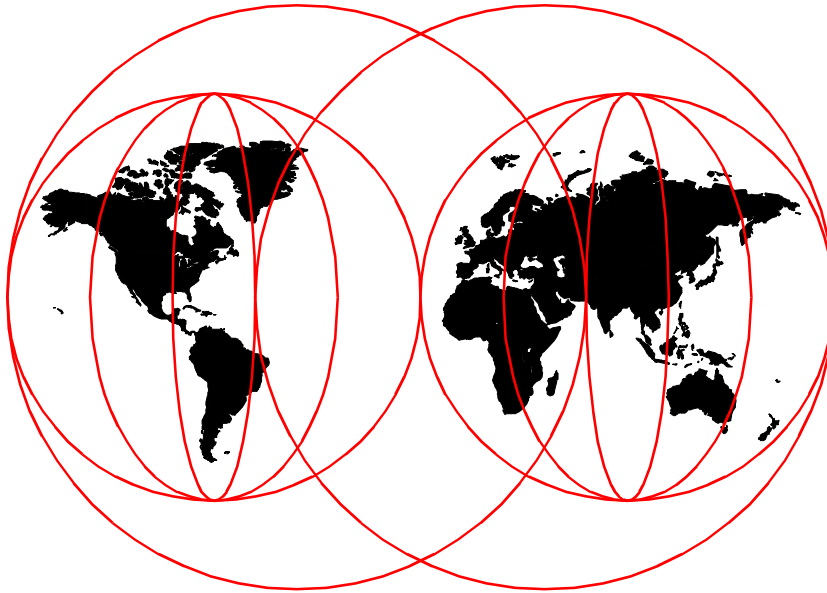


# **MQSeries Version 5.1 Administration and Programming Examples**

*Dieter Wackerow, David Armitage, Tony Skinner*



**International Technical Support Organization**

[www.redbooks.ibm.com](http://www.redbooks.ibm.com)

SG24-5849-00





International Technical Support Organization

**MQSeries Version 5.1**  
**Administration and Programming Examples**

December 1999

**Take Note!**

Before using this information and the product it supports, be sure to read the general information in Appendix G, "Special Notices" on page 237.

**First Edition (December 1999)**

This edition applies to the following products:

- MQSeries for AIX Version 5.1
- MQSeries for OS/2 Warp Version 5.1
- MQSeries for Windows NT Version 5.1

Comments may be addressed to:

IBM Corporation, International Technical Support Organization  
Dept. HZ8 Building 678  
P.O. Box 12195  
Research Triangle Park, NC 27709-2195

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

**© Copyright International Business Machines Corporation 1999. All rights reserved.**

Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Preface</b> .....	vii
The Team That Wrote This Redbook .....	vii
Comments Welcome .....	viii
<b>Chapter 1. About MQSeries Version 5.1</b> .....	1
1.1 The MQSeries Family .....	1
1.2 Platforms .....	2
1.3 What's New in MQSeries Version 5.1 .....	2
1.4 Publish/Subscribe .....	4
1.5 MQSeries and Java .....	4
1.5.1 Example .....	5
1.5.2 Connection Examples .....	5
<b>Chapter 2. About Clusters</b> .....	7
2.1 Why Clusters? .....	8
2.2 Cluster Concept .....	11
2.3 A Routing Example .....	16
2.4 How Clusters Work .....	17
2.4.1 Example with Two Queue Managers .....	20
2.4.2 Example with Three Queue Managers .....	21
2.4.3 Example with Four Queue Managers in Two Networks .....	24
2.5 RUNMQSC Commands for Clusters .....	26
<b>Chapter 3. MQSeries for Windows NT Version 5.1</b> .....	33
3.1 Installation .....	34
3.2 MQSeries First Steps .....	38
3.3 Default Configuration .....	39
3.4 MQSeries Postcard .....	42
3.5 MQSeries Explorer .....	45
3.6 MQSeries Services .....	47
3.7 MQSeries API Exerciser .....	49
<b>Chapter 4. Creating a Cluster with the MQExplorer</b> .....	55
4.1 Creating the Queue Managers .....	57
4.2 Creating a Cluster with Two Repository Queue Managers .....	63
4.3 Joining Queue Managers to a Cluster .....	70
4.4 Working with Local Queues in a Cluster .....	78
4.5 Creating a Shared Cluster Queue .....	83
4.6 Creating a Second Cluster Queue .....	87
4.7 Working with Clusters .....	89
4.7.1 Putting and Getting Messages .....	89

4.7.2	Disassembling the Environment with the MQ Explorer . . . . .	90
4.7.3	Stopping a Cluster . . . . .	90
4.7.4	Showing a Cluster . . . . .	91
4.7.5	Starting a Cluster . . . . .	92
4.7.6	Summary . . . . .	92
<b>Chapter 5. Creating a Cluster with Scripts . . . . .</b>		<b>95</b>
5.1	Some Comments about the Listener. . . . .	104
5.2	Some Comments about Cluster Objects . . . . .	104
<b>Chapter 6. Workload Management . . . . .</b>		<b>107</b>
6.1	Controlling the Workflow . . . . .	108
6.2	A Workload Distribution Example . . . . .	109
6.2.1	Getting Prepared . . . . .	109
6.2.2	Clearing a Cluster Queue . . . . .	110
6.2.3	Putting Using Bind On Open . . . . .	111
6.2.4	Putting Using Bind Not Fixed . . . . .	114
6.2.5	Putting to a Local Cluster Queue . . . . .	115
6.3	Writing a Workload Management Exit. . . . .	116
6.3.1	About the Example . . . . .	116
6.3.2	Commented Program Listing for Exit WLlogger.c . . . . .	118
<b>Chapter 7. MQSeries Administration and Service . . . . .</b>		<b>123</b>
7.1	Experiments with Runmqsc and Clusters . . . . .	125
7.1.1	Creating a Queue Manager . . . . .	126
7.1.2	Starting the Listener . . . . .	127
7.1.3	Starting the Channel Initiator . . . . .	128
7.1.4	Connecting QM_5 to the Existing Cluster . . . . .	129
7.2	Experiments with MQSeries Services . . . . .	132
7.2.1	Automatic or Manual Start-up . . . . .	133
7.2.2	How to Start a Queue Manager Manually . . . . .	134
7.2.3	Working with Queue Manager Properties . . . . .	135
7.2.4	Creating a Queue Manager from the Services GUI . . . . .	137
7.2.5	Adding a Trigger Monitor . . . . .	139
7.2.6	Using the MQSeries Alert Monitor. . . . .	142
7.3	Using MQSeries Control Commands with the New GUIs . . . . .	144
7.4	Remote Administration . . . . .	145
<b>Chapter 8. Web Administration . . . . .</b>		<b>147</b>
8.1	Enabling Web Administration . . . . .	148
8.2	Logging in . . . . .	149
8.3	Getting Help . . . . .	151
8.4	Using Commands . . . . .	152
8.5	Using Scripts . . . . .	154

<b>Chapter 9. Using the Performance Monitor</b> .....	159
9.1 Example 1: Track Cluster Queues .....	160
9.2 Example 2: Check Cluster Behavior .....	164
<b>Chapter 10. File Transfer Programs</b> .....	167
10.1 Design .....	168
10.1.1 putFile .....	168
10.1.2 getFile .....	169
10.2 Input Parameters .....	169
10.3 Message Types .....	171
10.3.1 Header Message .....	171
10.3.2 Data Message .....	171
10.3.3 Trailer Message .....	171
10.3.4 Instruction Message .....	171
10.3.5 Trigger Message .....	172
10.4 mqfm_defs.h .....	173
10.5 putMsg.c .....	174
10.6 putFile.c .....	178
10.7 getFile.c .....	192
<b>Chapter 11. MQSeries Security Changes</b> .....	203
11.1 MQSeries for Windows NT .....	203
11.2 MQSeries Client Identification .....	203
11.3 Long User IDs .....	204
11.4 Authorization Check .....	205
11.5 Security in Clusters .....	208
<b>Chapter 12. Using Dynamic Queues</b> .....	209
12.1 Temporary Dynamic Queues .....	209
12.1.1 Creating a Temporary Dynamic Queue .....	209
12.1.2 Writing to a Temporary Dynamic Queue .....	211
12.1.3 Getting from a Temporary Dynamic Queue .....	212
12.2 Report Messages .....	213
12.3 Permanent Dynamic Queues .....	213

<b>Appendix A. Sample Configuration Output</b> .....	215
<b>Appendix B. Log File Created by crt_str_all</b> .....	217
<b>Appendix C. Source Code for clusput.c</b> .....	221
<b>Appendix D. Source Code for fastget.c</b> .....	227
<b>Appendix E. MQSeries Processes</b> .....	233
<b>Appendix F. Diskette Contents</b> .....	235
<b>Appendix G. Special Notices</b> .....	237
<b>Appendix H. Related Publications</b> .....	241
H.1 International Technical Support Organization Publications .....	241
H.2 Redbooks on CD-ROMs .....	241
H.3 Other Publications .....	242
<b>How to Get IBM Redbooks</b> .....	243
IBM Redbooks Fax Order Form .....	244
<b>List of Abbreviations</b> .....	245
<b>Index</b> .....	247
<b>IBM Redbooks Evaluation</b> .....	255



---

## Preface

This redbook provides guidelines and hints for designing and managing networks used with MQSeries. It also helps you design and develop application programs that use the features of MQSeries Version 5.1. Even though the examples have been developed for the Windows NT platform, they apply to any other platform that supports MQSeries Version 5.1. This book is based on class exercises for an ITSO workshop and explains the following:

- How to create and manager clusters
- How to use workload management (load balancing)
- How to use the new GUIs to administer MQSeries on Windows NT
- How to use the Web Administration feature
- How to monitor queues using the Windows NT Performance Monitor
- How to write a workload management exit
- How to use MQSeries for file transfer
- How to use dynamic queues
- Changes in MQSeries security

The first chapter contains an overview of the functions released with MQSeries Version 5.1. The other chapters are dedicated to specific functions. They include programming hints and examples as well as guidelines for MQSeries administration. This redbook comes with a diskette that contains the source code of all examples.

---

## The Team That Wrote This Redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization, Raleigh Center.

**Dieter Wackerow** is the MQSeries expert at the International Technical Support Organization, Raleigh Center. His areas of expertise include application design and development, performance evaluations, capacity planning, and modelling of computer systems and networks. He also wrote a simulator for banking hardware and software. He teaches classes and writes on performance issues and application development.

**David Armitage** is from the IBM Transarc Laboratory in Sydney, Australia. He has 15 years of experience in level 2 support for VM, Series1, AIX (including AIX/370), TCP/IP, DCE, Encina, and distributed CICS. For several years he

was involved in service delivery and architecture consulting for distributed CICS and MQSeries in the Asia/Pacific region. He also developed courses and taught extensively in Australia, New Zealand and Japan. David holds a Science degree in Mathematics from the University of Sydney.

**Tony Skinner** is a System Designer and IT Specialist in Canada. He has 32 years of experience in IT with IBM. His areas of expertise include design of On-line Transaction Processing (OLTP) and database systems and applications. He has written extensively on CICS and MQSeries.

Thanks to the following people for their invaluable contributions to this project:

Alexandros Alexandrakis  
IBM United Kingdom

Andrew Banks  
IBM Hursley, England

Mike Brady  
IBM Transarc, Australia

---

## Comments Welcome

### Your comments are important to us!

We want our redbooks to be as helpful as possible. Please send us your comments about this or other redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks Evaluation” on page 255 to the fax number shown on the form.
- Use the online evaluation form found at <http://www.redbooks.ibm.com/>
- Send your comments in an internet note to [redbook@us.ibm.com](mailto:redbook@us.ibm.com)

---

## Chapter 1. About MQSeries Version 5.1

MQSeries already speeds implementation of distributed applications by simplifying application development and testing. With Version 5.1 for distributed platforms and Version 2.1 for MVS/ESA, MQSeries takes the next steps: It simplifies the installation and deployment of MQSeries itself and it enables quicker rollout and scale-up of business-critical applications across the enterprise.

---

### 1.1 The MQSeries Family

IBM's new family of MQSeries products addresses integration from a business as well as an information technology (IT) perspective. The product family offers information systems that support the way you work, that fit in with your business processes and workflow, and that deliver real business advantages.

The family of Business Integration solutions, based around IBM's award-winning MQSeries software, consists of:

**MQSeries:** The leader in message-oriented middleware for message-based applications. The base MQSeries product totally and reliably connects any application, or system, to any other. There are several redbooks available about MQSeries. Refer to Appendix H.1, "International Technical Support Organization Publications" on page 241 for the titles.

**MQSeries Integrator:** Powerful message brokering that centralizes knowledge of the enterprise (like business rules and application data formats) in a central hub. It makes it much quicker and simpler to distribute data relating to business events, and integrate applications to build new business.

MQSeries Integrator software enables integration of applications and systems into robust, flexible, and scalable information networks. Based upon MQSeries messaging and queuing capabilities, MQSeries Integrator features intelligent message routing that directs data according to conditions set by the business, and message formatting enables applications to exchange information. Its GUI allows users to perform integration tasks quickly and easily. Preconfigured templates for major packaged applications and e-business extensions are available. This product is now available under IBM terms and conditions worldwide. IBM will continue to add Business Integration capabilities to MQSeries Integrator and

provide a route for IBM Business Partners to offer additional functionality.

**MQSeries Workflow:** Aligns and integrates your organization's resources and capabilities with your business strategies, accelerating process flow, cutting costs, eliminating errors and improving workgroup productivity.

MQSeries Workflow is a workflow management middleware offering that automates workflow between applications. It automates business processes involving people and applications to give organizations more control of their business activities. MQSeries Workflow helps you in daily business operations, planning, and management, to align and integrate resources and applications, to improve efficiency, and to gain higher market share. In addition, it enables the design of applications tailored to your business needs.

---

## 1.2 Platforms

MQSeries Version 5.1 runs on the following platforms:

- MQSeries for AIX, V5.1: RS/6000 running AIX 4.2 or 4.3
- MQSeries for HP-UX, V5.1: HP machines or Stratus Continuum, all running HP-UX V10.20 or V11.0
- MQSeries for Sun Solaris, V5.1: SunSPARC or Sun UltraSPARC — desktop or server running
- Sun Solaris V2.6 or Sun Solaris 7
- MQSeries for OS/2 Warp, V5.1: Intel or compatible systems capable of running OS/2 Warp V4.0
- MQSeries for Windows NT, V5.1: Intel or compatible systems capable of running Windows NT, V4.0

Some Version 5.1 features, such as clusters are also available with MQSeries for OS/390 Version 2.1 and will be available with MQSeries for AS/400 Version 5.1.

---

## 1.3 What's New in MQSeries Version 5.1

The single most important new feature is clustering, which includes dynamic workload management, also referred to as load balancing. This feature also greatly reduces the administrative work of defining MQSeries objects, such as channels, remote queue definitions and transmission queues. Chapter 2,

“About Clusters” on page 7, Chapter 4, “Creating a Cluster with the MQExplorer” on page 55, and Chapter 5, “Creating a Cluster with Scripts” on page 95 are dedicated to clusters.

Users of MQSeries for Windows NT can take advantage of additional great features. MQSeries for Windows NT Version 5.1 is tightly integrated with the operating system. New GUIs make the administration of MQSeries on this platform much easier. You can even manage queues over the Web. Also, MQSeries First Steps provides a feature that automatically creates a cluster queue manager and an installation verification program that you can use to send messages from one queue manager to another without any additional definitions. The MQSeries for Windows NT functions are described in detail in this book.

The following is a summary of the new features in Version 5.1:

- Clusters (or groups) of queue managers dynamically share workload among themselves, balancing workload and rerouting if a system component fails or network path becomes unavailable.
- Administration of clusters of queue managers is made simpler and quicker and with less likelihood of operator error.
- Queue managers in the same cluster can be on different platforms or physically remote from one another.
- Publish/subscribe function ensures people and applications receive information on their chosen subjects.
- Subscribers specifying topics of interest have great flexibility and can specify not just a name but a range of names, "wild card", or just prefixes, for example.
- Publish/subscribe takes advantage of the robust messaging features of MQSeries, so delivery is assured, and transactional integrity is maintained when published information updates corporate databases.
- MQSeries for Windows NT uses the Windows NT Performance Monitor, allows valid Windows NT user IDs, uses the Windows NT registry for storing configuration data, and provides a set of Component Object Model (COM) classes that allow ActiveX applications to access MQSeries programming interfaces.
- Also for Windows NT there are graphical tools and applications for installing, administering, and exploring the product, and a Web-based administration server.
- Message queues can be up to 2 GB.

- Scalability and performance improvements include multiple application processes that speed throughput, and an improvement for persistent critical messages.
- Support for the new environments of HP-UX Version 11 and AIX Version 4.3, demonstrating IBM commitment to support the latest versions of the most popular platforms.
- Improved Java programming interface, and the Java client and bindings are combined in a single Java package to make the task of programming simpler.
- MQSeries for AS/400 will be enhanced to provide functional parity with the other MQSeries V5.1 products on UNIX platforms.

---

## 1.4 Publish/Subscribe

This MQSeries release introduces standard implementation of a new messaging model for applications: Publish and Subscribe, sometimes known as Event Services. It enables users to register interest in particular sets of information and then to receive such information when it becomes available. The Publish/Subscribe model provides direction in that the publisher of data need have no prior knowledge about the receivers of such data and vice versa. Subject to authorities, publishers may start or cease, and applications may create, vary or delete subscriptions without the need for administrative action. Publish/Subscribe exploits the strengths of the existing MQSeries infrastructure. Applications can combine Publish/Subscribe with other messaging calls and other subsystems (such as, databases) and have the same choice of qualities of service (non-persistent, persistent or transactional) as in other MQSeries applications.

This facility is available for MQSeries on the AIX, Sun Solaris, HP-UX and Windows NT platforms.

**Note:** Publish/Subscribe is available on the Internet. It is not on the product CD.

---

## 1.5 MQSeries and Java

In MQSeries Version 5.1, the MQSeries classes for Java are repacked to JAR files. There is one JAR file per transport option:

- com.ibm.mq.jar for client support
- com.ibm.mqbind.jar for bindings support

- com.ibm.mq.iiop.jar for VisiBroker support

You must explicitly set the required JAR files in the classpath. The root Java directory must also be included for properties files.

### 1.5.1 Example

```
classpath=%CLASSPATH%;
      c:\mqm\java\lib\com.ibm.mq.jar;
      c:\mqm\java\lib\com.ibm.mqbind.jar;
      c:\mqm\java\lib;
mqm\lava\lib is the properties file directory.
```

Include in your Java programs (applets, servlets, applications) the following statement:

```
import com.ibm.mq.*
```

### 1.5.2 Connection Examples

#### 1. Bindings connection

```
MQQueueManager = new MQQueueManager("ARROW");
```

#### 2. Client connection

```
MQEnvironment.hostname = "arrow.raleigh.ibm.com";
MQQueueManager = new MQQueueManager("ARROW");
```

#### 3. VisiBroker connection

```
MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                             MQC_TRANSPORT_VISIBROKER);
MQEnvironment.hostname = "arrow.raleigh.ibm.com";
MQQueueManager = new MQQueueManager("ARROW");
```





---

## Chapter 2. About Clusters

Clusters are groups of machines connected together. Even multiple queue managers on a single processor can build or be part of a cluster. Almost any collection of queue managers could be described as a cluster. Examples are IBM SP2, IBM Parallel Sysplex, and racks of Intel processors. More generally, clusters are described as WANs, LANs, Internets or intranets.

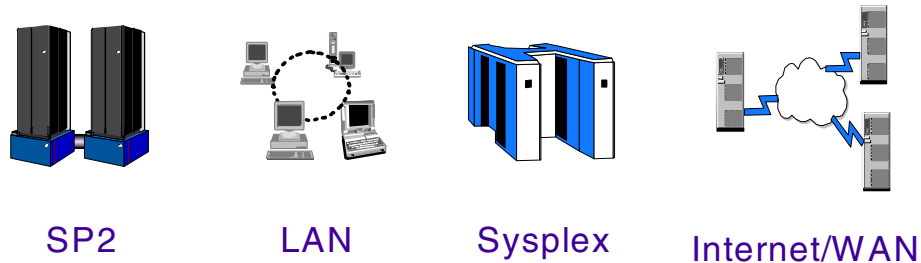


Figure 1. Clusters

What are we trying to achieve with MQSeries cluster support?

- We want to process very high message rates.
- We have to process very large numbers of messages.
- We want a single name space to describe queues and no conflict between names.
- We want a single system to control and administer.
- The system should always be available.

It is also important to remember the benefits for administration, even if no failover or scalability from multiple queue instances are required. Clusters allow for simple, scalable administration. Since there are fewer resources to define, there are fewer matching fields to enter, which reduces the opportunity to make errors. We will show that this is also valuable in small configurations.

Here are the reasons for reduced MQSeries administration:

- Communication channels are created automatically.
- Queues are administered cluster-wide.
- You don't have to define explicit remote queues any more.

- Only one cluster transmission queue is required to send messages to all queue managers in the cluster, and this queue is created automatically.
- There is no “master” queue manager to manage or that can fail. Each queue manager is still autonomous.
- Multiple instances of queue names are supported. So redundancy can be built into the systems.
- Multiple instances of queues support load balancing, also called workload management. Customizing is possible through workload management exit routines.
- No manual cleanup or deletion of redundant definitions.

## 2.1 Why Clusters?

Figure 2 shows very big systems running on a very big computer. You could take all of the applications you want to run with all their associated data and put them on a very big processor. Unfortunately, such processors do not exist, and, if they did exist the consequences of failure would be enormous.

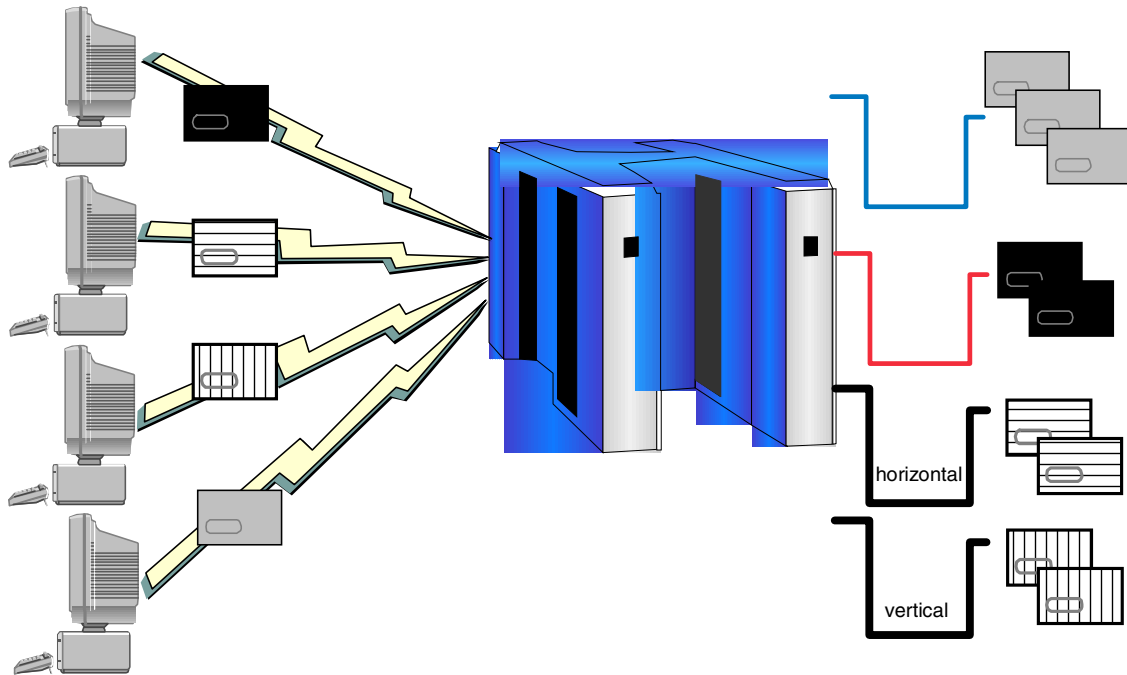


Figure 2. A Very Large Processor - An Impractical Solution

Even if we could build large enough processors that never failed, we may still not choose to use them. They would be very difficult to administer because the risk of making an administrative error would be considerable. Also, we would have to be very disciplined in the names we used.

Figure 2 gives a hint of this problem in that there are two striped queues, one horizontal and the other vertical striped. The applications on the left both intend to use a striped queue; However, one of them needs the horizontal and the other the vertical striped queue. Unfortunately, they have both chosen the name "striped".

Clustering is one way to help us to deploy several real machines as part of an application. In the configuration shown in Figure 3 several instances of the queue have been deployed on various processors. We achieve the processing power we need by using a number of smaller and less expensive processors working on the subset of the messages that are sent.

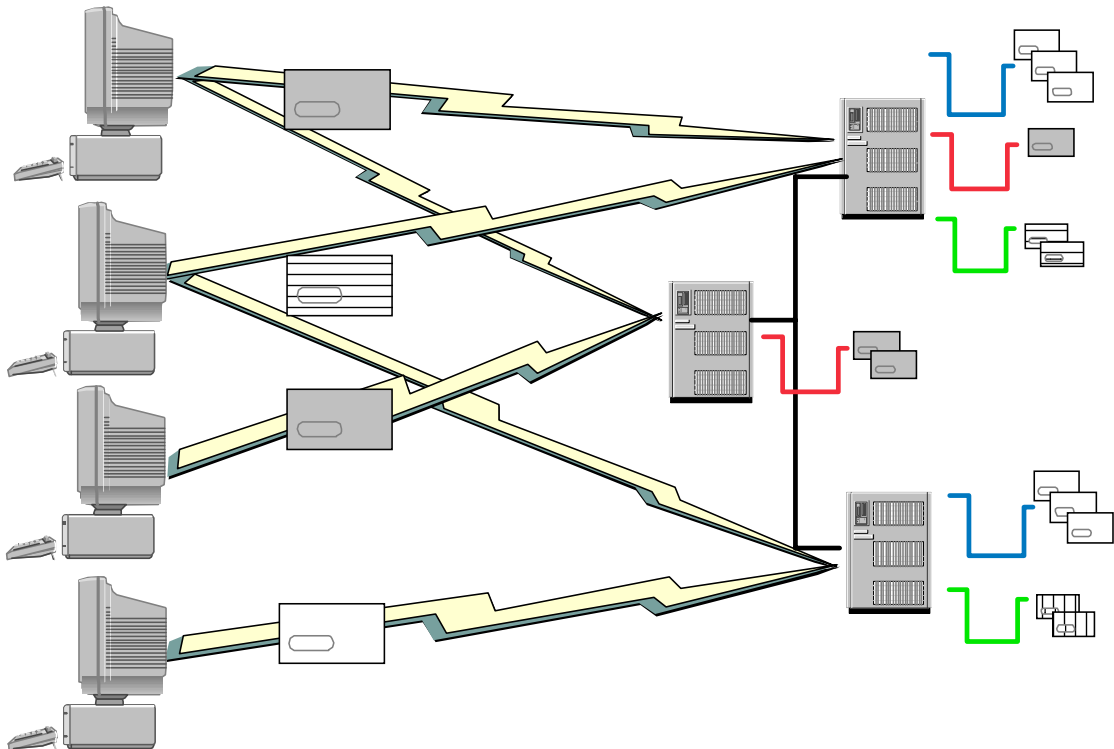


Figure 3. Cluster Solution

We achieve better availability because, if one of the smaller processors fails, or the network to it fails, we are still able to use others that have not failed.

The MQSeries cluster support is designed so that the consequences of an administrative error are confined to the machine that it is made on. MQSeries also manages the messages so that they go to a working instance of the queue as quickly as possible, without exposing the application programmers and the administrators to the complexity of achieving this.

MQSeries also offers a solution to the naming problems. Users of the striped queue in Figure 3 may use the specific striped queue that they know to be the one they are really interested in, or you can construct the cluster so that a particular application sees its own kind of striped queue only.

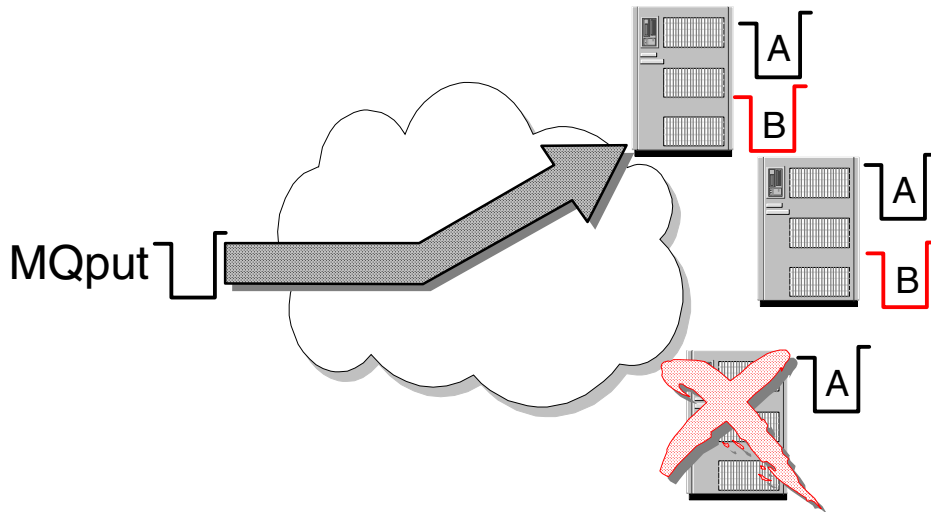


Figure 4. Goals of the Cluster Support

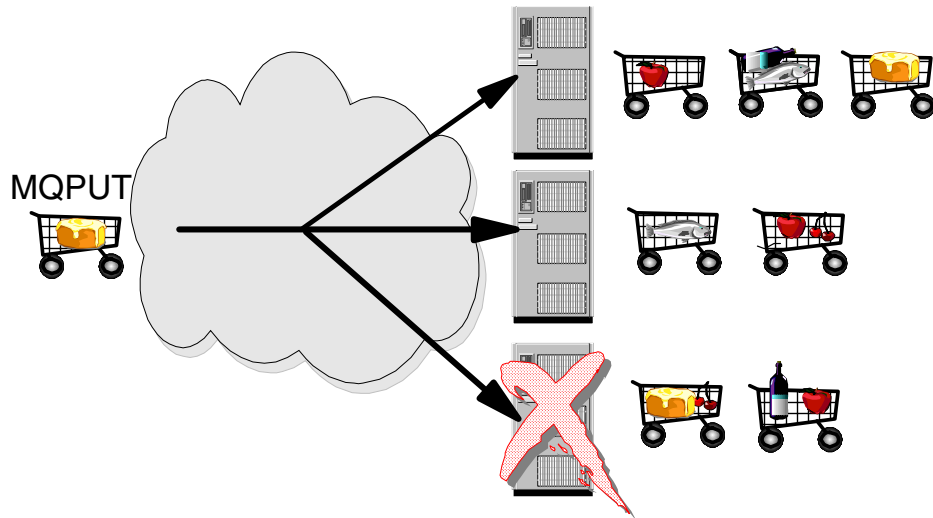
Now let us look at Figure 4. You see multiple queues, A and B, with a single image. The vision of an MQSeries cluster is as the place where multiple instances of a queue can exist. They come and go as an administrator requires in order to satisfy changing availability and throughput requirements. This has to be achieved completely dynamically and without placing the administrator under a great burden to configure and control it. In addition, the application programmer does not have to think about the multiple queues; he just treats them as if he were writing to a single queue.

This is not to say that there is no burden on the programmer or administrator. Enhanced levels of availability and exploitation of parallelism do require some planning. The administrator or system designer must ensure that there is enough redundancy in the configuration to meet their needs. The application designer must ensure that messages are capable of being processed in multiple places.

---

## 2.2 Cluster Concept

The concepts used for the MQSeries cluster support are very natural ones that we come across frequently in everyday life.



*Figure 5. Cluster Checkout*

Think about a supermarket checkout as illustrated in Figure 5. When we want our basket of goods processed we choose a checkout and get in the queue for it. We may not always make the ideal choice, in that another checkout may turn out to be working faster, but we do make a choice. If our initial choice turns out to be badly wrong then we can always make another choice. This is very similar to the way the MQSeries cluster support manages our messages. It makes a choice on a number of possible destinations and if it turns out to be a bad choice then it tries again.

In the above example, the application (customer) sees a single (checkout) queue which is actually implemented as multiple (cashier) queues.

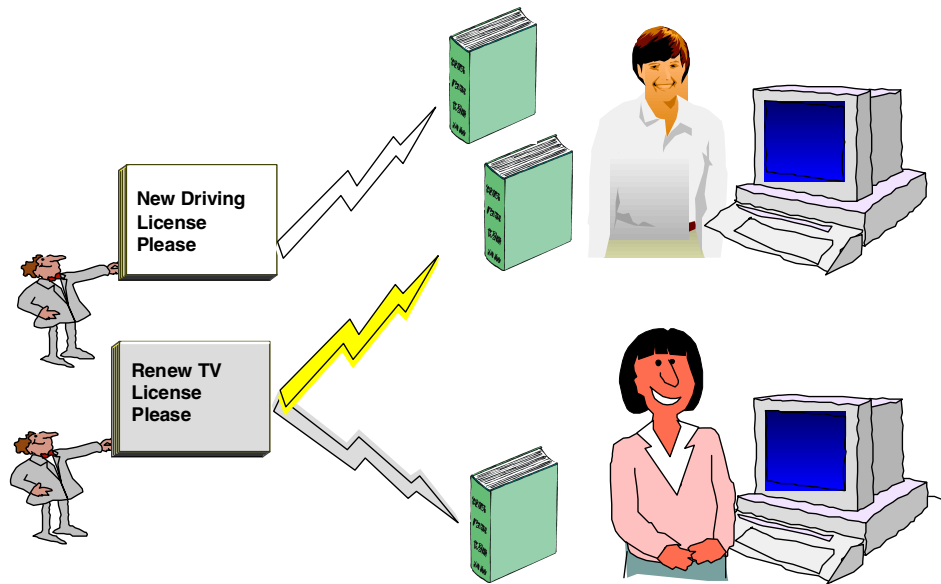


Figure 6. Workload Partitioning

Take another ever day example, this time a type of workload partitioning as shown in Figure 6. In a British post office, some of the counters offer full service and some only specialized functions. We have to make a choice based on the type of request we have, either a driving license application or television license application. The type of request decides which counter we must queue for.

We need three pieces of underlying technology to achieve clustering. These are shown in Figure 7 on page 13.

1. A repository containing the location of queues and queue managers
2. A means of communicating between queue managers that does not require prior definition, that is, automatic definition of dynamic channels
3. A way to choose where to send the messages, either to a specific queue or to any queue with the same name owned by any queue manager in the cluster

In summary, MQSeries cluster support provides automated definition of channels based on a single command. It supports multiple instances of queues so redundancy can be built into the system. There is no master queue manager that can fail; each queue manager is autonomous.

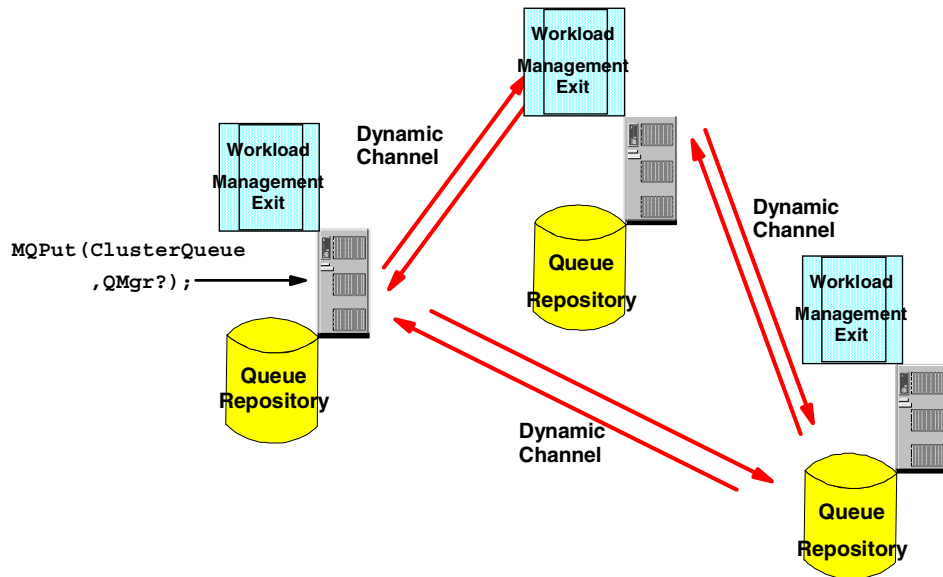


Figure 7. Components of a Cluster

You can see in Figure 7 that there are only dynamic channels defined between the queue managers for data transfer (message channels). There are also permanent channels that are used by the queue managers to exchange repository information. These are the cluster sender and cluster receiver channels.

In normal distributed messaging, we send messages to a specific queue owned by a specific queue manager. All messages destined for that queue manager are placed in a transmission queue at the sender's side. This transmission queue has the same name as the destination queue manager. The message channel agents move the messages across the network where it is placed in the destination queue. The picture below shows the relationship between the transmission (Xmit) queue and the target queue manager.

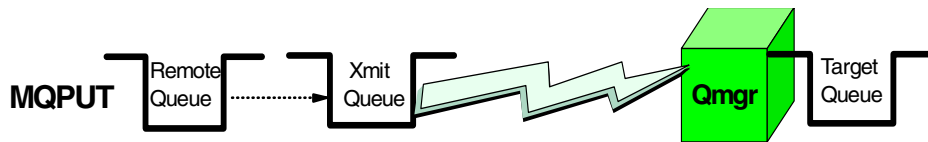


Figure 8. MQPUT to a Remote Queue

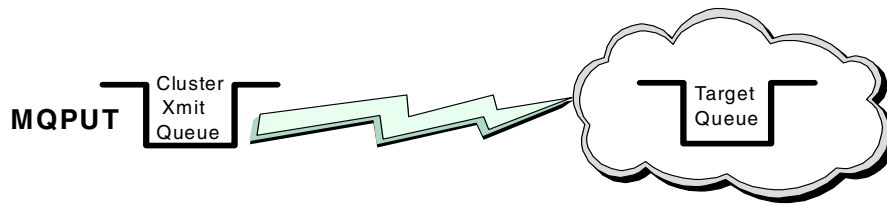


Figure 9. MQPUT to a Cluster Queue

With clustering, you send a message to a queue with a specific name somewhere in the cluster, here represented by a cloud. You specify the name of the target queue, not the name of a remote queue. Remote queues don't exist in clusters. They are only useful when the queue manager communicates with a queue manager that is not a member of the cluster. You may also want to specify the queue manager, but very often it is left to MQSeries to find out where the queue is (or the queues are) and where to send the message to.

Figure 10 on page 15 illustrates how clustering works. The queue manager finds out from the repository where the destination queue and the queue manager are located. If the queue is local then all messages will be placed in that local queue. Otherwise, it builds a transmission header and puts the message in the transmission queue. Instead of one transmission queue for each of the remote queue managers, in a cluster there is only one *cluster transmission queue* that hold all messages destined for all remote cluster queue managers.

The message channels between the sending and receiving queue managers are automatically created.

Figure 10 on page 15 shows what happens when MQSeries executes an MQOPEN, MQPUT and MQGET:

- If the queue is defined anywhere in the cluster the MQOPEN succeeds and opens it.
 

**Note:** When both queue and queue manager are specified, the existence of the queue is not checked. If the queue does not exist at that queue manager, the message will be put in the dead-letter queue, if there is one.
- When an MQPUT is executed, it can go either to a local instance of the queue or to a remote instance. MQSeries always puts the message in a local queue if there is one.



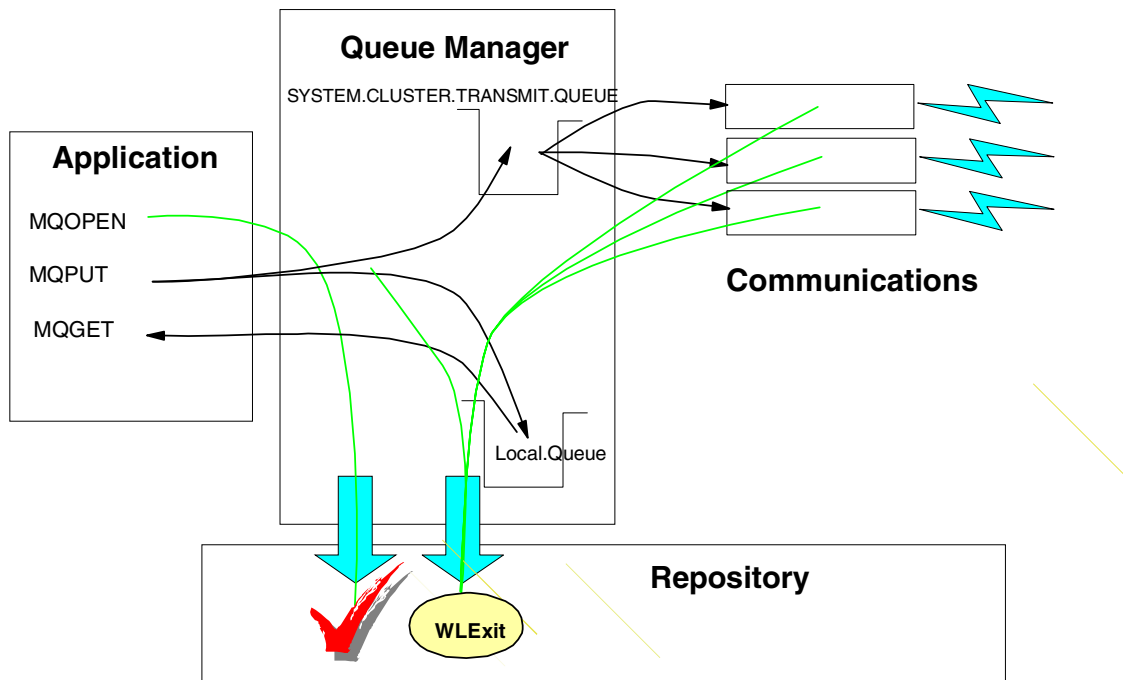


Figure 10. How Clustering Works

- A cluster workload exit can choose which queue instance to put the message to. By default, messages are distributed to the queue instances round-robin.
  - Note:** Workload balancing is determined by the open option.
    - The option MQOO\_BIND\_NOT\_FIXED causes the messages to be distributed round-robin to all queues with the same name.
    - If you specify MQOO\_BIND\_ON\_OPEN then all messages sent between the MQOPEN and the MQCLOSE are sent to the same queue. This conforms to the way MQSeries operates in a non-cluster mode.
- An MQGET can only be issued against the local instance of a queue. Here, nothing has changed from previous releases.

## 2.3 A Routing Example

Figure 11 shows an example of a cluster made from four servers connected using a local area network. There are four queue managers, one on each processor. Each queue manager hosts a variety of queues. Q3 for example is hosted on QM1 and QM3.

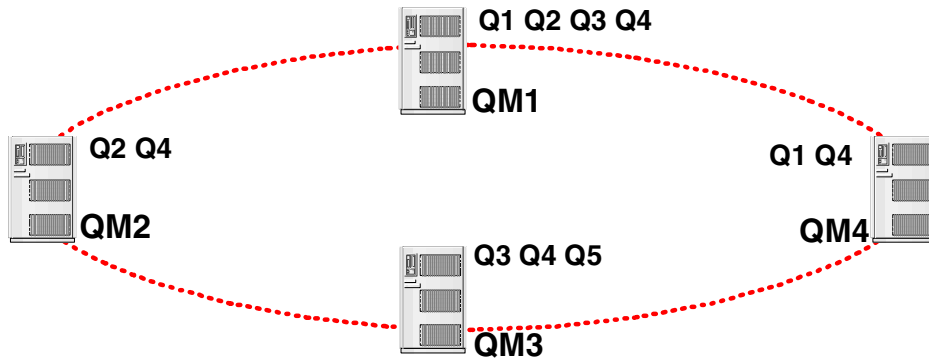


Figure 11. A Routing Example - Configuration

Now let us look at Table 1 and analyze the four examples for putting messages into Q3.

Table 1. A Routing Example - Results

Example No	QM Where Put	Requested Queue	Requested QM	Actual Destination QM	Status
1	QM2	Q3		QM1/QM3	OK
2	QM2	Q3	QM1	QM1	OK
3	QM2	Q3	QM4	Dead Letter Queue	Error
4	QM1	Q3		QM1/QM3	OK

1. Example 1 shows what happens if an application running on QM2 executes an MQPUT to Q3 without specifying a destination queue manager. Because Q3 exists on QM1 and QM3, the message could be sent to either QM1 and QM3.
2. Example 2 shows that if the same application had chosen to specify QM1 explicitly as the destination, then this choice would be honored and the message sent to QM1.

3. In example 3, QM4 was chosen explicitly as the destination queue manager. When the message arrives at QM4 it is put in the dead-letter queue because there is no instance of Q3 on QM4.
4. In example 4, an application running on QM1 requests Q3 without specifying a destination queue manager. Here either QM1 or QM3 could be the destination. In fact, if the default routing is used unaltered then QM1 will always be chosen because it hosts the local instance of the queue.

The queue manager identifies a queue as being in the cluster when the queue is opened. This is done by searching the repository. If the queue exists somewhere in the cluster then the application can open it.

**Watch out!**

If the application explicitly names a destination, this is used without checking that the queue exists at that destination.

The queue manager puts all messages for all queues in the cluster on a single transmission queue called `SYSTEM.CLUSTER.TRANSMIT.QUEUE`.

---

## 2.4 How Clusters Work

Two new channel types have been introduced for clusters:

- Cluster Sender Channel (CLUSSDR)
- Cluster Receiver Channel (CLUSRCVR)

There are also two types of repositories:

- A *full repository* holds information about all queues and queue managers in the cluster. Usually, two queue managers hold full repositories. Both queue managers exchange information so that both repositories contain the same data.
- All other queue managers have a *partial repository*, which contains only information the particular queue manager is interested in.

Data is kept in the repository for a specified time (90 days) or until it is refreshed by an operator command.

Figure 12 shows a cluster that consists of three queue managers. Two of them, QM2 and QM3, hold a full repository while QM1 holds a partial repository.

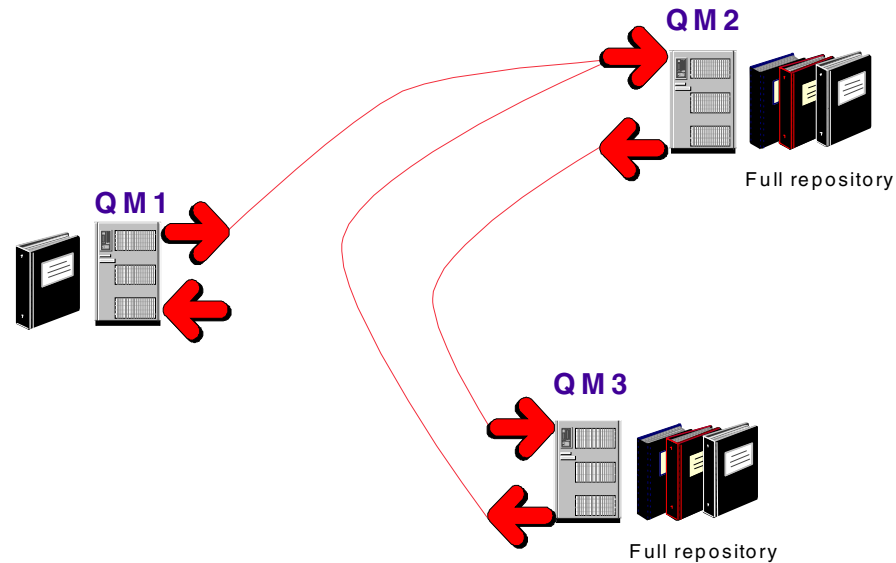


Figure 12. Cluster Definitions

To set up a cluster you first need to decide which two queue managers are going to keep a full copy of all the definitions in the cluster. Here we choose QM2 and QM3 to hold the full repositories; QM1 just holds the subset of the repository that it needs, a partial repository.

QM2 and QM3 are not master repositories in any sense; they do not control the cluster, nor does the cluster stop functioning if both are unavailable.

Each queue manager must have one cluster sender channel (CLUSDR) defined to give it the location of another repository. In the case of the full repository this must point to the other full repository. CLUSDR channels are represented by the arrows pointing away from the queue manager.

Each queue manager must also have a *cluster receiver channel* (CLUSRCVR) defined. Via this channel it receives messages from the other members of the cluster. And it is also used to advertise the queue manager to other queue managers (via the full repositories) so that they know how to connect to it. CLUSRCVR channels are represented by the arrows pointing to the queue managers.

A CLUSRCVR channel can talk only to a CLUSSDR channel. It cannot connect to a normal sender channel. Also, an MQ client cannot use it.

**Note:** The full repository queue manager QM2 must be running to distribute information about objects in the cluster that belong to a queue manager attached to it. For example, if you create a new queue on QM1, and QM2 is down, then QM3 and any other queue manager in the cluster will not become aware of the queue until QM2 is restarted again. You may, however, define cluster channels between QM1 and the other full repository, QM3.

Figure 13 shows a typical cluster setup from the point of view of the queue manager on the left. It has a single CLUSRCVR channel defined as well as a single CLUSSDR channel to the full repository queue manager on the top. All other definitions you see are created automatically.

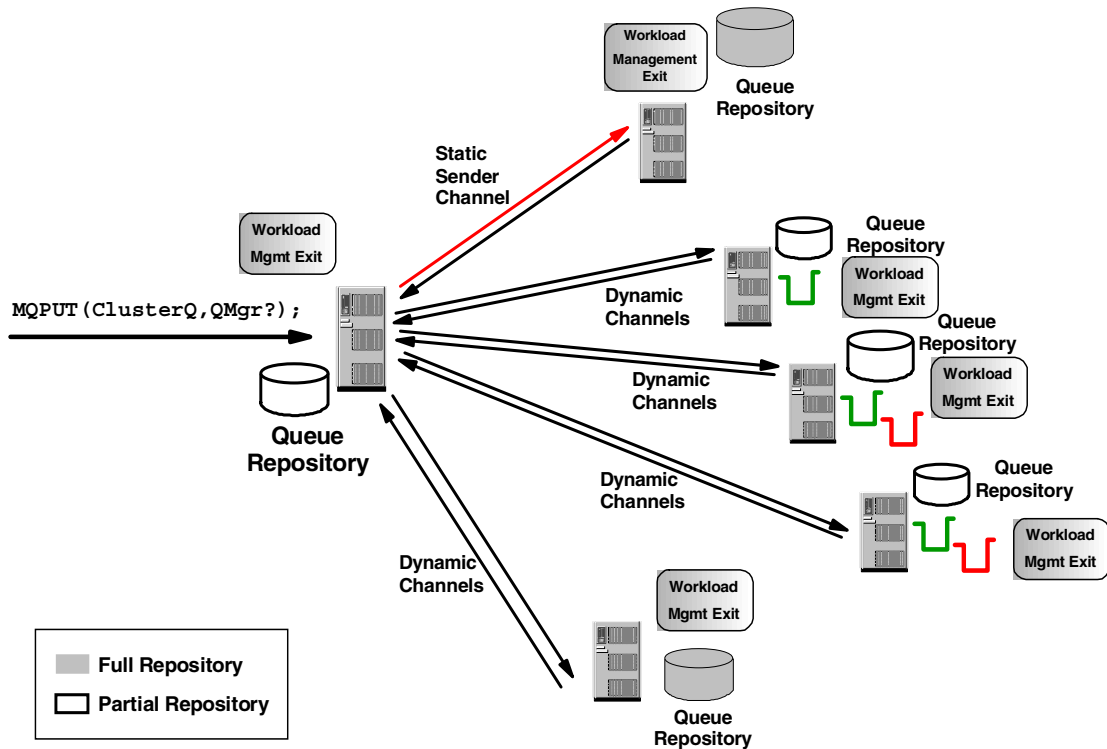


Figure 13. Typical Cluster Definition

## 2.4.1 Example with Two Queue Managers

Very little definition is required to use the MQSeries cluster support. In Figure 14 on page 20 we show an example using two queue managers connected using a local area network.

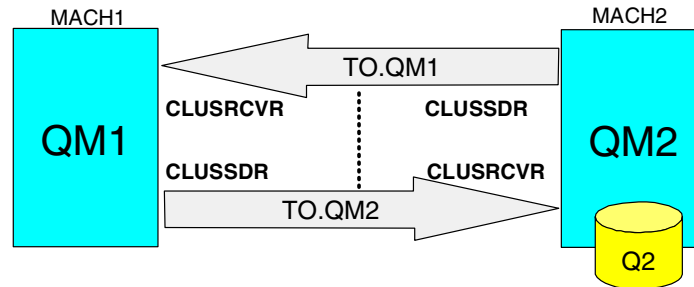


Figure 14. Example of a Cluster with Two Queue Managers

There are very few definitions, yet we have a high degree of autonomy at each queue manager and no single point of failure for either the configuration or the messages to queue Q2. Even the queues that do not have multiple instances have only been defined once.

```
ALTER QMGR REPOS (CLUS_1)
DEFINE CHANNEL (TO.QM1) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
          CONNAME (MACH1 . IBM . COM) CLUSTER (CLUS_1)
DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
          CONNAME (MACH2 . IBM . COM) CLUSTER (CLUS_1)
```

Figure 15. RUNMQSC Script File for QM1 in a Cluster

```
ALTER QMGR REPOS (CLUS_1)
DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
          CONNAME (MACH2 . IBM . COM) CLUSTER (CLUS_1)
DEFINE CHANNEL (TO.QM1) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
          CONNAME (MACH1 . IBM . COM) CLUSTER (CLUS_1)
DEFINE QLOCAL (Q2) CLUSTER (CLUS_1)
```

Figure 16. RUNMQSC Script File for QM2 in a Cluster

Now let us compare the definitions for the cluster above with the definitions we would have to make without clustering.

