34-5377

MQSeries for OS/390 Version 2 Release 1 Program Directory

MQSeries for OS/390 System Management Guide, SC34-5374

MQSeries for OS/390 Messages and Codes, GC34-5375

MQSeries for OS/390 Problem Determination Guide, GC34-5376

MQSeries link for R/3

MQSeries link for R/3 Version 1.2 User's Guide, GC33-1934

MQSeries for SINIX and DC/OSx

MQSeries for SINIX and DC/OSx System Management Guide, GC33-1768

MQSeries for Sun Solaris

MQSeries for Sun Solaris V5.1 Quick Beginnings, GC33-1870

MQSeries for Tandem NonStop Kernel

| *MQSeries for Tandem NonStop Kernel*
| *System Management Guide*, GC33-1893

MQSeries for VSE/ESA™

MQSeries for VSE/ESA Version 2 Release 1 Licensed Program Specifications, GC34-5365

MQSeries for VSE/ESA System Management Guide, GC34-5364

MQSeries for Windows

MQSeries for Windows V2.0 User's Guide, GC33-1822

MQSeries for Windows V2.1 User's Guide, GC33-1965

MQSeries for Windows NT

MQSeries for Windows NT V5.1 Quick Beginnings, GC34-5389

MQSeries for Windows NT Using the Component Object Model Interface, SC34-5387

MQSeries LotusScript Extension, SC34-5404

Softcopy books

Most of the MQSeries books are supplied in both hardcopy and softcopy formats.

BookManager® format

The MQSeries library is supplied in IBM BookManager format on a variety of online library collection kits, including the *Transaction Processing and Data* collection kit, SK2T-0730. You can view the softcopy books in IBM BookManager format using the following IBM licensed programs:

BookManager READ/2
BookManager READ/6000
BookManager READ/DOS
BookManager READ/MVS
BookManager READ/VM
BookManager READ for Windows

HTML format

Relevant MQSeries documentation is provided in HTML format with these MQSeries products:

- MQSeries for AIX V5.1
- | MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1 (compiled HTML)
- MQSeries link for R/3 V1.2

The MQSeries books are also available in HTML format from the MQSeries product family Web site at:

```
http://www.ibm.com/software/ts/mqseries/
```

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

PDF versions of relevant MQSeries books are supplied with these MQSeries products:

- MQSeries for AIX V5.1
- | MQSeries for AS/400 V5.1
- MQSeries for HP-UX V5.1
- MQSeries for OS/2 Warp V5.1
- MQSeries for Sun Solaris V5.1
- MQSeries for Windows NT V5.1
- MQSeries link for R/3 V1.2

PDF versions of all current MQSeries books are also available from the MQSeries product family Web site at:

```
http://www.ibm.com/software/ts/mqseries/
```

PostScript format

The MQSeries library is provided in PostScript (.PS) format with many MQSeries Version 2 products. Books in PostScript format can be printed on a PostScript printer or viewed with a suitable viewer.

Windows Help format

The *MQSeries for Windows User's Guide* is provided in Windows Help format with MQSeries for Windows Version 2.0 and MQSeries for Windows Version 2.1.

MQSeries information available on the Internet

The MQSeries product family Web site is at:

```
http://www.ibm.com/software/ts/mqseries/
```

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Index

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  Information Development Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–870229
  - From within the U.K., use 01962–870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The topic to which your comment applies
- Your name and address / telephone number / fax number / network ID.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC34-5349-01

Spine information:

MQSeries®

MQSeries Queue Manager Clusters