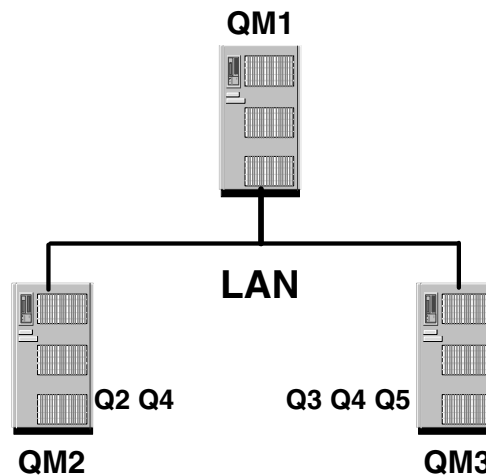
```
DEFINE QLOCAL(QM2) USAGE(XMITQ)
DEFINE QREMOTE(Q2) RNAME(Q2) RQMNAME(QM2)
DEFINE CHANNEL(TO.QM1) CHLTYPE(RCVR)
DEFINE CHANNEL(TO.QM2) CHLTYPE(SDR) XMITQ(QM2) +
    TRPTYPE(TCP) CONNNAME(MACH2.IBM.COM)
```

Figure 17. RUNMQSC Script File for QM1 without Clustering

```
DEFINE QLOCAL(QM1) USAGE(XMITQ)
DEFINE CHANNEL(TO.QM2) CHLTYPE(RCVR)
DEFINE CHANNEL(TO.QM1) CHLTYPE(SDR) XMITQ(QM1) +
    TRPTYPE(TCP) CONNNAME(MACH1.IBM.COM)
DEFINE QLOCAL(Q2)
```

Figure 18. RUNMQSC Script File for QM2 without Clustering

## 2.4.2 Example with Three Queue Managers



This example shows the definitions for a cluster consisting of three queue managers and five cluster queues. QM2 and QM3 hold a full repository.

```

DEFINE CHANNEL (TO.QM1) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
          CONNAME (MACH1 . IBM . COM) CLUSTER (CLUS_1)

DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
          CONNAME (MACH2 . IBM . COM) CLUSTER (CLUS_1)

```

Figure 19. Definitions for QM1

```

ALTER QMGR REPOS (CLUS_1)

DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
          CONNAME (MACH2 . IBM . COM) CLUSTER (CLUS_1)

DEFINE CHANNEL (TO.QM3) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
          CONNAME (MACH3 . IBM . COM) CLUSTER (CLUS_1)

DEFINE QLOCAL (Q2) CLUSTER (CLUS_1)
DEFINE QLOCAL (Q4) CLUSTER (CLUS_1)

```

Figure 20. Definitions for QM2 (Repository)

```

ALTER QMGR REPOS (CLUS_1)

DEFINE CHANNEL (TO.QM3) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
          CONNAME (MACH3 . IBM . COM) CLUSTER (CLUS_1)

DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
          CONNAME (MACH23 . IBM . COM) CLUSTER (CLUS_1)

DEFINE QLOCAL (Q3) CLUSTER (CLUS_1)
DEFINE QLOCAL (Q4) CLUSTER (CLUS_1)
DEFINE QLOCAL (Q5) CLUSTER (CLUS_1)

```

Figure 21. Definitions for QM3 (Repository)

Compare the above definitions with the ones on the next page. You can see that clusters make life easier.



```

DEFINE QLOCAL(QM2) USAGE(XMITQ)
DEFINE QLOCAL(QM3) USAGE(XMITQ)
DEFINE QREMOTE(Q2) RNAME(Q2) RQMNAME(QM2)
DEFINE QREMOTE(Q3) RNAME(Q3) RQMNAME(QM3)
DEFINE QREMOTE(Q4) RNAME(Q4) RQMNAME(QM3)
DEFINE QREMOTE(Q5) RNAME(Q5) RQMNAME(QM3)
DEFINE CHANNEL(TO.QM1) CHLTYPE(RCVR)
DEFINE CHANNEL(TO.QM2) CHLTYPE(SDR) XMITQ(QM2) +
    TRPTYPE(TCP) CONNAME(MACH2.IBM.COM)
DEFINE CHANNEL(TO.QM3) CHLTYPE(SDR) XMITQ(QM3) +
    TRPTYPE(TCP) CONNAME(MACH3.IBM.COM)

```

Figure 22. Definitions for QM1 without Clustering

```

DEFINE QLOCAL(QM1) USAGE(XMITQ)
DEFINE QLOCAL(QM3) USAGE(XMITQ)
DEFINE QREMOTE(Q3) RNAME(Q3) RQMNAME(QM3)
DEFINE CHANNEL(TO.QM2) CHLTYPE(RCVR)
DEFINE CHANNEL(TO.QM1) CHLTYPE(SDR) XMITQ(QM1) +
    TRPTYPE(TCP) CONNAME(MACH1.IBM.COM)
DEFINE CHANNEL(TO.QM3) CHLTYPE(SDR) XMITQ(QM3) +
    TRPTYPE(TCP) CONNAME(MACH3.IBM.COM)
DEFINE QLOCAL(Q2)
DEFINE QLOCAL(Q4)

```

Figure 23. Definitions for QM2 without Clustering

```

DEFINE QLOCAL(QM1) USAGE(XMITQ)
DEFINE QLOCAL(QM3) USAGE(XMITQ)
DEFINE QREMOTE(Q2) RNAME(Q2) RQMNAME(QM2)
DEFINE CHANNEL(TO.QM3) CHLTYPE(RCVR)
DEFINE CHANNEL(TO.QM1) CHLTYPE(SDR) XMITQ(QM1) +
    TRPTYPE(TCP) CONNAME(MACH1.IBM.COM)
DEFINE CHANNEL(TO.QM2) CHLTYPE(SDR) XMITQ(QM2) +
    TRPTYPE(TCP) CONNAME(MACH2.IBM.COM)
DEFINE QLOCAL(Q3)
DEFINE QLOCAL(Q4)
DEFINE QLOCAL(Q5)

```

Figure 24. Definitions for QM3 without Clustering

Doing this the old way we see that fewer definitions were required in the clustering case. It is also no longer possible to have multiple instances of the queues. The number of definitions is reduced and the number of matching fields is also reduced. The statistics are shown in Table 2.

## Number of definitions

	Clustering	Non Clustering
CHANNELS	6	9
QLOCAL	5	5
QREMOTE	0	6
XMIT QUEUE	0	6

## Number of matching fields

	Clustering	Non Clustering
CHANNELS	6	18
QLOCAL	0	0
QREMOTE	0	8
XMIT QUEUE	0	6

Table 2. Comparison of Definitions with and without Clustering

For clustering, the connection name and the channel name in the CLUSSDR channel must match the repository CLUSRCVR.

For nonclustering, the transmit queue name must match an actual transmit queue, as well as matching connection name and channel name. QREMOTE names must match QLOCAL names, and transmit queue names should match queue manager names.

### 2.4.3 Example with Four Queue Managers in Two Networks

Figure 25 on page 25 shows four queue managers. QM4 runs on MVS/ESA and holds a full repository. The other full repository is maintained by QM2, which runs on a distributed platform, such as Windows NT.

Notice that QM2 has two cluster receiver channels defined, one (CS2) using SNA to connect to QM4 and another (CT2) using TCP/IP. The cluster sender channel points to QM4 on the mainframe and uses SNA.

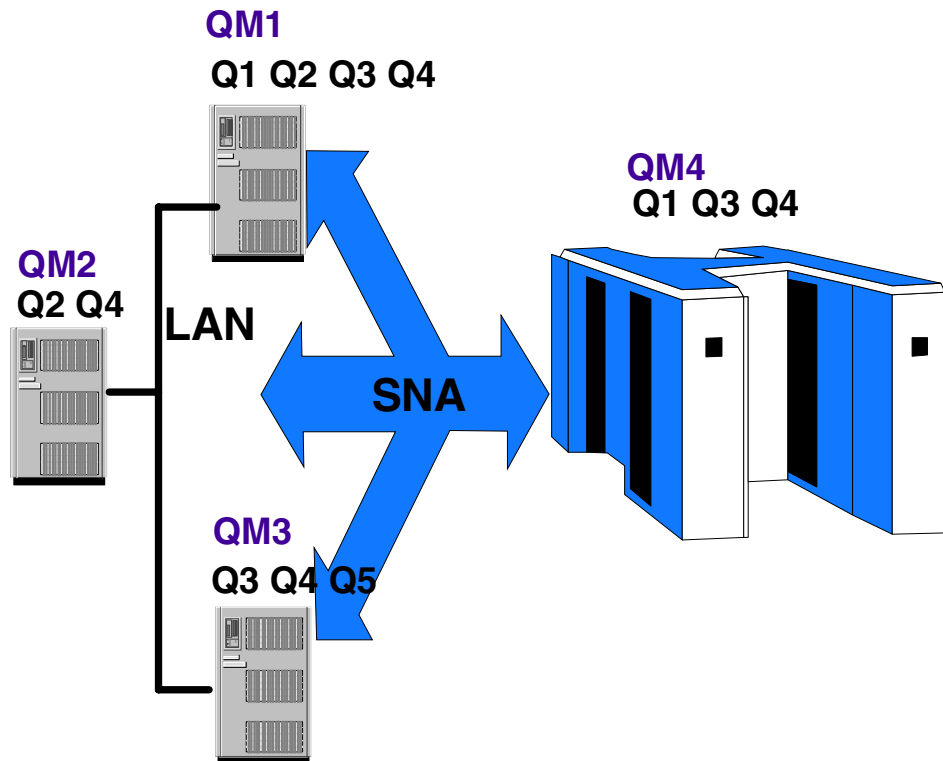


Figure 25. Multiple Networks

```

ALTER QMGR REPOS (DEMO)
DEFINE CHANNEL (CS2) CHLTYPE (CLUSRCVR) TRPTYPE (SNA) +
NETPRTY (1) CONNAME (SNAQM2) CLUSTER (DEMO)
DEFINE CHANNEL (CT2) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
NETPRTY (2) CONNAME (MACH2.IBM.COM(1414)) CLUSTER (DEMO)
DEFINE CHANNEL (CS4) CHLTYPE (CLUSSDR) TRPTYPE (SNA) +
CONNAME (SNAQM4) CLUSTER (DEMO)
DEFINE QLOCAL (Q2) CLUSTER (DEMO)
DEFINE QLOCAL (Q4) CLUSTER (DEMO)

```

Figure 26. Two Networks - Script File for QM2

```

DEFINE CHANNEL (CS3) CHLTYPE (CLUSRCVR) TRPTYPE (SNA) NETPRTY (1)
CONNAME (SNAQM3) CLUSTER (DEMO)
DEFINE CHANNEL (CT3) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) NETPRTY (2)
CONNAME (MACH3.IBM.COM) CLUSTER (DEMO)
DEFINE CHANNEL (CS4) CHLTYPE (CLUSSDR) TRPTYPE (SNA)
CONNAME (SNAQM4) CLUSTER (DEMO)
DEFINE CHANNEL (CT2) CHLTYPE (CLUSSDR) TRPTYPE (TCP)
CONNAME (MACH2.IBM.COM) CLUSTER (DEMO)
DEFINE QLOCAL (Q3) CLUSTER (DEMO)
DEFINE QLOCAL (Q4) CLUSTER (DEMO)
DEFINE QLOCAL (Q5) CLUSTER (DEMO)

```

Figure 27. Two Networks - Script File for QM3

```

ALTER QMGR REPOS (DEMO)
DEFINE CHANNEL (CS4) CHLTYPE (CLUSRCVR) TRPTYPE (SNA) +
CONNAME (SNAQM4) CLUSTER (DEMO)
DEFINE CHANNEL (CS2) CHLTYPE (CLUSSDR) TRPTYPE (SNA) +
CONNAME (SNAQM2) CLUSTER (DEMO)
DEFINE QLOCAL (Q1) CLUSTER (DEMO)
DEFINE QLOCAL (Q3) CLUSTER (DEMO)
DEFINE QLOCAL (Q4) CLUSTER (DEMO)

```

Figure 28. Two Networks - Script File for QM4

---

## 2.5 RUNMQSC Commands for Clusters

In this section, we briefly describe the RUNMQSC commands used with clusters. They are described, in detail, in the *MQSeries Command Reference*, SC33-1369.

In this case all three queue managers run in the same machine. Therefore, we can use the TCP/IP loopback address 127.0.0.1 as CONNAME. We also need three listeners listening on different ports. First let us have a closer look at the commands we already know, the ones we used to define the clusters in the previous examples.

```

DEFINE CHANNEL (TO.QM1) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1414)') CLUSTER (CLUS_1) REPLACE

DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1415)') CLUSTER (CLUS_1) REPLACE

```

Figure 29. First of Three Cluster Queue Managers in One Machine

QM1 does not keep a full repository. We specify two channels:

- Over the CLUSRCVR channel QM1 advertises itself to the cluster; it is also used to receive information about queues within the cluster.
- Over the CLUSSDR channel QM1 sends information about cluster objects it owns to its repository queue manager, which is QM2.

QM1 is associated with port 1414 (which is the default port) and QM2 listens to port 1515.

```

ALTER QMGR REPOS (CLUS_1)

DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSRCVR) TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1415)') CLUSTER (CLUS_1) REPLACE

DEFINE CHANNEL (TO.QM3) CHLTYPE (CLUSSDR) TRPTYPE (TCP) +
        CONNAME ('127.0.0.1 (1416)') CLUSTER (CLUS_1) REPLACE

DEFINE QLOCAL (Q2) CLUSTER (CLUS_1) REPLACE
DEFINE QLOCAL (Q4) CLUSTER (CLUS_1) REPLACE

```

Figure 30. Second of Three Cluster Queue Managers in One Machine

QM2 is a server in the cluster CLUS\_1 and it also maintains a copy of the repository. The commands cause the following:

- ALTER QMGR REPOS (CLUS\_1)  
This command declares that the queue manager is to own and maintain a copy of the repository for cluster CLUS\_1.
- DEFINE CHANNEL (TO.QM2) CHLTYPE (CLUSRCVR) ...  
This attaches the queue manager to the cluster.
- DEFINE CHANNEL (TO,QM3) CHLTYPE (CLUSSDR) ...  
This instructs the queue manager to send any updates for the repository

down this channel to QM3 where another copy of the repository is maintained.

- `DEFINE QLOCAL(Q2) CLUSTER(CLUS_1)`  
This defines a local queue. The queue manager is instructed to advertise the existence of an instance of Q2 on this queue manager to the rest of the cluster.

QM2 communicates with QM3, the other full repository queue manager in the cluster. They use port 1415 and 1416, respectively.

```
ALTER QMGR REPOS(CLUS_1)

DEFINE CHANNEL(TO.QM3) CHLTYPE(CLUSRCVR) TRPTYPE(TCP) +
          CONNAME('127.0.0.1(1416)') CLUSTER(CLUS_1) REPLACE

DEFINE CHANNEL(TO.QM2) CHLTYPE(CLUSSDR) TRPTYPE(TCP) +
          CONNAME('127.0.0.1(1415)') CLUSTER(CLUS_1) REPLACE

DEFINE QLOCAL(Q3) CLUSTER(CLUS_1) REPLACE
DEFINE QLOCAL(Q4) CLUSTER(CLUS_1) REPLACE
DEFINE QLOCAL(Q5) CLUSTER(CLUS_1) REPLACE
```

*Figure 31. Third of Three Cluster Queue Managers in One Machine*

QM3, the other full repository queue manager has cluster channels defined to communicate with QM2. Furthermore, it hosts three queues that are known throughout the cluster.

We typed the three sets of definitions in three different script files so that they can be used as input for RUNMQSC. To get communication going between the three queue managers, you first have to create the queue managers and start them:

```
crtmqm QM1
crtmqm QM2
crtmqm QM3
strmqm QM1
strmqm QM2
strmqm QM3
```

Next you have to start the three listeners:

```
start runmqtsr -t tcp -p 1414 -m QM1
start runmqtsr -t tcp -p 1415 -m QM2
```

```
start runmqtsr -t tcp -p 1416 -m QM3
```

This brings up three listener windows. After a while you will see that the channel programs have started. The following message will appear in each window:

```
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.  
09/09/99 15:05:58 Channel program started.  
09/09/99 15:07:06 Channel program started.
```

Now the repository queue managers QM2 and QM3 exchanged information about the queue in the cluster. QM1 will build a partial repository when it requests information about one of the queues, that is, when it tries to open one.

Let us use RUNMQSC on QM2 and display the queues we defined. Remember, all queues start with a Q.

```
C:\>runmqsc QM2  
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998.  
Starting MQSeries Commands.  
  
dis ql(Q*)  
 1 : dis ql(Q*)  
AMQ8409: Display Queue details.  
      QUEUE(Q2)  
AMQ8409: Display Queue details.  
      QUEUE(Q4)  
  
dis ql(Q*) cluster  
 2 : dis ql(Q*) cluster  
AMQ8409: Display Queue details.          QUEUE(Q2)  
      CLUSTER(CLUS_1)  
AMQ8409: Display Queue details.          QUEUE(Q4)  
      CLUSTER(CLUS_1)
```

Figure 32. Display Queues

- The first command lists all queue names starting with Q. This is no different from previous versions of MQSeries.
- The second command with the parameter cluster displays the same queues and the name of the cluster they belong to, if applicable.

In the examples above, we see only local queues or the local instances of the queues.

```

C:\>runmqsc QM2
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
Starting MQSeries Commands.

dis ql(Q*) cluster clusinfo clusqmgr
  1 : dis ql(Q*) cluster clusinfo clusqmgr
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q2)
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q4)
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q2)
  CLUSQMgr (QM2)
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q3)
  CLUSQMgr (QM3)
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q4)
  CLUSQMgr (QM3)
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q4)
  CLUSQMgr (QM2)
AMQ8409: Display Queue details.
  CLUSTER (CLUS_1)                QUEUE (Q5)
  CLUSQMgr (QM3)

```

Figure 33. Display Queues with Cluster information

The above screen shows all five queues and tells us which queue manager hosts which queues.

- In the first two lines we see the queues owned by QM2, Q2 and Q4. The additional lines are caused by the clusinfo parameter.
- There is one instance of Q2 at QMGR2.
- There are is instance each of Q3 and Q5 at QM3.
- There are two instances of Q4, one at QM2 and the other at QM3.
- QM1 does not own any cluster queues.

With the next command, `display CLUSQMgr` (Figure 35), we can display cluster information about all queue managers in the cluster. It displays the routes to queue managers in the cluster to which a queue manager is attached.

In the following example we display the information as seen from QM2.



```

C:\>runmqsc QM2
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
Starting MQSeries Commands.

dis clusqmgr(*) connname status
  1 : dis clusqmgr(*) connname status
AMQ8441: Display Cluster Queue Manager details.
  CLUSQMGR(QM1)                CLUSTER(CLUS_1)
  CHANNEL(TO.QM1)              CONNAME(127.0.0.1(1414))
  STATUS(RUNNING)
AMQ8441: Display Cluster Queue Manager details.
  CLUSQMGR(QM2)                CLUSTER(CLUS_1)
  CHANNEL(TO.QM2)              CONNAME(127.0.0.1(1415))
AMQ8441: Display Cluster Queue Manager details.
  CLUSQMGR(QM3)                CLUSTER(CLUS_1)
  CHANNEL(TO.QM3)              CONNAME(127.0.0.1(1416))
  STATUS(RUNNING)

```

Figure 34. Display Cluster Information (1)

The above screen shows the three queue managers in CLUS\_1. You can see the connection name (the TCP/IP loop back address) and the port. The channels between the queue managers are running.

```

C:\>runmqsc QM2
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
Starting MQSeries Commands.

dis clusqmgr(*) connname status qmtype deftype
  1 : dis clusqmgr(*) connname status qmtype deftype
AMQ8441: Display Cluster Queue Manager details.
  CLUSQMGR(QM1)                CLUSTER(CLUS_1)
  CHANNEL(TO.QM1)              CONNAME(127.0.0.1(1414))
  DEFTYPE(CLUSSDRA)            QMTYPE(NORMAL)
  STATUS(RUNNING)
AMQ8441: Display Cluster Queue Manager details.
  CLUSQMGR(QM2)                CLUSTER(CLUS_1)
  CHANNEL(TO.QM2)              CONNAME(127.0.0.1(1415))
  DEFTYPE(CLUSRCVR)           QMTYPE(REPOS)
AMQ8441: Display Cluster Queue Manager details.
  CLUSQMGR(QM3)                CLUSTER(CLUS_1)
  CHANNEL(TO.QM3)              CONNAME(127.0.0.1(1416))
  DEFTYPE(CLUSSDRB)           QMTYPE(REPOS)
  STATUS(RUNNING)

```

Figure 35. Display Cluster Information (2)

Figure 35 shows two additional pieces of information:

- `QMTYPE` displays whether it is a repository queue manager (REPOS) or a not (NORMAL).
- `DEFTYPE` shows how the cluster channels were defined:
  - `CLUSSDRA` is specified for the sender channel to QM1. This channel has been defined automatically.
  - `CLUSRCVR` is specified for the receiver channel connected to QM3, to the other repository queue manager. This channel has been defined explicitly.
  - `CLUSDRB` is specified for the sender channel to QM3, which hosts the other repository. We defined an explicit definition for it. However, it is also automatically defined.

Examples of how to build clusters using `RUNMQSC` and the new GUIs for Windows NT are given in Chapter 4, “Creating a Cluster with the MQExplorer” on page 55 and Chapter 5, “Creating a Cluster with Scripts” on page 95.

---

## Chapter 3. MQSeries for Windows NT Version 5.1

In this chapter we describe briefly the new features of Version 5.1 and the changes from Version 5.0. The new version is closely integrated with Windows NT Version 4. The most important features are the two GUIs for administration, the MQSeries Explorer and MQSeries Services. MQSeries exploits the Microsoft Management Console (MMC) for that. Detailed information on how to use them is provided in Chapter 4, "Creating a Cluster with the MQExplorer" on page 55.

Here are some of the important features:

- The code path length for non-persistent messages has been optimized and can save up to 20%.
- There can also be a 10% reduction in CPU time when the improved Microsoft V5 compiler is used.
- The scope of the MQCONN connection handle is expanded to allow it to be shared by multiple NT threads.
- The installation has been simplified. It can now actually be called "easy". A default configuration can get you going quickly.
- Version 5.1 includes the MQSeries Information Center, which provides good help information and lets you read MQSeries literature online.
- A Web browser interface for system administration lets you administer MQSeries objects on the local and on remote servers.
- MQSeries Lotus Script Extensions (LSX) include MQSeries calls in Lotus Scripts running on Windows NT. You can connect to Lotus Notes and Domino from NT using MQSeries messaging.
- There are also interfaces to MQSeries services from ActiveX and Visual Basic.
- In addition to Windows V3.1 and Windows 95 clients, there is now a Windows NT client.
- All valid Windows NT user IDs are now allowed. A user ID can now be longer than 12 characters.
- The Windows NT Performance Monitor can be used to gather queue statistics.

---

## 3.1 Installation

Before you can install MQSeries for Windows NT Version 5.1 you have to install some prerequisite software. For the examples developed for this book we installed:

1. Microsoft Service Pack 3 for Windows NT Version 4  
(We used Service Pack 4)
2. Microsoft Internet Explorer 4.0.1 with Service Pack 1  
(We used Version 5)
3. Microsoft HTML Help 1.2 (hhupd.exe)
4. Microsoft Management Console (MMC) 1.1
5. Active Directory Service Interface (ADSI) V2.0
6. Adobe Acrobat V3.02

The last four items are on the MQSeries installation CD. The others you may download from the Web or you may obtain a CD.

The following dependencies should be taken into consideration:

1. The Web Administration Server needs the MQSeries Server.
2. The Internet Gateway needs the Windows NT Client.
3. The documentation in other languages need the documentation in the installation language.

### Important

- When you install the Internet Explorer you must select the Microsoft Virtual Machine or the GUIs will not work. You will get a message saying that the MMC cannot be initialized.
- If you want to use Web administration, you must select **Custom install** and select **Web administration**. This feature is not installed by default.
- You may want to increase the virtual memory to over 100 MB. Select **Start -> Settings -> Control Panel -> System**. Then click the **Performance** tab and change the virtual memory.

MQSeries installs by default into the directory \Program Files\MQSeries. We changed it to \MQM as it was in previous versions. We recommend that you install the programs and data in the same directory. This may avoid problems with initializing the MMC.

Figure 36 shows the MQSeries menu. Later we will discuss the items in more detail. But first let us mention that there are no INI files any more. Information that used to be in the INI files is now in the registry. All stanzas become keys.



Figure 36. MQSeries Menu

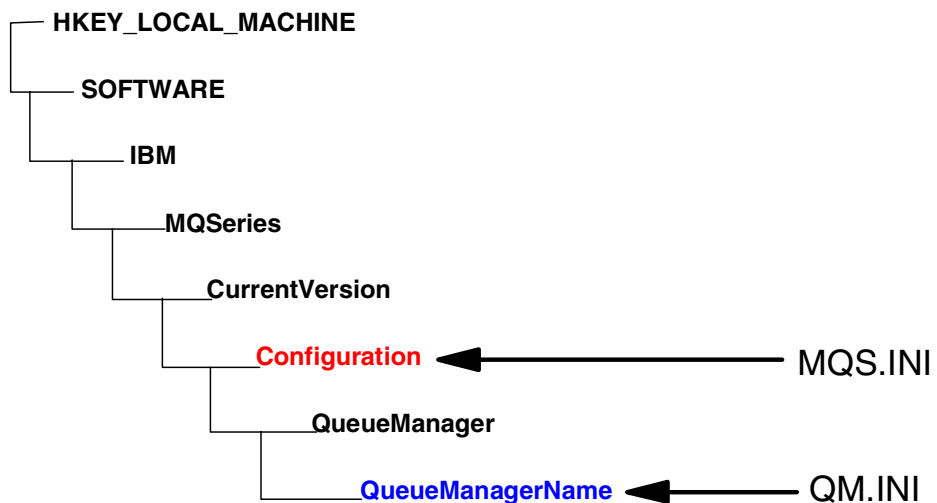


Figure 37. MQSeries Keys in Registry

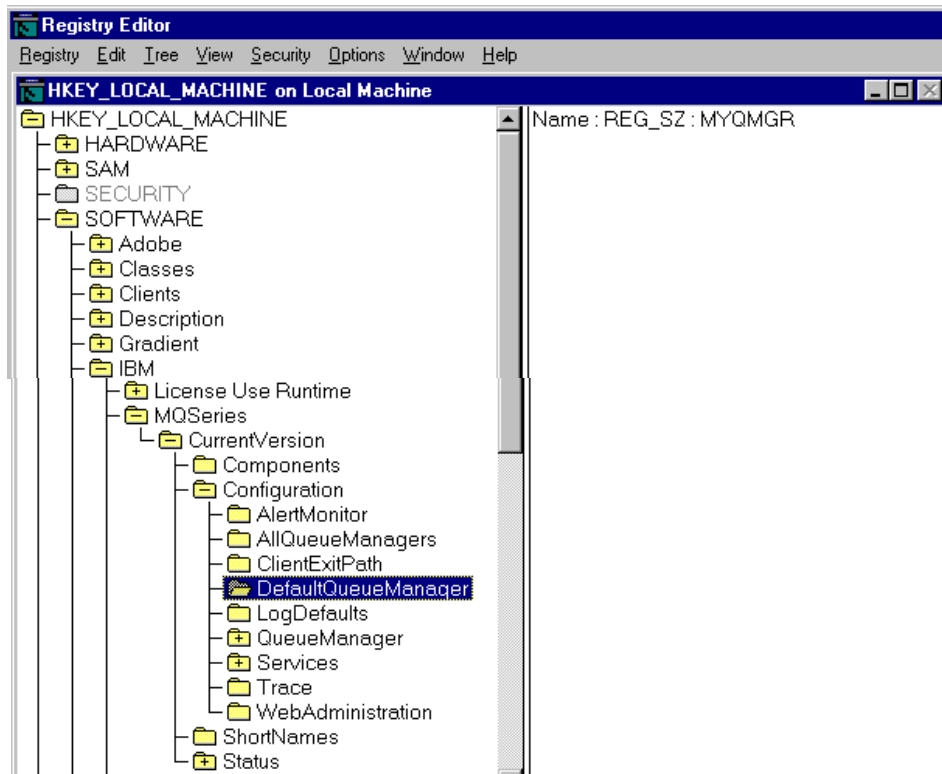


Figure 38. Registry Replacing MQS.INI File

Figure 38 shows the MQSeries configuration in the registry. Here the default queue manager is selected.

Figure 39 on page 37 shows registry keys for the queue manager, formerly stanzas in the QMgr.INI files. Here the logging-related values are displayed.

All values are stored as type REG\_SZ. You should not use REGEDIT to modify values in the registry. Use the MQServices Snap-in instead. If you must modify values by hand, use REGEDT32.

If a key has no values or subkeys, delete it. There should be no empty keys.

Figure 40 on page 37 shows the MQServices window. Through this window you should make your updates to the registry. The figure shows the tab with log file information.

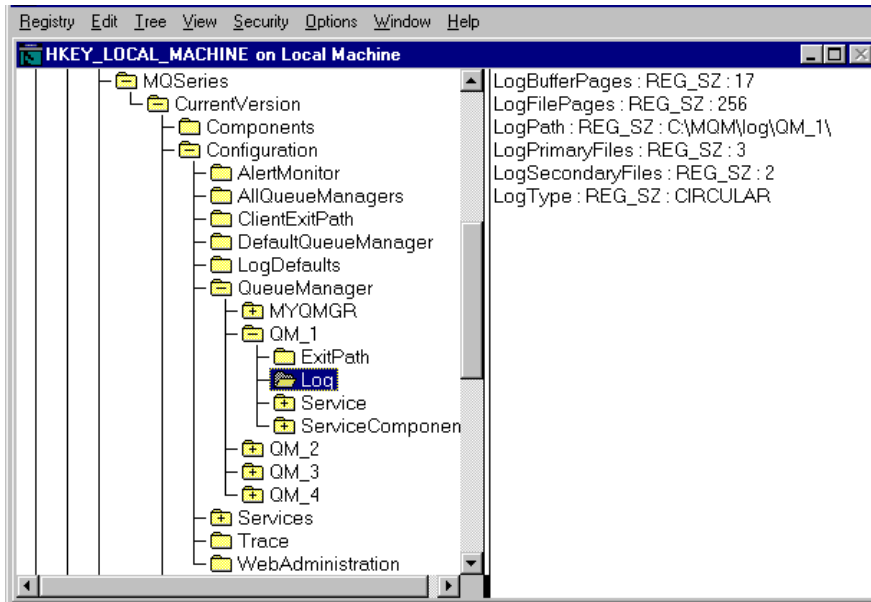


Figure 39. Queue Manager Keys in the Registry

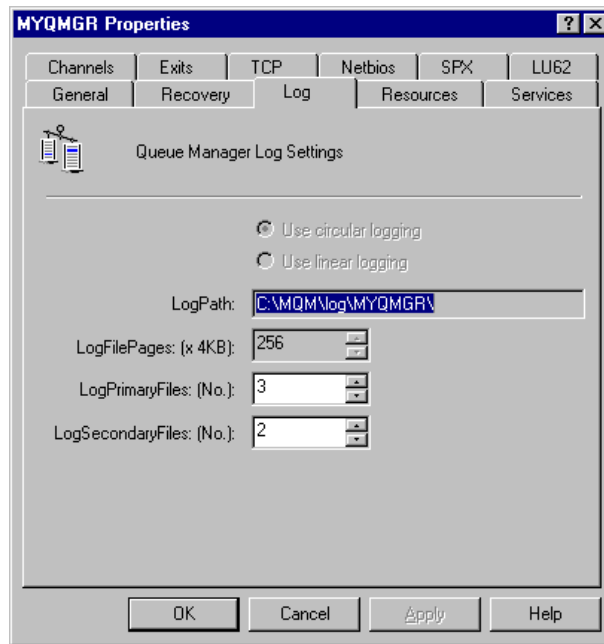


Figure 40. MQ Services Snap-In

---

## 3.2 MQSeries First Steps

For a short period of time in 1997, a product with this name could be downloaded from the MQSeries Web site on the Internet. We described this product in the redbook *Connecting the Enterprise to the Internet with MQSeries and VisualAge for Java*, SG24-2144. The new product available with MQSeries Version 5 has much improved. You can use it to get started with MQSeries and also use it as a tool to learn the MQSeries APIs.

When you start MQSeries First Steps you can choose from a list of options:

1. Default Configuration

This feature creates a queue manager and configures it for connecting with other computers on the same TCP/IP domain.

**Note:** This option can also be invoked from the installation menu.

2. Quick Tour

This feature demonstrates the basic concepts and functions of MQSeries. It provides an introduction to the MQSeries product and how to work with the MQSeries components, such as queue managers, cluster, queues, channels and listeners.

3. Postcard

You can use this program to try out messaging and communicate between different computers. This application uses the default configuration.

4. MQSeries Explorer

With this new GUI you can view and administer your MQSeries network. It is an alternative to `RUNMQSC`.

5. API Exerciser

With this tool you can explore the functions of messaging and queueing. A set of GUIs lets you execute the MQSeries APIs without writing one line of code.

6. Information Center

This gives you quick access to help information, reference material, and Web-based books and home pages. You can read the MQSeries manuals online.

In the following sections we discuss the above features in more detail.



---

### 3.3 Default Configuration

The default configuration is an application that creates a number of MQSeries objects to allow you to make use of the clustering abilities of Version 5.1 without having to set up the appropriate queues and channels.

The default configuration application can be invoked:

1. During the installation of Version 5.1
2. From MQSeries First Steps
3. From the Postcard application

The object names are based on the fully qualified TCP/IP machine name, for example, wtr05176.itso.ral.ibm.com. All machines must belong to the same TCP/IP domain. During the configuration process you have to specify a queue manager that will hold the repository.

You cannot create the default configuration under three circumstances:

- When DHCP addresses are used. Clusters cannot use the Dynamic Host Configuration Protocol.
- When queue managers already exist.
- When the DNS (Domain Name Server) is not set up.

The default configuration creates the following objects:

- Cluster
- Default queue manager
- Local queues with the names “default” and “postcard”
- Server connection channel
- Cluster sender channel
- Generic receiver channel
- Remote administration server connection channel with the name SYSTEM.ADMIN.SVRCONN.
- Listener on port 1414
- Channel initiator using the SYSTEM.CHANNEL.INITQ.

Some objects are named based on the domain name, the full local machine name, or the repository name. Queue names can be 48 characters long, channel names up to 20. Longer names will be truncated. If a truncated name

ends with a dot, this is also removed. Table 3 shows an example using the following names:

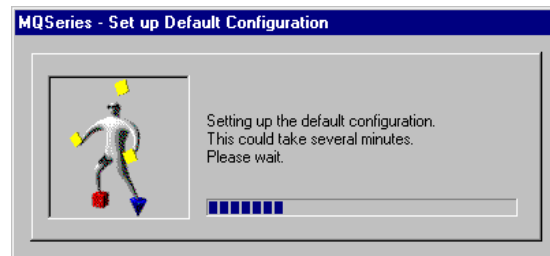
- Domain name is raleigh.ibm.com
- Local machine is dieter.raleigh.ibm.com
- Repository is in tony.raleigh.ibm.com

Table 3. Names in Default Configuration

Cluster	CL_domain_name <i>CL_raleigh.ibm.com</i>
Queue manager	QM_machine_name <i>QM_dieter.raleigh.ibm.com</i>
Server connection channel	S_machine name <i>S_dieter.raleigh.ibm.com</i>
Cluster sender channel	TO_repository_name <i>TO_tony.raleigh.ibm.com</i>
Generic receiver channel	TO_machine_name <i>TO_dieter.raleigh.ibm.com</i>

Follow these steps to create the default configuration:

1. Click **Start ->IBM MQSeries -> MQSeries First Steps -> Default Configuration**. This displays the GUI shown in Figure 41 on page 41.
2. Click **Set up Default Configuration** on the bottom left of the window.
3. Accept the defaults by clicking **Next**.
4. Mark the radio button to make the queue manager the repository queue manager. The setup may take several minutes.



5. Click **Close** when finished.

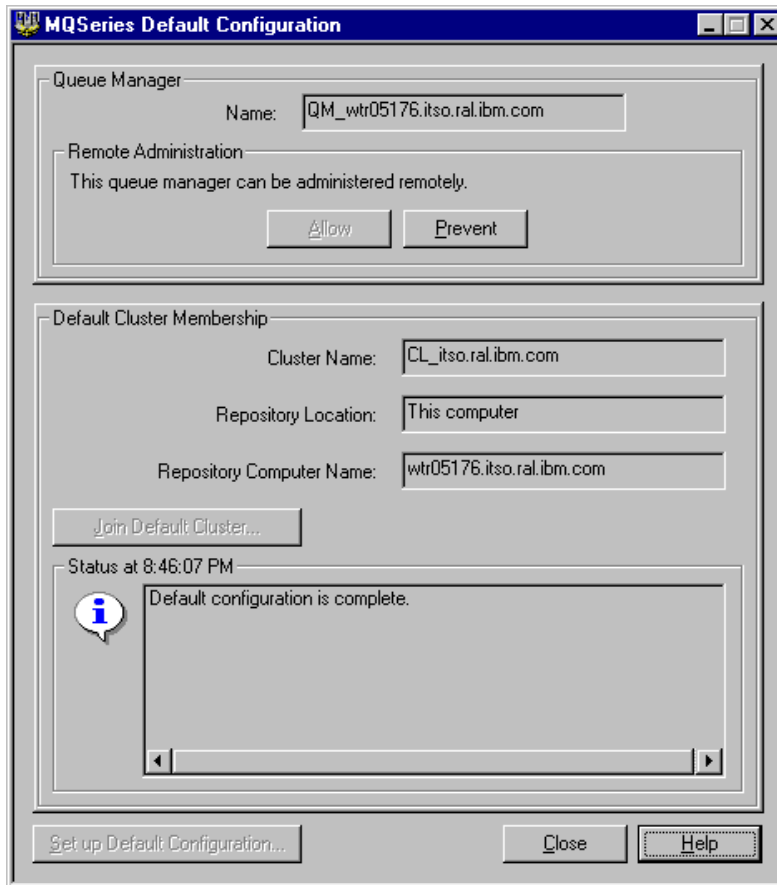


Figure 41. Creating the Default Configuration

Executing the `RUNMQSC` command `dis ql(*) cluster clusqmgr clusinfo` we see that the following queues are created:

```

CLUSTER(CL_itso.ral.ibm.com)          QUEUE(clq_default_wtr05176)
CLUSTER( )                            QUEUE(default)
CLUSTER( )                            QUEUE(postcard)
CLUSTER(CL_itso.ral.ibm.com)          QUEUE(clq_default_wtr05176)
CLUSQMGR(QM_wtr05176.itso.ral.ibm.com)

```

Figure 42 on page 42 shows the channels. We used the MQSeries Explorer to display them.

Name	Channel Type	Protocol Type	Connection Name
SYSTEM.ADMIN.SVRCONN	Server Connection	TCP/IP	
SYSTEM.AUTO.RECEIVER	Receiver	TCP/IP	
SYSTEM.AUTO.SVRCONN	Server Connection	TCP/IP	
SYSTEM.DEF.CLUSRCVR	Cluster Receiver	TCP/IP	
SYSTEM.DEF.CLUSSDR	Cluster Sender	TCP/IP	
SYSTEM.DEF.RECEIVER	Receiver	TCP/IP	
SYSTEM.DEF.REQUESTER	Requester	TCP/IP	
SYSTEM.DEF.SENDER	Sender	TCP/IP	
SYSTEM.DEF.SERVER	Server	TCP/IP	
SYSTEM.DEF.SVRCONN	Server Connection	TCP/IP	
S_wtr05176.itso.ral	Server Connection	TCP/IP	
TO_wtr05176.itso.ral	Cluster Receiver	TCP/IP	wtr05176.itso.ral...

Figure 42. Channels Defined by Default Configuration

### 3.4 MQSeries Postcard

This is a simple MQSeries application (amqpcard.exe) that can be used to verify the installation using the default configuration. It lets you send electronic postcards (see Figure 43 on page 43) to another instance of the program. The other instance can run on the same machine running under the same queue manager or on another machine in the same cluster. It is started from MQSeries First Steps.

If you want to run the Postcard application on your PC follow these steps:

1. **Click Start -> MQSeries First Steps -> Postcard.** The default configuration must exist and the queue manager must be running.
2. In the subsequent window enter a nickname, such as otto, and click **Enter**. This creates the first instance.
3. To create the second instance, start Postcard again and enter another nickname, such as hugo.



Figure 43. MQSeries Postcard

You will see two postcard images as shown in Figure 44 on page 44. When you click on the bottom right of the postcard you will see the other side. This is shown in Figure 43. If you have the right screen resolution the picture will even be animated.

4. Enter the nickname of the receiver (here hugo), type some text in the message field and click **Send**.

The program will fill in the name of the computer, display the message in the bottom of the window and send it to the other instance where it is also displayed on the bottom of the window.

5. To send a message back, type some text in the message area, click **Reply** to fill in the nickname of the receiver and then click **Send** to send the postcard.

You will see the reply message in both windows as you can see in Figure 45 on page 44.

6. You can double-click an entry in the list of sent and received postcards to view the message text.

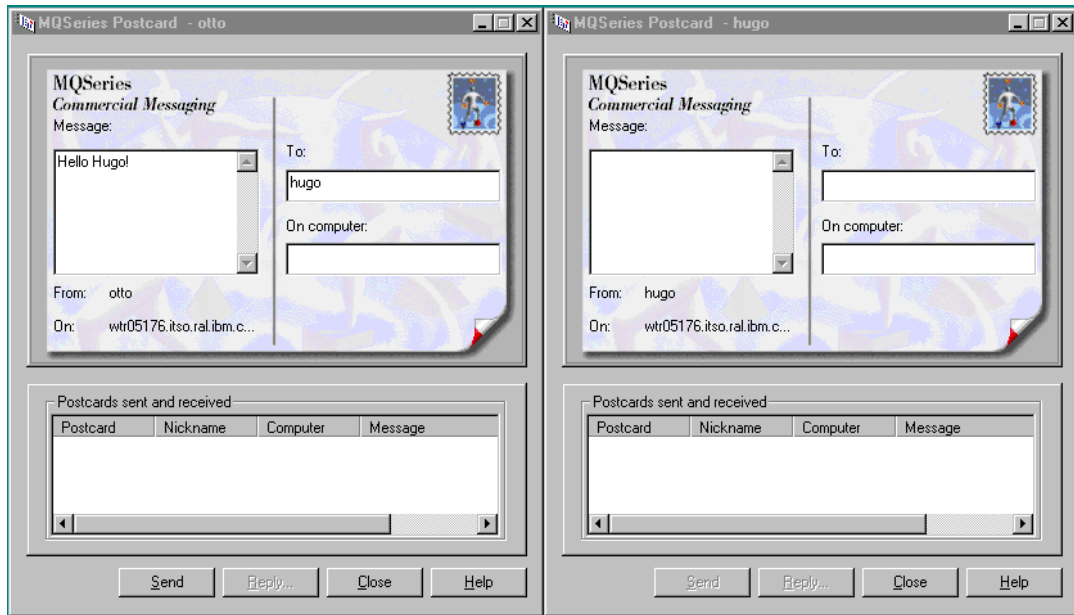


Figure 44. Sending a Postcard

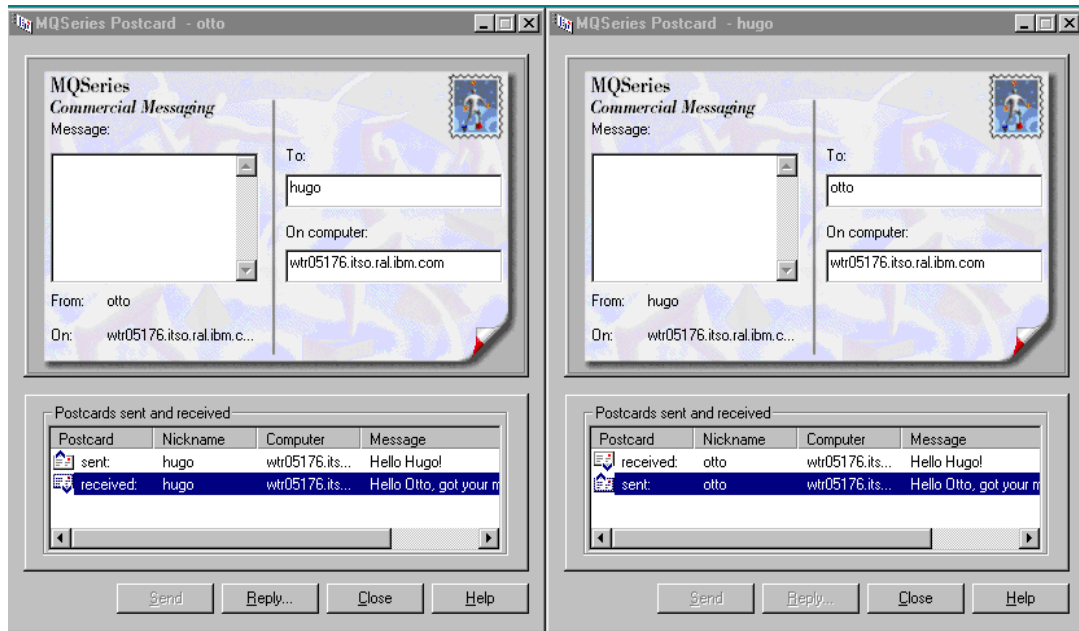


Figure 45. Receiving a Postcard

### 3.5 MQSeries Explorer

This Windows NT graphical user interface is concerned with *MQSeries objects*, while the other new GUI, MQSeries Services, is concerned with MQSeries processes.

The MQSeries Explorer obsoletes the use of control commands and `RUNMQSC` for creation and operation of multiple local and remote queue managers. It provides a convenient environment for experimentation and development. The product even includes a message browser. You can use the MQSeries Explorer to manually start and stop queue managers and channels. However, the product operates in interactive mode only; you cannot use any scripts.

Figure 46 shows the default configuration in the MQSeries Explorer window. You see the queue manager, the cluster, and what objects you can view.

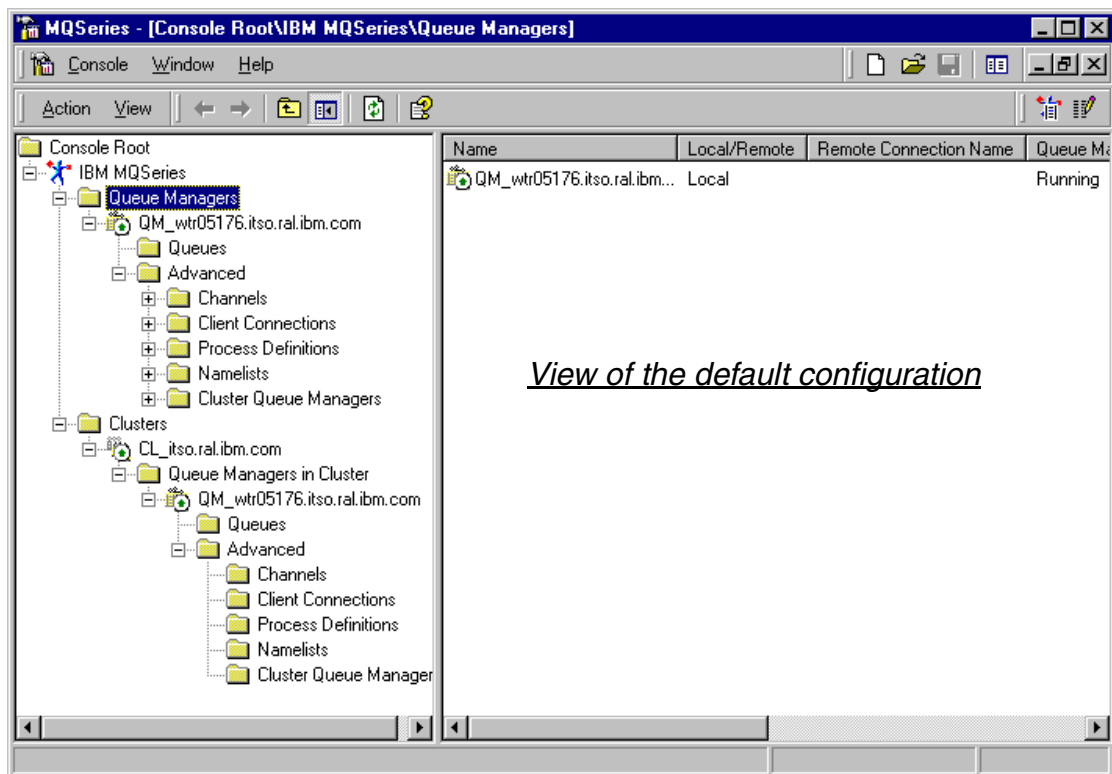


Figure 46. MQSeries Explorer

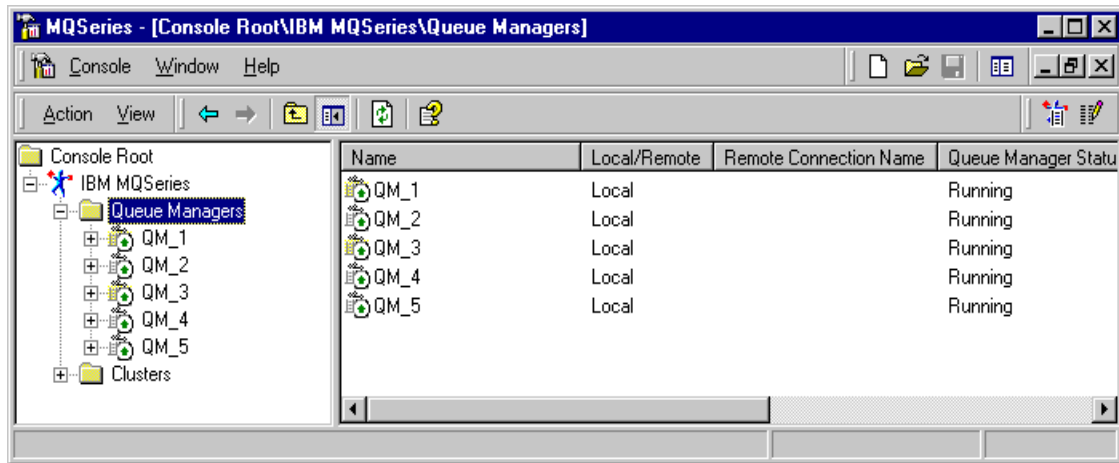


Figure 47. MQSeries Explorer - Multiple Queue Managers

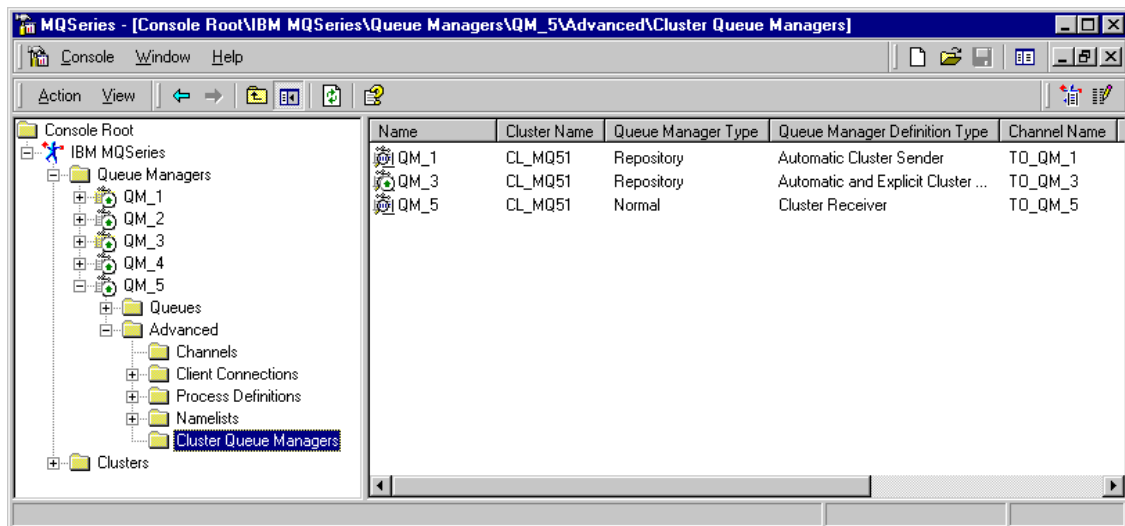


Figure 48. MQSeries Explorer - Cluster Queue Managers

The MQSeries Explorer uses only standard features of the MQI to obtain the objects it displays. Connections to other queue managers are running simultaneously. The connections are established when the tree in the windows is expanded for the first time. An explicit Connect option is available for subsequent use. The Explorer uses TCP/IP only and relies on the DNS (Domain Name System) server.



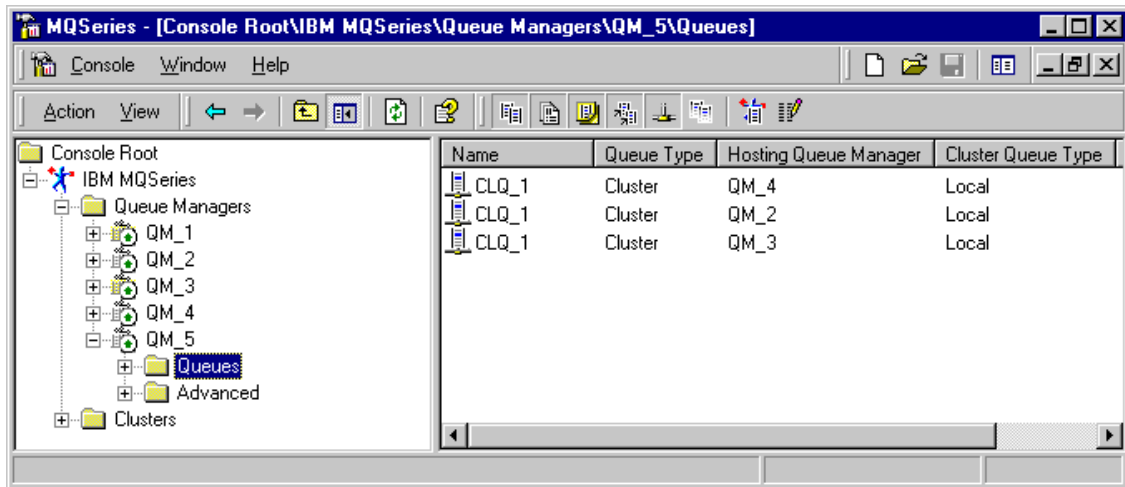


Figure 49. MQSeries Explorer - Cluster Queues

Even though a cluster is not an MQSeries object, it appears in the Explorer window as such. The Explorer presents information from many sources, that is, queue managers together. It sends PCF commands to command servers to obtain the information.

Figure 47 on page 46 shows that there are five queue managers running. Figure 48 on page 46 shows which queue managers are known to QM\_5. It knows itself, of course, and two full repository queue managers. QM\_2 and QM\_4 are not in the list because no messages have been sent to them or received from them, and, therefore, are not in its partial repository. Figure 49 shows that QM\_5 knows of three queue instances, all with the name CLQ\_1. They are hosted by three different queue managers.

We will explain the MQSeries Explorer in more detail in the next chapter when we describe how to create a cluster.

### 3.6 MQSeries Services

Like the MQSeries Explorer, MQSeries Services is a snap-in to the Microsoft Management Console (MMC). MQSeries Services is concerned with *MQSeries processes*. For example, you can use MQSeries Services to define what processes to start automatically at boot time, and whether a process should restart after it has failed. Notice that the command `SCMMQM` does not exist any longer. This program works in interactive mode only.

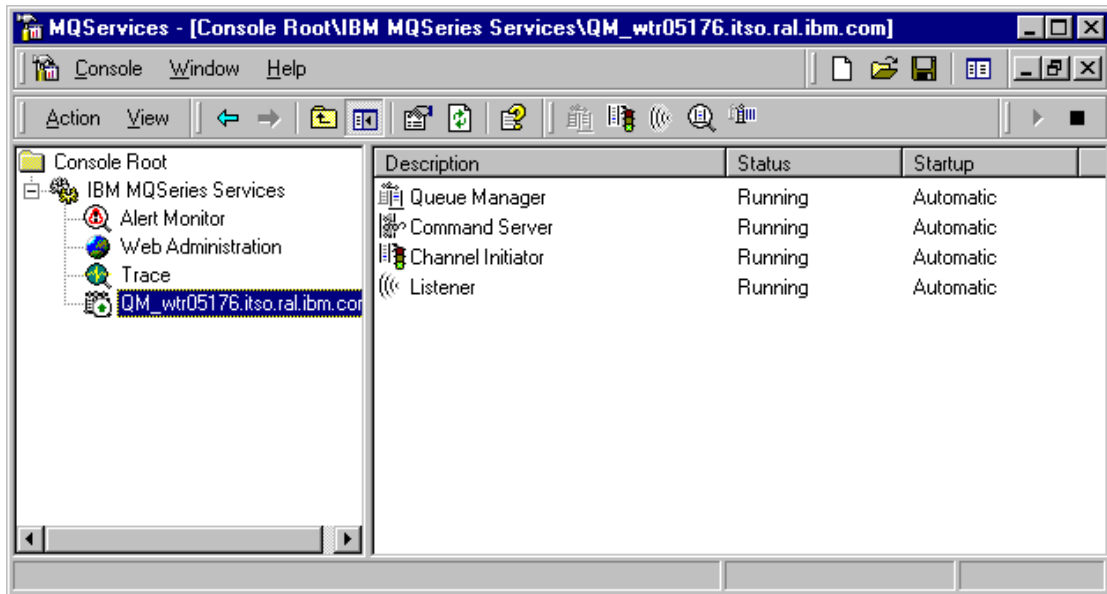


Figure 50. MQSeries Services

Figure 50 shows the MQSeries Services window showing the processes for the default configuration. You see that the queue manager is running and so are the command server, channel initiator and listener. You also see that the window includes an alert monitor, trace, and Web Administration.

From this window, you can do the following:

- Create, delete, start and stop processes
- Display and change properties (for example, to update the registry)
- Define recovery procedures

MQSeries Services works with the following objects:

- Queue managers
- Listeners
- Trigger monitors
- Channel initiators
- Web administration
- Command Server
- Alert Monitor
- Trace

### 3.7 MQSeries API Exerciser

This feature has been developed to provide novice MQSeries programmers with a basic understanding of how the 13 API calls work. The API exerciser product is NLS compliant and has two modes, basic and advanced. The MQINQ and MQSET APIs, for example, can only be executed in advanced mode. This mode also allows you also to test all API parameters.

This GUI is user friendly and intuitive; it incorporates a full HTML help as well as intelligible completion and return code interpretation.

This features requires that the command server is running. It supports clusters, but does not support segmented messages. Also, clients are not supported.

Figure 51 shows one of the GUIs. You can use it to connect and disconnect to a queue manager and to select more APIs by clicking on the tabs.

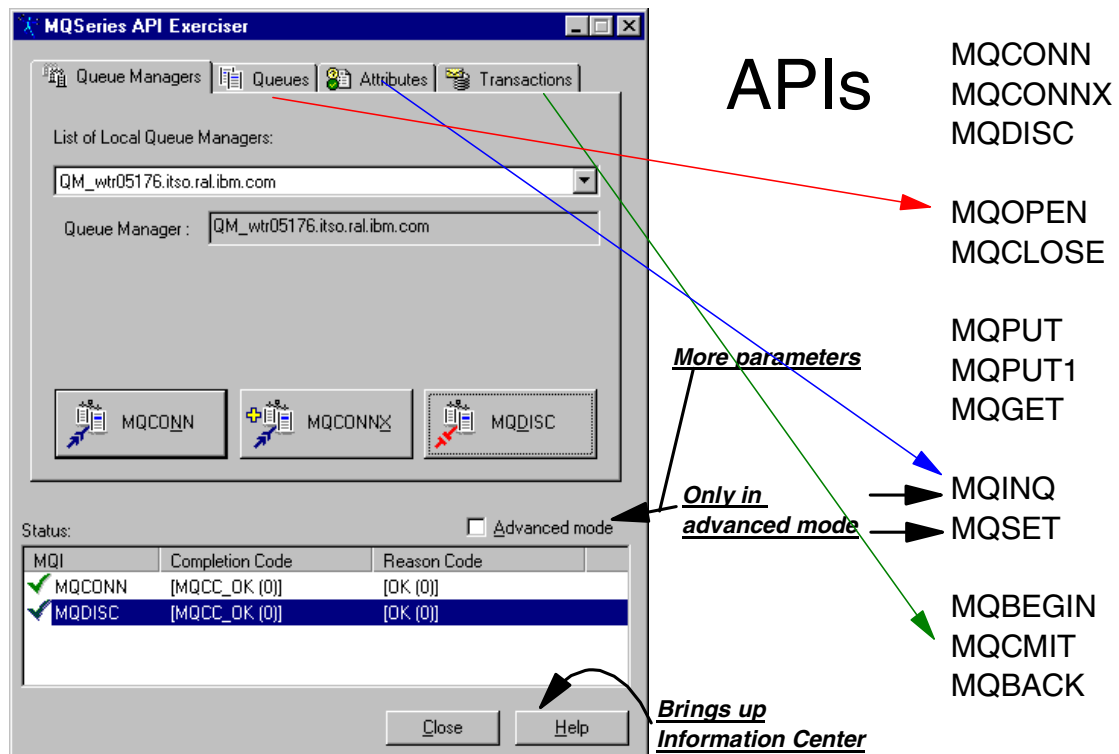


Figure 51. API Exerciser

Figure 51 on page 49 lists the 13 APIs. To work with MQINQ and MQSET you have to check Advanced mode. The four tabs let you work with four different kinds of APIs:

- Queue Managers** From this panel you connect to or disconnect from a queue manager. You can choose one of the existing queue managers from the drop-down list. You select the API by clicking one of the three push buttons. The scroll box on the bottom displays whether the call was successful.
- Queues** From this panel you can open and close queues and put or get messages.
- Attributes** From this panel you can execute MQSET and MQINQ calls.
- Transactions** When you put or get messages under syncpoint you can use this panel to commit them or roll them back. MQBEGIN supports database coordination.

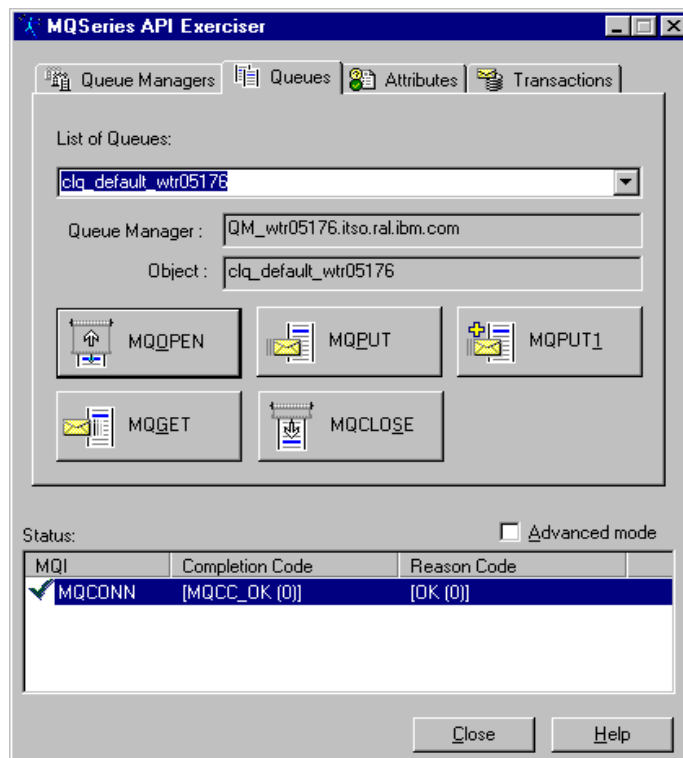


Figure 52. API Exerciser - Work with Queues

Figure 52 on page 50 shows the Queues panel. You select a queue from the drop-down list and then click **Open**. The return code will be displayed on the bottom of the panel. The you can click **MQPUT** and enter some text in the subsequent message window as shown below.

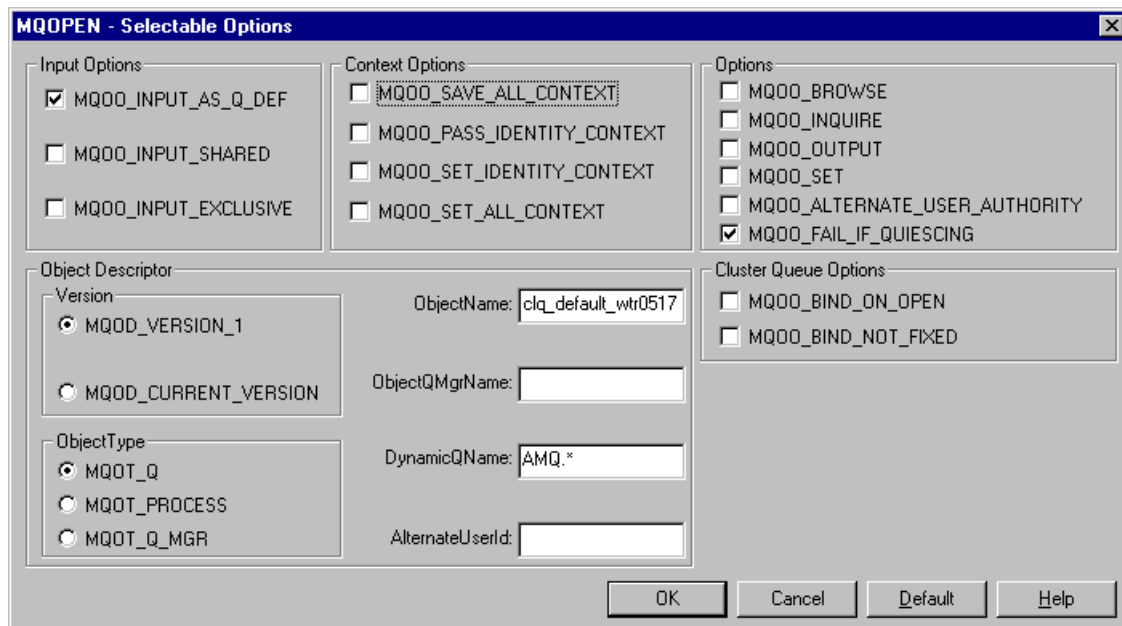
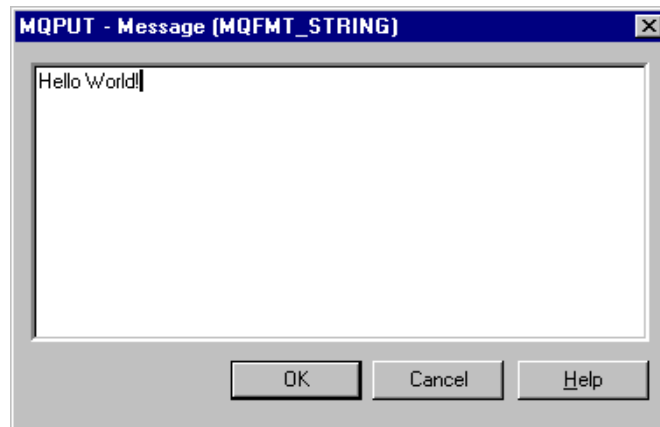


Figure 53. API Exerciser - Open Options

If you don't want to use the defaults for the MQOPEN, click **Advanced mode** and you are presented with the window shown in Figure 53 on page 51.

If you choose the same mode for the MQPUT, you will see the window below. Here you can enter the message text and also set values in the message descriptor and put message options.

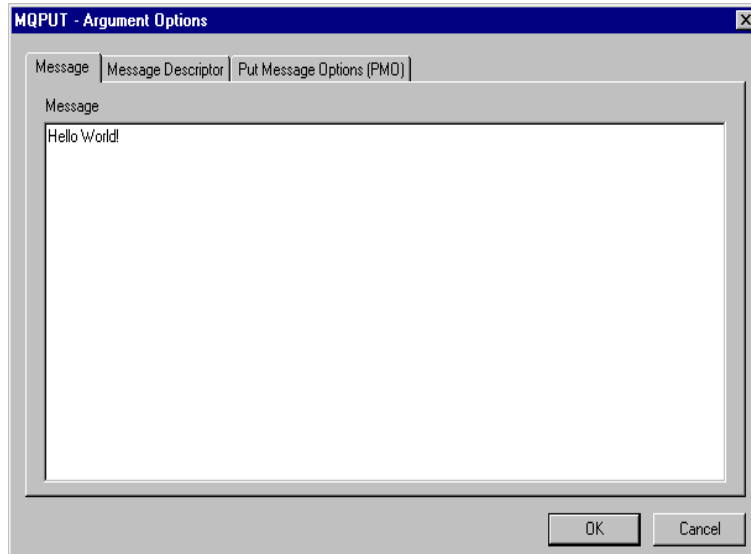


Figure 54 on page 53 shows the open options you can choose. For example, here you specify if you want to bind on open for a cluster queue.

Figure 55 on page 53 shows the first of five put option panels. Here you select what message type you want to send, for example, a reply or request message, and whether the message is persistent or not.

There are similar options for the other APIs. They are described, in detail, in *MQSeries Application Programming Reference*, SC33-1673.

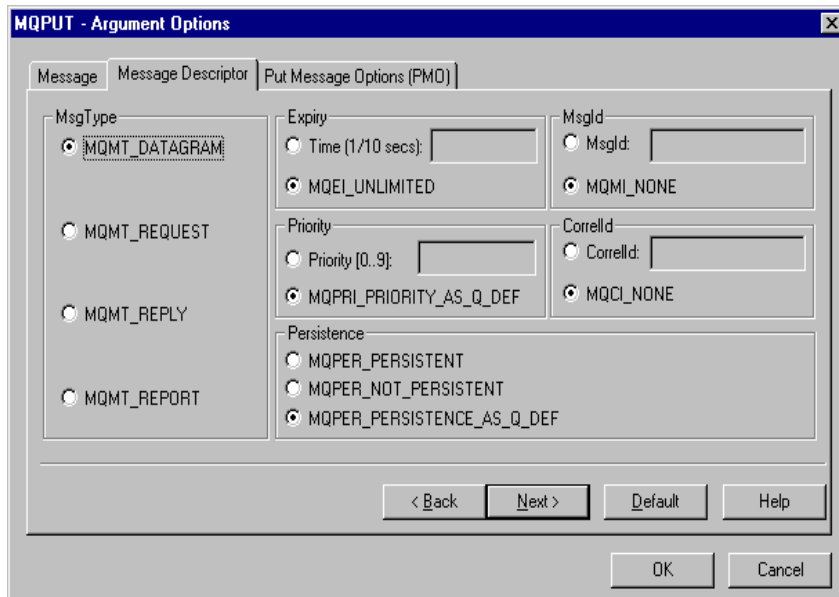


Figure 54. API Exerciser - Message Descriptor (One of Five Panels)

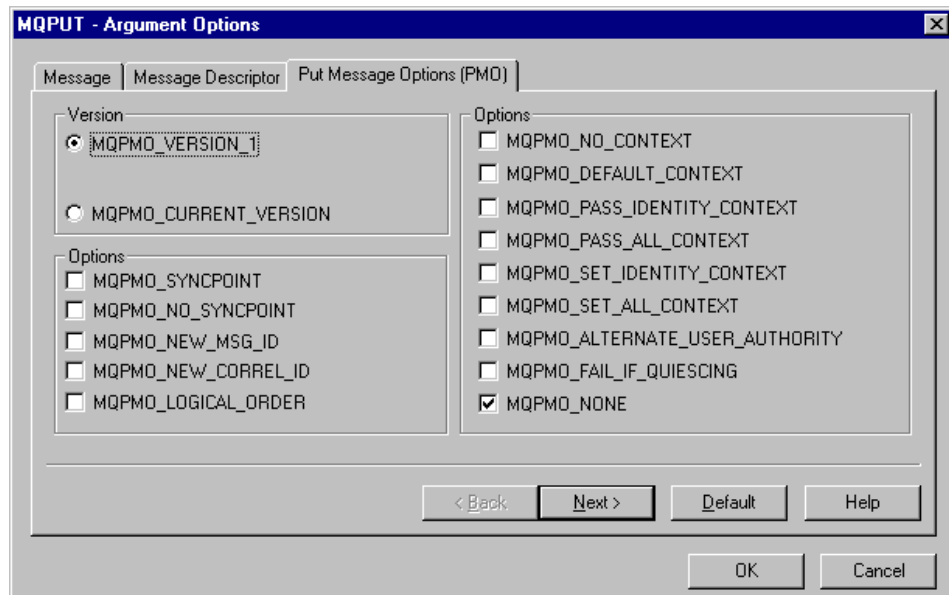


Figure 55. API Exerciser - PMO Options





## Chapter 4. Creating a Cluster with the MQExplorer

In this chapter we describe how to create a cluster of four queue managers on the Windows NT platform. The purpose of the following exercises is to introduce you to:

1. The MQExplorer interface
2. Some clustering and scripting ideas. Refer to SupportPac MC73 for some further ideas in this area.
3. Some ideas about how workload balancing works.

Figure 57 on page 56 shows the environment that we will create. Notice that not very many channel definitions will be made by us. This is one of the attractive administrative features of clusters; hand-made definitions are minimal and the cluster does the rest.

We will be building this environment twice, first using the MQExplorer and then using RUNMQSC. This is described in Chapter 5, "Creating a Cluster with Scripts" on page 95.

**Note:** Be aware that changes are not refreshed immediately within the MQSeries Explorer. Changes take time to be reflected through the cluster. When you have made changes and they do not appear on the screen, click the **Refresh** button. Figure 56 shows where the Refresh button is found (just above the mouse cursor).

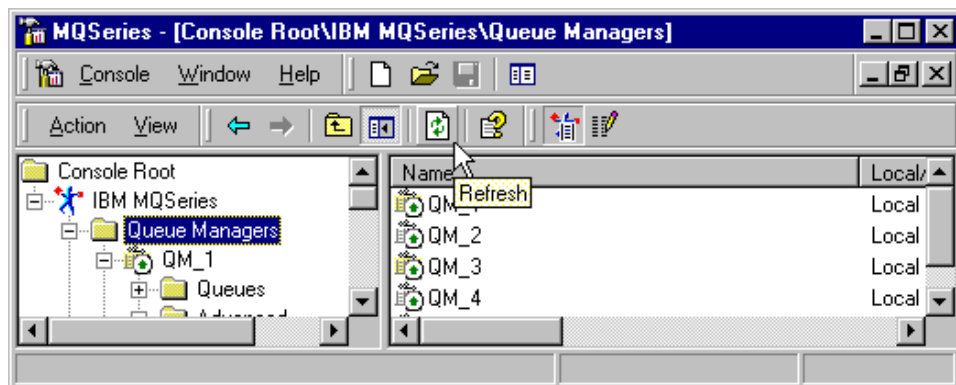


Figure 56. MQSeries Explorer - Refresh Button

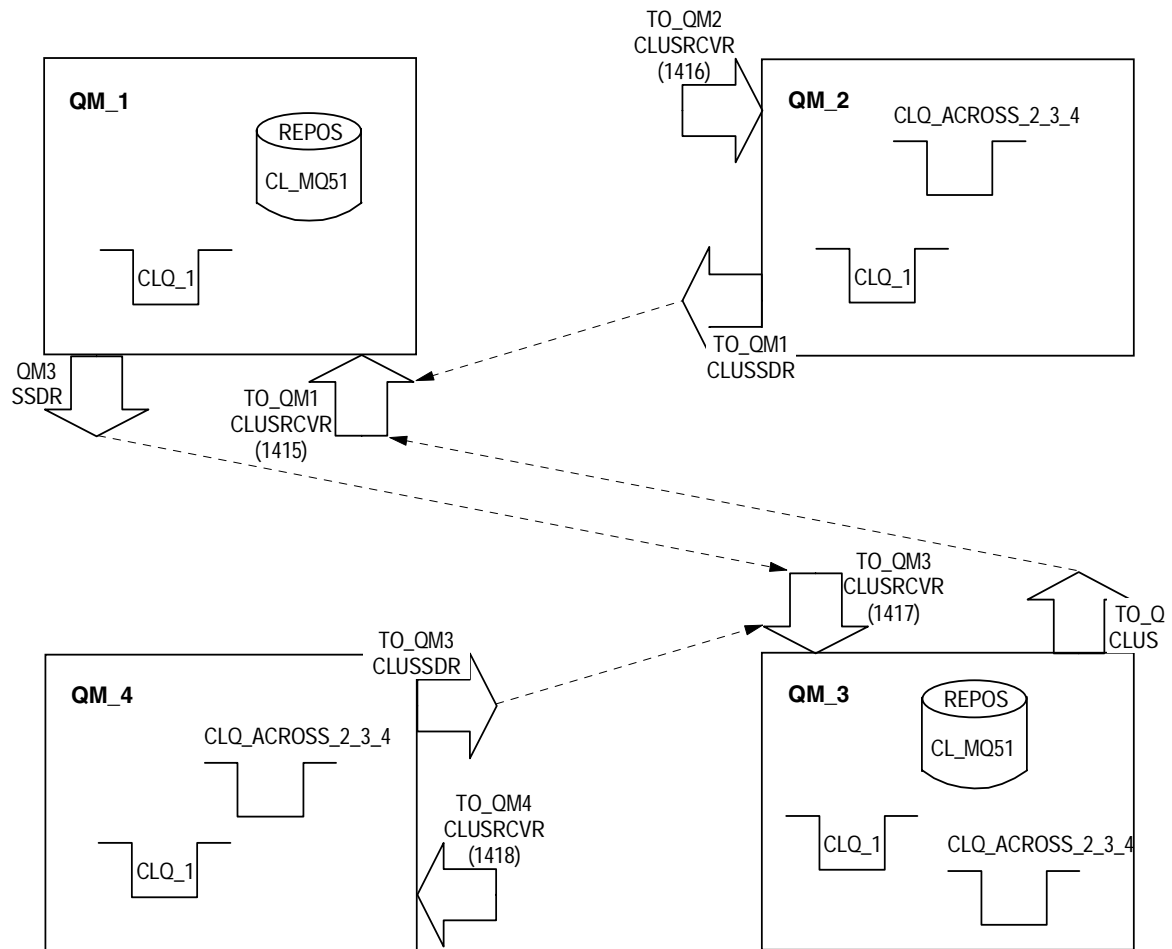


Figure 57. Cluster with Four Queue Managers

In this example, we build four queue managers in a single cluster. In the illustrations that follow, all the queue managers are built on the same physical machine. If you want to follow the instructions to create your own cluster, build at least one of the queue managers on a different machine, if possible. Bear in mind also that having multiple queue managers on a single machine implies that we will be using TCP/IP listener ports other than 1414.

If you have built the default configuration, or if you have already created a queue manager, then the MQExplorer window you see will be different from the one shown in Figure 58 on page 57.

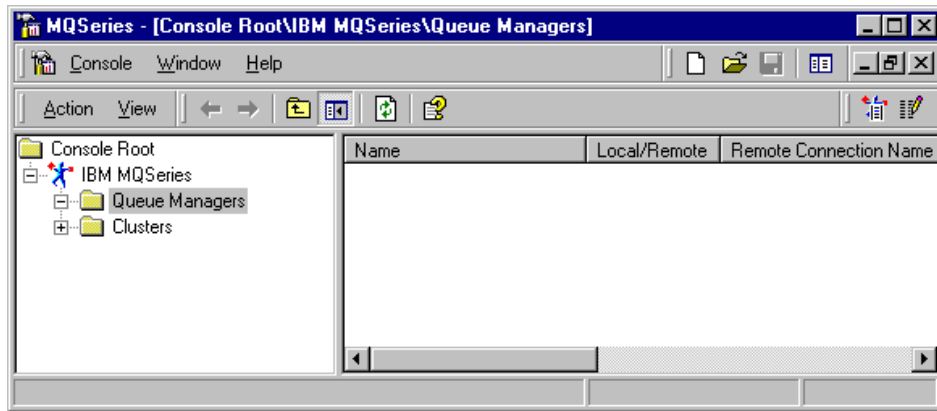


Figure 58. MQSeries Explorer - No Queue Managers Created Yet

In Figure 59 through Figure 63, all queue managers and clusters have been deleted. Feel free to retain any definitions you already have, but you will have to allow for the differences between the figures in this text and your own MQExplorer screen.

## 4.1 Creating the Queue Managers

First, let us create the first of the four queue managers, QM\_1. This is done with the following steps:

1. Right-click **Queue Managers** and select **New** and then **Queue Manager**, as shown in Figure 59 on page 58. This brings up the window shown in Figure 60 on page 58.
2. Type the queue manager name and SYSTEM.DEAD.LETTER.QUEUE as the name of the dead-letter queue. To avoid any errors in managing the cluster, we avoid having any default queue managers. So don't mark the check box. To continue, click **Next**.
3. In step 2, shown in Figure 61 on page 59, leave the settings as they appear and click **Next** again. Since we created a development environment, circular logging is fine. In a production environment, however, you would always use linear logging.
4. In step 3, shown in Figure 62 on page 59, select both check boxes. **Start Queue Manager** causes the queue manager to start automatically when you boot the system. **Create Server Connection Channel** allows for remote administration in this queue manager.

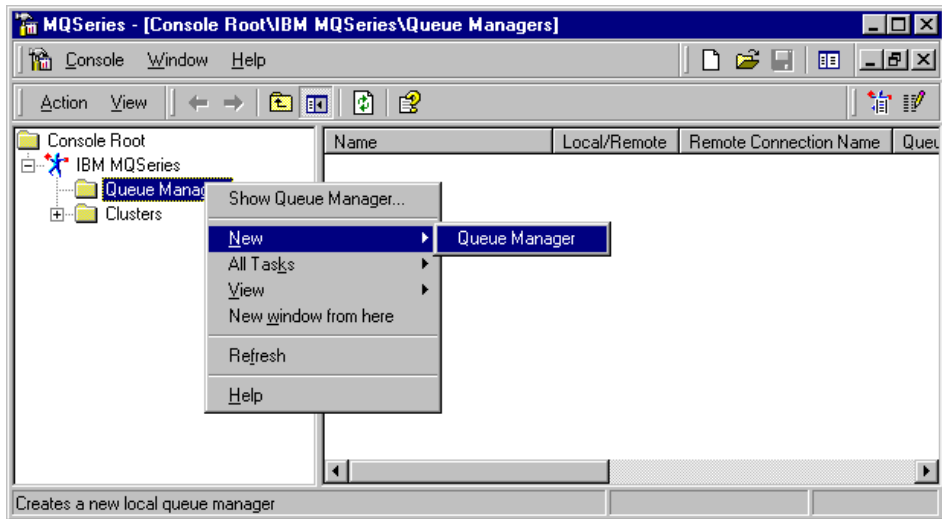


Figure 59. Creating a Queue Manager

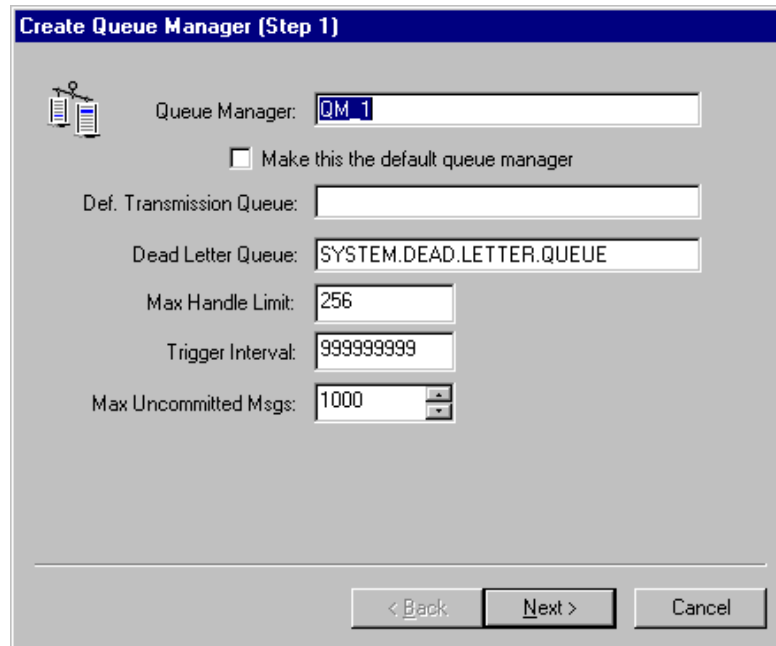


Figure 60. Creating a Queue Manager - Step 1

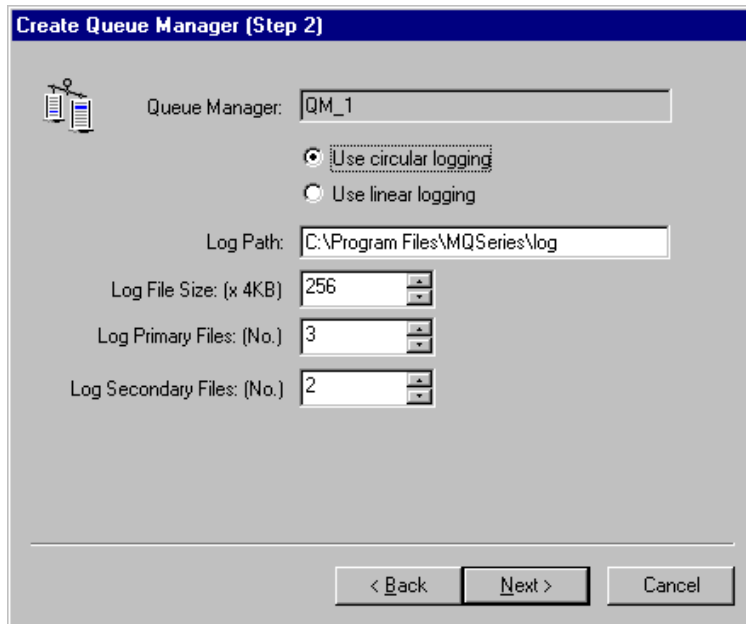


Figure 61. Creating a Queue Manager - Step 2



Figure 62. Creating a Queue Manager - Step 3

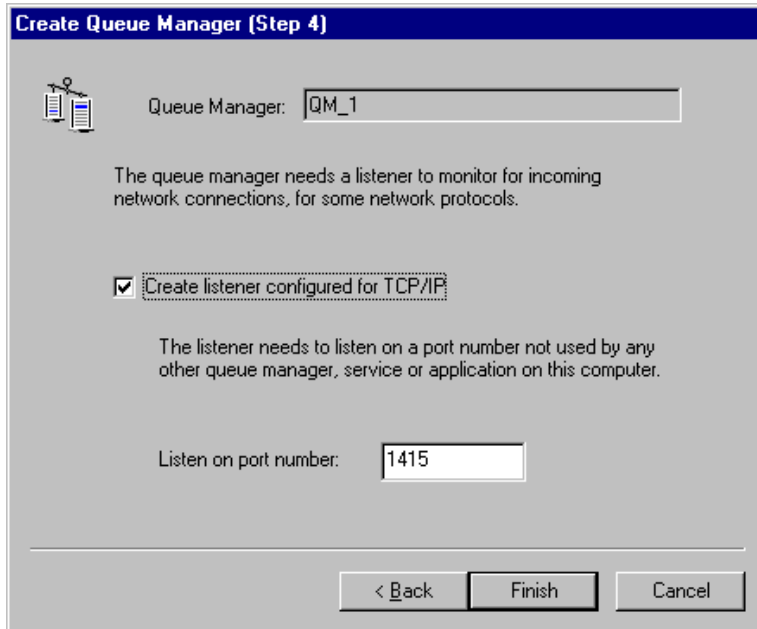


Figure 63. Creating a Queue Manager - Step 4

- In step 4, shown in Figure 63, you can have the MQExplorer create and maintain the listener for the queue manager. For this example, we mark this check box. The listener will be started automatically, too.

When considering the Listen On Port Number field, bear in mind that we will build the cluster in the same physical machine. Having multiple queue managers on a single machine implies that we will be using TCP/IP listeners ports other than 1414. We will use the following ports:

Queue Manager Name	TCP/IP Port
QM_1	1415
QM_2	1416
QM_3	1417
QM_4	1418

Table 4. Queue Managers and Port Numbers

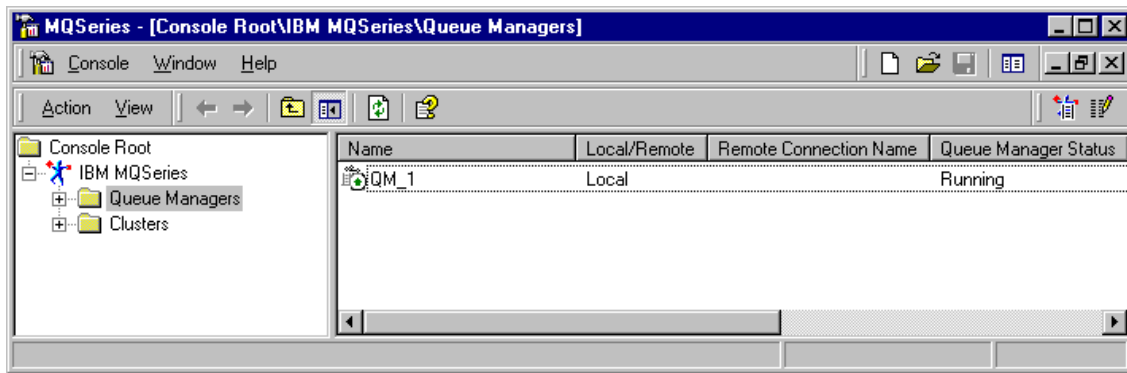


Figure 64. Creating a Queue Manager - Completed

When you have finished all of the above, your MQSeries Explorer window should look like the windows shown in Figure 64. Now let us look at the objects that have been created for you:

1. Expand **Queue Managers**.
2. Expand **QM\_1**.
3. Expand **Advanced**.
4. Click **Channels**.

You will see no channels unless you explicitly want system objects shown. To do that:

5. Click **View** on the menu bar.
6. Click **Show System Objects**.

Now you see the channels as shown in Figure 65 on page 62.

7. If you click **Queues**, you will see the windows shown in Figure 66 on page 62.

There are two new channels and three new queue definitions in Version 5.1:

- SYSTEM.DEF.CLUSSDR channel
- SYSTEM.DEF.CLUSRCVR channel
- SYSTEM.CLUSTER.COMMAND.QUEUE
- SYSTEM.CLUSTER.REPOSITORY.QUEUE
- SYSTEM.CLUSTER.TRANSMIT.QUEUE

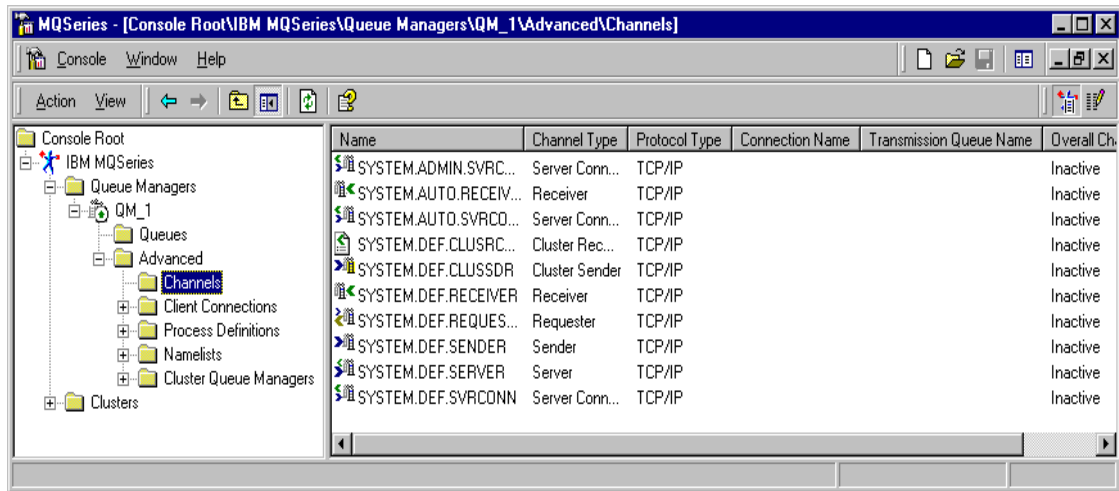


Figure 65. Creating a Queue Manager - Automatically Created Channels

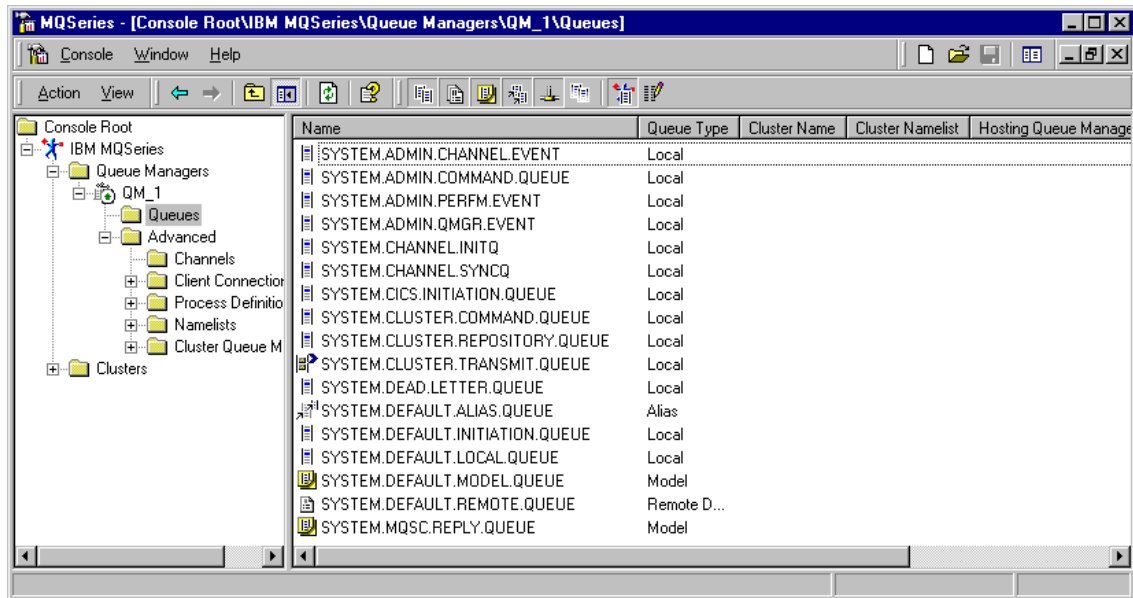


Figure 66. Creating a Queue Manager - Automatically Created Queues

Now, create the other three queue managers, QM\_2, QM\_3 and QM\_4, in exactly the same way. However, use the ports from Table 4 on page 60.



### Important

No two queue managers can share the same IP address *and* the same port number.

When you have created all four queue managers, your MQSeries Explorer window should contain the information shown in Figure 67. All queue managers should be running.

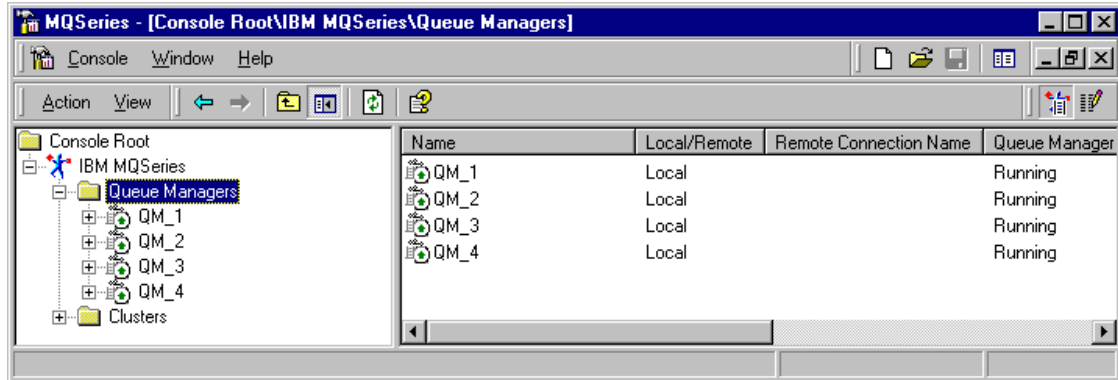


Figure 67. MQSeries Explorer Showing Four Queue Managers

How can you verify that you entered the right port numbers?

1. Click **Start ->Programs -> MQSeries Services**.
2. Expand **MQSeries Services**. You will now see the four queue managers.
3. Double-click the first queue manager.
4. Right-click the *listener* and select **Properties** from the menu.
5. Click the **Parameters** tab.
6. Verify the port.
7. Repeat steps 3 through 6 for the other three queue managers.

**Note:** On UNIX, you have to edit the etc/services file to define the port.

## 4.2 Creating a Cluster with Two Repository Queue Managers

At this time we have four queue managers, but we did not build the cluster yet. To create a cluster, click **Cluster** and follow these steps:

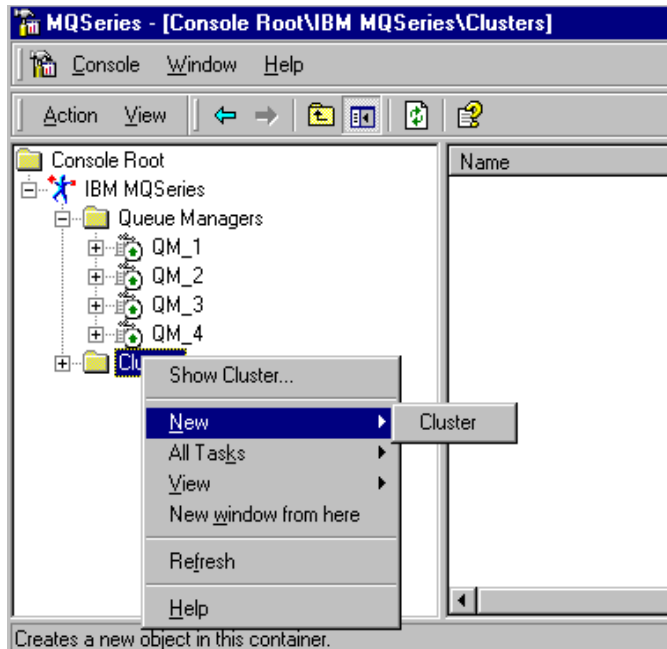


Figure 68. Creating a Cluster

1. Select **New -> Cluster**. This is shown above.
2. You will see the Create Cluster Wizard as in Figure 69 on page 65. You don't have to enter anything in this window. Read what the wizard does for you and click **Next**.
3. Now you have to name the cluster. As you can see, Figure 70 on page 65, we called our cluster CL\_MQ51. For this example, do likewise. Then click **Next**.
4. In the next window (Figure 71 on page 66) you are asked to enter the first repository queue manager. For this exercise, we let QM\_1 maintain the first repository. This is also the default.

Whether you select the check box for local or remote depends on where you have defined your queue manager. Since all our queue managers are on the same machine, we selected **Local on this computer**.

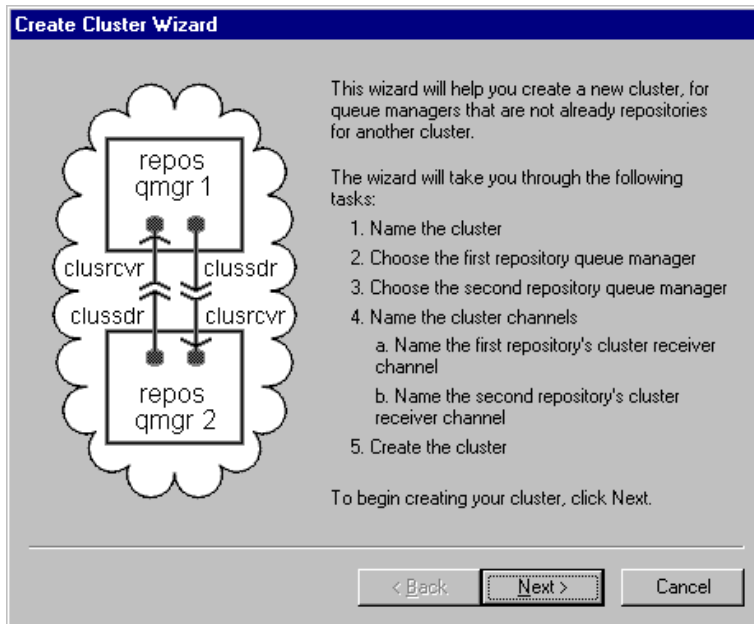


Figure 69. Create Cluster Wizard

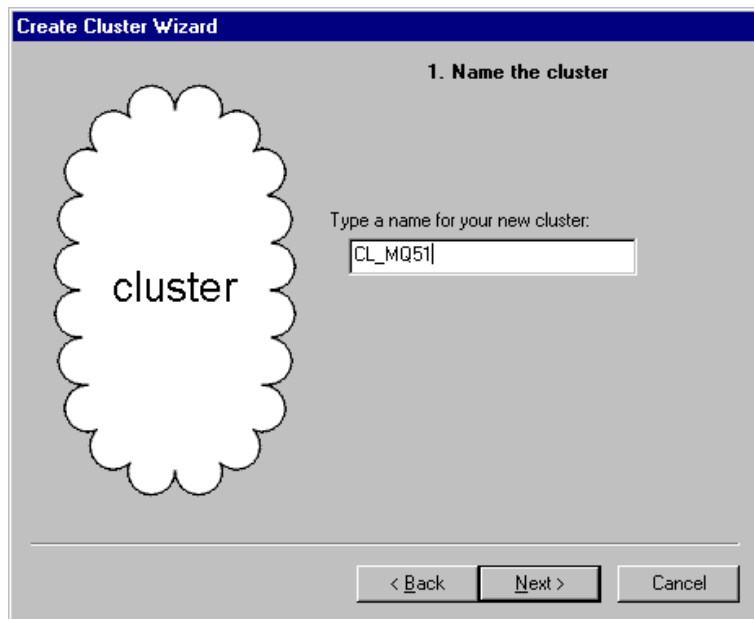


Figure 70. Create Cluster Wizard - 1

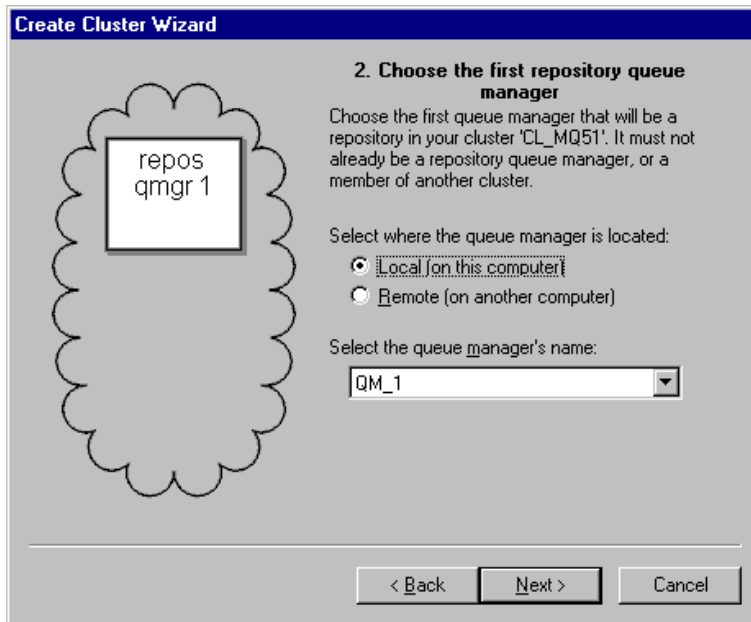


Figure 71. Create Cluster Wizard - 2

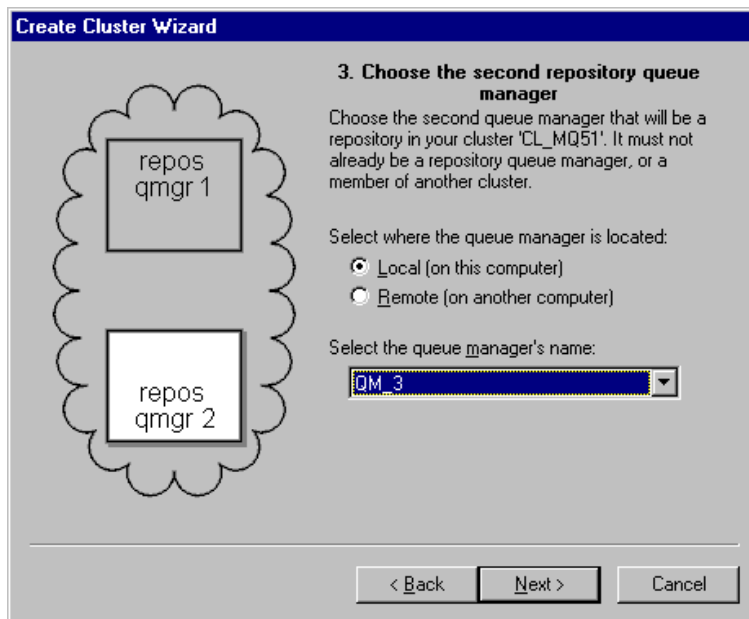


Figure 72. Create Cluster Wizard - 3

*Is the button Local (on this computer) disabled?*

If yes, you are using a DHCP server to obtain a TCP/IP address. Change the network settings before you create the cluster.

5. Click **Next** again and you will be asked to choose the second repository queue manager as in Figure 72 on page 66. For this exercise, choose QM\_3 from the drop-down list as the second repository. Also, mark for this queue manager **Local on this Computer**. All our queue managers are local. Then click **Next** again.
6. In the next window, Figure 73 on page 68, the wizard gives you some hints on naming cluster channels. Click **Next**.
7. Now the wizard will present window 4a shown in Figure 74 on page 68. Here you specify the cluster receiver channel for QM\_1, our first repository queue manager. We have chosen use TO\_QM1.

The Cluster receiver connection name is of the form <machine name>(<port>), where <machine name> is either a TCP/IP address or should resolve to one. The <port> is the IP port on which the listener for the queue manager (whose cluster receiver channel we are defining) is listening, such as WTR05246.itso.ral.ibm.com(1415).

8. Click **Next** when you have entered the cluster receiver name, and the wizard will take you to window 4b to name the second repository's cluster receiver channel. As you can see in Figure 75 on page 69, we followed the same pattern as for the first repository and named the channel TO\_QM3. Notice that the port is 1417.
9. Clicking **Next** again takes you to the last wizard window, shown in Figure 76 on page 69. You can read the summary and when you click **Finish** the cluster will be created.
10. Figure 77 on page 70 shows you the result that you should now see in the MQSeries Explorer window. Note that there are only two queue managers in the cluster at this stage. Remember, we have only added the repository queue managers. The next step will be to add the other two queue managers to the cluster, but not full repositories.

The channels TO\_QM1 and TO\_QM3 in both QM\_1 and QM\_3 should be running.

*How can you find out whether a queue manager holds a repository?*

Right-click a queue manager name, select **Properties** and then on the **Cluster** tab.

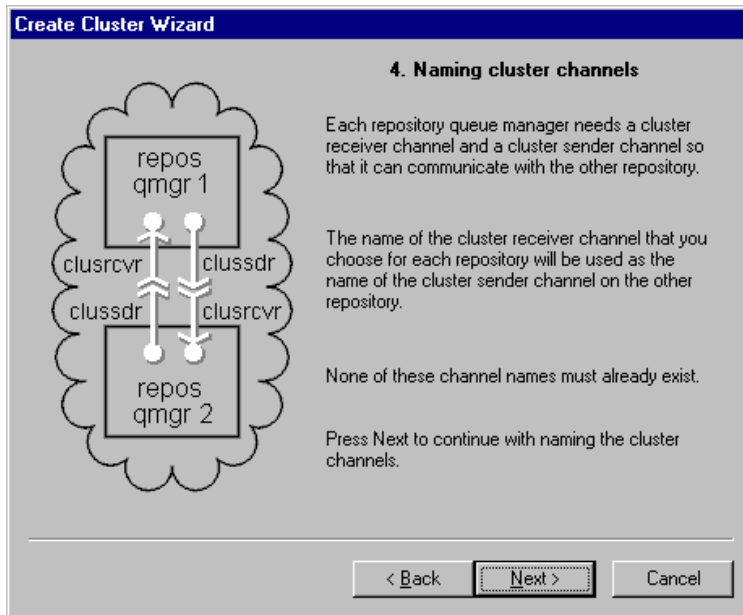


Figure 73. Create Cluster Wizard - 4

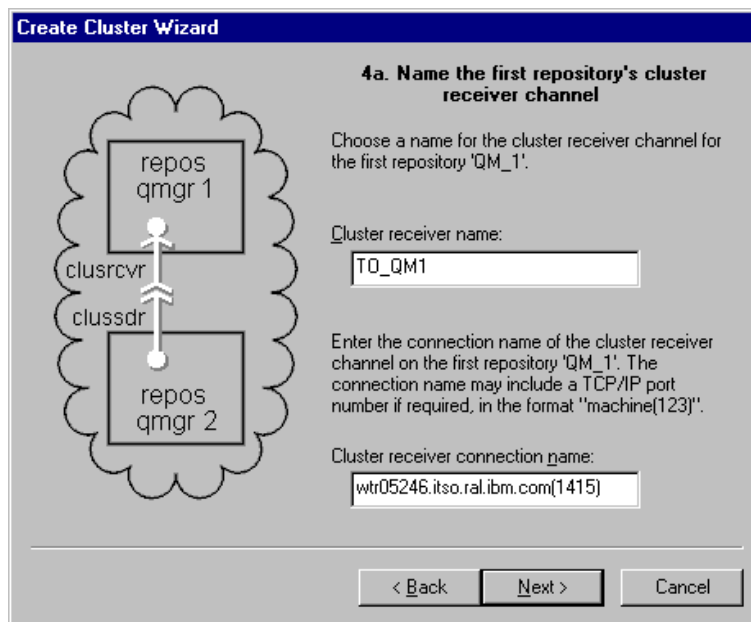


Figure 74. Create Cluster Wizard - 4a

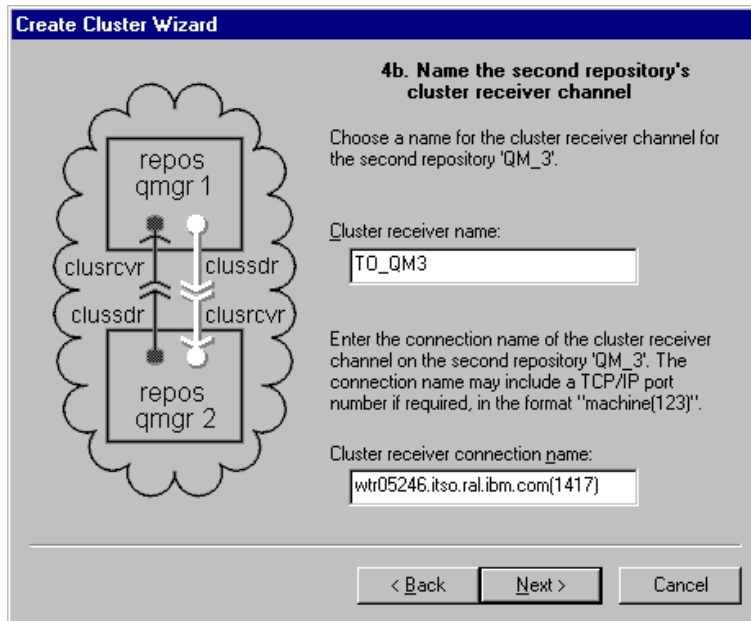


Figure 75. Create Cluster Wizard - 4b

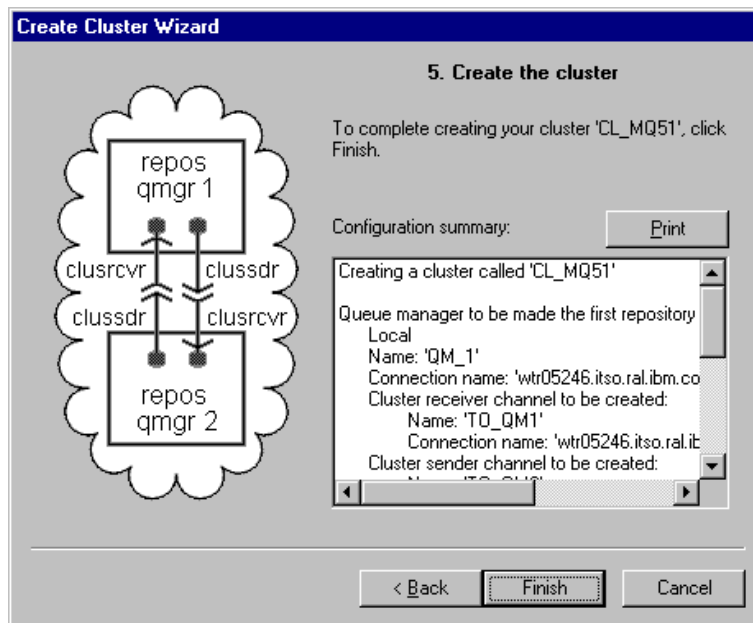


Figure 76. Create Cluster Wizard - 5

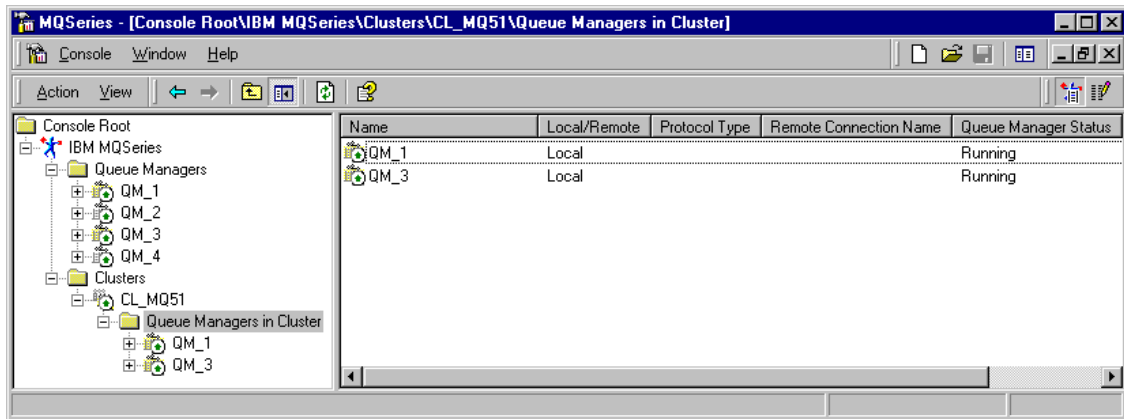


Figure 77. Creating a Cluster - Repository Queue Managers

### 4.3 Joining Queue Managers to a Cluster

Now the other two queue managers will join the cluster as partial-repository queue managers. To add a queue manager to a cluster, follow these steps:

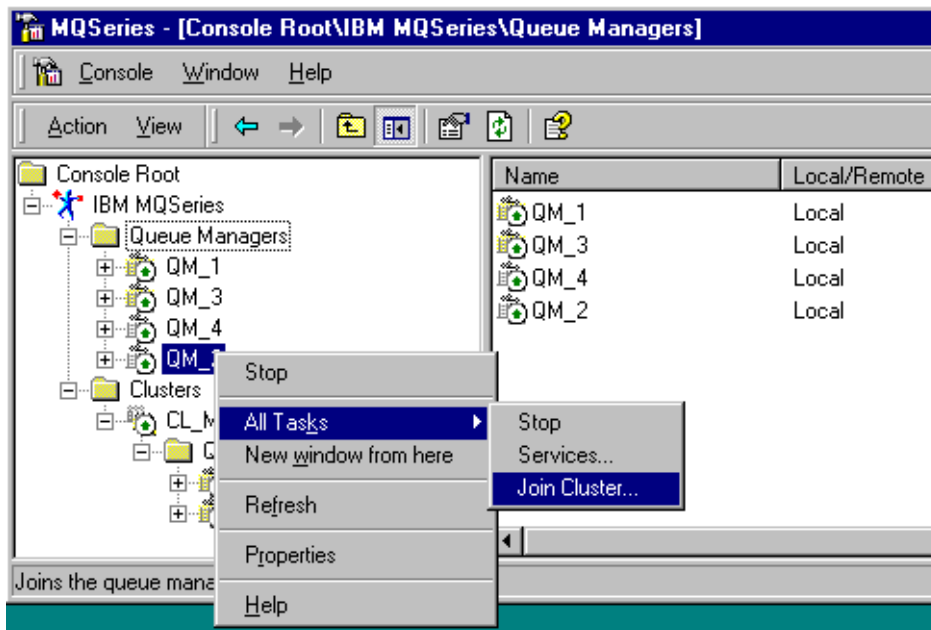


Figure 78. Joining a Cluster



1. First, we add QM\_2 to the cluster. From the MQSeries Explorer window, right-click **QM\_2** under **Queue Managers**, select **All Tasks ->Join Cluster**, as shown in Figure 78 on page 70.
2. Not surprisingly, this launches the Join Cluster Wizard, as shown in Figure 79. Click **Next** when you have finished reading what the wizard is about to do for you.

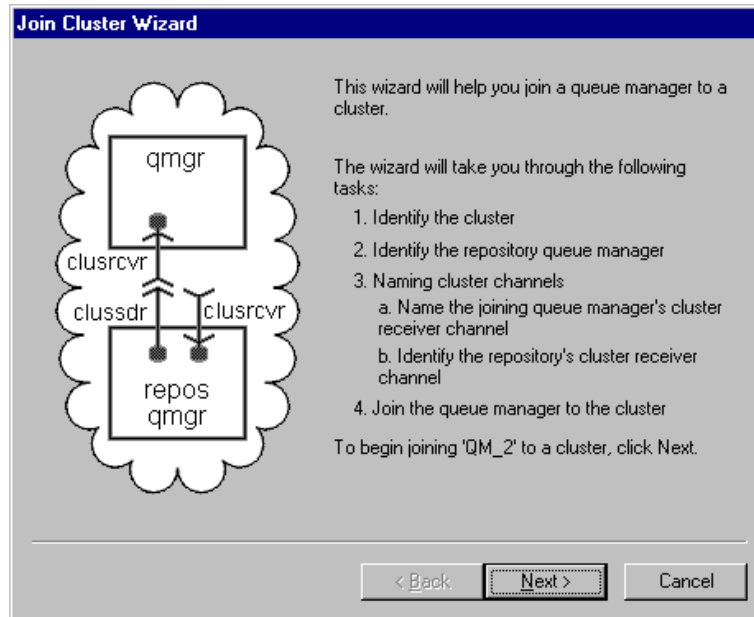


Figure 79. Join Cluster Wizard

3. The second window, shown in Figure 80 on page 72, requires that you enter the cluster name. This is CL\_MQ51. Then click **Next**.
4. Now you have to identify the *repository queue manager* for QM\_2. The window is shown in Figure 81 on page 72. You could choose QM\_1 or QM\_3 here, since both are full repositories. We chose QM\_1, as you can see. As in previous steps, the selection of Local or Remote will depend on whether you are creating your four queue managers on a network of machines, or a single machine, and whether there happens to be a full-repository queue manager on your local machine or not. If you select **Local**, as we did, a list of possible repositories will appear. If you select **Remote**, you will have to enter the queue manager's name and the queue manager's connection name.

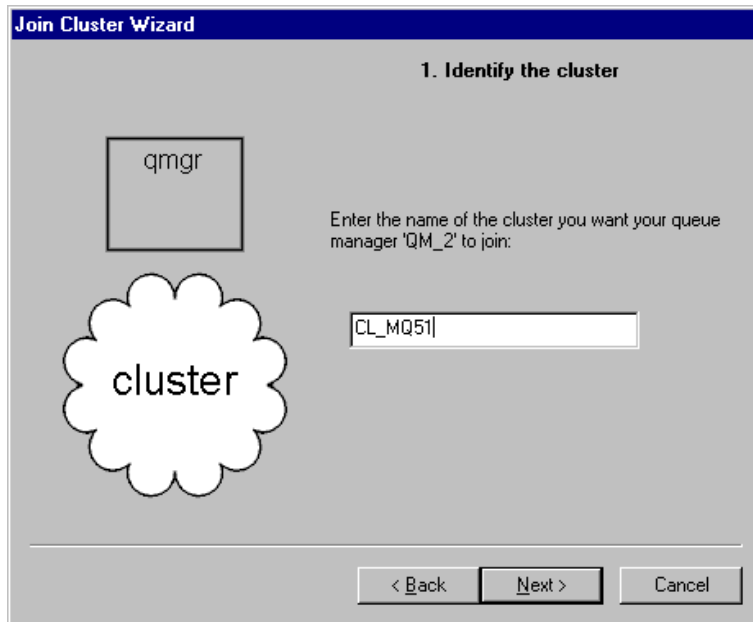


Figure 80. Join Cluster Wizard - 1

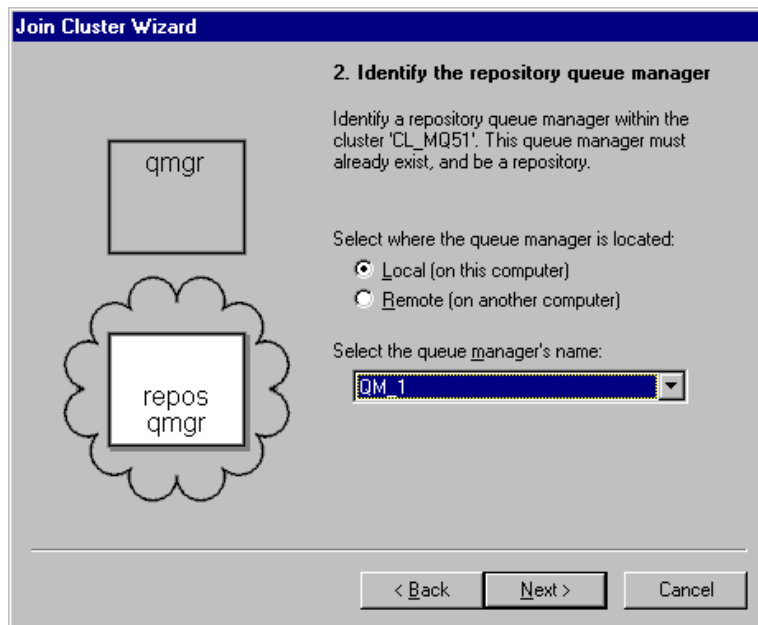


Figure 81. Join Cluster Wizard - 2

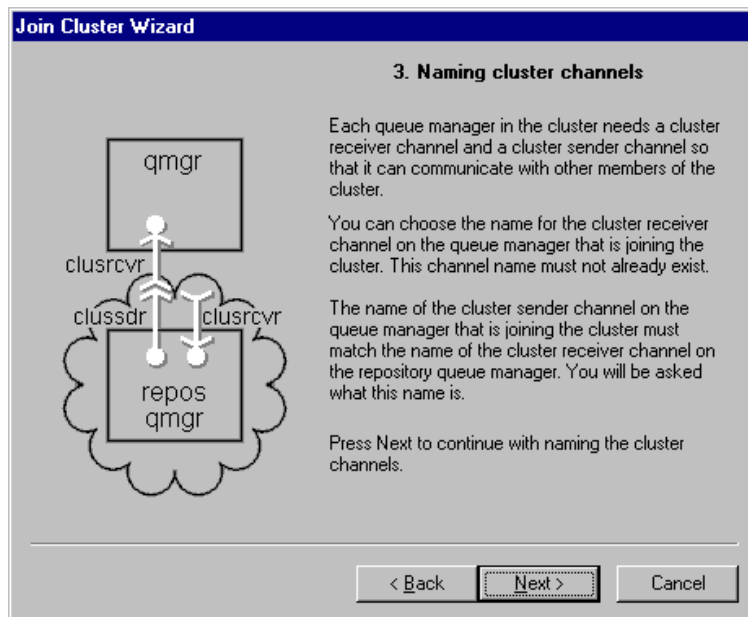


Figure 82. Join Cluster Wizard - 3

5. Clicking **Next** will bring you to the Naming cluster channels window, as shown in Figure 82. Read what the wizard has to tell you and then click **Next** again.

**Note:** Don't worry about the comment that you will have to remember the name of the cluster sender channel on the joining queue manager and matching it to the cluster receiver channel of the repository, the naming convention is very straight-forward and intuitive.

6. You have now arrived at window 3a. Name the joining queue manager's cluster receiver channel, as shown in Figure 83 on page 74. As the cluster receiver name, type `TO_QM2`. The wizard will have suggested the name `TO_2`, but if you have been following our naming scheme so far, type `TO_QM2`. The Cluster receiver connection name is the *IP address* (or resolvable name) and *listening port* of the joining queue manager.
7. Click **Next** and you will see the Identify the repository's cluster receiver channel window, as shown in Figure 84 on page 74. Once again, if the repository is local to the queue manager that is joining, things are very simple because the wizard correctly selects the name as **TO\_QM1** or **TO\_QM3**, depending on which full-repository you chose for this "joining session".

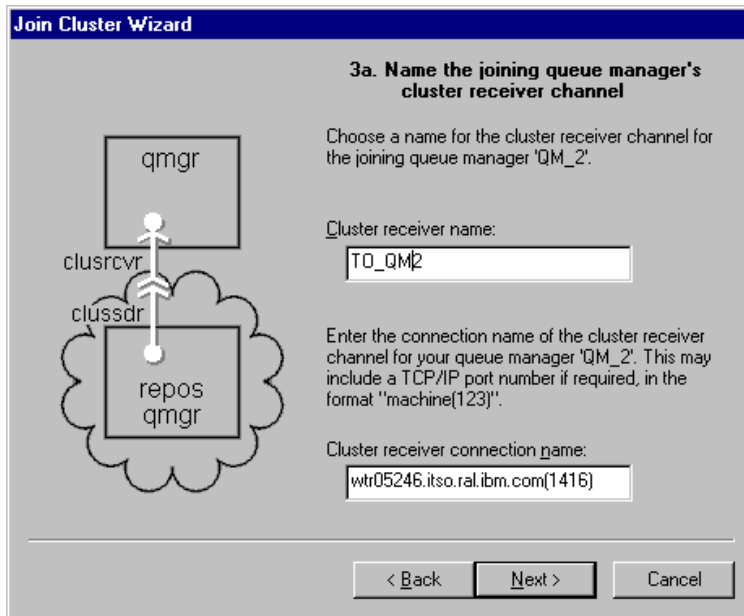


Figure 83. Join Cluster Wizard - 3a

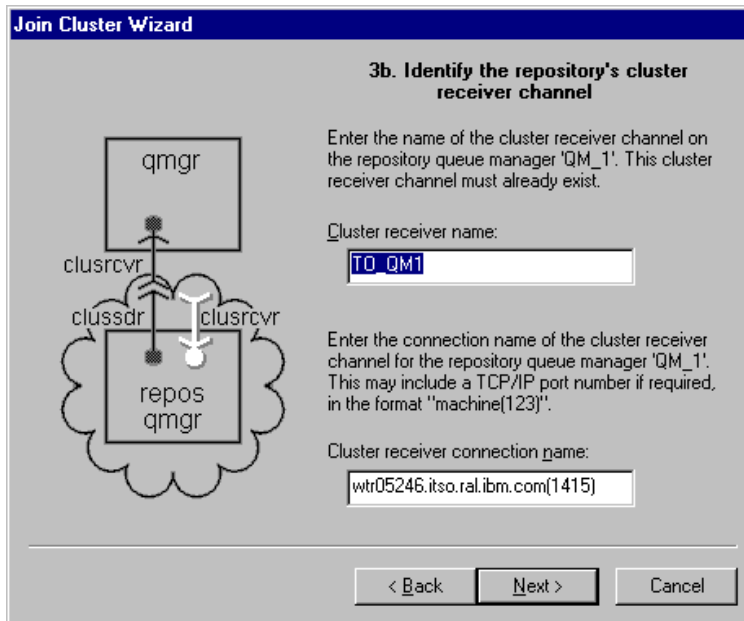


Figure 84. Join Cluster Wizard - 3b

The Cluster receiver connection name must be the IP address and listening port of the repository. Click **Next** when the information is correct.

8. This brings you to the Join the queue manager to the cluster window, as shown in Figure 85. There is nothing to do on this panel except check that your details are correct and then click **Finish**.

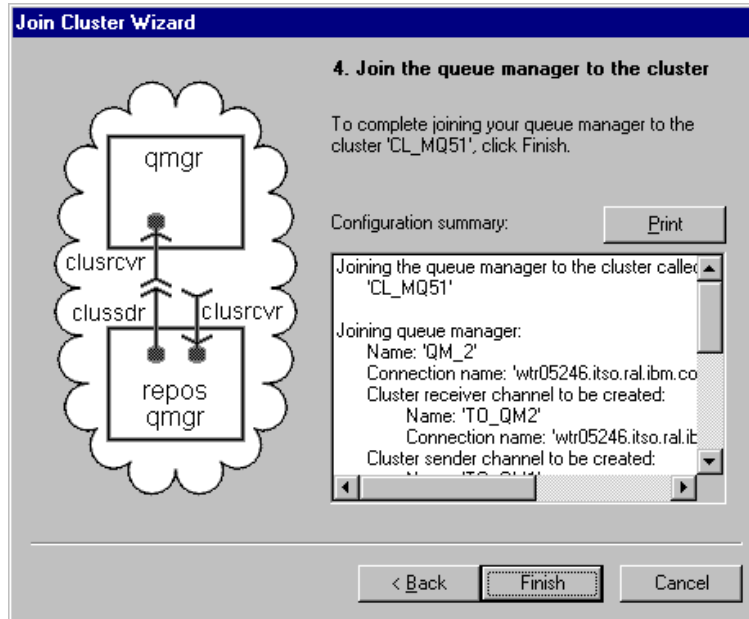


Figure 85. Join Cluster Wizard - 4

At this time you see in the MQExplorer window:

- Four queue managers running
- Three queue managers in the cluster

To complete this fairly long step, you need to add the last queue manager, QM\_4, to the cluster. Proceed as for QM\_2. The only difference is that you might want to select a different repository when joining QM\_4 to the cluster.

We have chosen QM\_3 instead of QM\_1 to be the repository queue manager for QM\_4. However, there is absolutely no *need* to choose a different one, as long as the queue manager specified maintains a full repository.

The cluster receiver name is TO\_QM4 and the port is 1418.

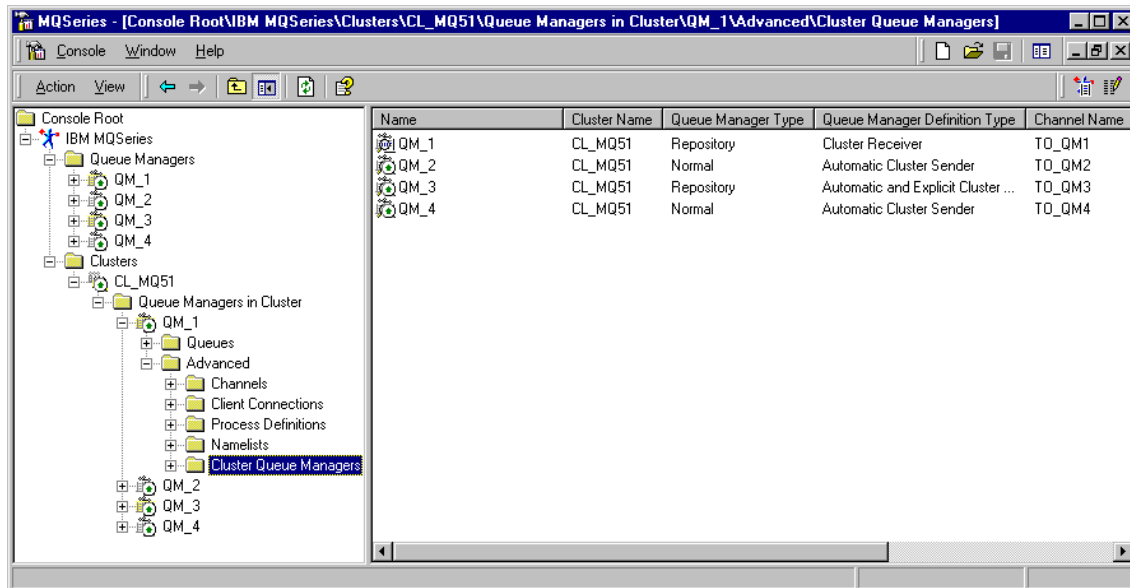


Figure 86. Cluster with Four Queue Managers

Now our cluster is built. Verify that your cluster looks the same as Figure 86.

**Note:** You may have to close and re-open the MQExplorer to see the added queue manager in the cluster.

In Figure 86 on page 76 you also see the cluster channels that QM\_1 knows. To display them click **Clusters ->CL\_MQ51->Queue Managers in Cluster->QM\_1, Advanced** and then click **Cluster Queue Managers**.

- QM\_1 has a cluster receiver channel TO\_QM1, which we defined in Figure 74 on page 68.
- The channel TO\_QM3 connects QM\_1 with the other full repository queue manager, QM\_3. We defined this channel for QM\_3 in Figure 75 on page 69; The sender part has been automatically defined.
- TO\_QM2 and TO\_QM4 are automatically defined cluster sender channels. The corresponding receiver channels were defined when QM\_2 and QM\_3 joined the cluster. Refer to Figure 83 and Figure 84 on page 74.

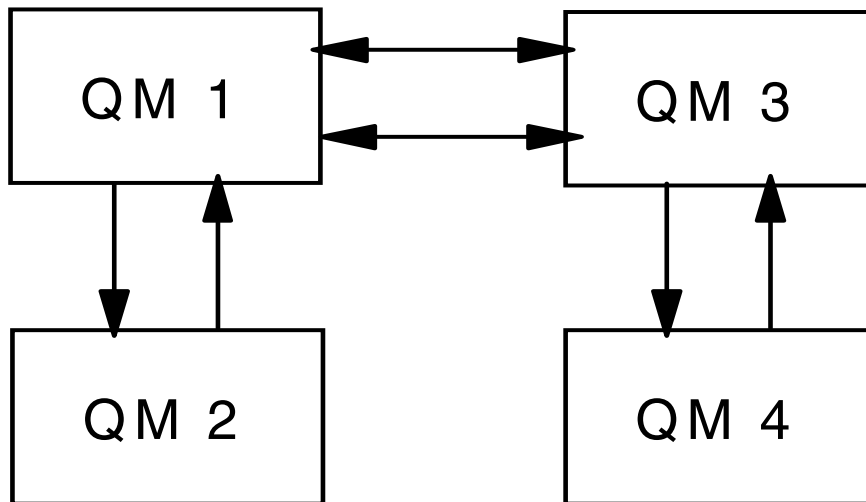


Figure 87. Cluster Channels

**Summary:**

Figure 87 shows the queue managers in the cluster and the connections defined between them.

- Queue managers in a cluster are connected with CUSSDR (cluster sender) and CLUSRCVR (cluster receiver) channels.
- The two repository queue managers QM\_1 and QM\_3 are connected with two channel pairs:
  - TO\_QM1 CLUSRCVR in QM\_1 and CLUSSDR in QM\_3
  - TO\_QM3 CLUSRCVR in QM\_3 and CLUSSDR in QM\_1
- QM\_2 is connected to QM\_1 with these channel pairs:
  - TO\_QM2 CLUSRCVR and automatic defined CLUSSDR in QM\_1
  - TO\_QM1 CLUSRCVR and automatic defined CLUSSDR in QM\_2
- QM\_4 is connected to QM\_3 with these channel pairs:
  - TO\_QM4 CLUSRCVR and automatic defined CLUSSDR in QM\_3
  - TO\_QM3 CLUSRCVR and automatic defined CLUSSDR in QM\_4

## 4.4 Working with Local Queues in a Cluster

We will now use the MQSeries Explorer to create some local test queues. These queues will not be shared within the cluster. They are only known to the particular queue manager. For each of the four queue managers we create one queue:

Queue Manager Name	Local Queue Name
QM_1	TQ_1
QM_2	TQ_2
QM_3	TQ_3
QM_4	TQ_4

Table 5. Local Queues for Cluster Queue Managers

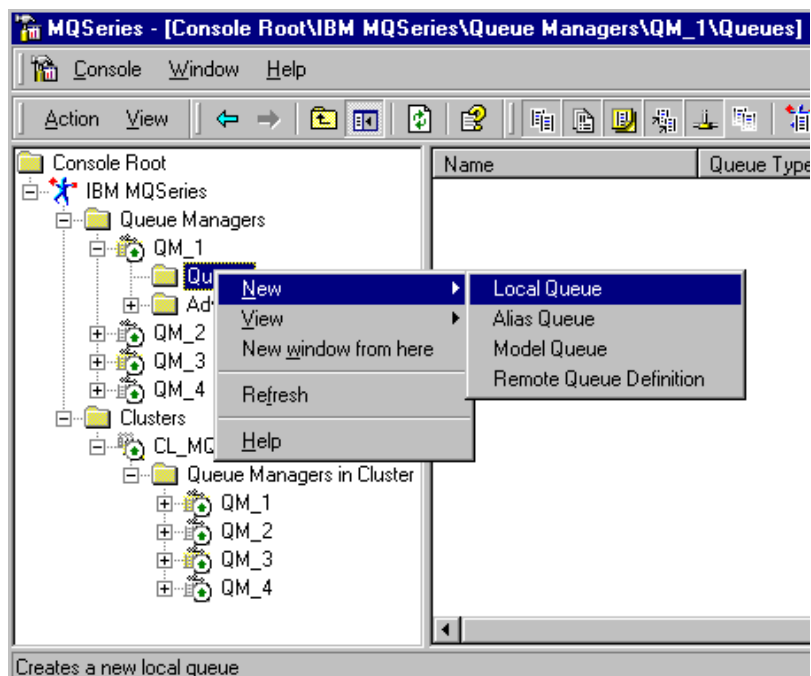


Figure 88. Defining a Local Queue

Use the following steps to create a queue:



1. As shown in the Explorer window in Figure 88 on page 78, expand **Queue Managers** and then **QM\_1**, the queue manager for which we want to create the queue TQ\_1.
2. Right-click the **Queues** folder of QM\_1, then select **New -> Local Queue**. This brings up the window in Figure 89.

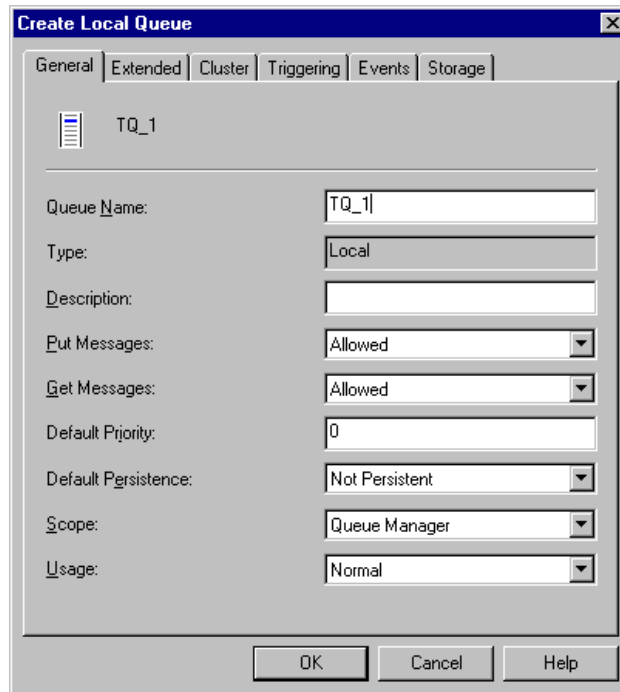


Figure 89. Create Local Queue - 1

3. Type TQ\_1 (for “test queue 1”) in the Queue Name field. No other changes need to be made on this panel. We will accept the defaults.
4. Next click the **Cluster** tab of the window shown in Figure 89.
5. In the Cluster tab, as shown in Figure 90 on page 80, we need to select that **Not shared in cluster** for our TQ\_1. That’s all. Click on **OK** and you have created the first queue.
6. Repeat the above steps for the other three queue managers. Name the test queues TQ\_2, TQ\_3 and TQ\_4.
7. Display the queues to ensure that they are local queues and not cluster queues. A queue should display only for a particular queue manager.

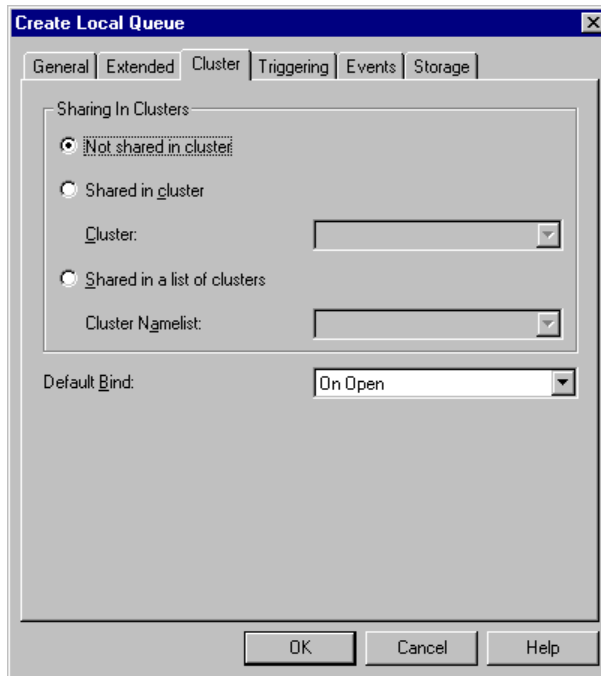


Figure 90. Create Local Queue - 2

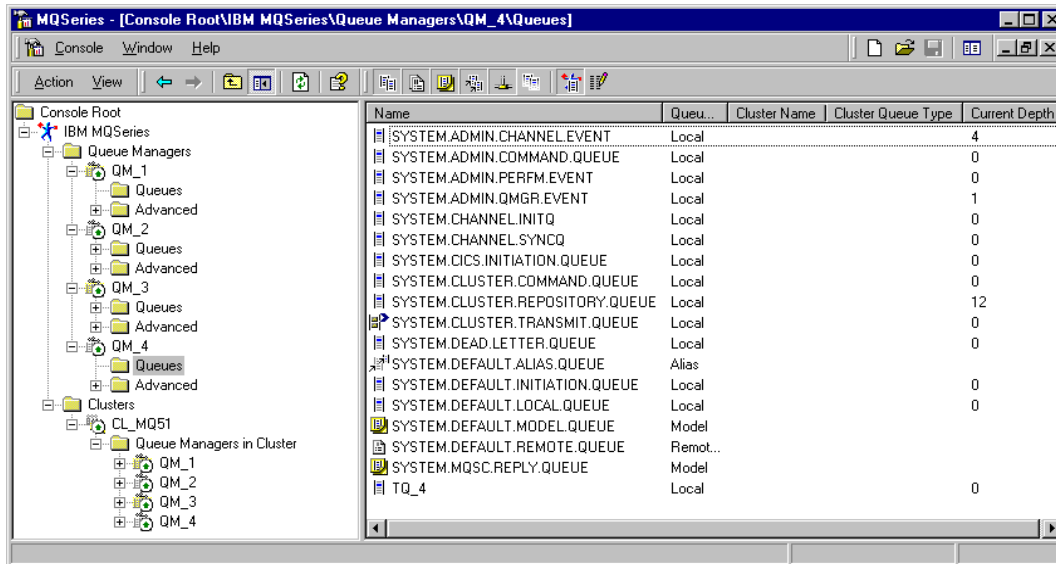


Figure 91. System Queues and a Local Queue

Figure 91 on page 80 shows all queues for QM\_4, the system queues and the local queue just defined. To hide system queues, click **View** in the task bar and uncheck **Show System Objects** by clicking on that menu item.

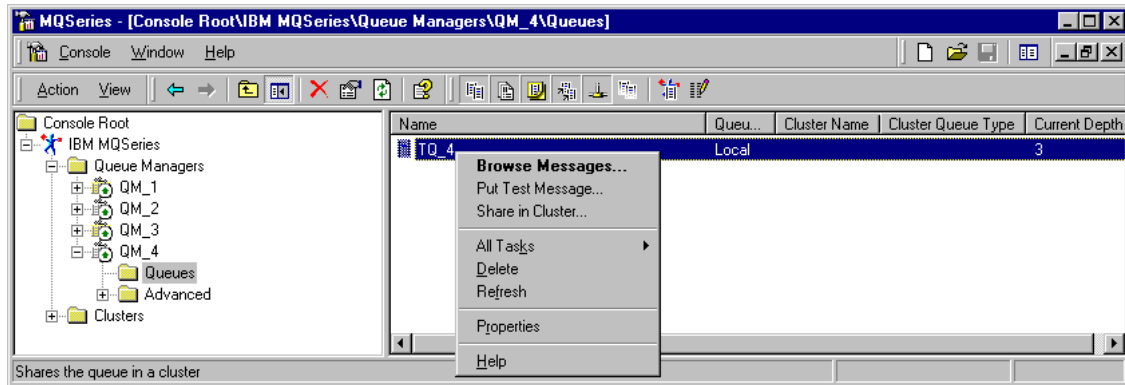


Figure 92. Put a Test Message - 1

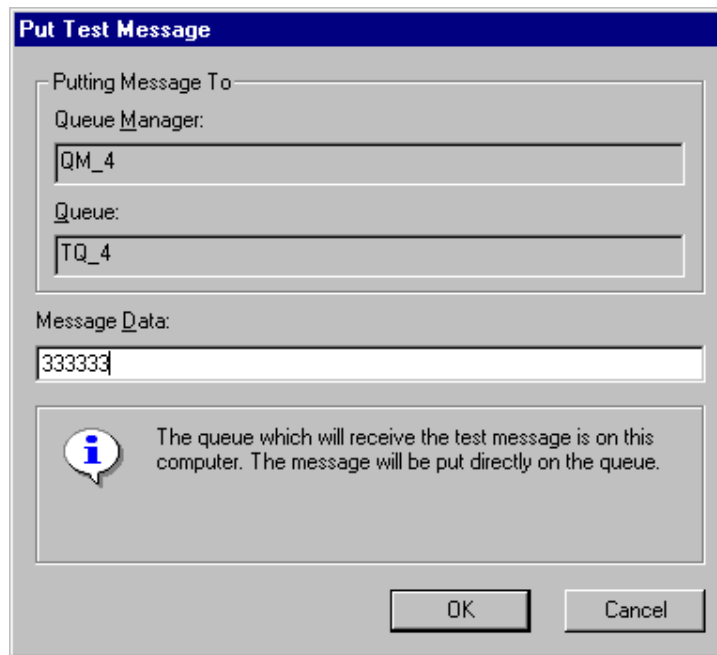


Figure 93. Put a Test Message - 2

As an alternative to amqsput and amqsgbr you can also put messages into the queue and browse the contents of a queue from the MQSeries Explorer.

To put a message in a queue, follow these steps:

1. Click **Queues** to display the queues on the right side of the Explorer window.
2. Right-click a queue, for example, **TQ\_4** as shown in Figure 92 on page 81 and select **Put Test Message** from the menu.
3. This brings up the window shown in Figure 93 on page 81. Type some message data and click **OK**.
4. You will see that the current depth of the queue will increase. You see this field in Figure 92 on page 81.

To browse messages in a queue, select **Browse Messages** from the menu. This displays a window such as shown in Figure 94. You can customize the columns you want to see.

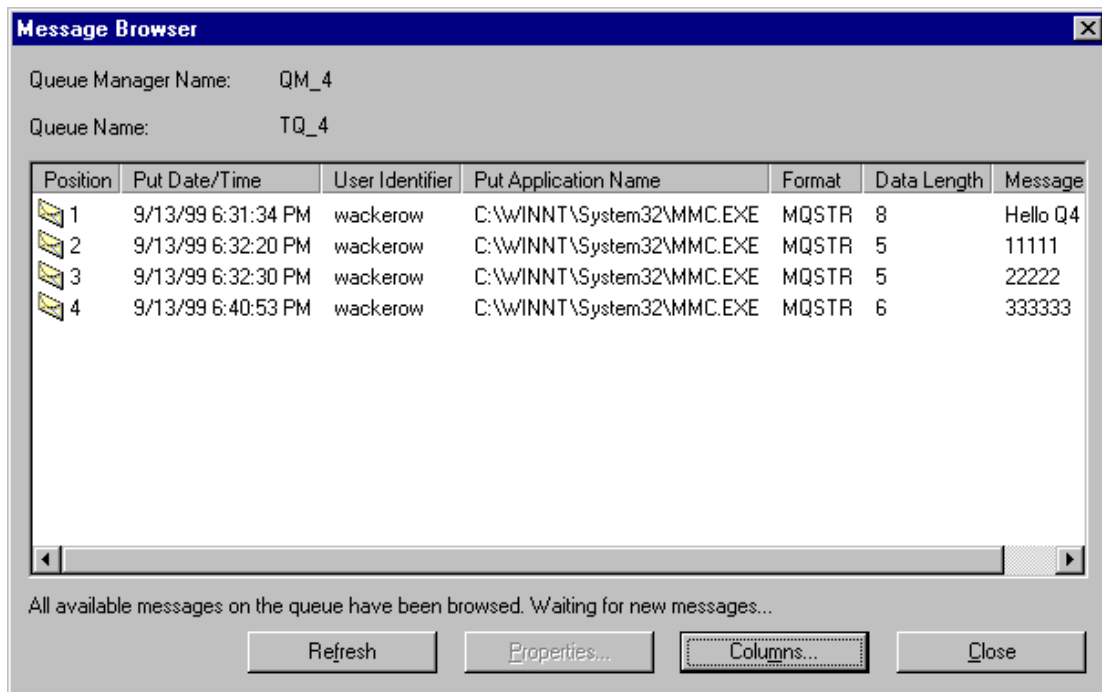


Figure 94. Browse Messages in a Queue

## 4.5 Creating a Shared Cluster Queue

Now let us create another test queue, CLQ\_1. This queue *will* be shared within the cluster. CLQ\_1 will not only be visible within the entire cluster, but we will create an instance of CLQ\_1 on every one of our four queue managers.

1. Bring up the Explorer window and right-click the **Queues** folder of **QM\_1**. Then select **New** and **Local Queue**, just as you did for TQ\_1 in Figure 88 on page 78.
2. Figure 95 shows that you then enter the Queue Name (CLQ\_1), just as we did before. Then, also as done before, click the **Cluster** tab.

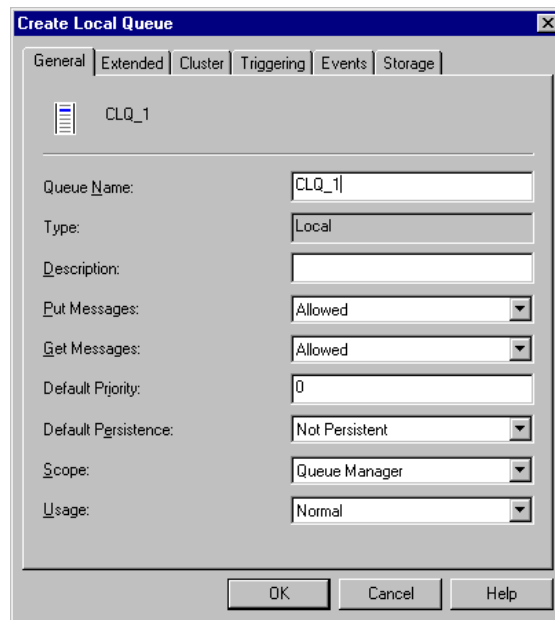


Figure 95. Creating a Shared Cluster Queue - 1

3. Figure 96 on page 84 shows that at this point we act differently from when we created a simple local queue. Select the **Shared in cluster** button and then select **cluster CL\_MQ51** from the **Cluster** selection box.
4. Click **OK** and you have created the copy of CLQ\_1, which is local to QM\_1. Figure 97 on page 84 shows the result. As you can see, CLQ\_1 now exists in QM\_1. Notice that the queue type is Local and that it is in cluster CL\_MQ51. Also notice that TQ\_1 does not belong to the cluster.

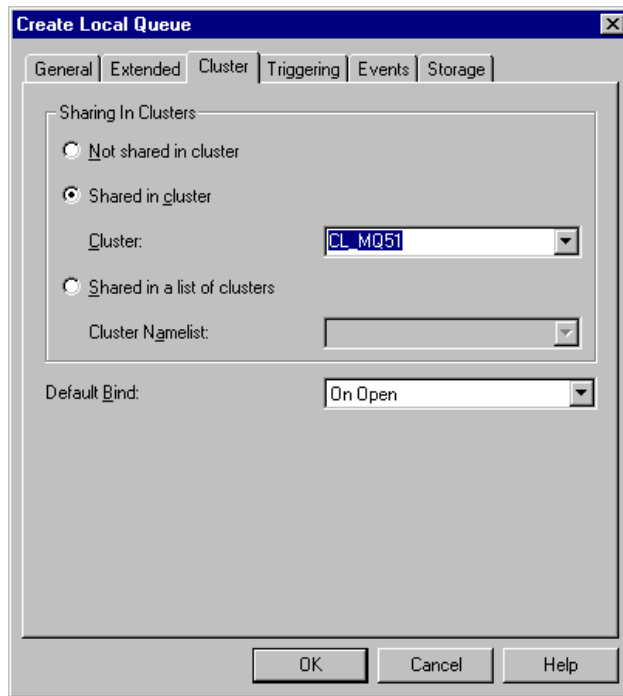


Figure 96. Creating a Shared Cluster Queue - 2

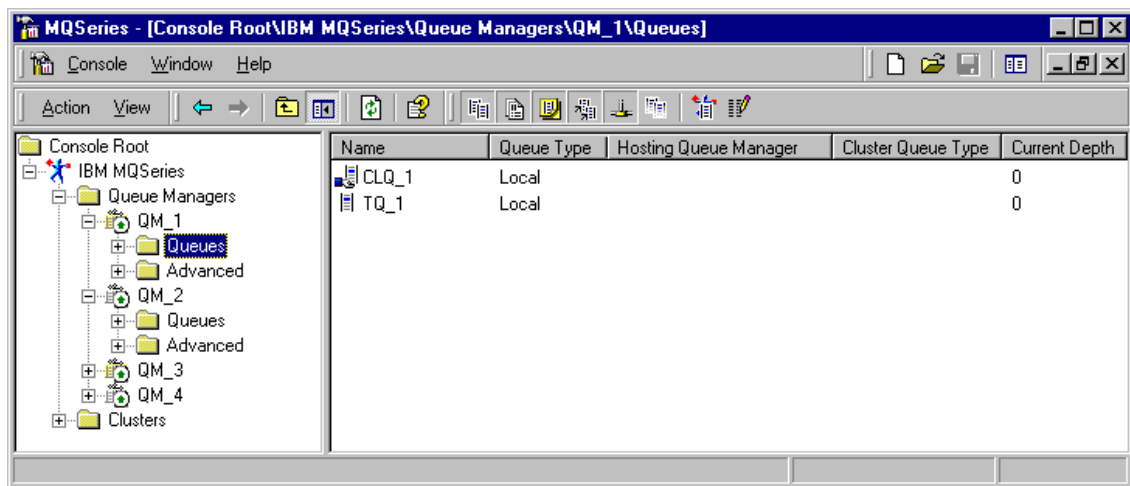


Figure 97. Local and Cluster Queues - QM\_1

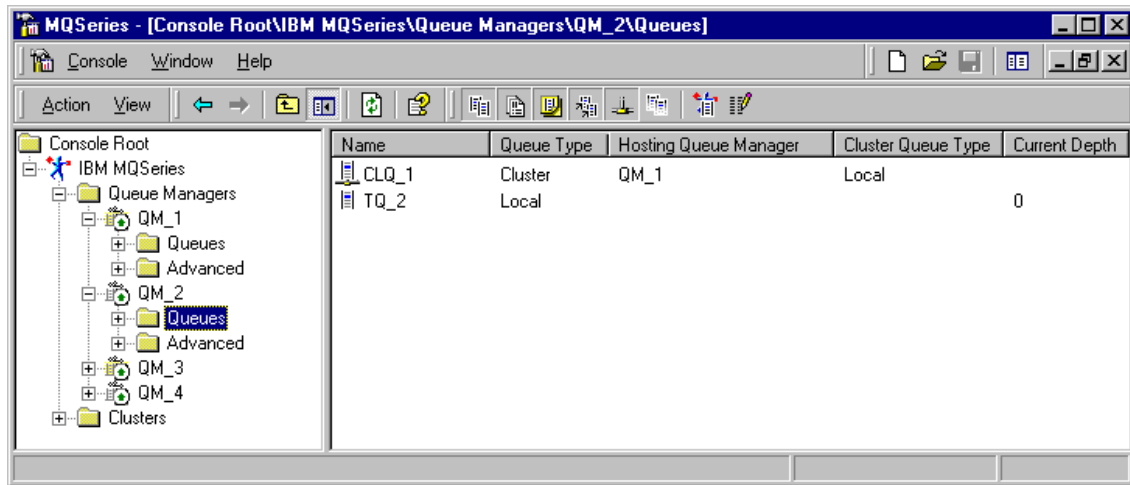


Figure 98. Local and Cluster Queue - QM\_2

Now, look at the Queues folder under QM\_2 (or any queue manager in the cluster other than QM\_1). Figure 98 presents the view from QM\_2. Notice that CLQ\_1 is displayed differently from here. That's because although it is visible (because it is a cluster queue) it is not local to QM\_2.

You may also notice that the current queue depth is displayed only for local queues, that is, queues hosted by the particular queue manager you selected for display.

**Note:** Click the **Refresh** button if you don't see the queue in all four queue managers.

Now perform exactly the same steps that you used to create CLQ\_1 in QM\_1, but this time create CLQ\_1 in QM\_2.

Figure 99 shows the result. Here you are looking at queues from the viewpoint of QM\_2. Notice that there are now two copies of CLQ\_1. One is local to QM\_2 (from where you are looking) and the other is local to QM\_1. Figure 100 is a view of exactly the same thing, but from the viewpoint of QM\_1.

Notice that although we can see CLQ\_1 from QM\_1 and QM\_2 (and the other queue managers), we can see TQ\_1 only from QM\_1 and likewise we can see TQ\_2 only from QM\_2 only .

Now create instances of CLQ\_1 in QM\_3 and QM\_4. Once again, you can refer to Figure 95 and Figure 96 on page 84.

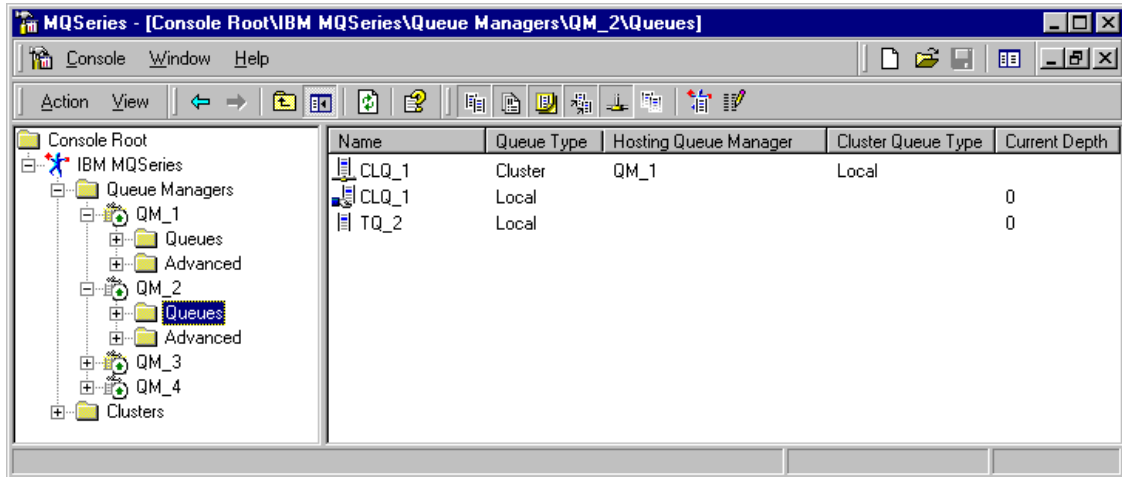


Figure 99. Two Cluster Queue Instances Seen From QM\_2

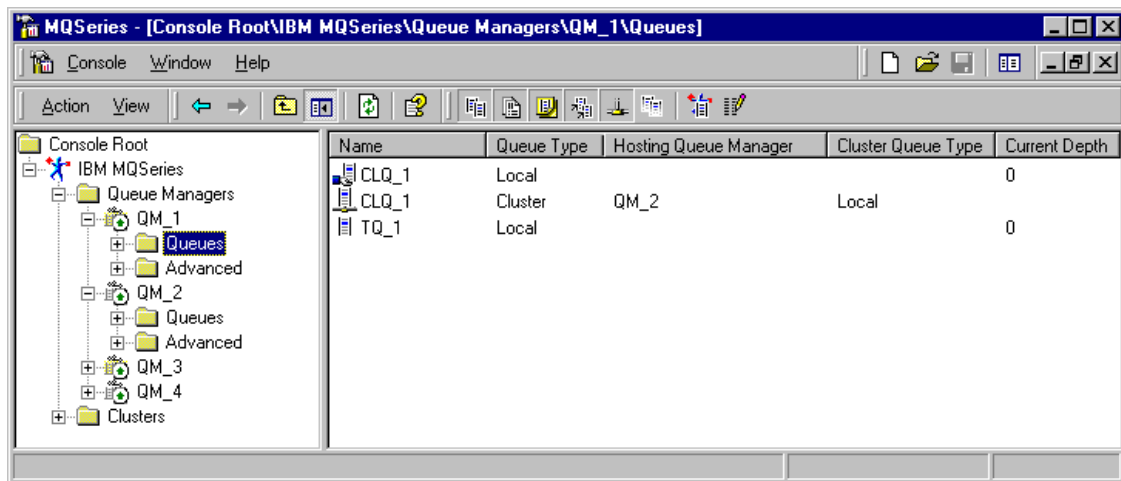


Figure 100. Two Cluster Queue Instances Seen From QM\_1

The final result is shown in Figure 101 on page 87. Notice that from the Queues folder of every one of our queue managers, we can now see all four copies of CLQ\_1. Remember also, these are not just views. There really are four queues, all called CLQ\_1.



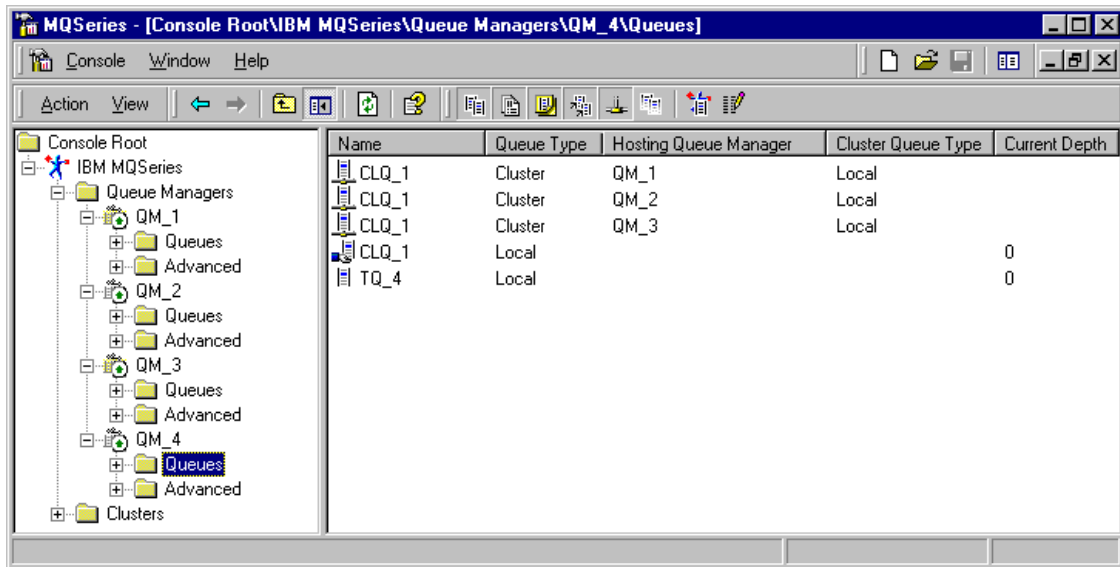


Figure 101. Four Cluster Queues Seen By QM\_4

## 4.6 Creating a Second Cluster Queue

This is the last step in this exercise to create queues using the MQSeries Explorer.

We are going to create another queue, which is duplicated within the cluster, but this time we will only have *three copies* of it, on QM\_2, QM\_3 and QM\_4. We will need this sort of queue for one of the subsequent exercises. You start out in the same way as for all the other queues created in this exercise.

1. Right-click the **Queues** folder under QM\_2 and then select **New -> Local Queue**.
2. Name this queue CLQ\_ACROSS\_2\_3\_4. This seems an unfortunate name perhaps, but it will remind us, from within the Explorer, that this queue has instances only on three of our four queue managers. When you have typed in the name, click the **Cluster** tab.
3. Select **Shared in cluster** and in the Cluster field select **CL\_MQ51**.
4. Click **OK**.

Repeat this process to define instances of CLQ\_ACROSS\_2\_3\_4 in QM\_3 and QM\_4. You see in Figure 102 on page 88 that QM\_4 sees all three instances and that one instance is local. QM\_1 shown in Figure 103 on page 88 sees all queues, too. However, there is no local instance.

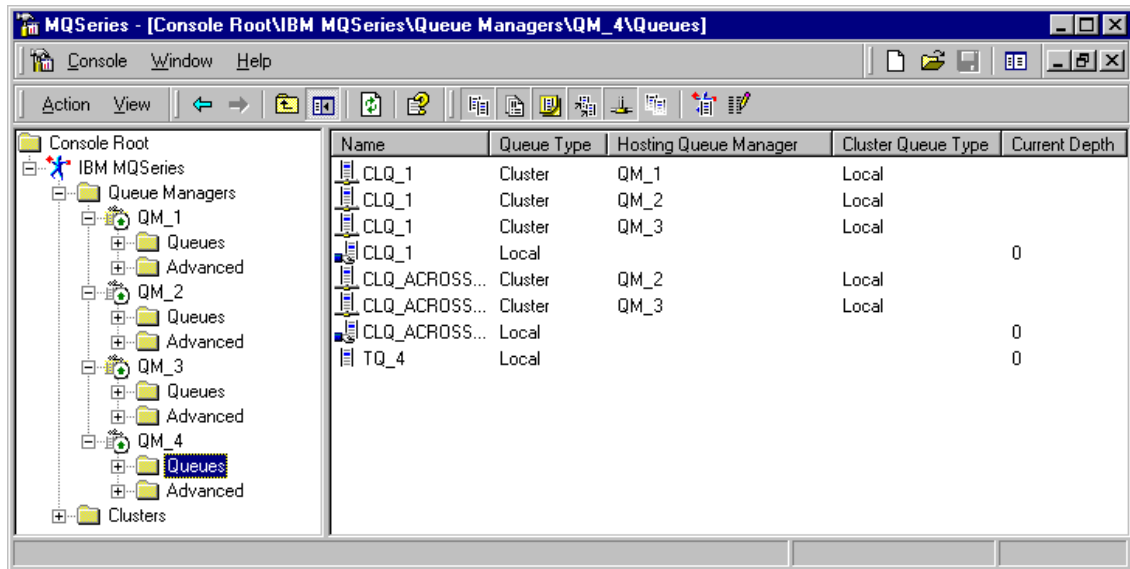


Figure 102. Queues Known To QM\_4

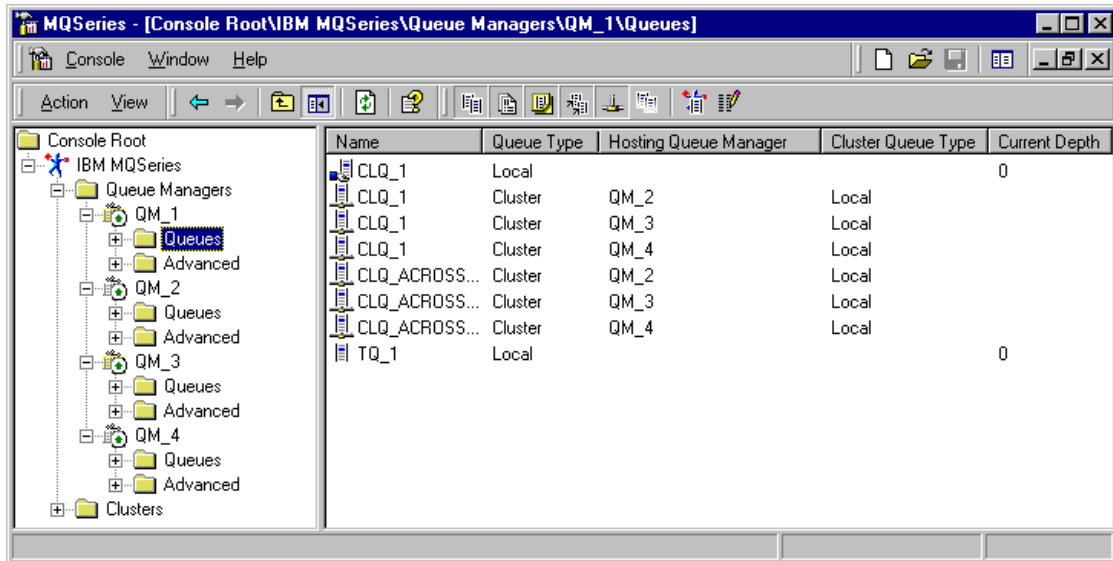


Figure 103. Queues Known To QM\_1

## 4.7 Working with Clusters

To become more familiar with clusters and the MQSeries Explorer, let us do some more exercises.

### 4.7.1 Putting and Getting Messages

Figure 104 below shows that when you right-click on a queue that is not local to the queue manager (whose folder you are within), then you get the option Put Test Message to put a test message into the queue.

Figure 105 on page 90 on the other hand, shows that when you right-click a queue that is local to the queue manager (whose folder you are in), then you get the additional option of Browse Message.

This conforms with what you already know about MQSeries; that you can get messages from a local queue only. It is important to remember, when looking at the wide view that clustering and the Explorer give you, that this is still the case.

All the changes that make clustering possible have been made with the PUT. The MQSeries GET is just as we have known it.

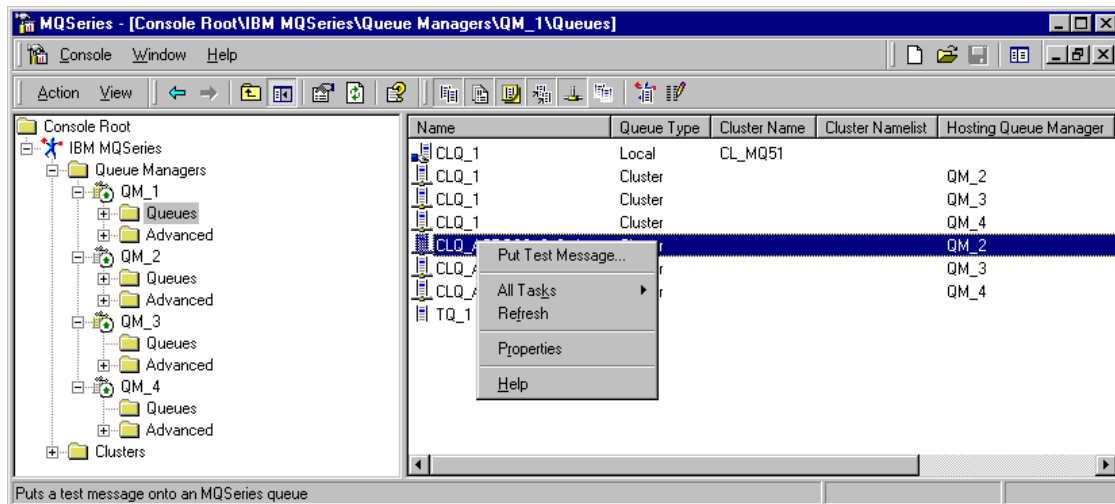


Figure 104. Put a Message on a Queue not Owned by the Queue Manager

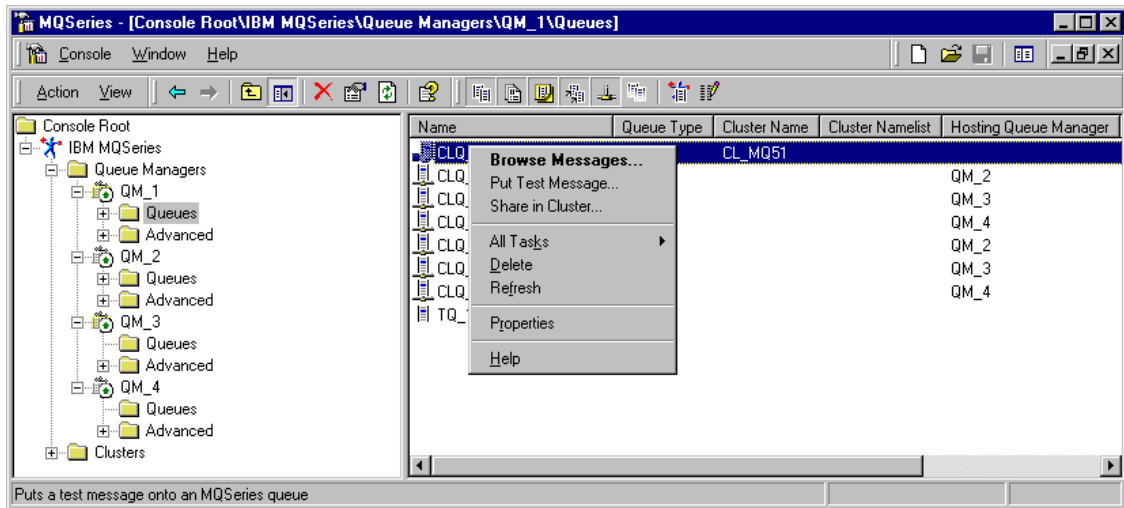


Figure 105. Put a Message on a Queue Owned by the Queue Manager

#### 4.7.2 Disassembling the Environment with the MQ Explorer

Bear in mind the following points if you are using the GUI to take apart the environment.

- To delete a queue manager, you must first right-click it in the MQSeries Explorer. A number of options will appear, one of which is Delete.
- A queue manager cannot be deleted while it is running. You must right click the queue manager and select **Stop**, then wait till it is stopped before you delete it. You may need to check the **Refresh** button to update the display.
- A cluster cannot be deleted. It is not an object in its own right.

#### 4.7.3 Stopping a Cluster

*How do you stop a cluster?*

You can't really. A cluster is deemed to have stopped (by the Explorer) when all its queue managers have stopped.

Try the following:

1. Using the Explorer, stop all the queue managers in cluster CL\_MQ51.

Notice how (perhaps after a refresh) the cluster has gone red too, into a stopped state. But notice also how there is no start option on the cluster when you right-click. That is because you can't *start* a cluster either.

2. Stop the Explorer by clicking **Console** and selecting **Exit** from the menu. Then start the Explorer again.

*Why is cluster CL\_MQ51 no longer displayed?*

Because the "cluster" is not an object in its own right and in the absence of any queue manager with an attribute of <clustername>, the Explorer is not able to know of any cluster, <clustername>.

#### 4.7.4 Showing a Cluster

Now try the following scenario:

- Right-click **Clusters**, and select **Show Cluster**.
- A Show Cluster dialog will pop up. In Cluster Name field enter **CL\_MQ51**.
- In the Queue Manager Name field, enter one of the non-repository queue managers, QM\_2 or QM\_4 from the list.
- Click **OK**.
- The message QM\_2 (or QM\_4) is not available is no surprise. Click **OK** again.

*Why are you now being asked whether you want to "show this cluster in the console?"*

Because the Explorer has no way of knowing whether a cluster called CL\_MQ51 exists *anywhere!*

- Click **Yes** (we *do* want to show this cluster on the console).

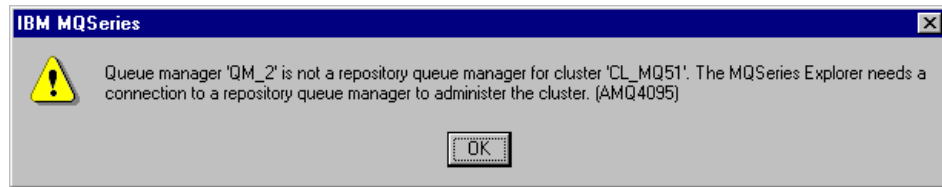
Start whichever non-repository queue manager (QM\_2 or QM\_4) you chose in the Show Cluster dialog. Wait until that queue manager has properly started. You may notice the CL\_MQ51 cluster icon go briefly green, but then it will change to a yellow warning sign.

*Why isn't the cluster regarded as "started" now?*

Right-click the **CL\_MQ51** icon and select **Connect**. Now you see the reason! QM\_2 or QM\_4 (whichever you chose) is not a repository queue manager for cluster CL\_MQ51.

## 4.7.5 Starting a Cluster

Start QM\_1 or QM\_3, or both, that is, one of the repository queue managers. Wait until it/they are started. If you don't get another error message either click the **Refresh** button, or right-click **CL\_MQ51** and choose **Connect**.



*Why is the error message talking about QM\_2 (or QM\_4)? We know they are not repository queue managers for cluster CL\_MQ51, but we have QM\_1 and/or QM\_3 running now, don't we?*

In 4.7.4, "Showing a Cluster" on page 91, you told the Explorer that QM\_2 (or QM\_4) was a repository queue manager for cluster CL\_MQ51. The Explorer has not forgotten this.

Now start the remaining queue managers.

*Does CL\_MQ51 still show the yellow warning sign?*

Yes, of course. You have to exit the Explorer and start it again. Then you will see that all the queue managers are running (green) and that the Clusters tree shows all queue managers.

More cluster administration examples are in Chapter 7, "MQSeries Administration and Service" on page 123.

## 4.7.6 Summary

- Clusters appear as objects in the Explorer. The Explorer presents information from many sources together.
- The Connection point to a cluster is a single repository. The first repository cluster queue manager is used. Only TCP/IP is supported. The Explorer supplies a list of cluster members and a complete set of cluster queues.
- The channel status for cluster channels is available from cluster queue managers. Double-click on a cluster queue manager or use the pop-up menu.

- Cluster queues shared in more than one cluster are displayed once per folder. This differs from MQSC, which displays it once per cluster.





## Chapter 5. Creating a Cluster with Scripts

In this chapter, we describe how to create a cluster with four queue managers using scripts and RUNMQSC. The configuration is the same we used in the previous chapter. It is shown again in Figure 106 below.

You may now spent some time thinking about how you would create runmqsc configuration scripts and batch (.BAT) files (or other interpreted scripts) to build exactly the same environment that we created with the GUI.

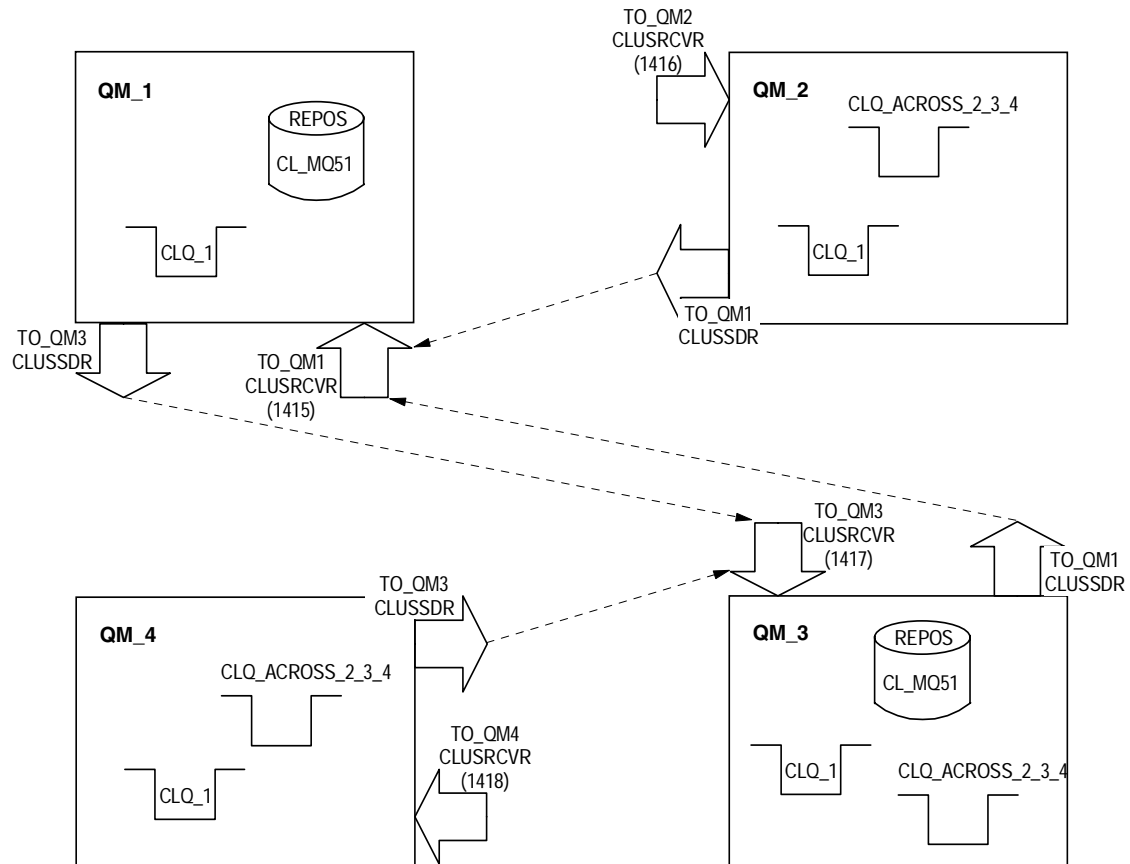


Figure 106. Cluster with Four Queue Managers

The easiest way is to look immediately at the examples provided in the files shown in Table 6 on page 96.

File Name	Description	See
QM_1.cfg	Configuration for queue manager QM_1	page 98
QM_2.cfg	Configuration for queue manager QM_2	page 99
QM3.cfg	Configuration for queue manager QM_3	page 100
QM_4.cfg	Configuration for queue manager QM_4	page 101
scripts.out	A sample of what the output should look like	page 215
crt_str_all.bat	A BAT file which builds the whole cluster CL_MQ51	page 102
crt_str_all.log	A log that crt_str_all.bat writes	page 217
end_dlt_all.bat	A BAT file that stops and destroys the whole of cluster CL_MQ51	page 103

Table 6. Configuration Files for a Cluster

If you already build the cluster CL\_MQ51 using the MQSeries Explorer as described in Chapter 4, “Creating a Cluster with the MQExplorer” on page 55, then there are two possible ways to proceed:

1. Destroy what you have already done and rebuild the *same* cluster from the scripts provided.
2. Edit the scripts before you start so that they will create *different* queue managers in a *different* cluster.

The choice is yours. One factor that may influence your decision is that the scripts provided with this exercise are designed to create/destroy only a cluster and its four queue managers, which are all running on a single Windows NT machine. If you went to a lot of trouble in the previous (GUI) exercise, creating a cluster across multiple machines, then you will probably want to edit the scripts and create a new cluster.

If you decide to destroy and rebuild the cluster, you can do this either through the MQSeries Explorer, or by running the end\_dlt\_all.bat script. Remember, end\_dlt\_all.bat will function properly only if:

- All the cluster resources are on one Windows NT machine.
- You named everything exactly as described in the notes as you progressed through.

Assuming you have spent some time trying some configuration files and .bat files of your own, let's now have a look at some of the ones provided. You may think of improvements to these. They do, however, have the endearing quality that they work!

The four configuration files for the four queue managers including some comments are shown on the next four pages.

These, when run through RUNMQSC, create queue managers that are the same as the ones we created manually using the Explorer, in Chapter 4, "Creating a Cluster with the MQExplorer" on page 55.

**Important:** You must change the CONNAME in the channel definitions. If you use TCP/IP and create all four queue managers on the same system (that is most likely in a test environment), you may use one of the following instead:

- The name of your workstation
- The TCP/IP address of your workstation
- The TCP/IP loop back address 127.0.0.1

So, having created and started queue managers QM\_1, QM\_2, QM\_3 and QM\_4, the commands to use these configuration files would be:

```
runmqsc QM_1 < QM_1.cfg
runmqsc QM_2 < QM_2.cfg
runmqsc QM_3 < QM_3.cfg
runmqsc QM_4 < QM_4.cfg
```

Or, if you want some output to a log file, as we did:

```
runmqsc QM_1 < QM_1.cfg > cluster_config.log
runmqsc QM_2 < QM_2.cfg >> cluster_config.log
runmqsc QM_3 < QM_3.cfg >> cluster_config.log
runmqsc QM_4 < QM_4.cfg >> cluster_config.log
```

A sample log file is shown in Appendix A, "Sample Configuration Output" on page 215. In this example, all queue managers were created on the same Windows NT machine. Therefore, we modified the CONNAME and used the TCP/IP loopback address 127.0.0.1 in the connection name parameter. This would work on your machine, too, provided you choose the same ports, 1415 through 1418, and the cluster name CL\_MQ51.

```

* This ALTER QMGR needs to run early. The scripts
* for QM_1 & QM_3 therefore need to be RUN first! 1

ALTER QMGR REPOS (CL_MQ51) 2

* This is the channel which would be created if you checked
* 'Create Server Connection Channel to allow remote administration
* of the queue manager over TCP/IP' in Step 3 of the Create
* Queue Manager wizard

* DEFINE CHANNEL (SYSTEM.ADMIN.SVRCONN) REPLACE + 3
* CHLTYPE (SVRCONN) TRPTYPE (TCP) REPLACE

DEFINE CHANNEL (TO_QM3) CHLTYPE (CLUSSDR) REPLACE + 4
TRPTYPE (TCP) CLUSTER (CL_MQ51) +
CONNAME ('wtr05246.itso.ral.ibm.com(1417)')

DEFINE CHANNEL (TO_QM1) CHLTYPE (CLUSRCVR) REPLACE + 5
TRPTYPE (TCP) CLUSTER (CL_MQ51) NETPRTY (0) +
CONNAME ('wtr05246.itso.ral.ibm.com(1415)')

DEFINE QLOCAL (TQ_1) REPLACE 6

DEFINE QLOCAL (CLQ_1) CLUSTER (CL_MQ51) REPLACE 7

```

Figure 107. QM\_1.cfg

#### Comments to the configuration files:

- 1** In theory it is preferable to bring up the cluster repository queue manager first. However, if you try changing the order, you can see for yourself that starting the non-repository queue managers first does not cause problems.
- 2** This makes the queue manager part of the cluster CL\_MQ51, and indeed a repository for the cluster.

```

* Do not run this script until AFTER the scripts
* for QM_1 & QM_3 have been run ! 1

* This is the channel which would be created if you checked
* 'Create Server Connection Channel to allow remote administration
* of the queue manager over TCP/IP' in Step 3 of the Create
* Queue Manager wizard

* DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) REPLACE + 3
* CHLTYPE(SVRCONN) TRPTYPE(TCP) REPLACE

DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSSDR) REPLACE + 4
TRPTYPE(TCP) CLUSTER(CL_MQ51) +
CONNAME('wtr05246.itso.ral.ibm.com(1415)')

DEFINE CHANNEL(TO_QM2) CHLTYPE(CLUSRCVR) REPLACE + 5
TRPTYPE(TCP) CLUSTER(CL_MQ51) NETPRTY(0) +
CONNAME('wtr05246.itso.ral.ibm.com(1416)')

DEFINE QLOCAL(TQ_2) REPLACE 6

DEFINE QLOCAL(CLQ_1) CLUSTER(CL_MQ51) REPLACE 7

DEFINE QLOCAL (CLQ_ACROSS_2_2_4) CLUSTER(CL_MQ51) + 8
REPLACE

```

Figure 108. QM\_2.cfg

- 3** We haven't defined this channel. It is here just as a comment, in case you would like to define it.
- 4** Necessary CLUSSDR queue to make the queue manager part of the cluster.
- 5** Necessary CLUSRCVR channel to make us part of the cluster.
- 6** An old-style queue, only visible to the particular queue manager.

```

* This ALTER QMGR needs to run early. The scripts
* for QM_1 & QM_3 therefore need to be RUN first! 1

ALTER QMGR REPOS (CL_MQ51) 2

* This is the channel which would be created if you checked
* 'Create Server Connection Channel to allow remote administration
* of the queue manager over TCP/IP' in Step 3 of the Create
* Queue Manager wizard

* DEFINE CHANNEL (SYSTEM.ADMIN.SVRCONN) REPLACE + 3
* CHLTYPE (SVRCONN) TRPTYPE (TCP) REPLACE

DEFINE CHANNEL (TO_QM3) CHLTYPE (CLUSSDR) REPLACE + 4
TRPTYPE (TCP) CLUSTER (CL_MQ51) +
CONNNAME ('wtr05246.itso.ral.ibm.com(1415)')

DEFINE CHANNEL (TO_QM1) CHLTYPE (CLUSRCVR) REPLACE + 5
TRPTYPE (TCP) CLUSTER (CL_MQ51) NETPRTY (0) +
CONNNAME ('wtr05246.itso.ral.ibm.com(1417)')

DEFINE QLOCAL (TQ_1) REPLACE 6

DEFINE QLOCAL (CLQ_1) CLUSTER (CL_MQ51) REPLACE 7

DEFINE QLOCAL (CLQ_ACROSS_2_2_4) CLUSTER (CL_MQ51) + 8
REPLACE

```

Figure 109. QM\_3.cfg

- 7** CLQ\_1 is the queue that we decided to create on every queue manager in the cluster. All four instances of this queue can be seen from each of the four queue managers.
- 8** CLQ\_ACROSS\_2\_3\_4 is a queue that we will use to experiment with clustered queues in Chapter 6, “Workload Management” on page 107.

```

* Do not run this script until AFTER the scripts
* for QM_1 & QM_3 have been run ! 1

* This is the channel which would be created if you checked
* 'Create Server Connection Channel to allow remote administration
* of the queue manager over TCP/IP' in Step 3 of the Create
* Queue Manager wizard

* DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) REPLACE + 3
* CHLTYPE(SVRCONN) TRPTYPE(TCP) REPLACE

DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSSDR) REPLACE + 4
TRPTYPE(TCP) CLUSTER(CL_MQ51) +
CONNAME('wtr05246.itso.ral.ibm.com(1417)')

DEFINE CHANNEL(TO_QM2) CHLTYPE(CLUSRCVR) REPLACE + 5
TRPTYPE(TCP) CLUSTER(CL_MQ51) NETPRTY(0) +
CONNAME('wtr05246.itso.ral.ibm.com(1418)')

DEFINE QLOCAL(TQ_2) REPLACE 6

DEFINE QLOCAL(CLQ_1) CLUSTER(CL_MQ51) REPLACE 7

DEFINE QLOCAL (CLQ_ACROSS_2_2_4) CLUSTER(CL_MQ51) + 8
REPLACE

```

Figure 110. QM\_4.cfg

Next, look at the .bat files crt\_str\_all and end\_dlt\_all in Figure 111 on page 102 and Figure 112 on page 103. These are simple Windows BAT files that automatically create and delete our cluster environment. The comments explain, in detail, what functions are performed.

```

crtmqm QM_1      1

crtmqm QM_3      2

crtmqm QM_2      3
crtmqm QM_4

strmqm QM_1      4
strmqm QM_3
strmqm QM_2
strmqm QM_4

echo "CLUSTER CONFIGURE QM_1..... 5...." > crt_str_all.log
runmqsc QM_1 < QM_1.cfg >> crt_str_all.log      6

echo "CLUSTER CONFIGURE QM_3....." >> crt_str_all.log
runmqsc QM_3 < QM_3.cfg >> crt_str_all.log      7

echo "CLUSTER CONFIGURE QM_2....." >> crt_str_all.log
runmqsc QM_2 < QM_2.cfg >> crt_str_all.log      8

echo "CLUSTER CONFIGURE QM_4....." >> crt_str_all.log
runmqsc QM_4 < QM_4.cfg >> crt_str_all.log      9

start "QM_1 Listening on 1415" runmqslsr -t TCP -p 1415 -m QM_1 10

start "QM_2 Listening on 1416" runmqslsr -t TCP -p 1416 -m QM_2 11
start "QM_3 Listening on 1417" runmqslsr -t TCP -p 1417 -m QM_3 12
start "QM_4 Listening on 1418" runmqslsr -t TCP -p 1418 -m QM_4 13

```

Figure 111. crt\_str\_all.bat

**Comments to the .bat files:**

- 1 Create queue manager QM\_1. Note that this definition (and the tree that follows) is a completely generic or default queue manager. As yet they have none of our special objects defined.
- 2 Create queue manager QM\_3. Do you remember why we want to start QM\_1 and QM\_3 before QM\_2 and QM\_4? See the comments in the configuration files starting on page 98.
- 3 Create the non-repository queue managers QM\_2 and QM\_4.
- 4 Start all four queue managers.
- 5 We insert a comment on the log file crt\_str\_all.log.



```
endmqm -i QM_2           14
endmqm -i QM_4           15
endmqm -i QM_1
endmqm -i QM_3

endmqlsr -m QM_2        16
endmqlsr -m QM_4
endmqlsr -m QM_1
endmqlsr -m QM_3

dltmqm QM_2             17
dltmqm QM_4
dltmqm QM_1
dltmqm QM_3
```

Figure 112. *end\_dlt\_all.bat*

- 6 This runs our definition file QM\_1.cfg using for QM\_1.
- 7 Similarly, RUNMQSC executes the definitions in QM\_3.cfg for QM\_3.
- 8 Similarly for QM\_2.
- 9 And lastly for QM\_4.
- 10 Start a listener on port 1415 for QM\_1. The quoted string is a parameter for the Windows start command and has nothing to do with MQSeries. It just puts a nice title on the windows in which the listener will run.
- 11 Start a listener on port 1416 for QM\_2.
- 12 Start a listener on port 1417 for QM\_3.
- 13 Start a listener on port 1418 for QM\_4.
- 14 Shut down the queue manager QM\_2. As you can see, the order is reversed for the shutdown. This time we prefer to take down the non-repository queue managers first. As before, things will work fine in any order. This is just a bit tidier.
- 15 Shut down the other three queue managers.
- 16 Close all listeners for queue managers QM\_2, QM\_4, QM\_1 and QM\_3.
- 17 Delete all four queue managers.

To delete the environment you created in Chapter 4, “Creating a Cluster with the MQExplorer” on page 55 execute the command:

```
end_dlt_all.bat
```

Be patient! This will take several minutes.

Instead of creating and starting the queue managers and running RUNMQSC to create the MQSeries objects, you can execute the bat file:

```
crt_str_all
```

This process, too, takes several minutes. You should see the four listener windows popping up. Also, check the log file to ensure that the environment has been created error free.

---

## 5.1 Some Comments about the Listener

If you have read the `crt_str_all.bat` file carefully, you might have noticed that four listeners are started manually with the `RUNMQLSR` command. In this respect, our new cluster is somewhat different from the one created using the MQSeries Explorer. In our present case, the Explorer is not around to build a listener for us and start it. Remember Figure 63 on page 60?

There are two points to make about this:

1. We could have unchecked **Create listener configured for TCP/IP** in Figure 63 on page 60. In that case we could have manually started our listeners just as you see now in `crt_str_all`.
2. Now that the objects have been created manually outside the Explorer, the Explorer will not start and stop listeners for the queue managers without further action. The further action is that you can go to the *MQSeries MQServices Console* (not the Explorer) and by right-clicking each queue manager you can select **New -> Listener**. If you create listeners this way, the Explorer will then manage them for you.
  - *The Advantage*: Explorer manages your listeners and you don't have windows on the desktop with listeners running in them.
  - *The Disadvantage*: You can't easily see the listeners' messages.

---

## 5.2 Some Comments about Cluster Objects

The scripts for QM\_1 and QM\_3 should be run *before* the scripts for QM\_2 and QM\_4. We want the two full repositories to be set up before other queue managers attempt to join the cluster.

The following queues and channels are among the default objects defined when you create a queue manager on V5.1 of MQSeries. Therefore we don't need to include them in our configuration scripts.

#### 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.

The above is from *MQSeries Queue Manager Clusters*, SC34-5349.



## Chapter 6. Workload Management

In this chapter, we use two programs to put messages to a queue, and we use the MQSeries Explorer to watch where the messages go. We will thus get some idea of how workload balancing functions.

The first program is amqsput.exe, which is exactly as provided in the tools directory when you installed MQSeries. This program simply puts messages on a queue within a queue manager. You name the queue and the queue manager as parameters when you invoke the command, thus:

```
amqsput <QueueName> <QueueManagerName>
```

The logic within amqsput flows like this:

- Do an MQCONN to a queue manager.
- Do an MQOPEN to a queue on that queue manager.
- Keep doing MQPUTS within a loop until the program user decides to stop.
- Do an MQCLOSE to the queue.
- Do an MQDISC to the queue manager.

The second program is clusput.exe. This program is really a copy of amqsput. The only change is that when the program sets its open options, this version sets the extra option MQOO\_BIND\_NOT\_FIXED described below.

```

/*****
/*
/*  Open the target message queue for output
/*
/*
/*****
O_options = MQOO_OUTPUT           /* open queue for output
             + MQOO_FAIL_IF_QUIESCING /* but not if MQM stopping
             + MQOO_BIND_NOT_FIXED; /* put to multiple clustered
                                     /* queues
MQOPEN(Hcon,                       /* connection handle
       &od,                        /* object descriptor for queue
       O_options,                  /* open options
       &Hobj,                      /* object handle
       &OpenCode,                 /* MQOPEN completion code
       &Reason);                  /* reason code

```

Figure 113. MQOPEN with BIND\_NOT\_FIXED

If you follow the program logic for amqsput (above), the difference between them is that:

- For amqsput, between the MQOPEN and the MQCLOSE all messages will be MQPUT to a single instance of the cluster queue <QueueName> (see above).

This doesn't mean that workload balancing isn't working! The balancing decision is made at MQOPEN time. If we run the whole program again, we should see a *different* queue instance selected.

*But*, once again, for this second invocation of amqsput, between the MQOPEN and the MQCLOSE all messages will be MQPUT to a single instance of the cluster queue.

- For clusput, a different instance of the cluster queue is selected for each MQPUT. That is to say, at MQOPEN time, no decision is made about the instance of the cluster queue to which messages will be MQPUT.

Having established all that, we are ready to work with the exercises. But before we begin, let us look at the new open options.

---

## 6.1 Controlling the Workflow

There are three bindings you can use to control how messages are distributed to cluster queues:

- MQOO\_BIND\_ON\_OPEN
- MQOO\_BIND\_NOT\_FIXED
- MQOO\_BIND\_AS\_Q\_DEF

The third option, bind as specified in the queue definition, is the default. When you define a queue you may specify for the DEFBIND keyword either the OPEN or NOTFIXED parameter, where OPEN is the default. Here is an example:

```
def q1 (MYQUEUE) CLUSTER (MYCLUSTER) DEFBIND (NOTFIXED)
```

The parameter specified in the queue definition is overwritten by what you choose in the MQOPEN.

- Programs that send a series of messages to another application should use BIND\_ON\_OPEN. It guarantees that all messages sent between the MQOPEN and MQCLOSE are put in the same instance of the queue.
- BIND\_NOT\_FIXED means that each single message is put into a different queue, either round-robin or according to the logic of a workload exit.

**Note:** If the local queue manager owns an instance of the queue, all messages will be placed in that queue.

---

## 6.2 A Workload Distribution Example

For this example, we use the configuration created in the previous chapter. Before we begin, let us clear the cluster queues CLQ\_ACROSS\_2\_3\_4 in QM\_2, QM\_3 and QM\_4. QM\_1 does not own an instance of this queue. Then let us see how the queue manager behaves when you use the bind options for the MQOPEN.

For this example we provide the following files:

*Table 7. Files for Workload Distribution Example*

File Name	Description	See
clusput.c	Source code of modified amqsput to put messages with BIND_NOT_FIXED	page 221
clusput.exe	Executable	
fastget.c	Source of modified amqsget to read messages from a queue even if they are too long for the buffer. This is useful as a simple way to clear a queue.	page 227
fastget.exe	Executable	
str_all.bat	A BAT file that starts all four queue managers and their listeners	page 110
end_all.bat	A BAT file that stops all four queue managers and their listeners	page 110

### 6.2.1 Getting Prepared

If you created the queue managers with scripts then the listeners are not automatically started unless you followed the guidelines in 5.1, “Some Comments about the Listener” on page 104.

We also include a handy program that you can run to clear queues instead of using the MQ Explorer. You invoke the program with the command:

```
fastget CL_ACROSS_2_3_4 QM_2
```

```

strmqm QM_1
strmqm QM_3
strmqm QM_2
strmqm QM_4
start "QM_1 Listening on 1415" runmqclsr -t TCP -p 1415 -m QM_1
start "QM_2 Listening on 1416" runmqclsr -t TCP -p 1416 -m QM_2
start "QM_3 Listening on 1417" runmqclsr -t TCP -p 1417 -m QM_3
start "QM_4 Listening on 1418" runmqclsr -t TCP -p 1418 -m QM_4

```

Figure 114. *str\_all.bat*

```

start endmqm -i QM_2
start endmqm -i QM_4
start endmqm -i QM_1
start endmqm -i QM_3
echo Hit ENTER WHEN ALL THE QUEUE MANAGERS HAVE STOPPED
pause
endmqclsr -m QM_2
endmqclsr -m QM_4
endmqclsr -m QM_1
endmqclsr -m QM_3
pause

```

Figure 115. *end\_all.bat*

## 6.2.2 Clearing a Cluster Queue

Obviously, if there are no messages on any instances of this queue, you can skip the next steps (at least for the first time).

You can use `fastget.exe` or the MQSeries Explorer as follows:

1. Inside MQSeries Explorer, look at the queue managers within cluster CL\_MQ51.  
Go to each of the queue managers QM\_2, QM\_3, and QM\_4.
2. Select **Queues**, then select (for each queue manager) the version of CLQ\_ACROSS\_2\_3\_4 which for *that queue manager* is displayed as Local under Queue Type.



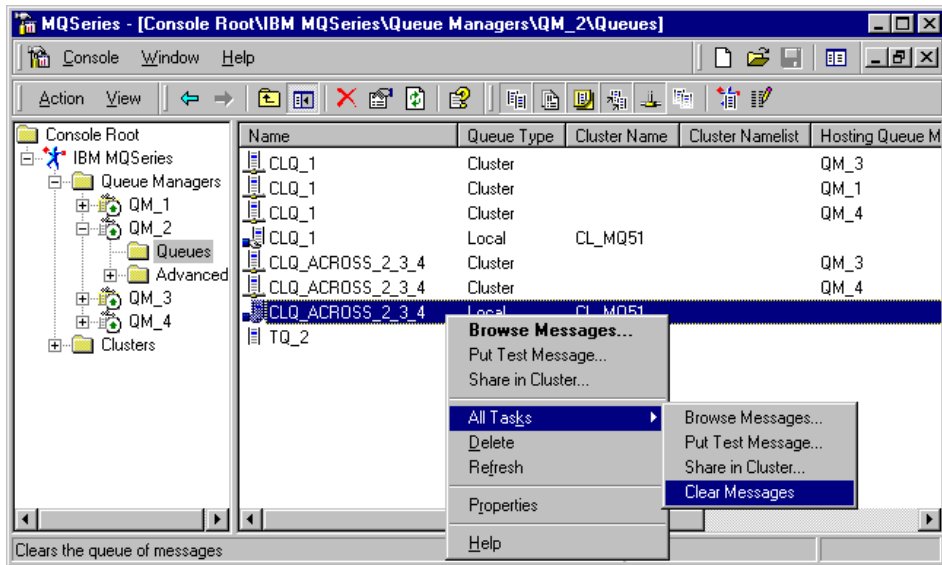


Figure 116. Clearing a Cluster Queue

3. Right click that queue. Select **All Tasks** from the menu and then **Clear Messages**.
4. Respond to Clear all messages from the queue? with a **Yes**.
5. Respond to The queue has been cleared of messages with **OK**.

### 6.2.3 Putting Using Bind On Open

We use the sample program amqsput to put some messages in a queue.

1. From a command prompt enter the command:

```
amqsput CLQ_ACROSS_2_3_4 QM_1
```

2. Next, you will see

```
Sample AMQSPUT0 start
target queue is CLQ_ACROSS_2_3_4
```

The program is ready for you to enter data (messages).

3. Type the following three lines on the command line: (You will need to press Enter after each line).

```
p1
p2
p3
```

- After the last line, press Enter a second time and you should see:

Sample AMQSPUT0 end

Figure 117 shows the window with the input in bold.

```

Microsoft (R) Windows NT (TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>amqsput CLQ_ACROSS_2_3_4 QM_1
Sample AMQSPUT0 start
target queue is CLQ_ACROSS_2_3_4
p1
p2
p3

Sample AMQSPUT0 end

C:\>

```

Figure 117. Putting Messages In a Queue

- Now go to MQSeries Explorer. *Where are your messages?*

If you haven't seen this aspect of Explorer yet, choose an instance of CLQ\_ACROSS\_2\_3\_4 (not the one on QM\_1, but one showing **Local** under Queue Type). The window shown in Figure 118 shows that CLQ\_ACROSS\_2\_3\_4 that belongs to QM\_3 contains three messages.

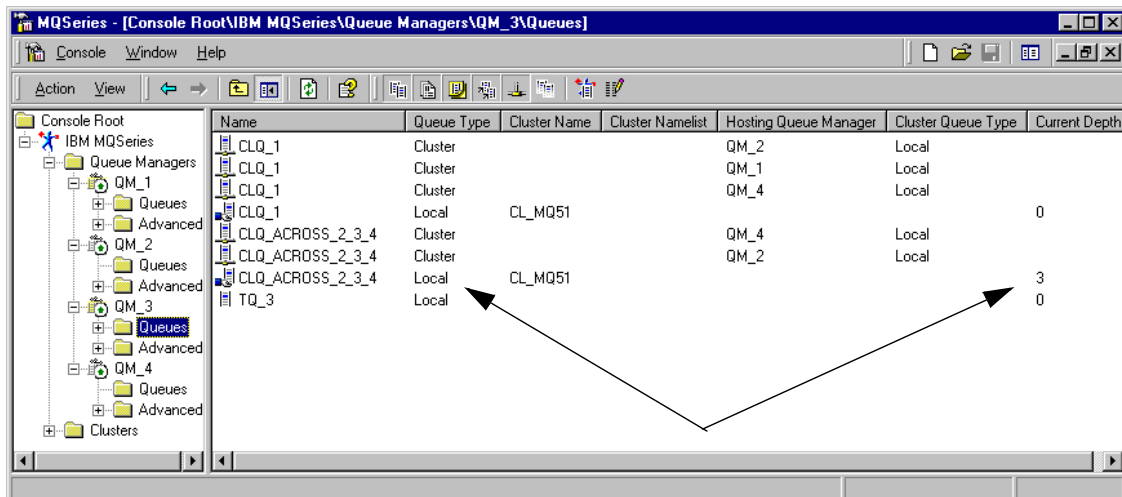
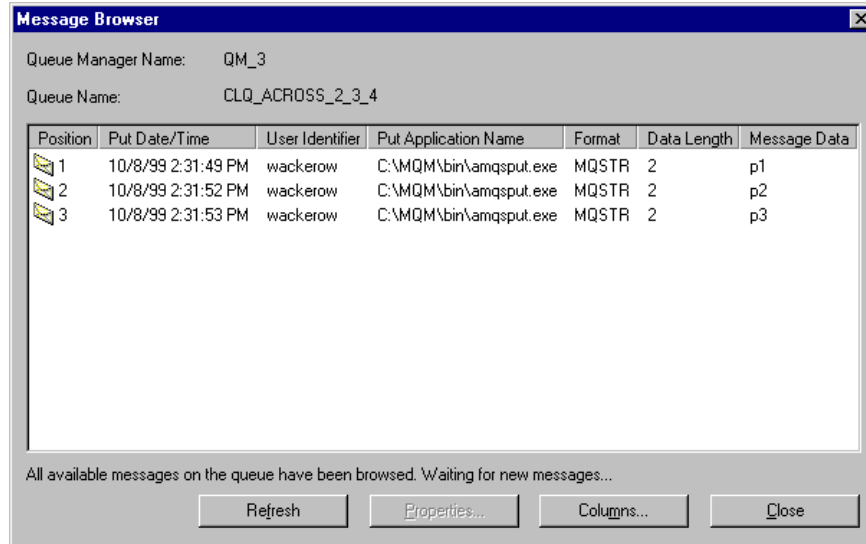


Figure 118. Displaying Depth of Cluster Queue

- Now right-click the queue and select **Browse Messages**. You can see the amount and the contents at the same time.



Why are the messages all on the same queue manager?

Before answering that, let us put some more messages.

- Return to the command prompt and enter three more messages, as you did before. You enter:

```
amqsput CLQ_ACROSS_2_3_4 QM_1
p4
p5
p6
```

- Now go to MQSeries Explorer.

*Why are p4, p5 and p6 on the same queue manager?*

*Are they on the same queue manager as p1, p2 and p3?*

*Why is that?*

**Note:** We had to exit the MQ Explorer and then start it again to make it show the current depths of the queue.

We have several instances of the same queue. The three messages (p4, p5, p6) are all in the same physical queue, p1, p2 and p3 are in a different physical queue on a different queue manager. This is because amqsput binds on open. If you put three more messages, then you will see that those are sent to the third queue manager. This proves that the workload distribution is truly round-robin.

## 6.2.4 Putting Using Bind Not Fixed

For this exercise we use `clusput.exe`, a modified version of `amqsput` that uses the option `MQOO_BIND_NOT_FIXED` as shown in Figure 113 on page 107.

Before you start you may clear the three instances of `CLQ_ACROSS_2_3_4` using either `fastget.exe` or the MQ Explorer as described in 6.2.2, “Clearing a Cluster Queue” on page 110.

1. Return to the command prompt and enter three (more) messages using `clusput` as shown below. You type the values shown in bold.

```
C:\>clusput CLQ_ACROSS_2_3_4 QM_1
MQ V5.1 Update Class: Workload-Balance enabled PUT
target queue is CLQ_ACROSS_2_3_4
c1
c2
c3

MQ V5.1 Update Class: Workload-Balance enabled PUT: end

F:\>
```

2. Go back to the MQSeries Explorer.

*Where are your messages?*

*It's a very different story with `clusput`, isn't it?*

Each instance of the queue has one (additional) message in it.

3. Return to the command prompt and enter three more messages, again with `clusput`. You enter:

```
clusput CLQ_ACROSS_2_3_4 QM_1
c4
c5
c6
```

4. Check with Explorer. *Is this what you expected?*

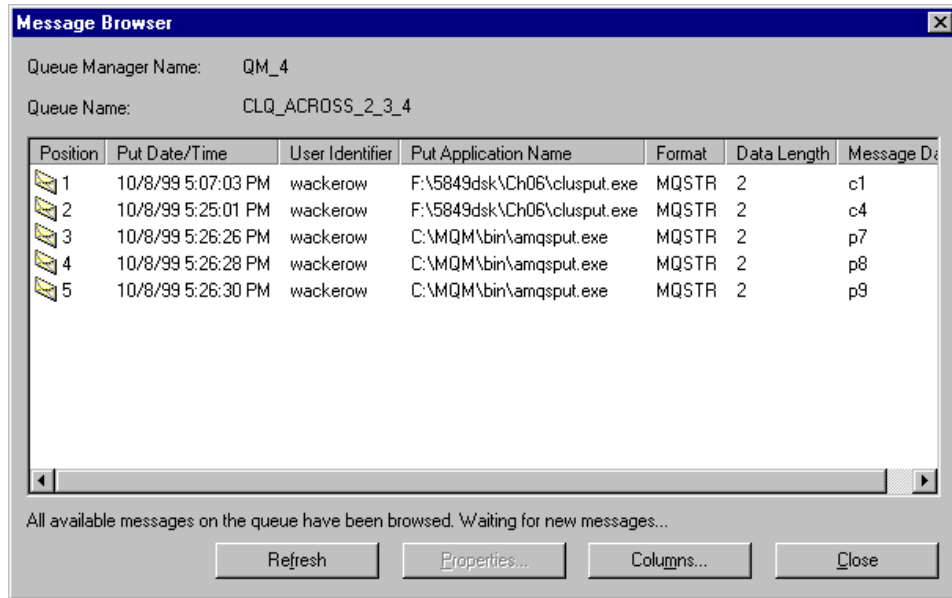
Yes, now there are two (additional) messages in each queue. The messages are distributed round-robin.

5. Return to the command prompt and enter three more messages, this time returning to using `amqsput`.

```
amqsput CLQ_ACROSS_2_3_4 QM_1
p7
p8
p9
```

6. Check with Explorer. *Did you predict this?*

The sample program amqsput uses the default MQOO\_BIND\_ON\_OPEN. Therefore, all three message go to the same instance of the queue. In the window below, you see that the first, fourth, and the three new messages are all in CLQ\_ACROSS\_2\_3\_4 of QM\_4.



### 6.2.5 Putting to a Local Cluster Queue

You can probably remember, back in 4.5, "Creating a Shared Cluster Queue" on page 83 or in Chapter 5, "Creating a Cluster with Scripts" on page 95 that you created a queue called CLQ\_1. This queue is very like CLQ\_ACROSS\_2\_3\_4, except that there exists a queue instance on every one of the four queue managers.

Repeat the steps described in the section above, but this time use CLQ\_1 as the queue name everywhere that you used CLQ\_ACROSS\_2\_3\_4.

*Do you understand the results?*

All messages have ended up on the same queue, which is QM\_1's instance of CLQ\_1. The queue manager always puts messages in a local instance if one exists.

---

## 6.3 Writing a Workload Management Exit

In this section, we describe how to write an MQSeries exit that will be called by the queue manager whenever it has to determine in which of a number of shared clustered queues it should put a message.

We will not actually make a decision about the queue. Our exit will be read only. It will examine some of the information that the queue manager has passed to it and then return without changing the DestinationChosen member of the MQWXP structure. This means that our exit accepts the choice that the queue manager had already made before our exit was called.

### 6.3.1 About the Example

We suggest that you begin by using the supplied workload exit shown in 6.3.2, “Commented Program Listing for Exit WLlogger.c” on page 118.

Also we suggest that you do your PUTs from QM\_1 to a cluster queue called CLQ\_ACROSS\_2\_3\_4. This is the same as in previous exercises with cluster queues. The reason is also the same. We want to work from a queue manager that doesn't have the named queue and we want to PUT to a queue name of which there are multiple instances in the cluster so that we can see how workload balancing works.

The emphasis here is not on seeing the workload balancing working. Proceed as in previous experiments with cluster queues if you want to do that. We now want to see how to drive our exit and look at the output it logs for various operations.

The exit supplied is in source form and executable (for Windows only). If you want to change/enhance the exit supplied, follow these instructions:

1. Compile the code using the command (for VisualAge C++):

```
icc /Ge- /Gm+ /Gf- WLlogger.c
```

2. You then need to copy the WLlogger.dll into the directory <mqmtop>\exits, for example:

```
c:\mqm\exits
```

3. Alter the queue manager so that it will drive your new exit:

```
ALTER QMGR CLWLDATA('<log_file_path_name>')  
ALTER QMGR CLWLEXIT('WLlogger(cwlFunction)')
```

**Notes:**

- The CLWLEXIT should be typed exactly as above, unless you rename your dll from WLlogger.DLL to something else.
- The CLWLDATA sets the <log\_file\_path\_name>. This is wherever you want your log to be. We used c:/temp/wllogger.log
- You could put these two ALTER QMGR statements in the scripts you built in Chapter 5, “Creating a Cluster with Scripts” on page 95 if you want. It would be a good idea because if you use end\_dlt\_all and crt\_str\_all you are bound to forget to put them in each time.
- These ALTER QMGR statements only need to be added to QM\_1 (or wherever you want to do your PUT from). Remember, the workload management (and associated exits) are driven from the MQPUT call.

REMEMBER: the queue manager passes the data in CLWLDATA to you program *as is*. It passes it in the ExitData field of MQWXP. It passes it to the clwFunction entry-point of DLL WLlogger.

Re-code WLlogger.c to do something you find interesting.

Remember, the queue manager hands your exit code an MQWXP. The queue manager will accept as your decision for the final destination *anything* that you put in the DestinationChosen field of that MQWXP. What would happen if you put some garbage characters in there?

You may think of some other interesting ways to decide on a DestinationChosen.

### 6.3.2 Commented Program Listing for Exit WLogger.c

```

/*****
/*
/* Original Program name: AMQSWLMO 1
/*
/*****
/* This exit has been altered for this book.
/* We use it now to print some interesting details of dynamic
/* workload balancing to a log.
/* We get the name of the log from the CLWLDATA
/* So you would set up the log name using:
/*
/* ALTER QMGR CLWLDATA('<log_file_path_name>') 2
/*
/* where <log_file_path_name> e.g. c:/temp/wlogger.log
/*
/* and you would enable the workload exit using:
/*
/* ALTER QMGR CLWLEXIT('WLogger(cwlFunction)') 3
/*****

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <cmqc.h>
#include <cmqxc.h>
#include <cmqfc.h>

__declspec (dllexport) void MQENTRY cwlFunction (MQWXP *parms); 4

void MQStart() {;} /* dummy entry point - for consistency only */

void MQENTRY cwlFunction (MQWXP *parms) 5
{
  MQLONG index;
  MQWDR * qmgr; 6
  MQWQR * queue; 7

```

- <sup>1</sup> This program was copied from a sample provided with MQSeries. It is now unrecognizable.
- <sup>2</sup> This is what you enter in the Cluster Workload Exit Data field. Right-click **Queue Manager** in the Explorer, select **Properties**, then click the **Cluster** tab. Now you see the Cluster Workload Exit Data field.
- <sup>3</sup> From the same Properties box as in step 2, enter literally WLogger(cwlFunction) in the Cluster **Workload Exit** field. This makes sure the queue manager calls our exit and our entry point.
- <sup>4</sup> This is for the preprocessor. We want it to know that we want *cwlFunction* to be exported in the DLL that we create. (Only exported functions can be called from outside the DLL.)
- <sup>5</sup> Out Entry Point. Notice that we are expecting an MQWXP structure when we are called and we are going to call it parms.
- <sup>6</sup> See the book “Queue Manager Clusters” Reference Section for the structure of MQWDR.
- <sup>7</sup> Likewise for MQWQR.



```

FILE *mylogger; 8
char myQmname[48], myQname[48]; 9

mylogger = fopen(parms->ExitData,"a"); 10
fprintf(mylogger, "\nWLlogger Workload Exit entered.\n"); 11

switch (parms->ExitReason) 12
{
case MQXR_INIT: fprintf(mylogger, "\tExit Initialisation\n"); 13
    break;

case MQXR_TERM: fprintf(mylogger, "\tExit Termination\n"); 14
    break;

case MQXR_CLWL_OPEN: fprintf(mylogger, "\tMQOPEN Processing\n");
    break;

case MQXR_CLWL_PUT: fprintf(mylogger, "\tMQPUT or MQPUT1 Processing\n");
    break;

case MQXR_CLWL_MOVE: fprintf(mylogger, "\tFrom MCA when msg state has changed\n");
    break;

case MQXR_CLWL_REPOS: fprintf(mylogger, "\tMQPUT or MQPUT1 for repos-mgr PCF msg\n");
    break;

case MQXR_CLWL_REPOS_MOVE: fprintf(mylogger, "\tFrom MCA for repos-mgr PCF msg\n");
    break;

default: fprintf(mylogger, "\tWarning: Cannot determine reason this Exit was called\n");
}

```

<sup>8</sup> We will use this file as our log file.

<sup>9</sup> Here we will store our own copies of qname and qmname. Basically we use these arrays to translate a type **MQCHAR48** into type **string**.

<sup>10</sup> We open the log file. Where do we get the file name? Do you understand `parms->ExitData`? Ask your lab instructor. This is how we get the data you stored in the queue manager's **Cluster Workload Exit Data**.

<sup>11</sup> A friendly message to make a log entry whenever we are called for *any* reason.

<sup>12</sup> Now starts the interesting bit. With `parms->ExitReason` we are looking into the `ExitReason` member of the `MQWXP` structure that the queue manager passed to us, when it called us. Look in the Reference Section of “*Queue Manager Clusters*”. Examine the layout of the **MQWXP** structure. Locate the `ExitReason` member. Consider the possible values. Now, even if you’re not sure of how a “C” program switch statement works, you should be able to work out what this switch statement is attempting to do.

<sup>13</sup> If `ExitReason` is `MQXR_INIT` we will write to the log that the exit has been called for Exit Initialization.

<sup>14</sup> And so on for the other case entries of the switch.

```

for (index = 0; index < parms->DestinationCount; index++) 15
{
    qmgr = parms->DestinationArrayPtr[index]; 16
    memcpy (myQMname, qmgr->QMgrName, 48); 17
    *(myQMname + strchr(myQMname, " ")) = 0x00; 18
    fprintf(mylogger, "\t\tFOR QUEUE MANAGER: %s\n", myQMname); 19

    switch (qmgr->QMgrFlags) 20
    {
        case MQQMF_REPOSITORY_Q_MGR:
            fprintf(mylogger, "\t\tDest is a Repos Qmgr\n");

        case MQQMF_CLUSSDR_USER_DEFINED:
            fprintf(mylogger, "\t\tClus sender channel was defined manually\n");

        case MQQMF_CLUSSDR_AUTO_DEFINED:
            fprintf(mylogger, "\t\tClus sender channel was defined automatically\n");

        case MQQMF_AVAILABLE:
            fprintf(mylogger, "\t\tDest Qmgr available to receive messages\n");
        default:
            fprintf(mylogger, "\t\tNothing meaningful in MQWDR:QMGrFlags\n");
    }
}

```

- <sup>15</sup> Look at **DestinationCount** in **MQWXP**. Can you see why we need a for loop here?
- <sup>16</sup> On pass **index** through the loop, we will be pulling the appropriate **qmgr** from the **DestinationArrayPointer** of **MQWXP**.
- <sup>17</sup> Copy 48 characters from array **QMGrName** in **MQWXP** into array **myQMname**.
- <sup>18</sup> Find the first blank (0x20) in myQMname and turn into a binary zero (0x00). This makes mqQMname a “proper” string in “C” terms. Strings in “C” must be terminated by a binary zero (0x00).  
 Actually this coding is not quite correct because the first parameter of strchr should be a string. mqQMname is not a string when strchr is called. This means that in theory we could “run off the end of mqQMname” into storage we shouldn’t be accessing, looking for the 0x00 which indicates the end of the string. We get away with it because we know there will be a blank (0x20) in the array before we run off the end. You should write better code than this.
- <sup>19</sup> Write a line (and indent it for clarity) to the log to say we are now looking at details of a queue manager.
- <sup>20</sup> You should be able to work out this **switch**, by thinking about the last one. We are making our decisions (in the **case** statements) based on the **QMGrFlags** member of structure **MQWXP**.

```

switch (qmgr->ChannelState) 21
{
  case MQCHS_INACTIVE:
    fprintf(mylogger, "\t\tChannel to %s not active\n", myQMname);
    break;

  case MQCHS_BINDING:
    fprintf(mylogger, "\t\tChannel to %s binding\n", myQMname);
    break;

  case MQCHS_STARTING:
    fprintf(mylogger, "\t\tChannel to %s starting\n", myQMname);
    break;

  case MQCHS_RUNNING:
    fprintf(mylogger, "\t\tChannel to %s running\n", myQMname);
    break;

  case MQCHS_STOPPING:
    fprintf(mylogger, "\t\tChannel to %s stopping\n", myQMname);
    break;

  case MQCHS_RETRYING:
    fprintf(mylogger, "\t\tChannel to %s retrying\n", myQMname);
    break;

  case MQCHS_STOPPED:
    fprintf(mylogger, "\t\tChannel to %s stopped\n", myQMname);
    break;

  case MQCHS_REQUESTING:
    fprintf(mylogger, "\t\tChannel to %s requesting connection\n", myQMname);
    break;
  case MQCHS_PAUSED:
    fprintf(mylogger, "\t\tChannel to %s paused\n", myQMname);
    break;
  case MQCHS_INITIALIZING:
    fprintf(mylogger, "\t\tChannel to %s initialising\n", myQMname);
    break;
  default:
    fprintf(mylogger, "\t\tCannot determine state of the channel to Qmgr %s\n", myQMname); 22
}

```

<sup>21</sup> You should be able to work the rest out for yourself now. Switching on the ChannelState field of the MQWDR structure pointed to by qmgr... right? Why are there break statements in this switch? There weren't any in the previous one.

<sup>22</sup> The default statement is there in case none of the case statements match. It is not mandatory to have a default.

```

if (parms->QArrayPtr) 23
{
    queue = parms->QArrayPtr[index]; 24
    memcpy (myQname, queue->QName, 48); 25
    *(myQname + strchr(myQname, " ")) = 0x00;
    fprintf(mylogger, "\t\t\tFOR QUEUE: %s\n", myQname); 26

    switch (queue->QFlags) 27
    {
        case MQQF_LOCAL_Q:
            fprintf(mylogger, "\t\t\tThis is a LOCAL queue\n");
            break;
        default:
            fprintf(mylogger, "\t\t\tCan't determine useful information from MQWQR:QFlags\n");
    }
switch (queue->DefBind) 28
{
    case MQBND_BIND_ON_OPEN:
        fprintf(mylogger, "\t\t\tDefault Binding; done on MQOPEN\n");
        break;
    case MQBND_BIND_NOT_FIXED:
        fprintf(mylogger, "\t\t\tDefault Binding; not Fixed\n");
        break;
    default:
        fprintf(mylogger, "\t\t\tCannot determine default binding for this queue\n");
    }

    switch (queue->DefPersistence)
    {
        case MQPER_PERSISTENT: fprintf(mylogger, "\t\t\tPersistent by default: YES\n");
            break;
        case MQPER_NOT_PERSISTENT: fprintf(mylogger, "\t\t\tPersistent by default: NO\n");
            break;
        default: fprintf(mylogger, "\t\t\tCannot determine default persistence of queue\n");
    }
    }
}
fclose(mylogger); 29
return;
}

```

- <sup>23</sup> In “C”, any integer that is non-zero is deemed to have a value TRUE. Zero is FALSE. Why do we need to test if the QArrayPtr field of MQWXP is greater than zero? We didn’t do that for DestinationArrayPtr. Read the “Queue Manager Clusters” book again.
- <sup>24</sup> Just as the last time, we used index. We want a different queue each time through the for loop. We get this by stepping through the array of pointers provided to us in QArrayPtr in MQWXP. (Yes we’re still in the for loop. The indentation should help you work this out.)
- <sup>25</sup> This is the same “fiddle” as we saw before. We have an MQCHAR48 and we want a string.
- <sup>26</sup> Tell the logfile that we are now looking at details that pertain to a queue.
- <sup>27</sup> You can work this out now.
- <sup>28</sup> It’s time you coded a few of these switch / case constructs yourself. See what other data you might like to log.
- <sup>29</sup> It is *most essential* to close the mylogger stream. Don’t rely on cleanup at program exit time to close open streams. Remember, this is a DLL.

---

## Chapter 7. MQSeries Administration and Service

There are many new functions and options for administration available in MQSeries Version 5.1; they are summarized below.

### MQSeries 5.1 Administration Interfaces

- **Web Administration**

A new NT web server that supports a 'browser' interface to RUNMQSC that includes a basic script file management facility. It enables you to construct more complex scripts that use conditional logic, looping, nesting and so on;

and it supports the configuration and administration of multiple local and remote Queue Managers on different platform types.

- **Microsoft Management Console (MMC)**

These facilities are available on *Windows NT only*.

- ▶ **MQSeries Explorer**

A full 'Windows-style' GUI interface that makes the use of Control Commands and RUNMQSC obsolete for the creation, configuration, and operation of multiple local and remote Queue Managers.

- ▶ **MQSeries Services 'snap-in'**

An MMC snap-in, accessible from the task-bar tray, for monitoring and controlling local Queue Managers.

Replaces the SCMMQM command (which is no longer available) for automating start up of MQSeries components.

- **Control Commands & RUNMQSC** (no change, may still be used from the NT command line.)

- **MQSeries Administration Interface (MQAI)**

A new programming interface, for procedural and OO languages, that makes it *much* easier to use PCFS.

- **Programmable Command Format (PCF)** (no change)

Which techniques are the most appropriate for a particular operation depends mainly on the platform type.

On Windows NT, you can carry out most common administration and operations tasks using the MQSeries Explorer and MQSeries Services GUI tools. These tools make Windows NT a very convenient environment for experimentation and development. However, it is much more efficient to use one of the scripting techniques to populate and manipulate queue managers once they have been created.

On other platforms where the MQSeries Explorer is not available, you must use MQSeries control commands for some tasks, for example, when you want to create a new queue manager.

If your network configuration includes a Windows NT server with MQSeries Version 5.1 installed on it, you can use the new *Web Administration* facility. Otherwise, you will have to use runmqsc interactively or with a script file, as you did with previous versions of MQSeries.

*Web Administration* enables you to use a Web Browser to do everything you can do using runmqsc. In addition, it enables you to construct more sophisticated scripts that can include conditional logic, looping, nesting, and so on. And it includes a simple file management capability that you can use to organize your script files in public and private data stores.

The *Web Server* that hosts these new facilities runs only on Windows NT, but the browser that provides the human interface can run on any platform that supports a *Java-enabled Web Browser*, such as Netscape Navigator<sup>1</sup> or the Microsoft Internet Explorer.

The MQSeries Services “snap-in” for the *Microsoft Management Console* (MMC) and the MQSeries Explorer provide new operations and administration interfaces for queue manager configurations on Windows NT. Although you can still use MQSeries control commands (for example, crtmqm, strmqm, and runmqsc) for many functions, the visual power and ease of use of the new GUI tools is likely to make them very attractive to customers who are accustomed to working with other products in the Windows NT environment.

The *MQSeries Explorer* is mainly concerned with MQSeries objects. Use it to define and configure queue managers, channels, queues, processes, namelists, and so on. It includes a message browser, and can be used to put “test” messages onto queues. You can also operate your MQSeries configuration from the Explorer. For example, manually start and stop queue managers and channels.

The *MQSeries Services* “snap-in” is mainly concerned with MQSeries processes. Use it to control how you want your configuration to behave, for example, which processes should be started automatically when Windows NT is booted, whether you want a process to be restarted when it fails, or should the NT system be rebooted. You can also operate the configuration manually from here.

**Note:** The scmmqm command, which you used in Version 5.0 and earlier versions when you wanted a queue manager or another MQSeries component to start automatically when Windows NT was booted, no longer

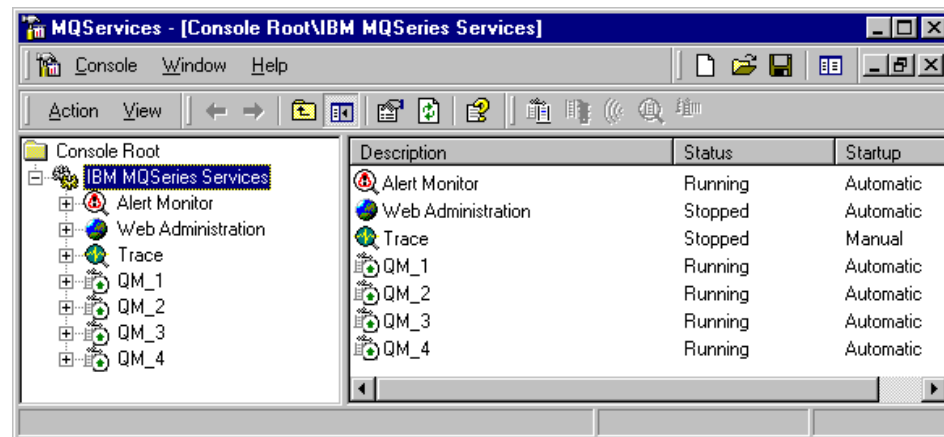
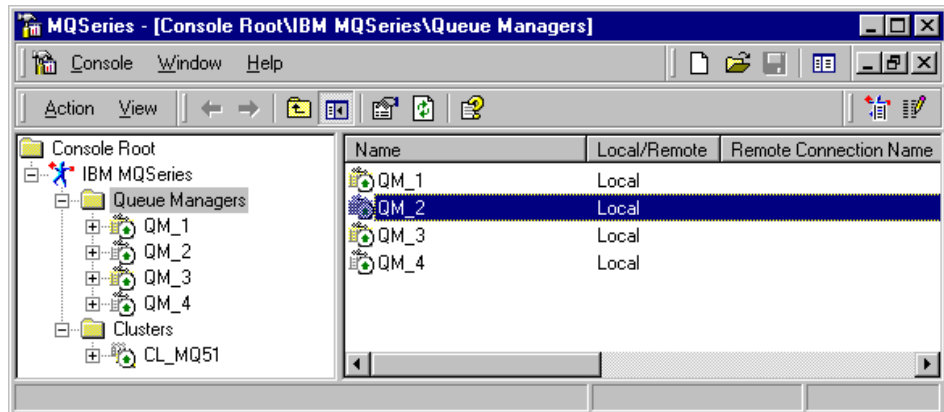
<sup>1</sup> We found that browser performance using the current Netscape product for Web Administration on the same platform as the Web Administration Server is poor compared to the Internet Explorer.

exists. In MQSeries Version 5.1 for Windows NT, the function of scmmqm is replaced by a function of the Services snap-in; the scmmqm command is no longer shipped with the product.

## 7.1 Experiments with Runmqsc and Clusters

The example described in this section is based on the configuration with the four queue managers we created in Chapter 4, “Creating a Cluster with the MQExplorer” on page 55. For this exercise, we use tools other than the MQSeries Explorer to create a fifth queue manager, QM\_5, and demonstrate that it can participate an the cluster CL\_MQ51 that has been set up earlier.

First, bring up the MQ Explorer and MQ Services GUIs. The windows should contain the following:



## 7.1.1 Creating a Queue Manager

Open a command prompt window and issue the command:

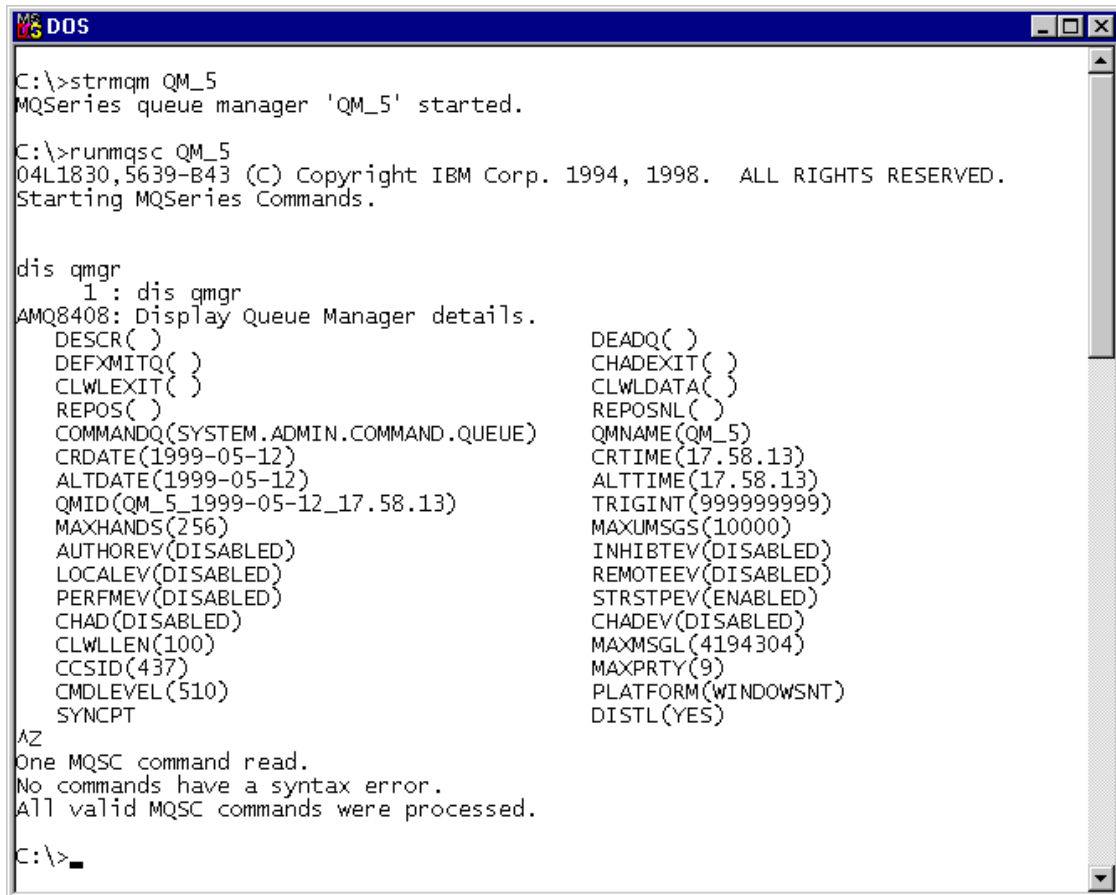
```
crtmqm QM_5
```

The new queue manager will appear automatically in the MQ Services GUI, but not in the MQ Explorer window, not even if you refresh it. You have to recycle the MQ Explorer.

To verify that the queue manager works, start it with the command:

```
strmqm QM_5
```

Run an application that uses it, such as runmqsc, and display the properties of the queue manager. You will see that it works.



```
DOS
C:\>strmqm QM_5
MQSeries queue manager 'QM_5' started.

C:\>runmqsc QM_5
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998.  ALL RIGHTS RESERVED.
Starting MQSeries Commands.

dis qmgr
  1: dis qmgr
AMQ8408: Display Queue Manager details.
DESCR( )
DEFXMITQ( )
CLWLEXIT( )
REPOS( )
COMMANDQ(SYSTEM.ADMIN.COMMAND.QUEUE)
CRDATE(1999-05-12)
ALTDATE(1999-05-12)
QMID(QM_5_1999-05-12_17.58.13)
MAXHANDS(256)
AUTHOREV(DISABLED)
LOCALEV(DISABLED)
PERFMEV(DISABLED)
CHAD(DISABLED)
CLWLLEN(100)
CCSID(437)
CMDLEVEL(510)
SYNCPT
DEADQ( )
CHADEXIT( )
CLWLDATA( )
REPOSNL( )
QMNAME(QM_5)
CRTIME(17.58.13)
ALTTIME(17.58.13)
TRIGINT(999999999)
MAXUMSGS(10000)
INHIBTEV(DISABLED)
REMOTEEV(DISABLED)
STRSTPEV(ENABLED)
CHADDEV(DISABLED)
MAXMSGL(4194304)
MAXPRTY(9)
PLATFORM(WINDOWSNT)
DISTL(YES)

AZ
One MQSC command read.
No commands have a syntax error.
All valid MQSC commands were processed.

C:\>_
```



Now check the status of QM\_5 using the GUI interfaces:

- Both GUIs recognize the new queue manager and show it running (provided you recycled the MQ Explorer).
- MQServices indicates that the star-up for QM\_5 is manual.
- When you click **QM\_5** you will see that the following two components are running:
  - Queue Manager
  - Command Server
- When you expand one of the queue managers created with the MQ Explorer, you will see two additional components:
  - Channel Initiator
  - Listener

*What can go wrong?*

- When `crtmqm` fails with error AMQ7077, you are not authorized to perform the requested operation.

When you want to use MQSeries control command, you must be logged on as a member of the `mqm` security group.

- When `strmqm` or `runmqsc` fail with error AMQ8118, the queue manager does not exist. Check *spelling* and *case* of the object names.

Before QM\_5 will be able to participate in the cluster it will need a listener to handle requests from partner queue managers to start inbound channels, and a channel initiator to start outbound channels.

### 7.1.2 Starting the Listener

Open a command prompt window and issue the command:

```
start runmqslsr -t tcp -p 1419 -m QM_5
```

This brings up another window showing messages from the channel initiator process.

**Note:** Since all five queue managers run in the same machine, their listeners must have different ports.

### 7.1.3 Starting the Channel Initiator

Try to start the channel initiator with the command:

```
runmqchi -m QM_5
```

You will see the error message shown below.

```
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

C:\>start runmqslsr -t tcp -p 1419 -m QM_5

C:\>runmqchi -m QM_5
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
10/11/99 13:45:51 AMQ9509: Program cannot open queue manager object.

C:\>
```

This command fails because the queue SYSTEM.CHANNEL.INITQ is already open for input. The clue for this is in the amqerr01.log file for QM\_5. You find this file in the directory c:\mqm\qmgrs\QM\_5\errors.

```
-----
10/11/99 13:45:51
AMQ9509: Program cannot open queue manager object.
```

EXPLANATION:

The attempt to open either the queue or queue manager object 'SYSTEM.CHANNEL.INITQ' on queue manager 'QM\_5' failed with reason code 2042.

ACTION:

Ensure that the queue is available and retry the operation.

```
-----
```

Error code 2042 means that an object is in use.

You only just created the queue manager. *Which process could have the SYSTEM.CHANNEL.INITQ open?*

The channel initiator starts *automatically* when you start a Version 5.1 queue manager.

***This is a change from Version 5.0 and earlier versions.***

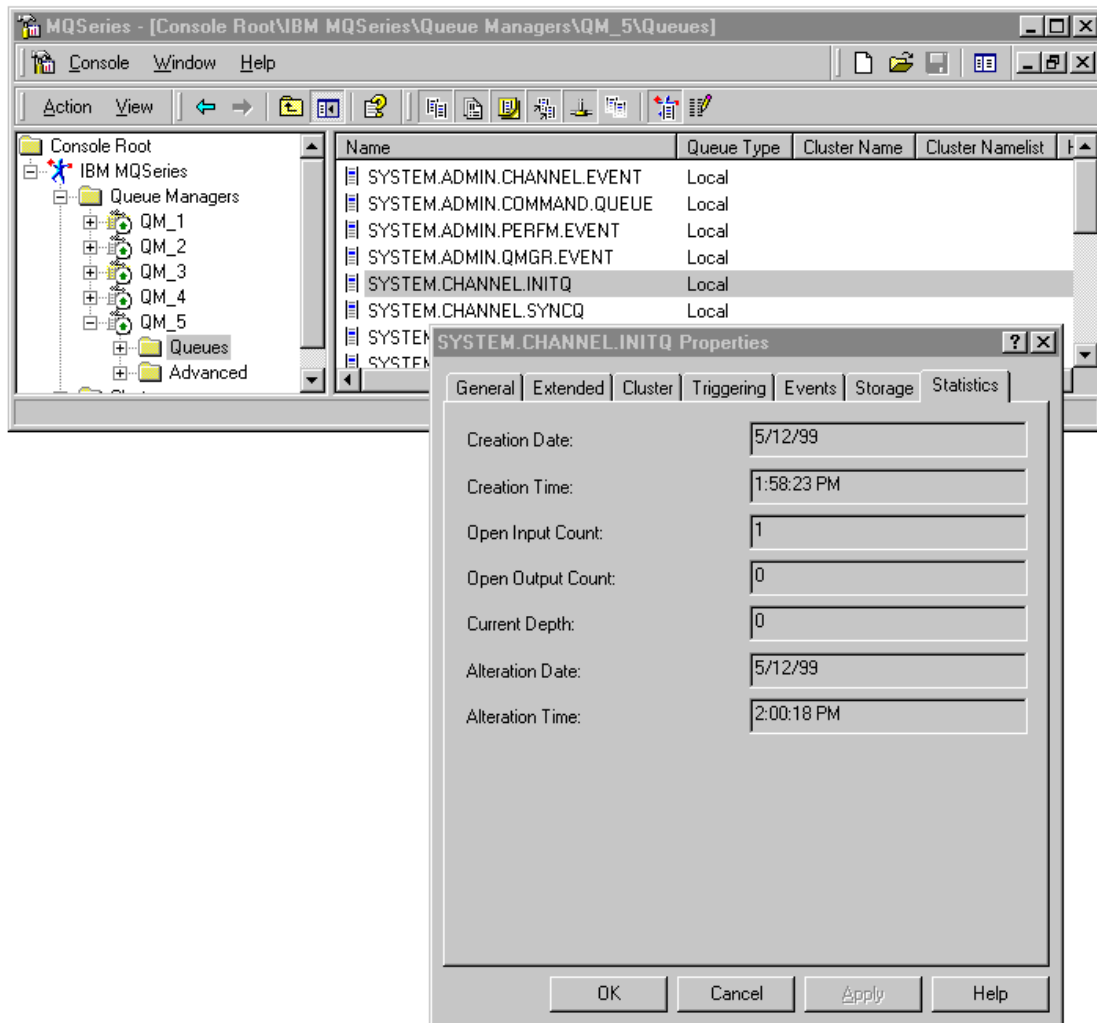


Figure 119. SYSTEM.CHANNEL.INITQ

#### 7.1.4 Connecting QM\_5 to the Existing Cluster

Now we have to make QM\_5 part of the cluster CL\_MQ51. To do this, we have to define two channels:

- A cluster sender channel pointing to one of the two existing repository queue managers in the cluster
- A cluster receiver channel pointing back to QM\_5

We use `runmqsc` to define the channels. Then we verify that `QM_5` is now part of the cluster by displaying the cluster queue managers as seen by `QM_5`. This is shown in Figure 120.

```
C:\>runmqsc QM_5
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
Starting MQSeries Commands.

def chl(TO_QM1) chltype(clussdr) cluster(CL_MQ51) trptype(tcp) +
  3 : def chl(TO_QM1) chltype(clussdr) cluster(CL_MQ51) trptype(tcp) +
conname('127.0.0.1(1415)') replace
  : conname('127.0.0.1(1415)') replace
AMQ8014: MQSeries channel created.
def chl(TO_QM5) chltype(clusrcvr) cluster(CL_MQ51) trptype(tcp) +
  4 : def chl(TO_QM5) chltype(clusrcvr) cluster(CL_MQ51) trptype(tcp) +
conname('127.0.0.1(1419)') replace
  : conname('127.0.0.1(1419)') replace
AMQ8014: MQSeries channel created.
dis clusqmgr(*)
  5 : dis clusqmgr(*)
AMQ8441: Display Cluster Queue Manager details.
      CLUSQMGR(QM_1)                CLUSTER(CL_MQ51)
      CHANNEL(TO_QM1)
AMQ8441: Display Cluster Queue Manager details.
      CLUSQMGR(QM_3)                CLUSTER(CL_MQ51)
      CHANNEL(TO_QM3)
AMQ8441: Display Cluster Queue Manager details.
      CLUSQMGR(QM_5)                CLUSTER(CL_MQ51)
      CHANNEL(TO_QM5)
```

Figure 120. Adding a Queue Manager to a Cluster Using `runmqsc`

We can see in Figure 120 that `QM_5` knows about both repository queue managers. At this time `QM_5` is not interested in `QM_3` and `QM_4`. Look in the listener's window and you will see that the channel programs have started.

Now go to the MQ Explorer GUI and expand the **QM\_5** tree.

- Click on **Channels**. Figure 121 on page 131 shows that the Explorer recognizes the channels and shows that they are running.
- Next expand the **Cluster Queue Manager** branch as shown in Figure 122 on page 131. You will see the two repository cluster queue managers and `QM_5` itself.
- When you expand the **Queue** branch of `QM_5` you will notice that all queue instances that exist in the other queue managers are visible, namely three instances of `CLQ_ACROSS_2_3_4` and four instances of `CLQ_1`.

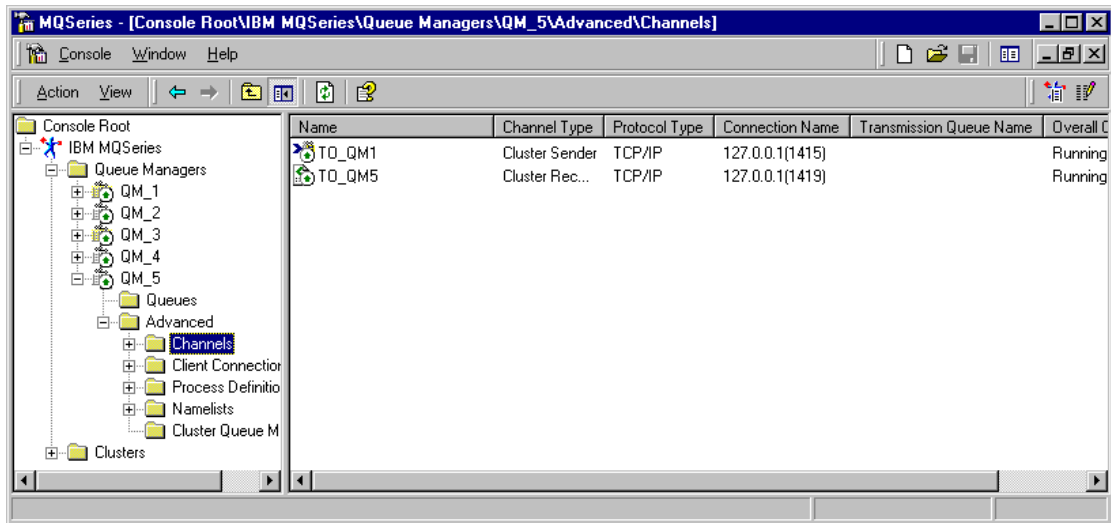


Figure 121. Channels for QM\_5

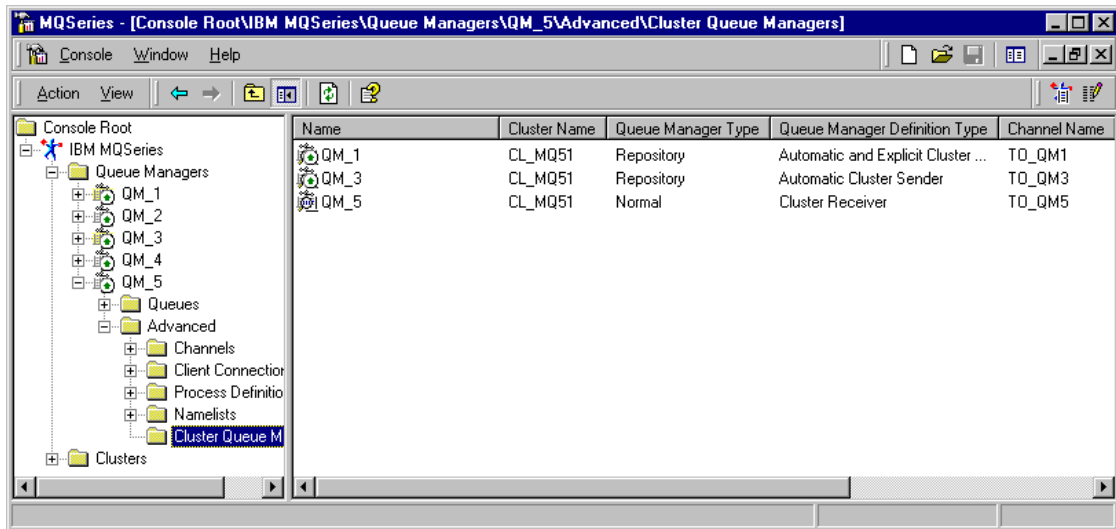


Figure 122. QM\_5's View of Cluster Queue Managers

Now refresh the Services GUI. You will see that the command server is running. Within a cluster, MQSeries uses PCF (programmable command format) to communicate with other cluster members. When a queue manager

joins a cluster, MQSeries starts the command server for the new queue manager automatically.

## 7.2 Experiments with MQSeries Services

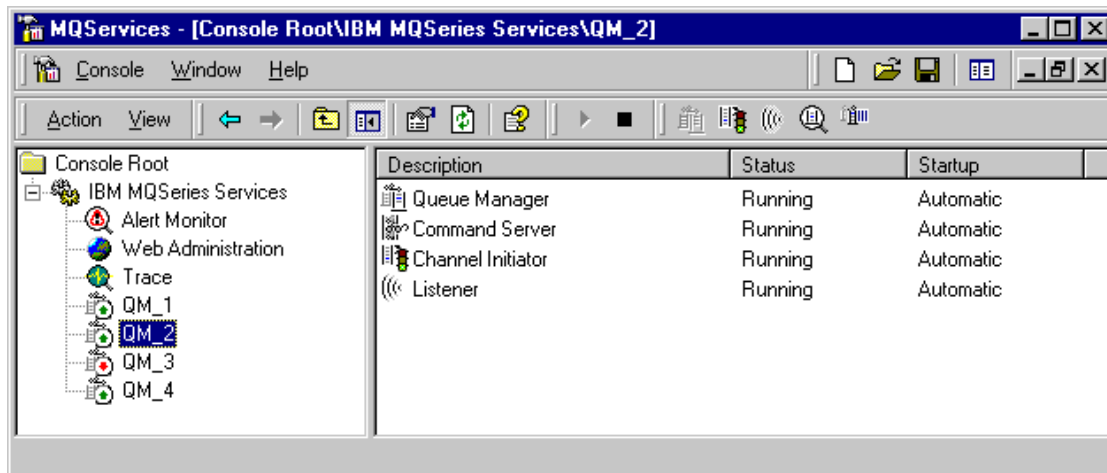
If you follow the exercises described in this section, you will become familiar with the mode of operation and basic functions of the MQSeries Services snap-in.

You can open the MQSeries Services window by right-clicking the icon shown on the right in the task bar. To use MQSeries Services, you must have the required level of authority.



If the icon is not visible, press Ctrl+Alt+Del and check which user you are currently logged in as. Make sure this user is in the mqm or Administrator group, or log out and in again as a user that is.

From the menu, select **MQSeries Services**.



The left-hand pane of the MQSeries Services window displays a tree containing icons for the Alert Monitor, Web Administration server, trace, and for each of the queue managers on the local machine.

The right hand pane shows an expanded view of the currently selected item in the tree. For example, when you select a queue manager, it lists the main processes defined for it, and their current status, for example the queue manager itself, listener, channel initiator, trigger monitor, whether they are running, and whether they are under manual or automatic control.

The view presented in the Services window is dependent on your actual configuration. It is built from the MQSeries information stored in the Registry. Usually, it is updated automatically. You cannot tailor the information displayed in the Services window. This is different from the Explorer, where you can configure many aspects of its display and, because it can contain so much more data, you often have to refresh it manually.

The following sections explain how to use the MQSeries Services GUI to perform the following tasks:

1. How to define that a queue manager starts automatically or manually
2. How to view manipulate queue manager properties using the Registry Editor and MQ Services
3. How to create a queue manager using MQ Services
4. How to add a trigger monitor
5. How to configure and use the Alert Monitor

### 7.2.1 Automatic or Manual Start-up

As mentioned before, in MQSeries for Windows NT Version 5.1 the command `scmmqm` does not exist any more. You use the MQSeries Services window to describe what components to start when you boot your system.

In the following scenario, we set QM\_1 to start up manually and leave all others to start automatically.

1. Right-click **QM\_1** and select **Properties** from the menu.
2. Click the **General** tab (if not already displayed).
3. From the list of start up types (shown in Figure 123 on page 134) select **Manual** and click **OK**.
4. You will notice that the MQSeries Services window changed and now shows Manual under Startup. QM\_5 is still running, of course.
5. Now stop all queue managers that are running<sup>2</sup>. Be patient and watch the animation.

<sup>2</sup> Right-click a queue manager, select **All Tasks** and then **Stop**.

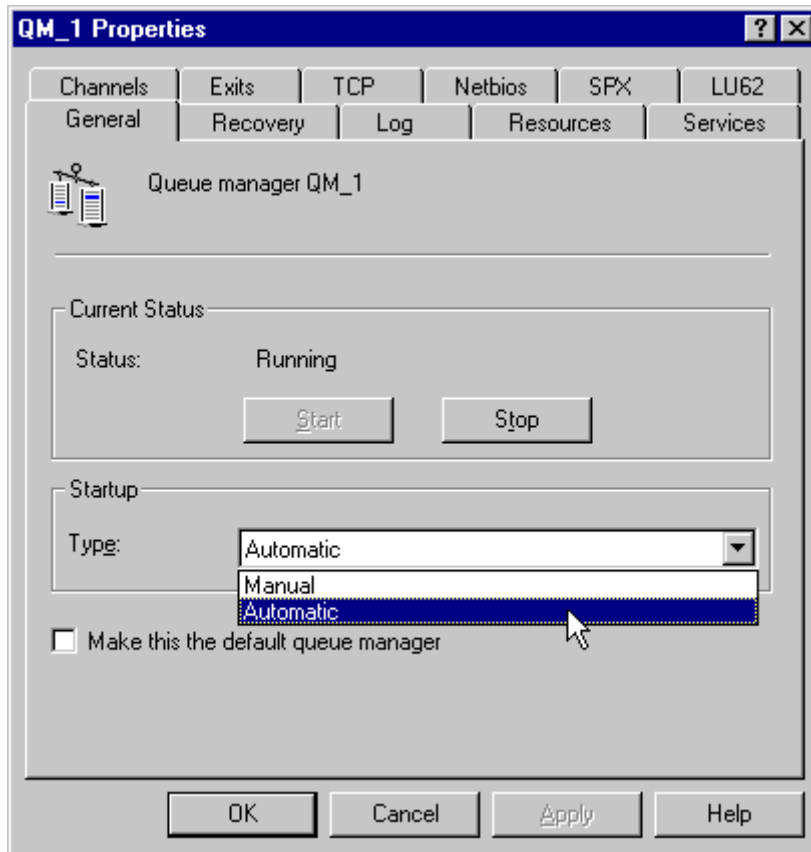


Figure 123. Queue Manager Properties

6. Reboot your Windows NT system. *Be patient again.* It may take a few minutes.
7. Verify that QM\_1 didn't start and that the other queue managers did.
8. Reboot Windows NT again without stopping any of the MQSeries components.

You will see again that all but QM\_1 are up and running.

*Are you impressed with the robustness of MQSeries Version 5.1?*

### 7.2.2 How to Start a Queue Manager Manually

To start a queue manager, say QM\_3, manually, right-click the **QM\_3** icon, select **All Tasks** and then **Start** as shown in Figure 124. Each of the



processes defined for QM\_3 will be started in turn, as displayed in the animated progress box.

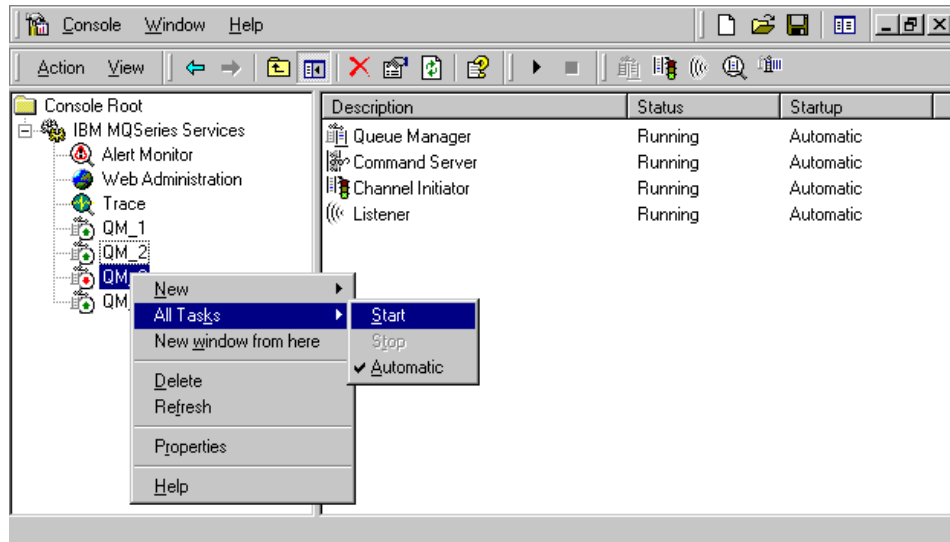
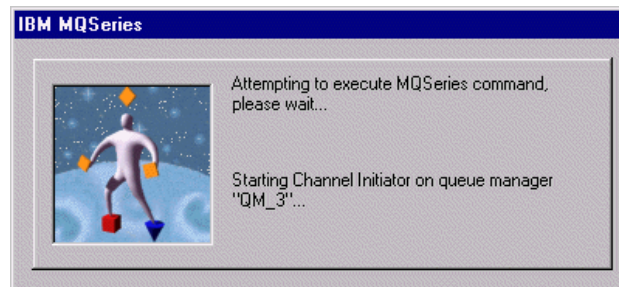


Figure 124. Starting a Queue Manager Manually



### 7.2.3 Working with Queue Manager Properties

Flip through the dialogue tabs shown in the Properties window in Figure 123 on page 134 and display some of the other Properties panels. Compare the configurable options in say, the Log Property page, with the MQS.INI and QM.INI files in MQSeries Version 5.0 or Version 2. You will notice that the entries look familiar.

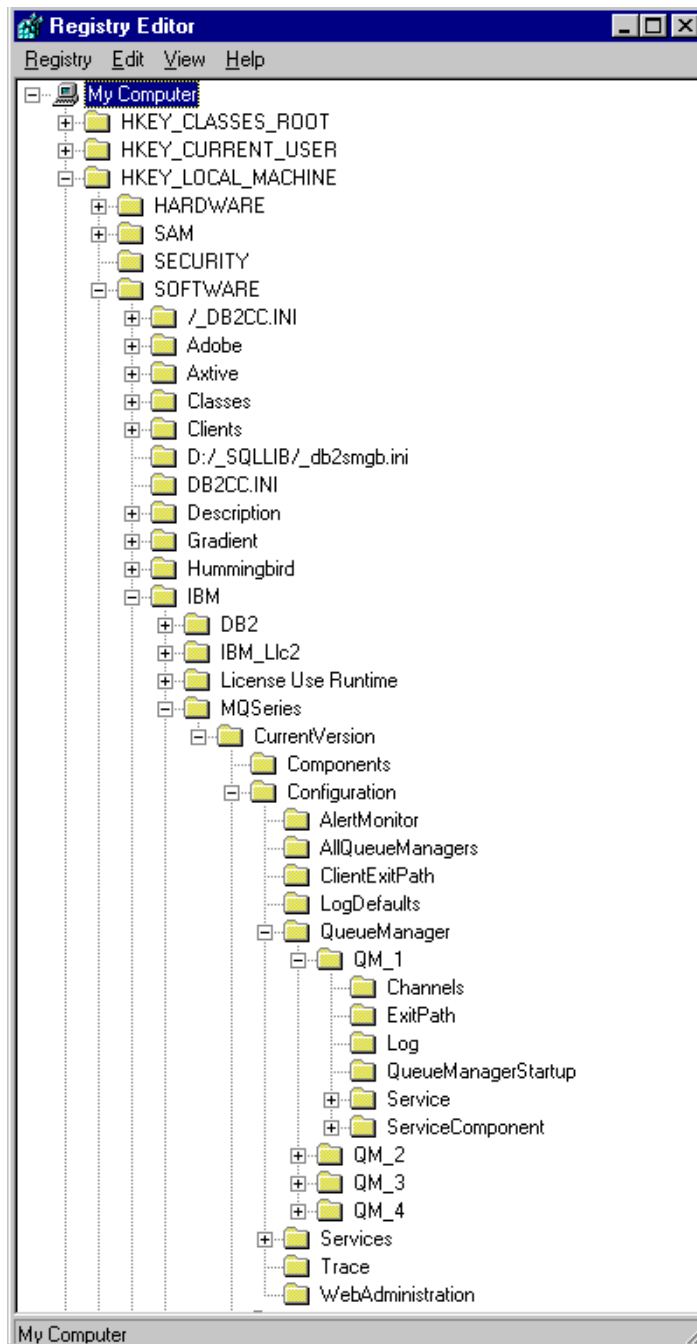


Figure 125. MQSeries Registry Entries

All of the .INI file information has been moved to the Windows NT Registry. MQSeries Services and Explorer display and enable you to update the Registry entries for MQSeries.

You can display the MQSeries entries in the Registry using the Windows NT regedt32 utility. The correspondence between the structures in the Registry and the MQSeries Services and Explorer displays is fairly obvious.

However, you should use *only* the Property pages in the Services tabbed dialogues to update the MQSeries information in the Registry.

If you update the Registry using regedt32 and make a mistake, you can make Windows NT unstable - you may even destroy it.

**Notes:**

- To start the Windows NT Registry Editor (regedt32) click the **Start** button, select **Run**, type `regedt32` in the text box and press Enter.
- To display the MQSeries Entries in the Windows NT Registry, expand the following keys in the left-hand window of the regedt32 display:

```
HKEY_LOCAL_MACHINE
  SOFTWARE
    IBM
      MQSeries
        CurrentVersion
          Configuration
            QueueManager
```

An example is shown in Figure 125 on page 136.

#### 7.2.4 Creating a Queue Manager from the Services GUI

Yes, you can use MQSeries Services to create a queue manager. Let us create the queue manager QM\_0.

1. In the Services window, right-click **IBM MQSeries Services**, select **New** from the menu and then **Queue Manager** as shown in Figure 126 on page 138.
2. Type the name `QM_0`.
3. Check the boxes **Start Queue Manager** and **Create a Server Connection Channel**.
4. Enter a free port, such as 1420.

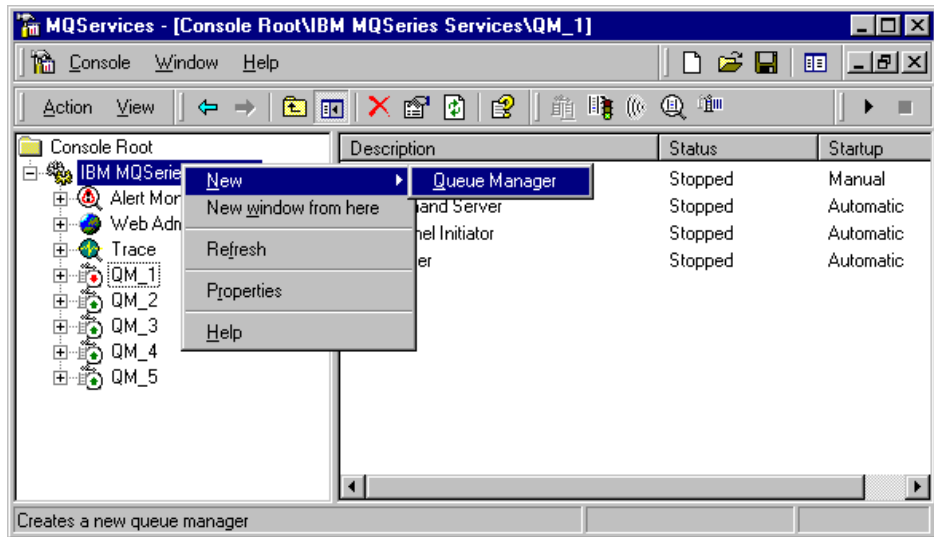


Figure 126. MQ Services - Creating a Queue Manager

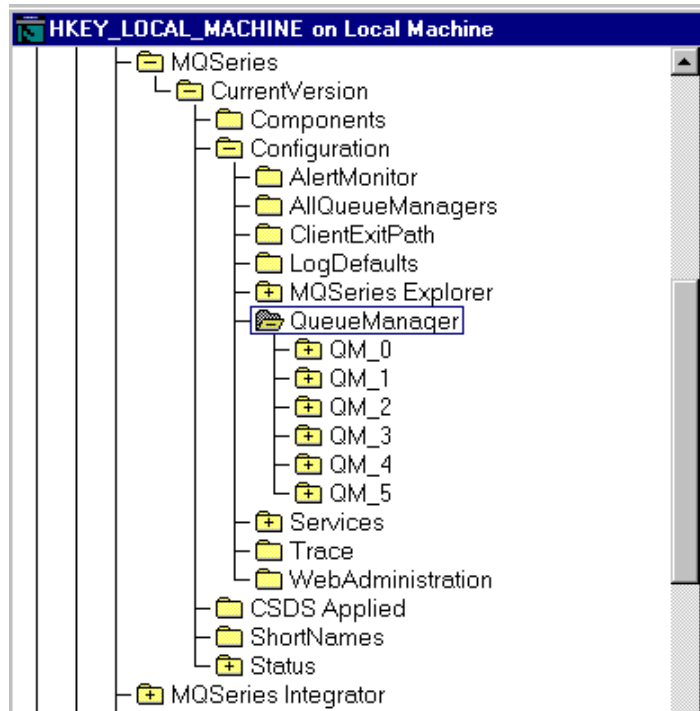


Figure 127. Registry - Queue Manager Objects

Now you can refresh the regedit display and verify that the new queue manager appears. An example is given in Figure 127 on page 138.

In the MQ Services window, the new queue manager is added to the bottom of the list. To display the queue managers in alphabetical order, close and then restart MQ Services.

## 7.2.5 Adding a Trigger Monitor

Now let us add a Trigger Monitor to watch the default trigger queue (SYSTEM.DEFAULT.INITIATION.QUEUE) on QM\_0. We will set up the service to start automatically, and also define the recovery options so that Windows NT will reboot if the Trigger Monitor fails.

To add a process to the queue manager, or change a process, right-click the item and select from the menu as shown in Figure 128.

- Here click **Trigger Monitor**. The Create Trigger Monitor Service dialog shown in Figure 129 on page 140 appears.
- Choose either **Automatic** or **Manual** startup.
- Next, click the **Parameters** tab shown in Figure 130 on page 140. This is where you specify the SYSTEM.DEFAULT.INITIATION.QUEUE.

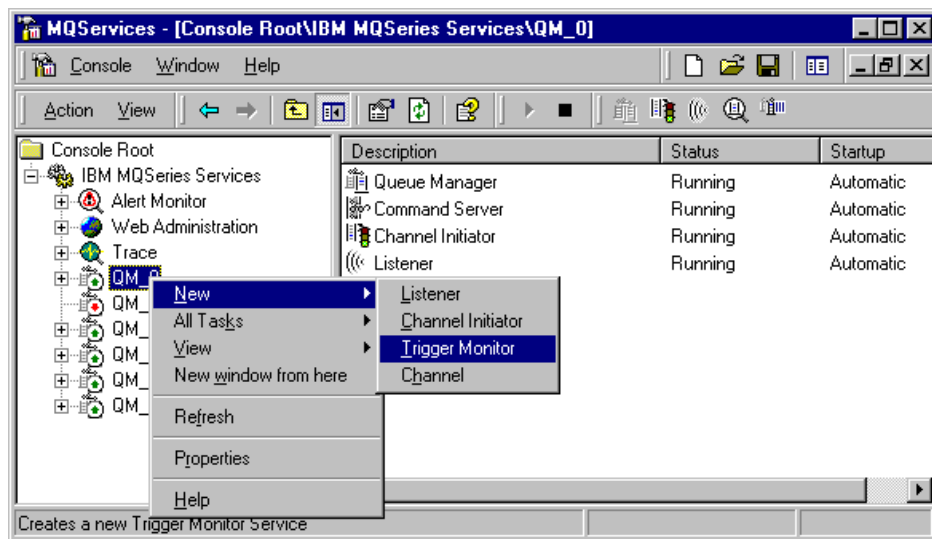


Figure 128. Adding a Trigger Monitor

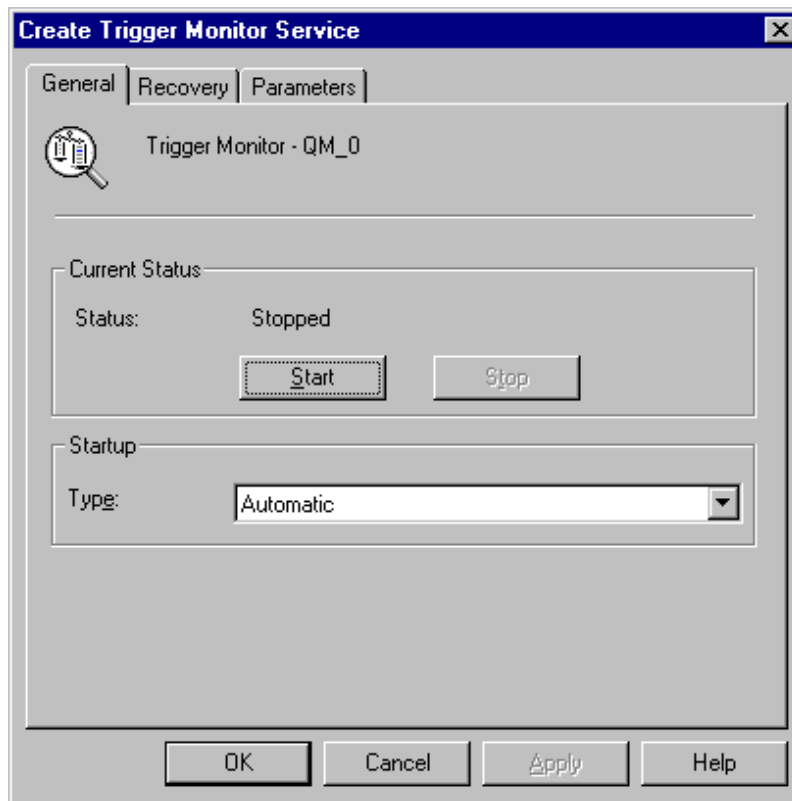


Figure 129. Create a Trigger Monitor Service - General Tab

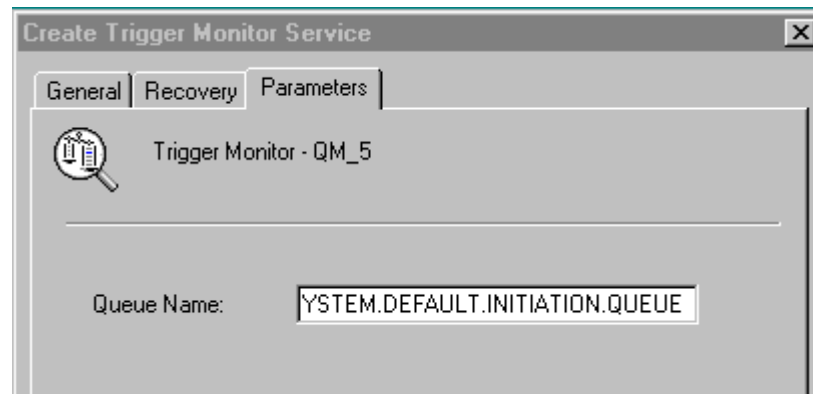


Figure 130. Create a Trigger Monitor Service - Parameters Tab

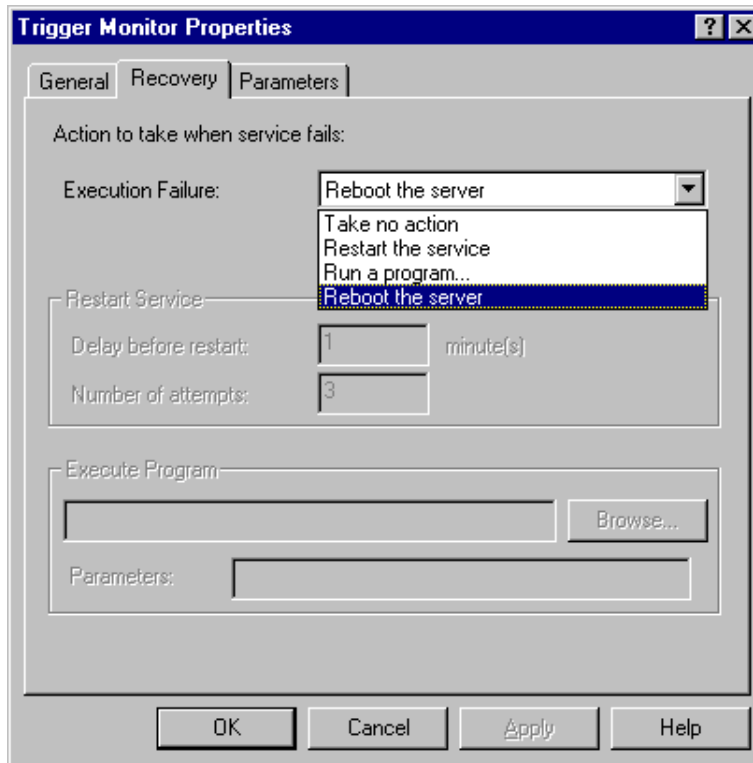


Figure 131. Create a Trigger Monitor Service - Recovery Tab

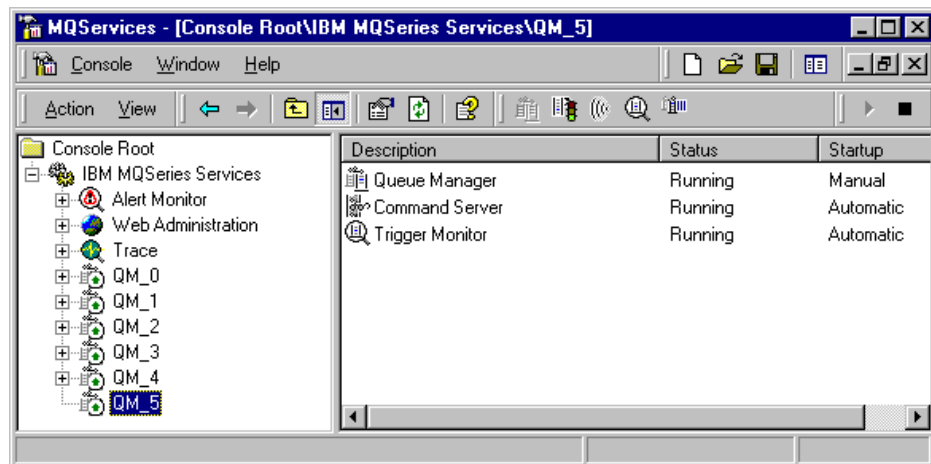


Figure 132. Queue Manager with Trigger Monitor

- To specify what to do in case of a failure click the **Recovery** tab. Here you have four options to choose from to tell the IBM MQSeries Service what action to take when a service had been successfully started, but has now stopped. The choices are:
  - a. **Restart the service** (in this case the Trigger Monitor)  
Here you can specify a delay (in minutes) if you need to allow time for the system to settle, for example, to allow for long timeouts in other applications.
  - b. **Reboot the server** (this computer)
  - c. **Run a program**  
Here you specify the program you want to run and any parameter it needs.
  - d. **Take no action**
- You can start the Trigger Monitor manually by clicking **Start** in the General tab.

### 7.2.6 Using the MQSeries Alert Monitor

You can configure the Alert Monitor to send notifications to any user. These alerts refer to the failure of a service for which you have configured an action. For example, if you want MQSeries for Windows NT Services to monitor a Trigger Monitor, tell someone if it fails, and restart it automatically, right-click the Trigger Monitor in the Services GUI and set the Recovery property accordingly. If you have enabled alerts, messages are sent to the appropriate user when a failure is detected and after the requested recovery has been carried out.

In this example, we send a notification to the user ID we used to log on when the Trigger Monitor fails. To accomplish that we have to do two things:

- Configure the Alert Monitor so that the alert message is sent to the right user ID.
- Configure the Trigger Monitor to perform an action after the service fails. In this case we will reboot the system. This is probably not what you would do in the real world, but it demonstrates that the alert function works and that MQSeries Version 5.1 is robust.

The Alert Monitor uses Windows NT messaging to notify a user when something drastically goes wrong with the MQSeries operational configuration. You can update its Property pages to send the messages to you.



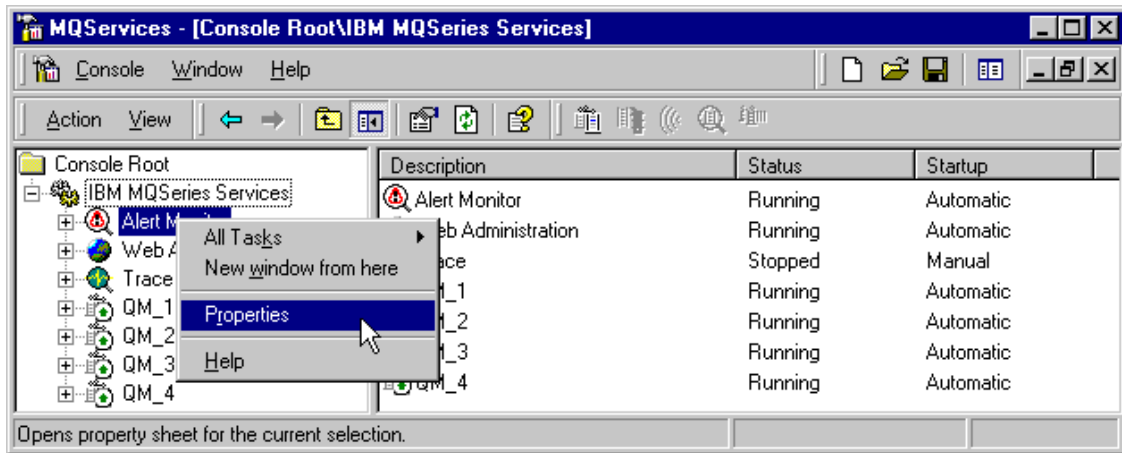


Figure 133. Configuring the Alert Monitor

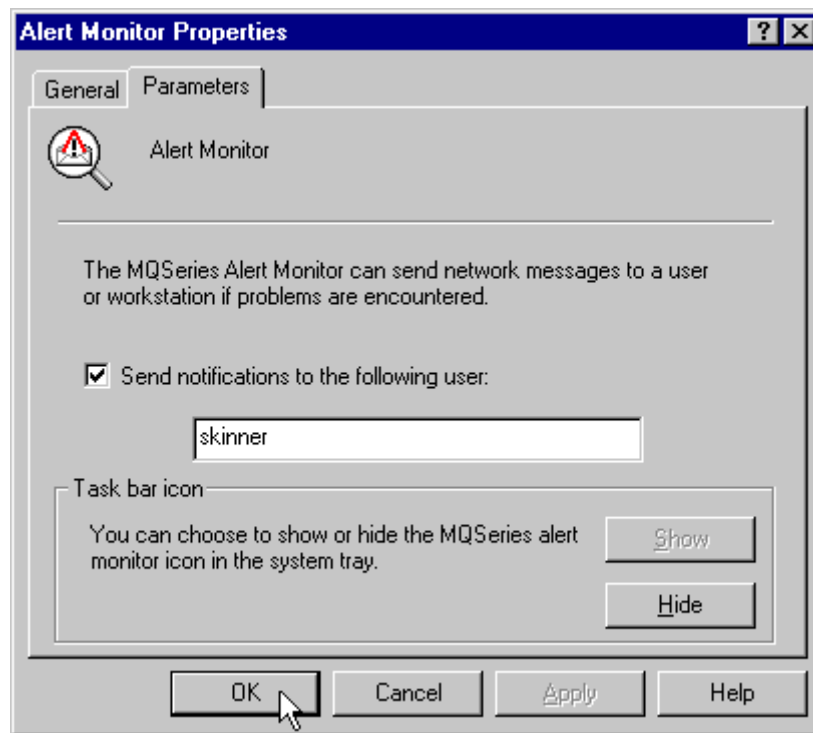


Figure 134. Alert Monitor Properties

1. Right-click the **Alert Monitor** icon and select **Properties** as shown in Figure 133 on page 143.
2. Click the **Parameters** tab to display the panel shown in Figure 134 on page 143.
3. Mark the check box **Send notification to the following user**.
4. Enter your user ID in the text box and click **OK**.

After having configured the Alert Monitor and the Trigger Monitor restart service, test if the function works. Use the Windows NT Task Manager to kill the new Trigger Monitor.

**Note:** To start the Windows NT Task manager right-click an empty portion of the task bar and select **Task Manager** from the menu.

In the task list, look for runmqtrm.exe and end the process. It may take a minute or two for your blatant act of sabotage to be noticed but *be patient* and resist the temptation to touch the mouse or the keyboard.

When the system restarts, log in and open the MQSeries Services window. Give the services time to start up again; it shouldn't take more than a few minutes.

*Now are you impressed?*

---

### 7.3 Using MQSeries Control Commands with the New GUIs

When you administer and operate a queue manager configuration using only the Explorer and Services GUIs, the displays show a consistent view of all the relevant MQSeries processes and their status. However, if you start MQSeries processes from the command line, the view presented by the Services “snap-in” does not correspond exactly with reality. Here are some examples:

1. The channel initiator, which is automatically started when you use strmqm to start a Version 5.1 queue manager, doesn't appear in the Services display.
2. When you start a queue manager using strmqm, an icon appears in the Services display for a “phantom” command server which does not yet exist. Starting the command server manually, either from the command line or through the Services GUI, updates the status of the icon correctly.

3. The listener for a queue manager started at the command line must be started and stopped at the command line, and doesn't appear in the Services display.

These inconsistencies come about because the control commands and the Services “snap-in” work with different data sources. The control commands have not been changed to take account of the Services “snap-in”; and the Services “snap-in” cannot reliably discover the existence, state, or relationships between all MQSeries processes when they are controlled from the command line, because the Control Commands do not update the Registry.

**Recommendation:** To avoid possible confusion, use either the Services “snap-in” or control commands to manage MQSeries processes, not both.

However, there is no problem mixing the use of runmqsc-based administration commands (this includes Web Administration, MQAI, and PCF) and the MQSeries Explorer, because they all use the same base data source.

**Recommendation:** Use the Explorer for displaying and making temporary changes to MQSeries configuration data on Windows NT, but use Web Administration with scripts for repeatability in production systems and for its cross-platform support.

---

## 7.4 Remote Administration

The MQSeries Explorer can remotely administer the following products:

AIX and UNIX	Command level 221 and above
MQWin 2.1	Not supported
OS/400	Command level 320 and above
OS/2 and Windows NT	Command level 201 and above
VMS and Tandem	Command level 221 and above

It can't do MQ/390 though.

An important point to be aware of is that both the MQ Explorer and Web Admin interfaces can only transmit commands that you would in the past have entered through runmqsc, or the platform equivalent. That is, you *can't* use them to create or delete queue managers.

Although the Services GUI allows you to control starting and stopping queue managers and their associated processes (channel initiators, listeners,

trigger monitors, and the like) that only works within the Windows NT context. That is, you can't manage a queue manager on an AS/400 from the Windows NT Services GUI.

In our opinion, it is reasonable to propose remote administration of existing queue managers, where administration means “looking after the object definitions”. As long as you can do a remote login, you can also administer security. And it seems reasonable to consider remote operation in an Windows NT-only queue manager network. You can do a limited amount of remote operations of queue managers on non-Windows NT platforms, such as starting and stopping channels, but full-blown operations in a heterogeneous multi-platform configuration require more than the MQ Explorer and MQ Services or Web Admin can deliver.

---

## Chapter 8. Web Administration

The MQSeries Explorer and Services user interfaces are a huge advance in ease of use over the DOS prompt used for MQSeries control commands, such as `runmqsc`. Attractive as they are, these new interfaces have two key limitations, namely:

- They are available only on Windows NT.

The new MQSeries Web Administration interface in Version 5.1 uses a standard Java-enabled browser that can run on any platform supported by your choice of browser.

- They operate only in interactive mode.

MQSeries Web Administration includes a scripting and script management facility that enables you to construct scripts, which may include complex logic<sup>1</sup>. You can store the scripts in a public or private library from which they can be invoked interactively, or called from within other MQSeries Web Administration scripts.

Web Administration uses `runmqsc` “under the covers”. Therefore, any operations that are valid in `runmqsc` will also work in Web Administration.

An important point to be aware of is that both the MQSeries Explorer and the MQSeries Web Administration interfaces can only transmit commands that you would in the past have entered through `runmqsc`, or the platform equivalent. That is, you can’t use them to create or delete queue managers.

Although the MQSeries Services GUI allows you to control starting and stopping queue managers and their associated processes, such as channel initiator, listener and trigger monitor, that only works in Windows NT. This means that you can’t manage a queue manager on an AS/400 from the MQSeries GUI. Here remote administration means administration of existing queue managers, that is, looking after object definitions.

The configuration in Figure 135 on page 148 shows clients attached to a Web Administration Server running on a Windows NT machine. This machine hosts queue managers of which one must be a default queue manager. Using the default queue manager, you can administer remote queue managers that run either in the same or different machines.

<sup>1</sup> You can use the new `-w` parameter option on the `endmqm` command to delay a script until a queue manager has stopped, but doesn’t wait until the registry has been updated. A `dlmqm` command following `endmqm -w` will fail if the registry update hasn’t had enough time to “catch up”.

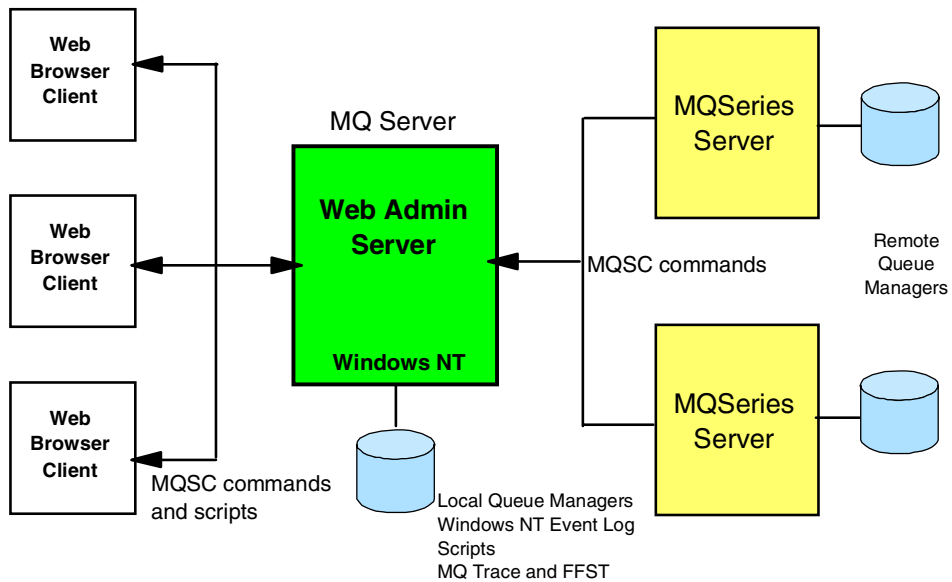


Figure 135. Web Administration Architecture

## 8.1 Enabling Web Administration

To enable MQSeries Web Administration, you need the following:

- An active user ID with authority to log in to the Web Administration Server, and on each machine that hosts queue managers that you want to administer. That is, your user ID should belong to the mqm security group, and be known globally.
- A Java-enabled Web browser, such as Netscape Navigator 4.04 with the Java AWT upgrade, or Microsoft Internet Explorer Version 4.01 (SP1) on your local machine.
- An MQSeries Web Administration Server installed on a Windows NT machine, either local to your browser or accessible to it across the network. This requires a custom installation of MQSeries.
- The Web Administration Server must be running on a dedicated IP port.
- Channel connections from the default queue manager on the machine where the Administration Server is running to queue managers on other machines that you want to administer remotely.
- A default queue manager on the Administration Server machine.

The following describes how to log in to the Web Administration, to find out which queue managers are available to you, and to how to execute `runmqsc` commands.

## 8.2 Logging in

Log on to Windows NT as a user who is a member of the `mqm` or Administrators security group. If you log on as a user that is not a member of one of those groups, you get this error message: The selected file is not a Microsoft Management Console document.

Then open an MQSeries Services window, and check the current status of the Web Administration server. Start it, if necessary. Figure 136 shows that the Web Administration server is running and the default port is uses, 8081.

Start your Web browser using the URL that points to the MQSeries Web Administration Server: `http://hostname:8081`, where `hostname` is the name of the NT server machine that hosts the MQSeries Web Administration Server, 8081 is the default port that the Web Administration Server listens on.

The Web Administration Server can be on your local machine or on another machine that can be addressed over a TCP/IP connection. If the server is remote, the following conditions must be met:

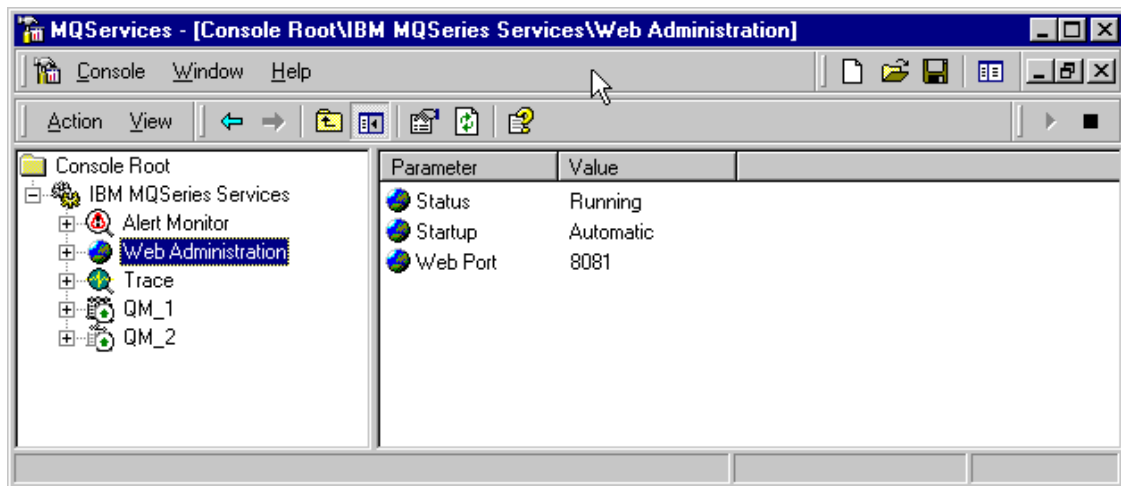


Figure 136. MQSeries Services - Web Administration

- The MQSeries Web Administration Server must be running there.

- hostname must be accessible. Use `ping hostname` at a command prompt to verify this.
- You need to be properly authorized on that machine, too.

The Web Administration Interface is implemented using Java. Be patient while the Java virtual machine is started and the code downloaded and initialized. Figure 137 shows the Web Administration in the Microsoft Internet Explorer.

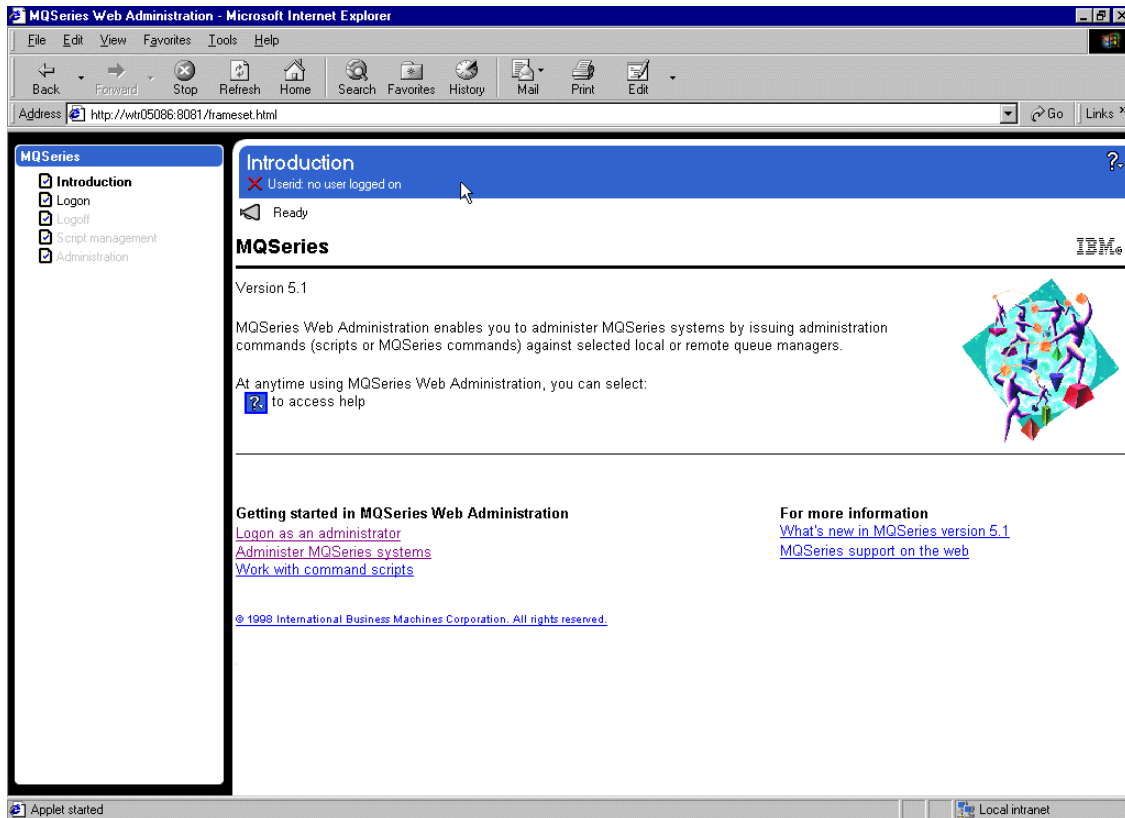


Figure 137. Web Administration Server

The Web Administration Interface comprises three distinct areas:

1. The left-hand pane lists the two available options, Introduction and Logon.
2. At the top of the right-hand pane you see the status of the Interface (Ready), and your current relationship with it (no user logged on).



3. The lower part of the right-hand pane contains the introduction. Selecting a hypertext link takes you into the comprehensive documentation built into the interface itself.

Click the **Logon** button in the left-hand pane.

When prompted in the lower right-hand pane, log in to Web Administration.

The user name you enter here must be in the mqm or Administrators group on the machine (or machines) that host the queue manager(s) that you want to administer, because you need the same level of authorization as you would need to use runmqsc. It need not be the same as the one you used to log on to Windows NT.

---

### 8.3 Getting Help

When you are working with the Web Interface and need help with the script language and management functions, open a second instance of the Web Administration browser and refer to the introduction. There is very little information about the practical aspects of scripting in the MQSeries Information Center.

Web Administration understands two types of commands:

1. MQSC commands

You find them in the online help under Commands when you click “?” in the top right-hand side in the window.

2. Script statements

From the script management help, select **How do I?** and then the link **script language**. The script statements are:

- CALL
- ECHO
- EXIT
- FOR-IN-ENDFOR
- IF-ELSE-ELSEIF-ENDIF
- LIST
- RUN
- SELECT
- SET

For detailed information, refer to the online documentation.

## 8.4 Using Commands

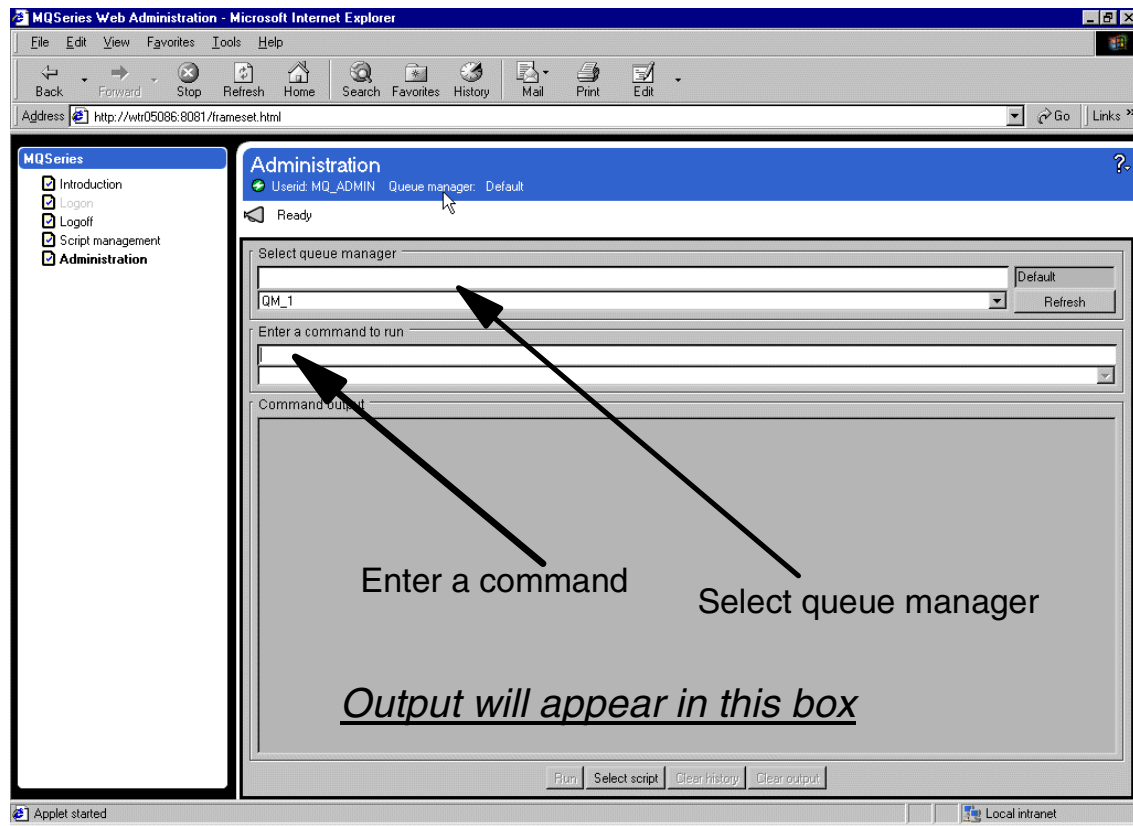


Figure 138. Web Administration

After logging on, you are presented with a window as shown in Figure 138. Notice the two data entry fields labelled “Select a queue manager” and “Enter a command”. Each is paired with a list box. If you don't populate the first text box with a queue manager name, the default queue manager will be selected automatically.

**Note:** If you have not defined one of the queue managers as the default, your commands will fail.

You can enter any valid queue manager name in the first text box; you should also be able to pick a local one from the list box. You may try to enter a simple MQSeries command in the second text box, for example, `dis qmgr`. Then press Enter or click the **Run** button.

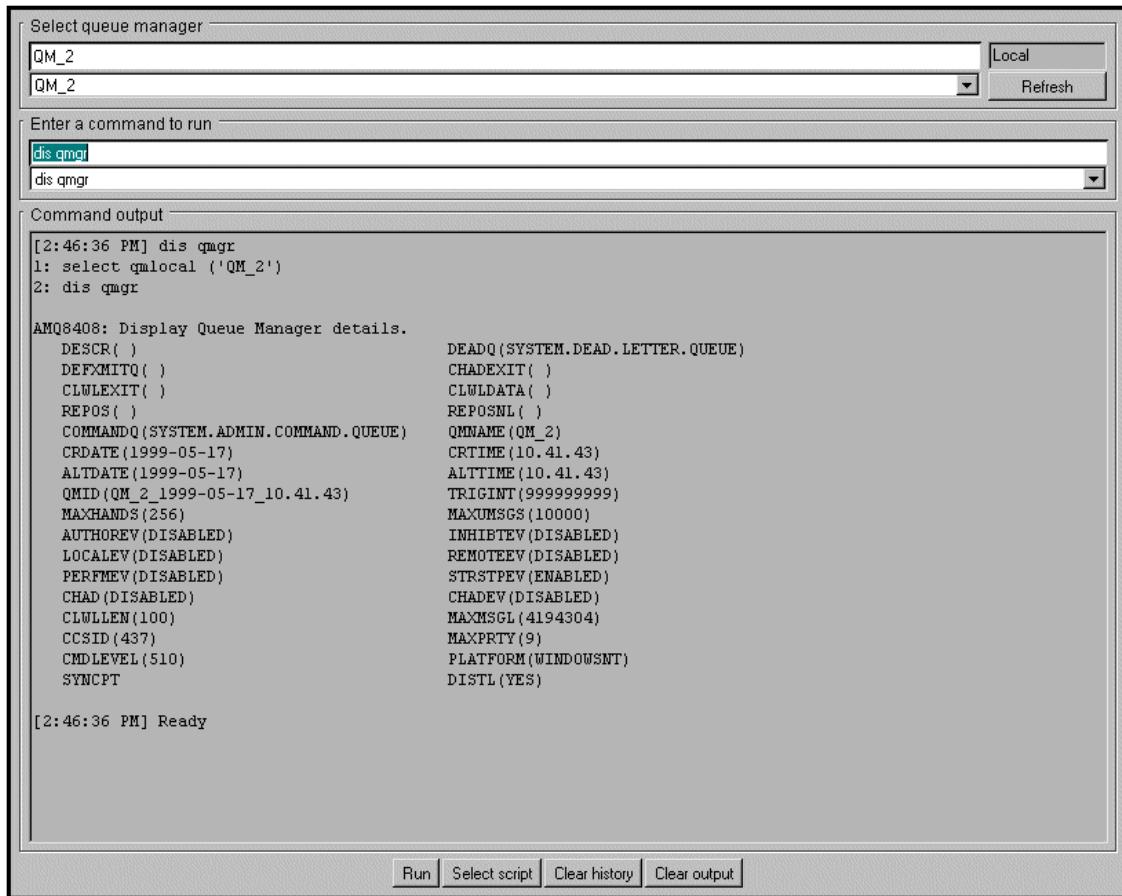


Figure 139. Web Administration - Using Commands

The results should be displayed in the command output area, as shown in Figure 139.

If you have access to a remote queue manager, enter its name in the Select queue manager text box and execute the command again. Your command was saved in the list box so you don't need to enter it again. Notice that, if the remote queue manager selection is valid, its name is added to the first list box so you can recall it later.

When you have finished with Web Administration, select the **Logoff** button in the left-hand navigation pane, and then press Enter or select the **Logoff** button in the lower right-hand pane.

## 8.5 Using Scripts

In this example, we will build a script that defines a second cluster queue, CLQ\_2, in cluster CL\_MQ51. We will call this script from an “outer script” that installs an instance of the new queue in each of the queue managers QM\_1, QM\_2, and QM\_3.

If not already done so, log on to Web Administration and select the **Script Management** option in the left-hand navigation pane.

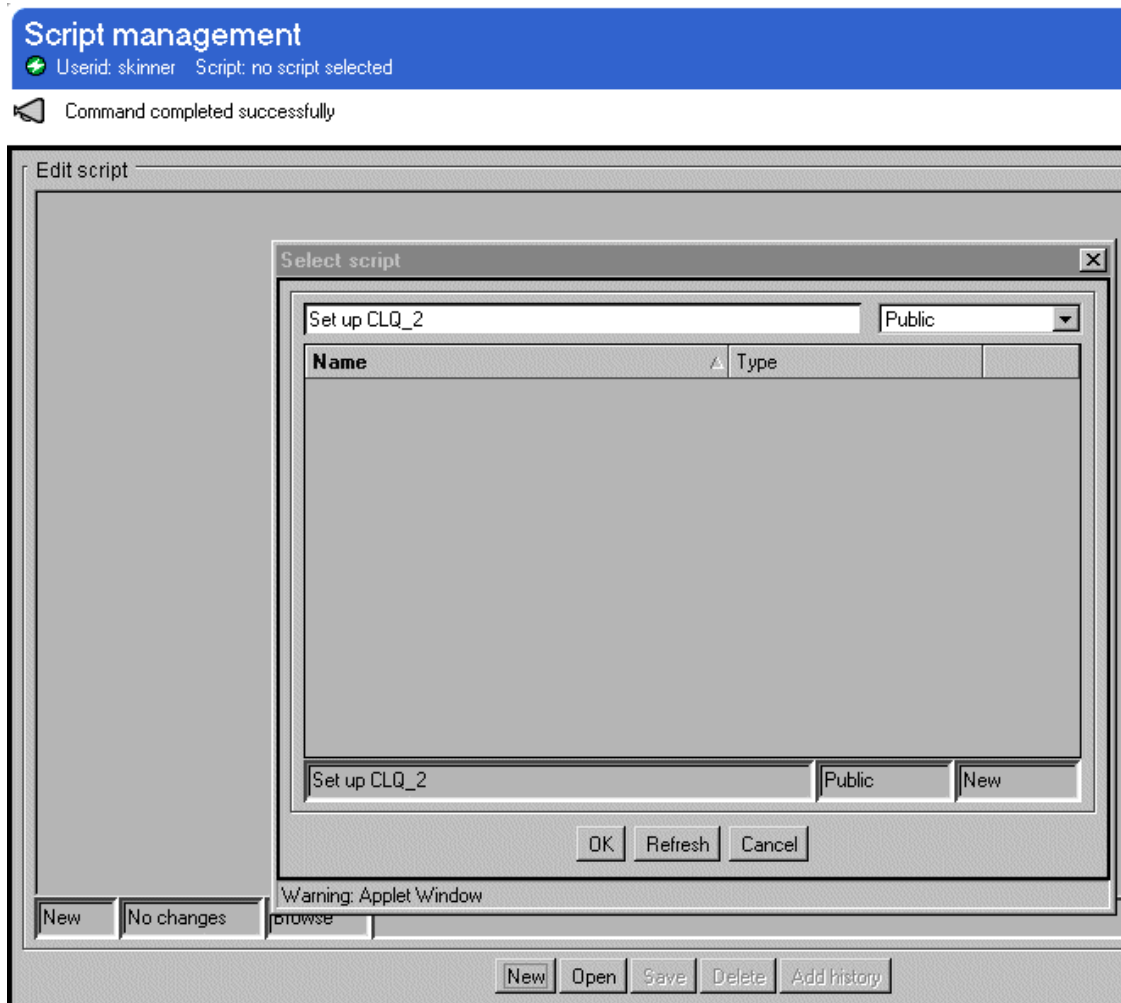


Figure 140. Web Administration - Script Management

When the Edit script pane opens, click **New** on the bottom of the pane, and name the inner script, for example, `Set up CLQ_2`. This example is shown in Figure 140 on page 154. Then click **OK** and the Select script window disappears.

In the Edit script window, enter the command to create the cluster queue `CLQ_2`:

```
define ql(CLQ_2) cluster(CL_MQ51) replace
```

Now save the script by clicking **Save** on the bottom of the pane. By default, it will be created as a Public script. If you wish, you may choose to save it as Private instead.

Now create and save the outer script “Install `CLQ_2` in `CL_MQ51`” that will call your first script once for each of the three queue managers `QM_1`, `QM_2`, and `QM_3`. You can include a display command at the end of your script to verify that your commands had the intended results.

If you saved your inner script as Private, remember this when coding the call statements in the outer script. The outer script is shown in Figure 141.

```
* set all the output options ON so you can see what's happening

list all

select qmlocal(QM_1)
call public 'Set up CLQ_2'

select qmlocal(QM_2)
call public 'Set up CLQ_2'

select qmremote(QM_3)
call public 'Set up CLQ_2'

* check that CLQ_2 ended up where you intended

dis qcluster(CLQ_2) cluster clusqmgr
```

Figure 141. Web Administration - Outer Script

## Administration

Userid: skinner Queue manager: Default

Command completed successfully

```
Select queue manager
QM_1

Enter a command to run
call public 'Install CLQ_2 in CL_MQ51'
call public 'Install CLQ_2 in CL_MQ51'

Command output
[6:34:55 PM] call public 'Install CLQ_2 in CL_MQ51'
1: select qmdefault
2: call public 'Install CLQ_2 in CL_MQ51'
Install CLQ_2 in CL_MQ51(PUBLIC) 6: select qmlocal(QM_1)
Install CLQ_2 in CL_MQ51(PUBLIC) 7: call public 'Set up CLQ_2'
AMQ8006: MQSeries queue created.
Install CLQ_2 in CL_MQ51(PUBLIC) 9: select qmlocal(QM_2)
Install CLQ_2 in CL_MQ51(PUBLIC) 10: call public 'Set up CLQ_2'
AMQ8006: MQSeries queue created.
Install CLQ_2 in CL_MQ51(PUBLIC) 12: select qmremote(QM_3)
Install CLQ_2 in CL_MQ51(PUBLIC) 13: call public 'Set up CLQ_2'
AMQ8006: MQSeries queue created.
Install CLQ_2 in CL_MQ51(PUBLIC) 16: * check that CLQ_2 ended up where you intended
Install CLQ_2 in CL_MQ51(PUBLIC) 18: dis qcluster(CLQ_2) cluster clusqmgr

AMQ8409: Display Queue details.
CLUSTER(CL_MQ51)                QUEUE(CLQ_2)
CLUSQMGR(QM_3)

AMQ8409: Display Queue details.
CLUSTER(CL_MQ51)                QUEUE(CLQ_2)
CLUSQMGR(QM_1)

AMQ8409: Display Queue details.
CLUSTER(CL_MQ51)                QUEUE(CLQ_2)
CLUSQMGR(QM_2)

[6:34:57 PM] Ready
```

Now click **Administration** in the left-hand pane and run your outer script. The command is:

```
call public 'Install CLQ_2 in CL_MQ51'
```

## CL\_MQ51

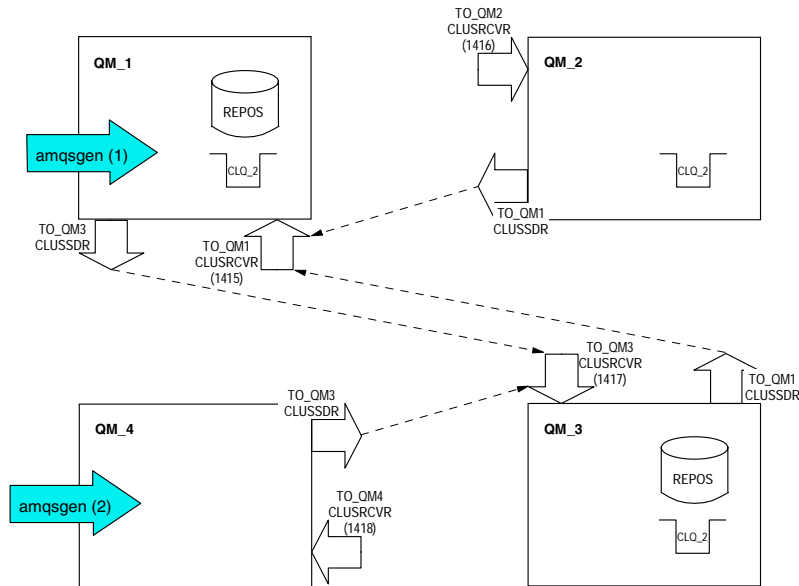


Figure 142. Web Administration - Verify Cluster Traffic

Figure 142 shows the configuration developed and used in the previous chapters. To verify that this cluster works properly you can run two tests.

1. Use a program such as amqspout or amqsgen (see page 161) to pump messages to CLQ\_2 from QM\_1 and use Web Administration to check where the messages actually go.
2. Stop the program and restart it at QM\_4. Then track the message traffic again.

*Why are the traffic patterns different?*

QM\_1 owns an instance of CLQ\_2 and therefore all messages are put into this local queue. QM\_4 on the other hand does not have an instance of CLQ\_2. Therefore the messages are distributed to the three instances on the other three queue managers.





## Chapter 9. Using the Performance Monitor

The Performance Monitor is a standard component of Windows NT. It enables you to select and display a variety of data about the performance of the Windows environment, as tabular reports or graphs. You can use it to monitor the depth of messages on MQSeries queues, and the rates of message arrival and removal.

You access the Performance Monitor from the Start menu: **Programs, Administrative Tools (Common), Performance Monitor.**

When you first start it, the display is empty. To add a resource that you want to monitor, select **Edit** and then **Add to Chart**.

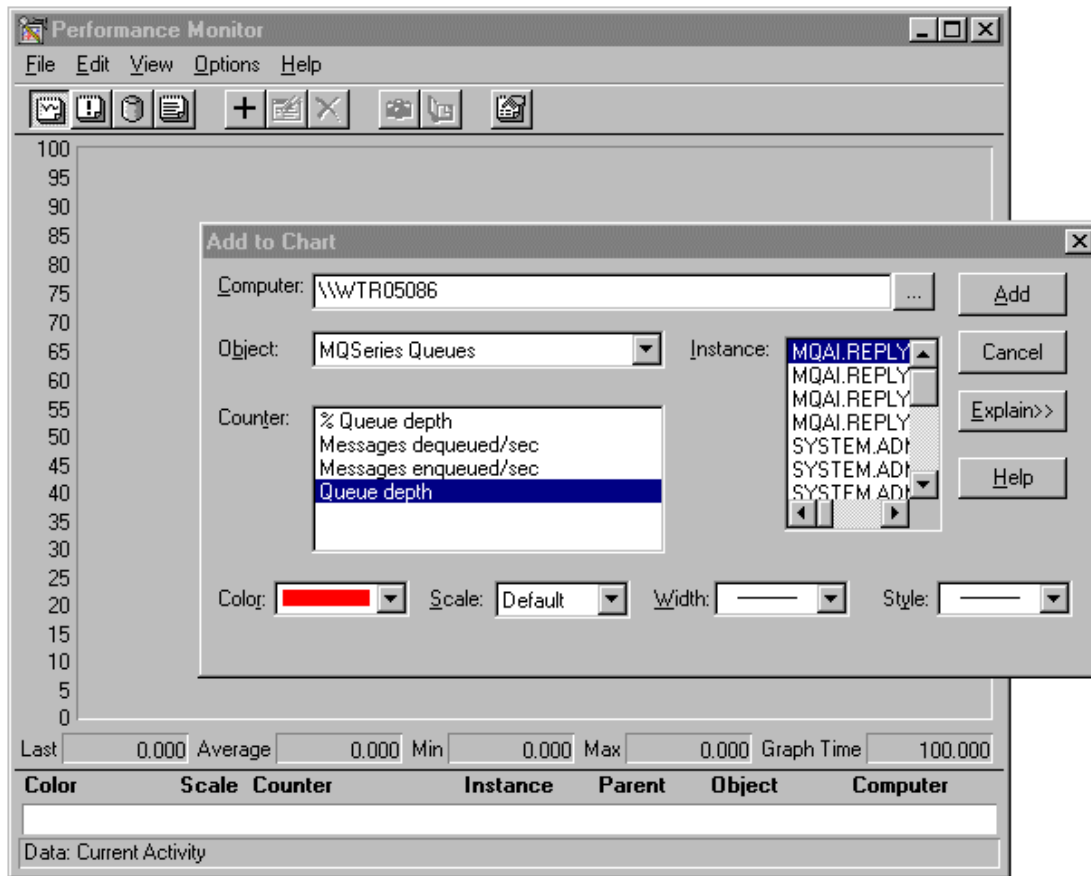


Figure 143. Performance Monitor - Setup

In the Add to Chart window shown in Figure 143 on page 159, select **MQSeries Queues** from the list of objects.

Next, choose what you want to monitor:

1. The current queue depth, that is, how many messages are in the queue.
2. The queue depths as a percentage of the maximum queue depth, that is, how full the queue is.
3. The enqueue rate in messages per second, that is, the number of messages placed in the queue. This is not necessarily the number of MQPUTs; each message segment counts as one message.
4. The dequeue rate in messages per second, that is, the number of messages removed from the queue.

Then select a queue from the instance list. The instance list contains only queues that have had messages inserted or removed before the Performance Monitor started.

**Note:** All counters are installed in the Windows NT registry.

As an example, we will use the Windows NT Performance Monitor to track the rate of message flow to multiple instances of a queue across a cluster and observe how the traffic flow changes when a queue manager (or queue instance) drops out of the cluster and rejoins it.

---

## 9.1 Example 1: Track Cluster Queues

The objective is to use the Windows NT Performance Monitor to track the activity on the instances of the cluster queue, CLQ\_ACROSS\_2\_3\_4, on queue managers, QM\_2, QM\_3, and QM\_4.

To do this we use the configuration built in Chapter 4, “Creating a Cluster with the MQExplorer” on page 55. Before we start, use the MQSeries Explorer to check that the configuration is up and running, and that the cluster queue instances are accessible from QM\_1. The Explorer window should contain the information you see in Figure 144 on page 161.

**Note:** If you don’t see the queue instances in the Performance Monitor, use amqsput or any other program to put a message into them.

Use the modified sample application amqsgen to generate message traffic on QM\_1 destined for the cluster queue CLQ\_1. This GUI is shown in Figure 145 on page 161.

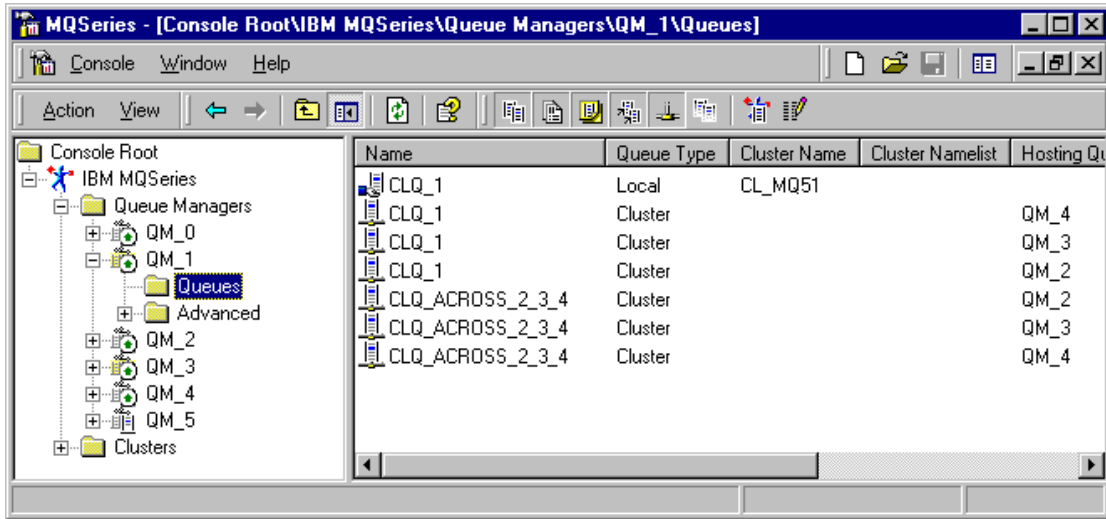


Figure 144. Accessible Cluster Queues

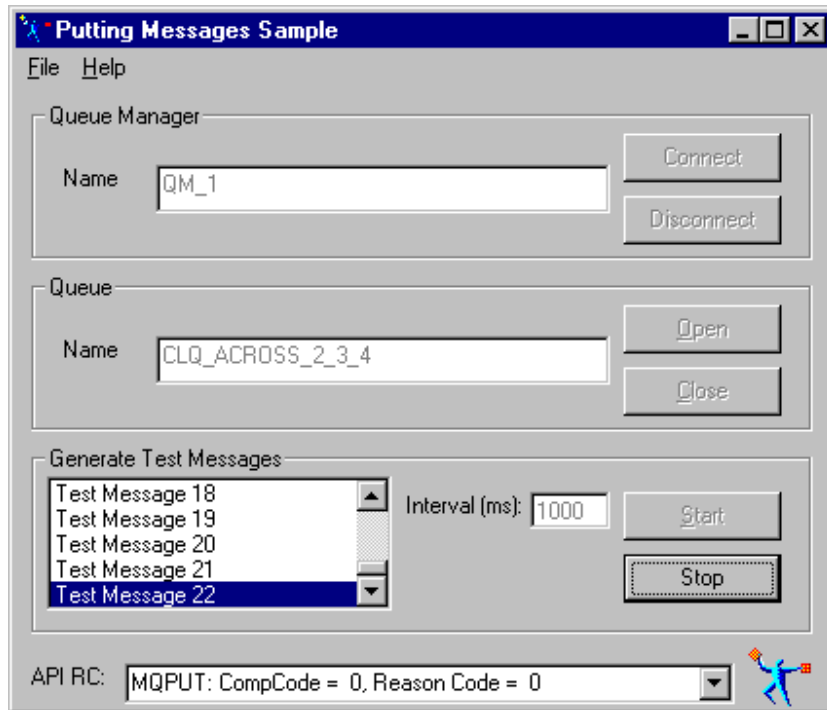


Figure 145. Put Message Sample - amqsgen

The program amqsgen is a modified sample program written in Visual Basic and is supplied with this book. You use it this way:

1. Type the queue manager name (here QM\_1) and click **Connect**.
2. Type the queue name (here CLQ\_ACROSS\_2\_3\_4) and click **Open**.
3. You may modify the time interval to put messages in the queue. The default is set to one message per second.
4. Click **Start** to begin putting messages. The program generates messages as shown in Figure 145 on page 161.
5. Click **Stop** to end putting messages.
6. To end the program click **Close** and then **Disconnect**.

**Note:** If you use the configuration you created in the earlier chapters, make sure that you use CLQ\_ACROSS\_2\_3\_4 and not CLQ\_1. Since QM\_1 also owns an instance of this queue, all messages would be put into that one.

If you use the defaults of the Performance Monitor and amqsgen, you will see a flat line on the bottom in the Performance Monitor window, as shown in Figure 145. You distribute messages at a rate of one per second over the three queues and display them in the chart using the default of .1 messages. You can use the MQSeries Explorer to verify that messages are really put in all three queues. Expand each queue manager, click **Queues**, right-click the local instance of **CLQ\_ACROSS\_2\_3\_4** and then select **Browse Messages** from the menu.

To get a better graph in the Performance Monitor, adjust the scaling factor of each of the graphs. Select an instance at the bottom of the graph as you see in Figure 146. Then click **Edit**, choose **Edit Chart Line** and set Scale to 1.0 as shown in Figure 147 on page 163. You can see that the curves now give a better view.

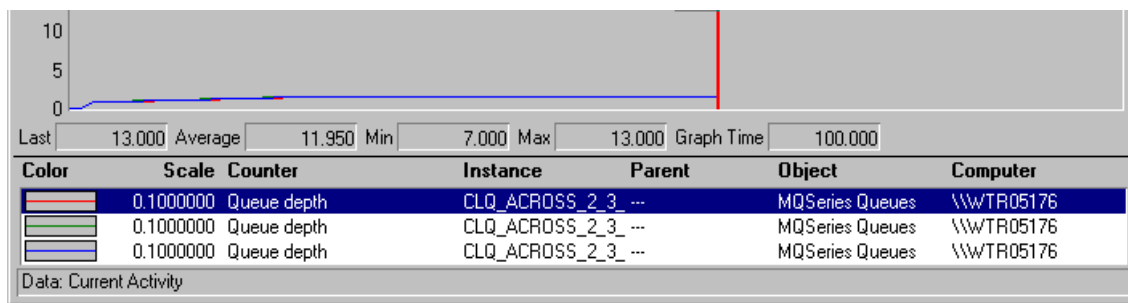


Figure 146. Performance Monitor - Counters

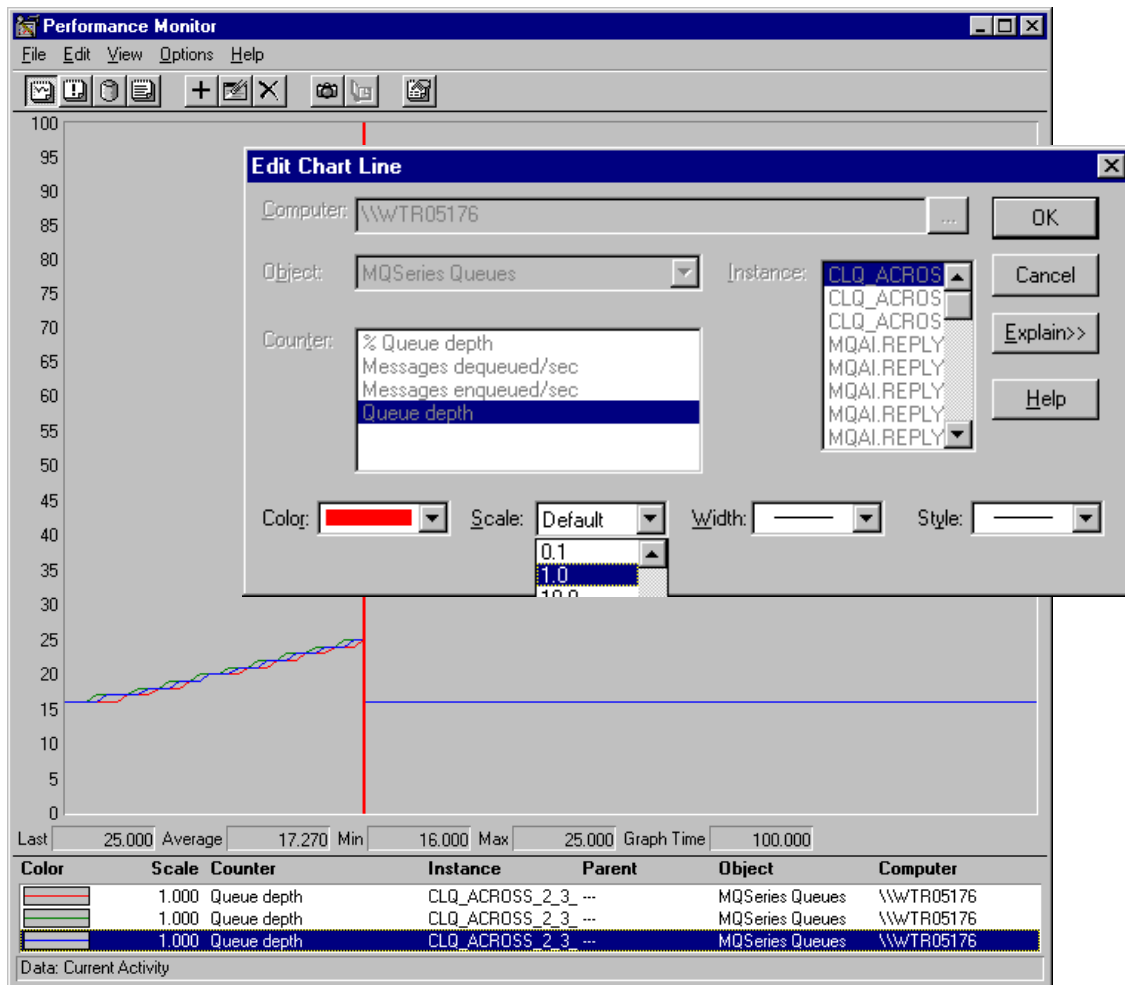


Figure 147. Performance Monitor - Scale Factor

Once you have all three cluster queue instance added and being displayed, drain the queues and reset the Performance Monitor display using the **Clear Display** command from the **Edit** menu. You can use the amqsget sample program to empty each of the queues. The commands are:

```
amqsget CLQ_ACROSS_2_3_4 QM_2
amqsget CLQ_ACROSS_2_3_4 QM_3
amqsget CLQ_ACROSS_2_3_4 QM_4
```

You may also change the vertical optimum, for example, to 50. Click **Options** and then **Chart**. Figure 148 on page 164 shows the output.

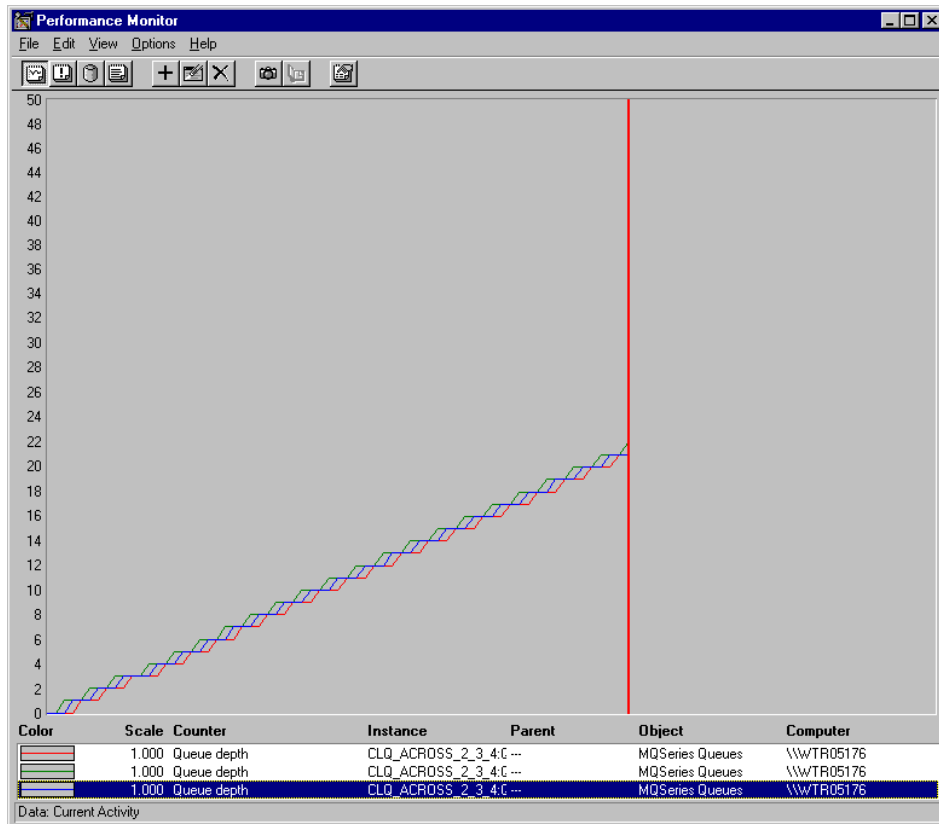


Figure 148. Performance Monitor - Chart Showing Current Queue Depth

## 9.2 Example 2: Check Cluster Behavior

Now we use the Performance Monitor to observe what happens when a cluster queue instance becomes inaccessible and when it rejoins the cluster.

We use amqsgen to distribute messages to CLQ\_ACROSS\_2\_3\_4. The Performance Monitor shows curves as shown in Figure 148. Then we interrupt the flow of messages to one of the cluster queue instances. There are various methods you might try to simulate different kinds of situations, for example:

- Set the queue to Put Disabled.
- Stop the receiver channel.
- Suspend the queue manager.
- Terminate the queue manager that owns the queue.

The chart below shows when one queue manager was suspended and when it resumed its work.

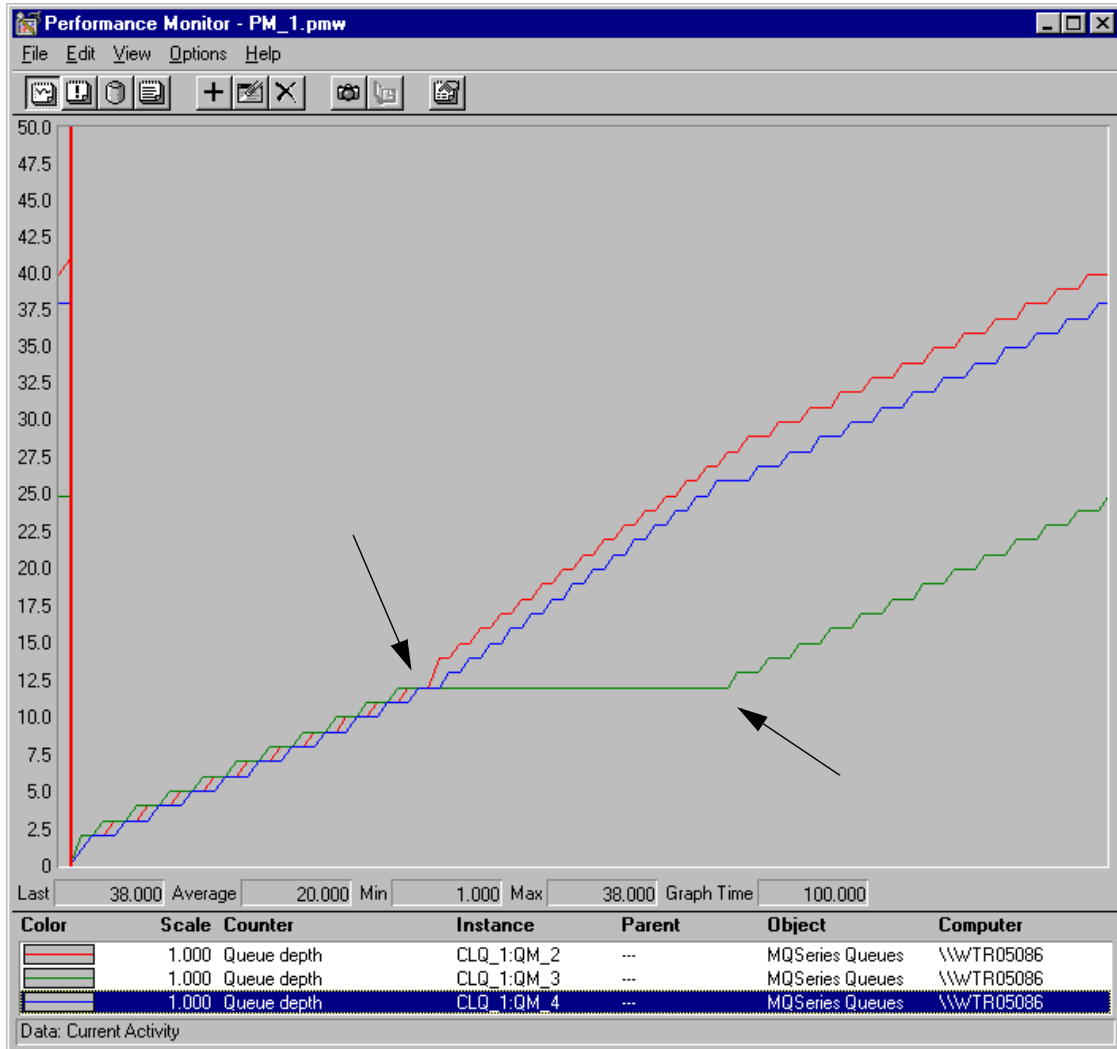


Figure 149. Performance Monitor - Interrupted Message Flow





---

## Chapter 10. File Transfer Programs

Although there are usually better ways of solving application problems than by using MQSeries as a file transfer mechanism, the realities of deadlines and schedules often mean that file transfer-based solutions must be put in place in the short to medium term.

The programs described here (or ones very like them) were created in response to such requirements.

Originally the programs were based on MQSeries Version 5.0 and were therefore able to make use of Version 5 features such as message segmentation and message groups. As it often happens, however, it was discovered that some of the MQSeries servers would in fact be Version 2 type servers.

A number of solutions were considered as a result of the Version 2 compatibility requirement. In the end, the mechanism shown here was used. It uses only Version 2 features. What this means is that we have to implement our own (application-specific) way of "grouping" the messages that make up a single file, of "ordering" them properly during re-assembly at the receiving end, and of indicating the END-OF-FILE condition.

This chapter will guide you through the simple design of the solution that was used.

The text that follows assumes that you can read C code and that you have at least browsed some of the C based sample code that comes with MQSeries.

The files we will explore are as follows:

- mqfm\_defs.h A header file that contains some handy values that are shared among the other files.
- putMsg.c A very simple program that allows us to create MQSeries messages whose contents is the text of the parameters on the command line of putMsg.
- putFile.c The program that chops up files into messages and sends them.
- getFile.c The program that reassembles files from messages sent by putFile.

We will walk through the code simply by annotating the C code itself with footnotes. The source code for the programs themselves are available on the diskette at the back of the book.

---

## 10.1 Design

The key programs are `putFile` and `getFile`. Each of these programs can be run either from the command line, or as MQSeries triggered programs. You will see comments (footnotes) below the code pointing out the differences.

### 10.1.1 `putFile`

This program chops the source file into chunks of a standard size<sup>1</sup>. It turns the chunks into a series of MQ message with the following characteristics:

Table 8. MQSeries Message Header Values for File Transfer Messages

Message	MsgId	CorrelId
<i>First (header)</i>	Allocated by MQSeries	'000000000000000000000000'
<i>nth (2nd to last - 1)</i>	Same as header	n left padded with zeroes
<i>Last</i>	Same as header	'999999999999999999999999'

As you can see, we allow MQSeries to allocate a unique MsgID on the header message only. We then reuse that MsgID for all messages in the group for transfer of a particular file. Meanwhile, we use the CorrelID as a "counter" for the file chunks. The CorrelID zero message (000000000000000000000000) is the header. It contains no file data.

The header message does contain the *filename* of the file being transferred and optionally, a *destination directory* (which, if present, will override the default destination directory at the receiving end).

The header message communicates three pieces of data.

- The unique MsgID for the new file transfer (which is in the MQ header).
- The filename and destination directory (in the data portion of the header message), as follows:

FILENAME=<filename> DESTDIR=<destination directory>

<sup>1</sup> MQFM\_MESSAGE\_SIZE in `mqfm_defs.h`. Default value is 5000 bytes.

### 10.1.2 getFile

This program uses the following logic to reassemble the messages (as above) into files:

1. Do an MQGET with CorrelID "000000000000000000000000". This will certainly find the next header message, with a new MsgID <OurMsgid> for us and no data.
2. Do an MQGET with MsgID <OurMsgid>. We are thus getting the *next sequential message* with MsgID of <OurMsgid>. This assumes FIFO queue processing. We then check that the CorrelID is the correct CorrelID for the next chunk of the file we are expecting.<sup>2</sup>
3. Repeat step 2 until CorrelID = "999999999999999999999999". This indicates that the current message is the final piece of the file.
4. Write the message out to DestinationDirectory / filename where:
  - *filename* arrived in the data portion of the file's header message.
  - *DestinationDirectory* defaults to the value of "UserData" in the PROCESS definition of the triggered process on the receiver, unless it has been overridden by a DestDir value arriving in the data portion of the file's header message.

---

## 10.2 Input Parameters

Obviously both putFile and getFile need to know the names of objects, such as files and queues in order to do their job. As indicated earlier, the programs can be initiated from the command line or started as MQSeries triggered programs.

As you read the following code, Table 9 on page 170 may help you to keep track of where each of the programs finds these parameters. The headings indicate how putFile and getFile are started, either from the command line or triggered. The row titles on the left show where to specify the various parameters the programs need.

<sup>2</sup> Note that better logic would be to SPECIFY both a MsgID of <OurMsgid> and the CorrelID we require next. This would get MQ to fully manage the reordering of files at the receiver. The only problem with such a change would be that whereas currently we do not need to know which is the LAST chunk, in such a revised algorithm we would. Currently we know the last chunk has arrived when we see a CorrelID of "999999999999999999999999". The revised program would have to request the last chunk by CorrelID. Perhaps you would like to revise the program. Could you put the number of chunks (messages) for the complete file in the header message?

	putFile		getFile	
	Command Line	Triggered	Command Line	Triggered
<i>Queue Manager</i>	Default	Default	Default	Default
<i>Instruction Message Queue<sup>a</sup></i>		Trigger Message		
<i>Transferred File's Queue<sup>b</sup></i>	Command Line	Instruction Message	Command Line	Triggered Message
<i>Directory of File<sup>c</sup></i>	Possible part of FILE <sup>d</sup>	Instruction Message	Header Message <b>or</b> part of FILE (see footnote d)	Trigger Message <sup>e</sup> <b>unless overridden by</b> Header Message
<i>Name of File<sup>f</sup></i>	Command Line	Instruction Message	Header Message <b>unless overridden by</b> Command line (optional)	Header Message only
<i>Destination Directory<sup>g</sup></i>	Command Line (optional) <sup>h</sup>	Instruction Message (optional) <sup>i</sup>		

- a. The queue from which an Instruction Message can be read.
- b. The queue to which the file chunks are written (putFile) or from which the file chunks are read (getFile).
- c. That is, the <directory> portion of the fully qualified pathname of the file being transferred.
- d. The code would need to be changed to allow directory names as part of filenames, but this could easily be done. That is to say, we would qualify <filename>, with a <directory> component. So <filename> might be "c:\directory\file". The code just before footnote 31 on page 185 is where we currently prevent this option.
- e. This is transferred from the UserData attribute of the PROCESS statement of the triggered process.
- f. The <filename> portion of the fully qualified pathname of the file being transferred.
- g. The <directory> portion of the fully qualified pathname of the file being transferred **AS IT WILL BE AT THE RECEIVING SIDE**. Obviously, we cannot assume the same directory structure at both ends. If it is specified, it travels in the Header Message.
- h. Placed in the Header Message IF it is specified.
- i. Same as footnote h.

Table 9. Source of Input Parameters for getFile and putFile

---

## 10.3 Message Types

The File Transfer application uses quite a few message types. This section is a summary of the terms we have used and definitions of the message types. We list them in roughly the logical order in which they would be created during the file transfer process.

Note that Header Message, Data Message, Trailer Message, and Instruction Message are defined by the File Transfer application. Trigger Message on the other hand is a standard MQSeries object.

### 10.3.1 Header Message

This is created by putFile and contains an MQ-generated MsgID that is to be the message ID for all the chunks of this particular file. The CorrelID is "000000000000000000000000".

A Header Message is of the form:

```
FILENAME=<file> DESTDIR=<destdir>
```

**Note:** The "<>" characters, are of course not included. Also, the DESTDIR is optional. That is to say the message is valid if it does not contain a DESTDIR.

### 10.3.2 Data Message

This is created by putFile and carries the data of the file transfer.

It has MsgID equal to the MsgID in the Header Message. The Nth Data Message for an individual file transfer has CorrelID of <N left padded to 24 char with zeroes>. It carries file data of MQFM\_MESSAGE\_SIZE (which, by default, in mqfm\_defs.h, is 5000 bytes).

### 10.3.3 Trailer Message

This is created by putFile as the last of the data messages. It has MsgID equal to the MsgID in the Header Message. It has CorrelID of "999999999999999999999999". It carries file data of less than or equal to MQFM\_MESSAGE\_SIZE (which, by default, in mqfm\_defs.h, is 5000 bytes).

### 10.3.4 Instruction Message

This can be created by any means. You could use putMsg, which just makes a message from its input command line parameters. The Instruction message should land on a triggered queue. The triggered queue should be set up to trigger a process that would (for our purposes) be getFile. When the Instruction Message lands on the triggered queue, the local queue manager

automatically creates a Trigger Message whose contents is made available to the triggered program (getFile, in our case). If this is all a mystery to you, you need to read Section 14.1 "What is Triggering?" in *MQSeries Application Programming Guide*, SC33-0807.

An Instruction Message is of the form:

```
FILENAME=<file> QUEUENAME=<queue> DIRECTORY=<dir> DESTDIR=<destdir>
```

**Note:** The "<>" characters, are of course not included. Also, the DESTDIR is optional. That is to say the message is valid if it does not contain a DESTDIR.

### 10.3.5 Trigger Message

This message is created automatically when a message arrives on a triggered queue. In the File Transfer application we use the Trigger Message to carry the UserData field of the PROCESS definition to the triggered application (getFile). We use UserData to carry the default "dropoffDirectory" on the receiving side. Refer to footnote e in Table 9 on page 170.

## 10.4 mqfm\_defs.h<sup>3</sup>

```
#define MQFM_MESSAGE_SIZE 50004
#define MQFM_MAX_FILENAME 200
#define MQFM_FILENAME_EYECATCHER "FILENAME="5
#define MQFM_DIR_EYECATCHER "DIRECTORY="
#define MQFM_DESTDIR_EYECATCHER "DESTDIR="
#define MQFM_QUEUE_EYECATCHER "QUEUE="
#define MQFM_TRIGM_USERD_EYECATCHER "::<-DIR"6
#define MQFM_INSTRUCTION_MESSAGE_SIZE 2007
#define MQFM_MAX_DIRECTORY 200
#define MQFM_FIRSTMSG_FLAG "000000000000000000000000"8
#define MQFM_LASTMSG_FLAG "999999999999999999999999"

#if !defined(TRUE)
    #define TRUE 1
#endif
#if !defined(FALSE)
    #define FALSE 0
#endif
```

<sup>3</sup> This file contains definitions shared among the programs.

<sup>4</sup> This is the chunk size we will use as we turn files into messages, here 5000 byte messages.

<sup>5</sup> These programs make use of control or "instruction" messages in order to know about queues, directories, filenames etc. These are the definitions of the eyecatchers we use to sift out the various "instructions".

<sup>6</sup> This eyecatcher matches a definition on the receiving server which looks like this:

```
* define the process for triggered data queue
define process (HP.GETFILE.PROCESS) +
descr('process for getting files from queue') +
apptype (UNIX) +
applicid('/dda/getFile3') +
userdata('/ddaHPtest::<-DIR') +
replace
```

The purpose of the "::<-DIR" is to make absolutely sure that we would not pick up any trailing garbage characters in the userdata field.

This userdata is what ends up in the trigger message as the dropoffDirectory. See Table 9 on page 170 and footnote 54 on page 194.

<sup>7</sup> We won't create instruction messages (see footnote 5 above) larger than this.

<sup>8</sup> Because we have restricted ourselves to MQSeries V2 function we need a "home-grown" method of ordering the messages which represent any one file. We use these two values in the CorrelId of our file transfer messages in order to ensure we can identify first and last messages of our logical "message group".

## 10.5 putMsg.c<sup>9</sup>

```
/*
*****
* NAME:          putMsg.c - write msg data to specified MQ queue
*
* SYNOPSIS:
* #include <mqfm_defs.h>
*
* DESCRIPTION:  putMsg <queueName> <msgData>
*
* NB: This application assumes it will use the default queue manager.
*     Therefore, the queue manager on the machine running this application
*     must have been defined with the -q option, i.e.
*         crtmqm -q <queue manager name>
*
*****
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <cmqc.h>
#include <mqfm_defs.h>

int main(int argc, char **argv)
{
    char msgData[sizeof(MQFM_FILENAME_EYECATCHER) + MQFM_MAX_FILENAME +
                    sizeof(MQFM_QUEUE_EYECATCHER) + 48 +
                    sizeof(MQFM_DIR_EYECATCHER) + MQFM_MAX_DIRECTORY], 10
            *queueName;          /* target queue name */
    /* various MQI structures needed */ 11
    MQOD od = {MQOD_DEFAULT};    /* Object Descriptor */
    MQMD md = {MQMD_DEFAULT};    /* Message Descriptor */
    MQPMO pmo = {MQPMO_DEFAULT}; /* put message options */

    MQHCONN hCon;                /* connection handle          */
    MQLONG  compCode;             /* completion code            */
    MQLONG  reason;              /* reason code                 */
}
```

<sup>9</sup> We use this program to create the "Instruction Messages" (see footnote 5). This program simply creates a small MQ message from its command-line parameters (see specifically, the msgData in the comments at the top of the program).

<sup>10</sup> We allocate the msgData array to be large enough to contain all the fields we might want to put in there. (Tidy programmers might like to change the literal "48" to something like MQFM\_MAX\_QUEUE and retrieve its value from the `mqfm_defs.h` file.)

<sup>11</sup> A lot of this is just standard MQ stuff, so we won't comment on it.



```

MQHOBJ  Hobj;                /* object handle          */
MQLONG  openOptions;        /* MQOPEN options        */
int  i;

printf("%s program running\n", argv[0]);
if (argc < 3)
{
    printf("Required parameter(s) missing\n");
    printf("Usage: %s <queueName> <msgData>\n", argv[0]);
    exit(1);
}

/* extract input parameters for convenience */
queueName = argv[1];

/* build message data by concatenating all subsequent parms */12
msgData[0] = '\0';
for (i=2; i<argc; i++)
{
    strcat(msgData, argv[i]);
    strcat(msgData, " ");
}
printf("msgData value is '%s'\n", msgData);

/*****/ 13
/* Connect to queue manager */
/*****/
MQCONN "",                /* default queue manager */
    &hCon,                /* connection handle */
    &compCode,            /* completion code */
    &reason);            /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCONN failed with reason code %ld\n", reason);
    exit(1);
}

/*****/
/* Open the target message queue */
/*****/
strncpy(od.ObjectName, queueName, (size_t)MQ_Q_NAME_LENGTH);

```

<sup>12</sup> All command line parameters after the first one (queueName) we are going to treat as elements of the message (including the spaces between the parameters).

<sup>13</sup> The remainder of this program is standard MQ coding as found in the sample programs. If its flow is not clear to you, then you need to read through some of the earlier examples in this book.

```

printf("target queue is %s\n", od.ObjectName);
openOptions = MQOO_OUTPUT          /* open queue for output */
              + MQOO_FAIL_IF QUIESCING; /* but not if MQM stopping */
MQOPEN(hCon,                        /* connection handle */
       &od,                        /* object descriptor for queue */
       openOptions,                /* open options */
       &Hobj,                      /* object handle */
       &compCode,                  /* completion code */
       &reason);                  /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQOPEN failed with reason code %ld\n",reason);
    exit(1);
}

/* set message format to STRING */
memcpy(md.Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

/* make sure we get new message and correl ids allocated */
memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));

/* finally, put the message on queue */
MQPUT(hCon,                        /* connection handle */
      Hobj,                        /* object handle */
      &md,                        /* message descriptor */
      &pmo,                        /* put options */
      strlen(msgData),            /* buffer length */
      msgData,                    /* segment buffer */
      &compCode,                  /* completion code */
      &reason);                  /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQPUT failed with reason code %ld\n",reason);
    exit(1);
}

/*****
/* Close the target queue */
*****/
MQCLOSE(hCon,                      /* connection handle */
        &Hobj,                    /* object handle */
        0,                        /* close options */
        &compCode,                /* completion code */
        &reason);                /* reason code */

```

```

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCLOSE failed with reason code %ld\n", reason);
}

/*****
/* Disconnect from queue manager */
*****/
MQDISC(&hCon,          /* connection handle */
       &compCode,     /* completion code */
       &reason);      /* reason code */
if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQDISC failed with reason code %ld\n", reason);
}

/* dump out some stats */
printf("\n");
printf("Input msgData: %s\t\t\n"
       "Output queue: %s\t\t\n",
       msgData, queueName);

return(0);
}

```

---

## 10.6 putFile.c

```
/*
*****
* NAME: putFile.c - read specified file into one or more messages and
*               write to nominated MQ queue as segmented message. 14
* SYNOPSIS:
* #include <mqfm_defs.h>
*
* DESCRIPTION: putFile <queueName> <fileName> <destDir>
*
* NB: This application assumes it will use the default queue manager.
*     Therefore, the queue manager on the machine running this application
*     must have been defined with the -q option, i.e.
*         crtmqm -q <queue manager name>
*****
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <cmqc.h>
#include <mqfm_defs.h>

int main(int argc, char **argv)
{
    FILE *inFile;                /* input file handle */

    char fileNameBuffer[MQFM_MAX_FILENAME + 15
        sizeof(MQFM_FILENAME_EYECATCHER) + 1],
        readBuffer[MQFM_MESSAGE_SIZE], /* our message buffer */
        instructionMessage[MQFM_INSTRUCTION_MESSAGE_SIZE],
        *fileName,                /* source file name */
        *directoryName = NULL,    /* source file directory name */
        *inputQueueName,         /* input queue name */
        *chPos,
        *queueName,              /* target queue name */
        *destDir = NULL; /* optional dest directory in message */
    int bytesRead,               /* bytes read from input file */
        totalBytesRead;
```

<sup>14</sup> Originally this program used V5 style segmented messages. The current version uses only V2 function. This means we have to write more code but it means we can run across any of the commonly installed MQSeries platforms.

<sup>15</sup> These three lines once again use definitions from our `mqfm_defs.h` header file.

```

/* various MQI structures needed */
MQOD od = {MQOD_DEFAULT};          /* Object Descriptor */
MQMD md = {MQMD_DEFAULT};          /* Message Descriptor */
MQOD od_original = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD md_original = {MQMD_DEFAULT}; /* Message Descriptor */
MQPMO pmo = {MQPMO_DEFAULT};       /* put message options */
MQGMO gmo = {MQGMO_DEFAULT};       /* get message options */

MQHCONN hCon;                       /* connection handle      */
MQLONG compCode;                     /* completion code        */
MQLONG messageLength;                /* returned message length */
MQLONG reason;                       /* reason code            */

MQHOBJ hObj;                         /* object handle          */
MQHOBJ hObj_input;                   /* object handle          */
MQLONG openOptions;                  /* MQOPEN options        */
MQTIM *triggerMsgP;                  /* trigger message */

int triggeredProcess = FALSE;
int messageCount;
int msgSeqNo;
char msgSeqNoText[25];
char savedMsgId[25];

printf("%s program running\n", argv[0]);

if (argc < 2) 16
{
    printf("Required parameter(s) missing\n");
    printf("Usage: %s <queueName> <fileName> [<destDir>]\n", argv[0]);
    exit(1);
}
/* have we been triggered? */

if (argc == 2)
{
    if (!(memcmp(argv[1], "TMC ", 4))) 17
    {
        triggeredProcess = TRUE;
    }
}

```

<sup>16</sup> This is a fairly standard way of checking that the parameters passed to the program are as we expect. You have probably seen such processing numerous times before.

<sup>17</sup> Why are we comparing one of the parameters passed to us with the literal "TMC"? For a full explanation, see the *MQSeries Application Programming Guide*, SC33-0807-09, Section 14.6.3 "MQSeries for OS/2 Warp, Digital OpenVMS, Tandem NSK, UNIX systems, and Windows NT trigger monitors". More specifically, see the *MQSeries Application Programming Reference*, SC33-1673-05, Section 6.1.157 "MQTMC\_\* (Trigger message character format structure identifier)".

This comparison is made because (if we are triggered) we are being passed the MQTMC2 structure. The first field in MQTMC2 is "StrucId" (MQCHAR4) and its value is "TMC".

```

    }
    else
    {
        printf("memcmp failed\n");
    }
}

if (triggeredProcess)
{
    /* map parameter to trigger message structure */
    triggerMsgP = (MQTM *)argv[1];
    inputQueueName = triggerMsgP->QName; 18
}

else 19
{
    if (argc < 3)
    {
        printf("Required parameter(s) missing\n");
        printf("Usage: %s <queueName> <fileName> [<destDir>]\n", argv[0]);
        exit(1);
    }
    if (argc == 4)
    {
        destDir = argv[3];
    }
    queueName = argv[1];
    fileName = argv[2];
}

/*****
/* Connect to queue manager */
*****/
MQCONN("", /* default queue manager */
        &hCon, /* connection handle */
        &compCode, /* completion code */
        &reason); /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCONN failed with reason code %ld\n",reason);
    exit(1);
}

```

<sup>18</sup> If we were triggered, then we get the inputQueueName from the QName field of the MQTMC2 structure we were passed. If you don't understand the (MQTM \*)argv[1] or the triggerMsgP->QName constructs, you really need to consult a good book on the "C" programming language.

<sup>19</sup> If we were not triggered, then we just process the command-line parameters we expect. See the header of this program (putFile <queueName> <fileName> <destDir>)

```

    }

/* open our input queue and instruction message */
if (triggeredProcess)
{ 20

    /******
    /* Open the instruction message queue */
    /******
    strncpy(od.ObjectName, inputQueueName, (size_t)MQ_Q_NAME_LENGTH);
    printf("input queue is %s\n", od.ObjectName);
    openOptions = MQOO_INPUT_AS_Q_DEF /* open queue for input */
        + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
    MQOPEN(hCon, /* connection handle */
        &od, /* object descriptor for queue */
        openOptions, /* open options */
        &hObj_Input, /* object handle */
        &compCode, /* completion code */
        &reason); /* reason code */

    if (compCode == MQCC_FAILED)
    {
        fprintf(stderr, "Failed to open input queue\n");
        fprintf(stderr, "MQOPEN failed with reason code %ld\n", reason);
        exit(1);
    }

    fprintf(stderr, "input queue opened\n");

    /******
    /* get the instruction message containing the file name and q name */
    /******
    gmo.Options = MQGMO_NO_WAIT /* expect message to be there */
        + MQGMO_SYNCPOINT; /* take msgs off under uow */

    MQGET(hCon, /* connection handle */
        hObj_Input, /* object handle
        &md, /* message descriptor */
        &gmo, /* get message options */
        (MQLONG)MQFM_INSTRUCTION_MESSAGE_SIZE, /*size of receive buffer */
        instructionMessage, /* message buffer */
        &messageLength, /* returned message length */

```

<sup>20</sup> All of the code in this block (until footnote 21 on page 183) is executed only if we are triggered. It is designed to process an instruction message (of the type we might have sent with the putMsg.c program). We won't comment any further on this block of code. If you want to try putFile.c in the triggered mode, you will need to read this block carefully in order to understand how to create the instruction message with putMsg.c.

```

        &compCode,          /* completion code */
        &reason);          /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQGET failed with reason code %ld\n", reason);
    exit (1);
}

/* check that message is in format we expect */
if (strcmp(instructionMessage, MQFM_FILENAME_EYECATCHER,
           strlen(MQFM_FILENAME_EYECATCHER))
    {
        fprintf(stderr, "Instruction message does not contain file name\n");
        exit(1);
    }

queueName = strstr(instructionMessage, MQFM_QUEUE_EYECATCHER);
if (!queueName)
{
    fprintf(stderr, "Instruction message does not contain queue "
                "name\n");
    exit(1);
}

directoryName = strstr(instructionMessage, MQFM_DIR_EYECATCHER);
if (!directoryName)
{
    fprintf(stderr, "Instruction message does not contain directory "
                "name\n");
    exit(1);
}

destDir = strstr(instructionMessage, MQFM_DESTDIR_EYECATCHER);

/* extract file, directory and queue name */
/*
 * format of instruction message we expect is
 *
 * FILENAME=xxxxxxx QUEUENAME=yyyyyyy DIRECTORY=zzzzzzzz DESTDIR=wwwwwww
 *
 */
fileName = instructionMessage+strlen(MQFM_FILENAME_EYECATCHER);
*(queueName-1) = '\0'; /* replace blank with string terminator */
fprintf(stderr, "extracted file name is '%s'\n", fileName);

queueName += strlen(MQFM_QUEUE_EYECATCHER);

```



```

/* add terminator which removes any trailing blanks */
for (chPos=queueName; *chPos; chPos++)
{
    if (*chPos == ' ')
    {
        *chPos = '\0';
        break;
    }
}

fprintf(stderr, "extracted queue name is '%s'\n", queueName);

*(directoryName-1) = '\0'; /* replace blank with string terminator */
directoryName += strlen(MQFM_DIR_EYECATCHER);

/* add terminator which removes any trailing blanks */
for (chPos=directoryName; *chPos; chPos++)
{
    if (*chPos == ' ')
    {
        *chPos = '\0';
        break;
    }
}

fprintf(stderr, "extracted directory name is '%s'\n", directoryName);

if (destDir)
{
    *(destDir-1) = '\0'; /* replace blank with string terminator */
    destDir += strlen(MQFM_DESTDIR_EYECATCHER);

    /* add terminator which removes any trailing blanks */
    for (chPos=destDir; *chPos; chPos++)
    {
        if (*chPos == ' ')
        {
            *chPos = '\0';
            break;
        }
    }
    fprintf(stderr, "extracted destination directory name is '%s'\n",
            destDir);
}
}^21
/* open source file for binary reading */

```

<sup>21</sup> This is the end of the if(triggeredProcess) block.

```

fileNameBuffer[0] = '\0'; 22
if (*directoryName != NULL) 23
{
    strcat(fileNameBuffer, directoryName); 24
    strcat(fileNameBuffer, "/");
}

strcat(fileNameBuffer, fileName); 25
fprintf(stderr, "about to open file '%s'\n", fileNameBuffer);

inFile = fopen(fileNameBuffer, "rb");
if (!inFile)
{
    fprintf(stderr, "failed to open file '%s'\n", fileNameBuffer);
    perror("");

    if (triggeredProcess) 26
    {
        /* Commit the duff message to remove it from queue */
        MQCMIT(hCon, /* connection handle */
              &reason); /* reason code */
        if (compCode == MQCC_FAILED)
        {
            fprintf(stderr, "MQCMIT failed with reason code %ld\n",
                    reason);
            exit(1);
        }
    }
    exit(1);
}

```

<sup>22</sup> Make fileNameBuffer a NULL string (in "C" terms).

<sup>23</sup> Note that directoryName was set to NULL back at the start, so if you were watching carefully, it can only be non-NULL at the moment if we executed the if(triggeredProcess) block.

<sup>24</sup> We have made fileNameBuffer NULL. If directoryName is not NULL then we copy directoryName into fileNameBuffer and add "/" to the end of fileNameBuffer. ie We are constructing a fully qualified path name. If your directories are delimited by characters other than "/", you might have some changes to make here!

<sup>25</sup> If you read back through the code you will see that we do have a value for fileName, either from the command line or from the instruction message (if triggered). So now we do have a fully qualified file name to open.

<sup>26</sup> Once again we won't comment much on the "triggered" code path. Note, however, that we need to commit the instruction message because we used the following get message options when we got it from the queue:

```

gmo.Options = MQGMO_NO_WAIT /* expect message to be there */
             + MQGMO_SYNCPOINT; /* take msgs off under uow */

```

```

memcpy(&od, &od_original, sizeof(od)); 27
memcpy(&md, &md_original, sizeof(md));

    /*****
    /* Open the target message queue */
    *****/
strncpy(od.ObjectName, queueName, (size_t)MQ_Q_NAME_LENGTH); 28
printf("target queue is %s\n", od.ObjectName);

openOptions = MQOO_OUTPUT           /* open queue for output */
              + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */

MQOPEN(hCon,                          /* connection handle */ 29
        &compCode,                    /* completion code */
        &od,                          /* object descriptor for queue */
        openOptions,                  /* open options */
        &hObj,                        /* object handle */
        &compCode,                   /* completion code */
        &reason);                    /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQOPEN failed with reason code %ld\n", reason);
    exit(1);
}

    /*****
    /* put the header message containing the file name */
    *****/

/* set message format to STRING */
memcpy(md.Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);
/* Header msg will always have correlid 0 */ 30
msgSeqNo = 0;
memcpy(msgSeqNoText, MQFM_FIRSTMSG_FLAG, sizeof(md.CorrelId)); 31

/* make sure we get new message and correl IDs allocated */

```

<sup>27</sup> This is just a way of copying MQOD\_DEFAULT and MQMD\_DEFAULT back into "od" and "md" respectively. The "od" (object descriptor) may be overwritten during the MQOPEN. The "md" (message descriptor) was overwritten during the MQGET.

<sup>28</sup> Whether we have arrived here via triggering or the command line, we do have a value for queueName.

<sup>29</sup> The queue that we are opening here is the one on which we will put the messages which make up the file that we are transferring.

<sup>30</sup> Our logic is that the header message for a file being transferred will always have value zeroes (see footnote 8 on page 173 and 10.1, "Design" on page 168, including Table 8 on page 168).

<sup>31</sup> Same as footnote 30 above.

```

memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId)); 32
memcpy(md.CorrelId, msgSeqNoText, sizeof(md.CorrelId)); 33

    /* set up put message options */
    pmo.Options |= MQPMO_SYNCPOINT;

    /* make sure we get rid of any directory path info from file name */
    {
    int i;
    for (i=strlen(fileName)-1; i>=0; i--)
        {
            if (fileName[i] == '\\\' || fileName[i] == '/')
                {
                    fileName = fileName+i+1;
                    break;
                }
        }
    }
    fileNameBuffer[0] = '\0'; 34
    strcpy(fileNameBuffer, MQFM_FILENAME_EYECATCHER); 35
    strcat(fileNameBuffer, fileName);
    if (destDir) 36
        {
            strcat(fileNameBuffer, "MQFM_DESTDIR_EYECATCHER"); 37
            strcat(fileNameBuffer, destDir);
        }
}

```

<sup>32</sup> The MQMI\_NONE ensures that the Queue Manager will create a new and unique MsgId which will be available (in md.MsgId) after the MQPUT.

<sup>33</sup> We are setting md.CorrelId equal to msgSeqNoText which is equal to MQFM\_FIRSTMSG\_FLAG which is equal to "000000000000000000000000". This has the effect of forcing the CorrelId to that value of zeroes in the header of the message. This is what will allow us (at the receiving end) to recognize this message as a "header message" type. See Table 8 on page 168.

<sup>34</sup> fileNameBuffer was used in the "fopen" call to open the file which we are sending. It won't be used again for the duration of the program. Here we reuse it as a buffer to build the data portion of the Header Message. Here I set it to the NULL string. (You may consider this poor programming and prefer to define another buffer).

<sup>35</sup> These two lines copy into fileNameBuffer both MQFM\_FILENAME\_EYECATCHER (ie FILENAME=) and the value in fileName.

<sup>36</sup> If we have a value for destDir" or in other words "if a value for the destination directory has been specified". What this means is that it was decided to specify the "destination directory" from the file-sending end rather than picking up the default destination directory at the file-receiving end.

See footnote 6 on page 173 for an example of how a default destination directory would be specified at the file-receiving end (using the UserData attribute of an MQSeries process definition - which ends up in the generated Trigger Message - see footnote 54 on page 194).

<sup>37</sup> If there was a destDir then we append it and its eyecatcher to fileNameBuffer. There is a logical error here. Back when we specified the size of fileNameBuffer (near the head of the program) we allowed for the filename and its eyecatcher but not for the destDir and its eyecatcher! Since there are 200 bytes for MQFM\_MAX\_FILENAME it's not usually a problem, but you could correct this error for yourself.

```

        /* finally, put the message on queue */

MQPUT(hCon,          /* connection handle */ 38
      hObj,          /* object handle */
      &md,           /* message descriptor */
      &pmo,          /* put options */
      strlen(fileNameBuffer), /* buffer length */
      fileNameBuffer, /* segment buffer */
      &compCode,    /* completion code */
      &reason);     /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQPUT failed with reason code %ld\n",reason);
    exit(1);
}

/* save the MsgId for later */ 39

memcpy(savedMsgId, md.MsgId, sizeof(md.MsgId));

/* set message format to NONE to avoid message data conversion */

memcpy(md.Format, MQFMT_NONE, (size_t)MQ_FORMAT_LENGTH);

/* set up put message options */

pmo.Options |= (MQPMO_SYNCPOINT);

messageCount = 0; 40
totalBytesRead = 0; 41

        /*****
        /* read file until eof reached */
        *****/
while(!feof(inFile))

```

<sup>38</sup> This is the header message we are about to put. It has a CorrelId of zeroes (which identifies it as a header message). Note that it contains none of the data from the file we are seeking to transfer.

<sup>39</sup> It is essential that we save the MsgId. Although we allowed the queue manager to allocate this MsgId (see footnote 32) it will now be saved and passed as the MsgId of EVERY message that constitutes a part of the file we are transferring. That is our logic. A file in transfer is represented by three TYPES of message (see Table 8 on page 168):

Header	MsgId=<ReturnedOnHeaderPut>	CorrelId=000000000000000000000000
Intermediate	MsgId=<ReturnedOnHeaderPut>	CorrelId=N (value of N for the Nth msg)
Trailer	MsgId=<ReturnedOnHeaderPut>	CorrelId=999999999999999999999999

<sup>40</sup> The number of messages we've sent.

<sup>41</sup> The number of bytes of the file we've sent.

```

{
    /* read maximum of MQFM_MESSAGE_SIZE bytes from file into buffer */ 42
    bytesRead = fread((void *)readBuffer,
                     (size_t)1,
                     (size_t)MQFM_MESSAGE_SIZE,
                     inFile);

    /* did we get a full buffer? */ 43
    if (bytesRead != MQFM_MESSAGE_SIZE)
    {
        /* check for stream errors */
        if(ferror(inFile))
        {
            fprintf(stderr, "read failure on file '%s'\n",
                    fileName);

            perror("");
            exit(1);
        }

        /* no stream error, so assume its end of file */
    }

    printf("Read %d bytes from file\n", bytesRead);
    totalBytesRead += bytesRead;

    /******
    /* Build message segment and write to queue */
    /******
    /* increment our msgSeqNo - and text version */
    msgSeqNo++; 44
    sprintf(msgSeqNoText, "%024d", msgSeqNo); 45

    /* force MsgId and CorrelId to values we want */
    memcpy(md.MsgId, savedMsgId, sizeof(md.MsgId)); 46
    if (!feof(inFile))
    { /* intermediate segment */
        memcpy(md.CorrelId, msgSeqNoText, sizeof(md.CorrelId)); 47
    }

    else

```

<sup>42</sup> We read 5000 bytes at a time. Each message will have 5000 bytes of file data (except for the header, which has no file data and is quite small, and possibly the trailer, which could be 5000 bytes or less).

<sup>43</sup> If we didn't get a full buffer then after a bit more checking we assume that we have reached the end of the file.

<sup>44</sup> Easy-to-increment MsgSeqNo, which is integer.

<sup>45</sup> Not-so-easy-to-increment msgSeqNoText, which is string data.

<sup>46</sup> We want every message in the file transfer group to have the same MsgId (see footnote39 on page 187 and Table 8).

<sup>47</sup> For intermediate messages (not header or trailer) we want the CorrelId to be the text representation of MsgSeqNo which is incrementing by one for each message. See Table 8 on page 168.

```

    { /* final segment */
        memcpy(md.CorrelId, MQFM_LASTMSG_FLAG, sizeof(md.CorrelId)); 48
    }

    /* finally, put the message on queue */
    MQPUT(hCon,          /* connection handle */
          hObj,          /* object handle */
          &md,           /* message descriptor */
          &pmo,         /* put options */
          bytesRead,    /* buffer length */
          readBuffer,   /* segment buffer */
          &compCode,    /* completion code */
          &reason);     /* reason code */

    if (compCode == MQCC_FAILED)
    {
        fprintf(stderr, "MQPUT failed with reason code %ld\n", reason);
        exit(1);
    }

    /* bump the message count */
    messageCount++;
}

/*****
/* if we got to here, then we've been successful thus far, */
/* so commit msg                                           */ 49
/* *****/
MQCMIT(hCon,          /* connection handle */
       &compCode,    /* completion code */
       &reason);     /* reason code */
if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCMIT failed with reason code %ld\n", reason);
    exit(1);
}

```

<sup>48</sup> For the trailer we want a CorrelId value of MQFM\_LASTMSG\_FLAG which is "99999999999999999999". See Table 8 on page 168.

<sup>49</sup> The point here is that we have just put all the messages under syncpoint. That is, the header, all "N" intermediate messages and the trailer were all put under a logical unit of work. At this point we MQCMIT to commit all the work, thus making all the messages available for transport to the destination. The reason is that we really don't want incomplete message sets getting to the receiving end. We could handle that scenario at the receiving end but it would make our code more complex. By using the syncpoint we get MQSeries to do the worrying for us about delivery. We know we have successfully put ALL our messages so we trust that MQSeries will get them ALL to the destination.

```

/*****/ 50
/* Close the input file */
/*****/
fclose(inFile);

if (triggeredProcess)
{
/*****/
/* Close the input queue */
/*****/
MQCLOSE(hCon, /* connection handle */
        &hObj_Input, /* object handle */
        0, /* close options */
        &compCode, /* completion code */
        &reason); /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCLOSE failed with reason code %ld\n",
            reason);
}
}

/*****/
/* Close the target queue */
/*****/
MQCLOSE(hCon, /* connection handle */
        &hObj, /* object handle */
        0, /* close options */
        &compCode, /* completion code */
        &reason); /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCLOSE failed with reason code %ld\n", reason);
}

/*****/
/* Disconnect from queue manager */
/*****/
MQDISC(&hCon, /* connection handle */
        &compCode, /* completion code */
        &reason); /* reason code */
if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQDISC failed with reason code %ld\n", reason);
}

```

<sup>50</sup> The remainder of this program is standard MQSeries code for closing down.



```
    }

    /* dump out some stats */

    printf("\n");
    printf("Input file: %s\t\t%d bytes read\n",
           "Output queue: %s\t\t%d segments written\n",
           fileNameBuffer, totalBytesRead,
           queueName, messageCount);

    return(0);
}
```

---

## 10.7 getFile.c

```
/*
*****
* NAME:      getFile.c - read specified MQ queue and recover named file
*            from retrieved segmented messages. 51
*
* SYNOPSIS:
* #include <mqfm_defs.h>
*
* DESCRIPTION: getFile <queueName> <fileName>
*
* NB: This application assumes it will use the default queue manager.
*     Therefore, the queue manager on the machine running this application
*     must have been defined with the -q option, i.e.
*         crtmqm -q <queue manager name>
*
*****
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <cmqc.h>
#include <mqfm_defs.h>

int main(int argc, char **argv)
{
    FILE *outFile;                /* output file handle */
    char fileNameBuffer[MQFM_MAX_FILENAME +
                        sizeof(MQFM_FILENAME_EYECATCHER) + 1];
    char writeBuffer[MQFM_MESSAGE_SIZE], /* our message buffer */
        *fileName, /* target file name */
        *userFileName = NULL, /* user supplied file name */
        *dropoffDirectory=NULL, /* output directory name */
        fullFileName[MQFM_MAX_FILENAME + MQFM_MAX_DIRECTORY],
        *queueName, /* source queue name */
        *chPos,
        *destDir; /* optional dest directory in message */
    int bytesWritten, /* bytes read from input file */
        totalBytesWritten;
    int fileComplete; /* set when complete file read */
    MQLONG messageLength;
```

<sup>51</sup> As was noted at the head of the putFile program, this is not V5 segmentation, but our own coding. By now you will be getting familiar with the style of these programs, so we will comment less from here on.

```

/* various MQI structures needed */
MQOD od = {MQOD_DEFAULT};           /* Object Descriptor */
MQMD md = {MQMD_DEFAULT};           /* Message Descriptor */
MQGMO gmo = {MQGMO_DEFAULT};        /* get message options */

MQHCONN hCon;                        /* connection handle */
MQLONG compCode;                     /* completion code */
MQLONG reason;                       /* reason code */

MQHOBJ hObj;                          /* object handle */
MQLONG openOptions;                  /* MQOPEN options */
MQTM *triggerMsgP;                   /* trigger message */

int messageCount;
int triggeredProcess = FALSE;        /* are we triggered? */
int moreMessages = TRUE;            /* any more messages? */
int msgSeqNo, SeqNoFromMsg;         /* seq numbers to be compared */
char savedMsgId[25];                /* we use the same MsgId many times */
char CorrelIdBuff[25];              /* a scratchpad field for the sequence number */

printf("%s program running\n", argv[0]);

if (argc < 2)
{
    printf("Required parameter(s) missing\n");
    printf("Usage: %s <queueName> [<fileName>]\n", argv[0]);
    exit(1);
}

/* have we been triggered? */
if (argc == 2)
{
    if (!(memcmp(argv[1], "TMC ", 4))) 52
    {
        triggeredProcess = TRUE;
    }
}

if (triggeredProcess)
{
    /* map parameter to trigger message structure */
    triggerMsgP = (MQTM *)argv[1];
    queueName = triggerMsgP->QName; 53
}

```

<sup>52</sup> See footnote 17 on page 179 for an explanation of what's happening here. Note that both putFile and getFile are able to run triggered, or from the command line.

<sup>53</sup> See footnote 18 on page 180.

```

        dropoffDirectory = triggerMsgP->UserData; 54
    }
else
    {
        /* extract input parameters for convenience */
        queueName = argv[1];
        if (argc == 3)
            {
                userFileName = argv[2]; 55
            }
    }

/*****
/* Connect to queue manager */
*****/
MQCONN("", /* default queue manager */
        &hCon, /* connection handle */
        &compCode, /* completion code */
        &reason); /* reason code */

if (compCode == MQCC_FAILED)
    {
        fprintf(stderr, "MQCONN failed with reason code %ld\n",reason);
        exit(1);
    }

/*****
/* Open the source message queue */
*****/
strncpy(od.ObjectName, queueName, (size_t)MQ_Q_NAME_LENGTH);
printf("source queue is %s\n", od.ObjectName);

openOptions = MQOO_INPUT_AS_Q_DEF /* open queue for input */
              + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
MQOPEN(hCon, /* connection handle */
        &od, /* object descriptor for queue */
        openOptions, /* open options */
        &compCode, /* completion code */
        &reason); /* reason code */

```

<sup>54</sup> This is new. See Table 9 on page 170 (footnote i) and footnote 6 on page 173, to understand what this is used for. The default value for the directory part of the fully qualified pathname for the file we are about to reconstruct reaches us in this way. It was set as the UserData attribute of the PROCESS statement and only applies if we have been triggered. When getFile is triggered, this UserData is transferred into the Trigger Message. What you see here is getFile retrieving that data from the Trigger Message.

<sup>55</sup> If getFile was started from the command line and if a filename was specified, this is retrieved. It will later be used to override any filename that may have arrived in the Header Message.

```

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQOPEN failed with reason code %ld\n",reason);
    exit(1);
}

fprintf(stderr, "queue opened\n");

/*****
/* We loop here. When we have finished processing a file,
/* we go back for another header message,
/* which MUST have CorrellId "MQFM_FIRSTMSG_FLAG".
/* If such a one does not exist, we conclude
/* that there are no more messages to process.
*****/

while (moreMessages) 56
{

    /*****
    /* get the header message containing the file name */
    /*****
    msgSeqNo = 0;          /* our hdr msg will always have CorrelId 0 */ 57
    memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId)); 58
    memcpy(md.CorrelId, MQFM_FIRSTMSG_FLAG, sizeof(md.CorrelId)); 59

    gmo.Options = MQGMO_NO_WAIT          /* expect message to be there */
                + MQGMO_SYNCPOINT;      /* take msgs off under uow */

    MQGET(hCon,                          /* connection handle */
          hObj,                          /* object handle */
          &md,                          /* message descriptor */
          &gmo,                          /* get message options */
          (MQLONG)sizeof(fileNameBuffer), /* size of receive buffer */
          fileNameBuffer,                /* message buffer */
          &messageLength,               /* returned message length */
          &compCode,                    /* completion code */
          &reason);                      /* reason code */

```

<sup>56</sup> This is the outer loop for processing all the messages for an individual file transfer.

<sup>57</sup> We have already covered this well. See Table 8 on page 168.

<sup>58</sup> Get the MQ-generated MsgID from the Header Message of the file. This will represent the file through all the messages that comprise it.

<sup>59</sup> We recognize a Header Message because it has CorrelId = MQFM\_FIRSTMSG\_FLAG = "000000000000000000000000".

```

if (compCode == MQCC_FAILED)
{
    moreMessages = FALSE;
    printf(stderr, "No more files to process: MQRC %ld\n", reason);
    exit(1);
}

/* save the MsgId for later */
memcpy(savedMsgId, md.MsgId, sizeof(md.MsgId)); 60

fprintf(stderr, "header message got\n");
/* check that message is in format we expect */
if (strncmp(fileNameBuffer, MQFM_FILENAME_EYECATCHER,
            strlen(MQFM_FILENAME_EYECATCHER)) 61
    {
        fprintf(stderr, "Header message does not contain file name\n");
        exit(1);
    }
/* check for optional destination directory in message */
destDir = strstr(fileNameBuffer, MQFM_DESTDIR_EYECATCHER); 62

/* extract file name */
fileName = fileNameBuffer + strlen(MQFM_FILENAME_EYECATCHER);
if (destDir)
    {
        *(destDir-1) = '\0'; /* replace blank with string terminator */
        destDir += strlen(MQFM_DESTDIR_EYECATCHER);

        /* add terminator which removes any trailing blanks */
        for (chPos=destDir; *chPos; chPos++)
            {
                if (*chPos == ' ')
                    {
                        *chPos = '\0';
                        break;
                    }
            }
    }
if (userFileName) 63
    {
        fileName = userFileName;
    }

```

<sup>60</sup> See footnote 58 on page 195.

<sup>61</sup> In the Header Message, the filename is mandatory.

<sup>62</sup> The Destination Directory is optional.

<sup>63</sup> If getFile was started from the command line and a filename was specified, then this overrides the filename passed in the Header Message. See footnote 55 on page 194 and Table 9 on page 170.

```

printf("using %sfilename '%s'\n",
userFileName ? "user supplied " : "", fileName);

/* open target file for binary writing */
fullFileName[0] = '\0';

/* build file name */
if (destDir) 64
{
    strcat(fullFileName, destDir);
    strcat(fullFileName, "/");
}
else
{ 65
    if (dropoffDirectory)
    {
        strcat(fullFileName, dropoffDirectory);
        /* strip rubbish from end of where dropoffDirectory points */
        *strstr(fullFileName, MQFM_TRIGM_USERD_EYECATCHER) = 0x00;
        strcat(fullFileName, "/");
    }
}
strcat(fullFileName, fileName);

outFile = fopen(fullFileName, "wb");
if (!outFile)
{
    printf(stderr, "failed to open file '%s'\n", fullFileName);
    perror("");
    exit(1);
}

messageCount = 0;
totalBytesWritten = 0;

/*****
/* Get messages from the message queue until we've retrieved the */
/* complete file */
*****/
fileComplete = FALSE;
while (!fileComplete) 66
{

```

<sup>64</sup> IF a destination directory was supplied in the Header Message

<sup>65</sup> THEN it overrides the destination directory supplied via the Trigger Message (which comes from the UserData attribute of the PROCESS statement of the triggered process). See Table 9 on page 170.

<sup>66</sup> This is the inner loop where we cycle through all the file transfer messages that contain file data. (that is, all but the Header Message).

```

gmo.Options = MQGMO_NO_WAIT      /* expect message to be there */
              + MQGMO_SYNCPOINT; /* take msgs off under uow    */

/*****/
/* In order to read the messages in sequence, MsgId and      */
/* CorrelID must have the correct values. MQGET sets them    */
/* on every call, so we re-initialise them before every call */
/*****/
memcpy(md.MsgId, savedMsgId, sizeof(md.MsgId)); 67
memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId)); 68

/*****/
/* MQGET sets Encoding and CodedCharSetId to the values in  */
/* the message returned, so these fields should be reset to */
/* the default values before every call.                      */
/*****/
md.Encoding      = MQENC_NATIVE;
md.CodedCharSetId = MQCCSI_Q_MGR;

MQGET(hCon,                /* connection handle */
      hObj,                /* object handle */
      &md,                 /* message descriptor */
      &gmo,               /* get message options */
      (MLONG)sizeof(writeBuffer), /* size of receive buffer */
      writeBuffer,        /* message buffer */
      &messageLength,    /* returned message length */
      &compCode,         /* completion code */
      &reason);          /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQGET failed with reason code %ld\n", reason);
    exit(1);
}

messageCount++;
msgSeqNo++;

/*****/
/* check for final segment */
/*****/
/* convert sequence no in message to integer */
memcpy(CorrelIdBuff, md.CorrelId, sizeof(md.CorrelId));
seqNoFromMsg = atoi(CorrelIdBuff);

```

<sup>67</sup> We specify to the MQGET the MsgID that it must match on the GET.

<sup>68</sup> We READ from the resulting Message Header the CorrelID.



```

if (!memcmp(md.CorrelId, MQFM_LASTMSG_FLAG, sizeof(md.CorrelId)))69
{
    fileComplete = TRUE;
}
else
{
    if (msgSeqNo != SeqNoFromMsg)70
    {
        fprintf(stderr, "MQ message contains invalid sequence number.\n");
        fprintf(stderr, "Next Seq No should have been: (%d)\n",
            msgSeqNo);
        fprintf(stderr, "The message actually contained: (%d)\n",
            SeqNoFromMsg);
    }
}

/*****/
/* write segment to file */71
/*****/
bytesWritten = fwrite((void *)writeBuffer,
                    size_t)1,
                    (size_t)messageLength,
                    outFile);

/* did we write the whole buffer? */
if (bytesWritten != messageLength)
{
    /* check for stream errors */
    if (ferror(outFile))
    {
        fprintf(stderr, "write failure on file '%s'\n",
            fullFileName);

        perror("");
        exit(1);
    }
}
/* can't think how we could ever get here, but we'd better exit */
fprintf(stderr, "unexpected write failure on file '%s'\n",
        fullFileName);

exit(1);
}
printf("Wrote %d bytes to file\n", bytesWritten);
totalBytesWritten += bytesWritten;
} /* end of second WHILE

```

<sup>69</sup> We check to see if this is the LAST chunk of the file transfer data. See footnote 48 on page 189.

<sup>70</sup> If not, we check that it is the next expected chunk in the sequence. See footnote 2 on page 169.

<sup>71</sup> In this block we write the chunks out to the file on this, the receiving side.

```

/* if we got to here, then we've been successful thus far, */
/* so commit msg                                          */ 72
    MQCMIT(hCon,          /* connection handle */
           &compCode,    /* completion code */
           &reason);     /* reason code */
if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCMIT failed with reason code %ld\n",reason);
    exit(1);
}

/*****/ 73
/* Close the output file */
/*****/
fclose(outFile);

} /* end of first WHILE */

/*****/
/* We have processed ALL messages (logical files) from the queue */
/*****/

/*****/
/* Close the source queue */
/*****/
MQCLOSE(hCon,          /* connection handle */
        &hObj,        /* object handle */
        0,            /* close options */
        &compCode,    /* completion code */
        &reason);     /* reason code */

if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQCLOSE failed with reason code %ld\n",
reason);
}

/*****/
/* Disconnect from queue manager */
/*****/

```

<sup>72</sup> We have been doing all our GETs under syncpoint, so if we have successfully written the file, we MQCMIT that does the final transactional delete of the messages from the queue.

**Note:** If the program fails before this MQCMIT, then all the messages will be rolled back onto the queue and the file chunks will be available once again to be processed when/if getFile is re-invoked.

<sup>73</sup> All standard MQSeries processing for termination, from here on.

```

MQDISC(&hCon,                /* connection handle */
       &compCode,           /* completion code */
       &reason);           /* reason code */
if (compCode == MQCC_FAILED)
{
    fprintf(stderr, "MQDISC failed with reason code %ld\n",
reason);
}

/* dump out some stats */
printf("\n");
printf("Input queue: %s\t\t%d segments read\n"
       "Output file: %s\t\t%d bytes written\n",
       queueName, messageCount,
       fullFileName, totalBytesWritten);

return(0);
}

```



---

## Chapter 11. MQSeries Security Changes

There are significant changes in the security function with MQSeries Version 5.1. If you are migrating from MQSeries Version 5.0 or an earlier release, you will need to consider the impact of the following changes on your existing procedures.

---

### 11.1 MQSeries for Windows NT

A user ID can be up to 20 characters long and can be domain-qualified, for example, user@domain. A domain-qualified user ID can be up to 64 bytes long.

MQSC, PCF, and SETMQAUT have been changed to allow long names.

**Note:** Although the architecture can now accommodate 64-character-long names, MQSeries (and Windows NT) support user IDs of no more than 20 characters, and domain names no more than 15 characters long.

The Windows NT Security Identifier (SID) is used to supplement the user ID. The following structures and formats are changed to accommodate the SID:

- mqmd: changed use of the AccountingToken field
- mqod: new field
- mqcd: new field

The Object Access Manager (OAM) has been rewritten to allow SID and domain information to be specified in addition to a user ID. A new caching mechanism has been implemented to improve performance. And new event log messages give details of authorization failures.

---

### 11.2 MQSeries Client Identification

Windows NT and UNIX clients send the currently logged-in user ID when they connect to a queue manager. The Windows NT Client sends the logged-in user's Security Identifier (SID) also.

These clients no longer read the mq\_user\_id and mq\_password environment variables.

Windows 95 and 98 Clients send mq\_user\_id (if set), or the current user ID (if logged in).

If neither is available, no client identification is passed.



Figure 150 on page 204 display SIDs for three users of different types:

- A user connected locally to QM\_2 accessing LQ\_1.QM\_2
- A user connecting to QM\_2 as a Client and accessing LQ\_1.QM\_2
- A user connected to QM\_4, accessing LQ\_1.QM\_2 remotely

## 11.4 Authorization Check

When MQSeries for Windows NT performs authorization checks, it uses the SID to query the full information from the SAM database. The issuing SAM can be identified from the SID, so SID resolution is efficient; the correct SAM can be queried directly.

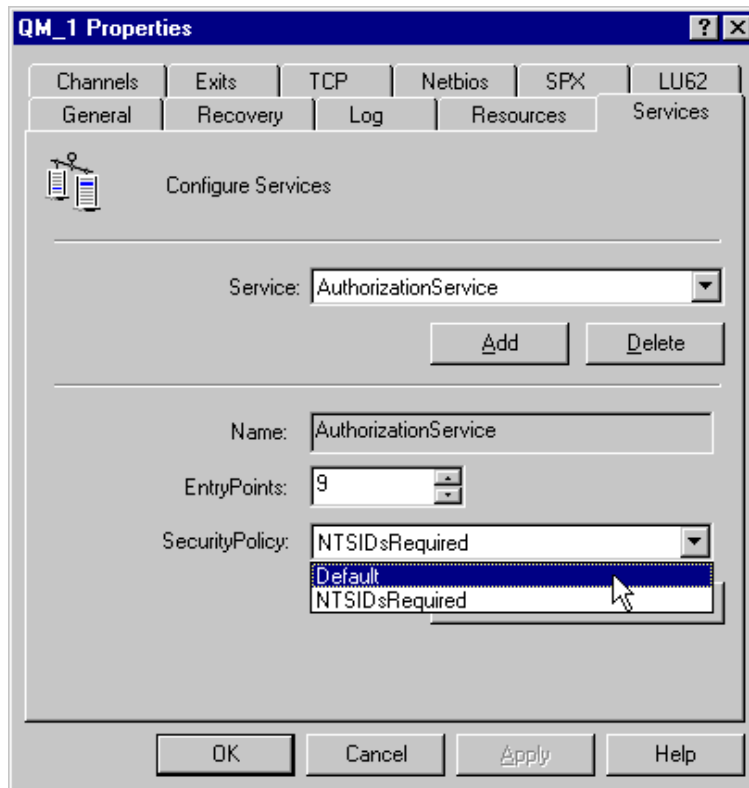


Figure 151. Security Policy

The SAM database in which the user was defined must be accessible for this query to succeed. An NT system that belongs to a domain has access to the

SAMs on the primary domain and all trusted domains, as well as the local system SAM.

A security policy can be specified for each queue manager by setting the SecurityPolicy attribute in the Queue Manager Properties.

The security policy dictates how the OAM behaves when it receives authority requests which do not contain Windows NT security identifier (NT SID) information, for example, in a multi-platform environment. When using the default security policy, it is permissible for the OAM to receive authority requests that do not contain SID information. In such situations, the OAM attempts to resolve the user ID into a Windows NT SID by searching:

- The local security database
- The security database of the primary domain
- The security database of trusted domains

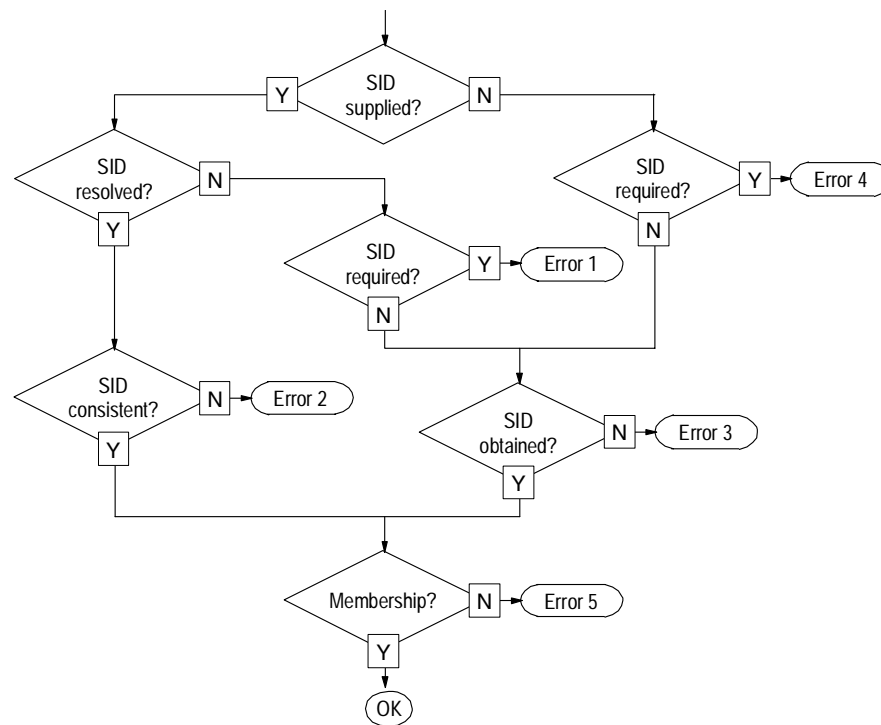


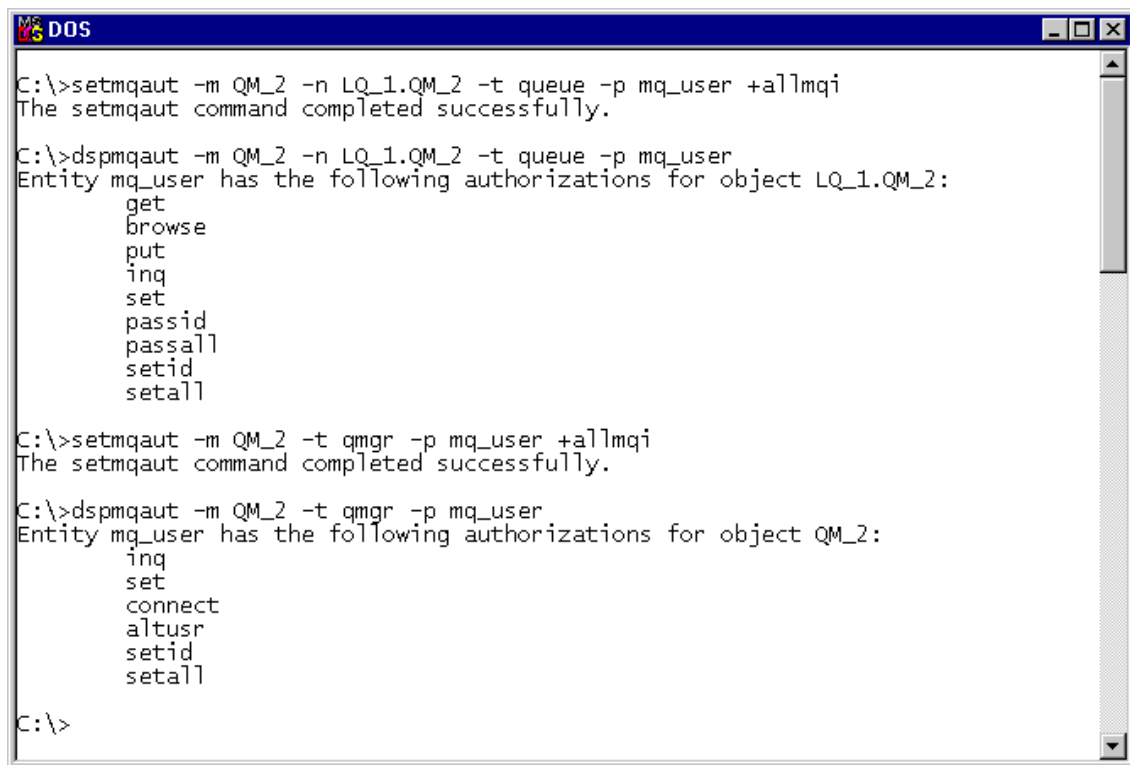
Figure 152. SID Processing



If the security policy is set to the value NTSIDsRequired, then both the user ID and NT SID information must be passed to the OAM. In cases where both a user ID and NT SID information are passed to the OAM, a check is made to ensure that the two are consistent. The supplied user ID is compared with the user ID (or the first 12 characters if the user ID is longer than 12 characters) associated with the NT SID. If the two are unequal, then authorization fails. This consistency check is performed regardless of the security policy setting. The logic flow of how the SID is handled is summarized in Figure 152 on page 206.

**Errors:**

- UNABLE\_TO\_RESOLVE\_SID (AMQ8073)
- INCONSISTENT\_ENTITY (AMQ8074)
- UNABLE\_TO\_OBTAIN\_SID (AMQ8075)
- NO\_SID (AMQ8076)
- NOT\_AUTHORIZED (various messages corresponding to reason code 2035)



```
C:\>setmqaut -m QM_2 -n LQ_1.QM_2 -t queue -p mq_user +allmqi
The setmqaut command completed successfully.

C:\>dspmqaut -m QM_2 -n LQ_1.QM_2 -t queue -p mq_user
Entity mq_user has the following authorizations for object LQ_1.QM_2:
    get
    browse
    put
    inq
    set
    passid
    passall
    setid
    setall

C:\>setmqaut -m QM_2 -t qmgr -p mq_user +allmqi
The setmqaut command completed successfully.

C:\>dspmqaut -m QM_2 -t qmgr -p mq_user
Entity mq_user has the following authorizations for object QM_2:
    inq
    set
    connect
    altusr
    setid
    setall

C:\>
```

Figure 153. Authorization Example

The setmqaut and dspmqaut control commands are still used to set, change, and display authorizations in MQSeries 5.1. For example, authorize mq\_user as client to access local queue LQ\_1.QM\_2 on queue manager QM\_2 is shown in Figure 153 on page 207.

---

## 11.5 Security in Clusters

Here we provide answers to some frequently asked questions:

*How do I stop certain queue managers sending messages to my queue manager?*

Use existing channel security exits.

*How do I stop certain user IDs in the cluster putting to my queues?*

Use existing queue security, in conjunction with PUTAUTH(DEFICTX) on the CLUSRCVR.

*How do I force an unwanted queue manager out of a cluster?*

Use the command:

```
RESET CLUSTER (name) QMNAME (QueueManagerName) ACTION (FORCEREMOVE)
```

*How do I limit activity from a specific queue manager in the cluster?*

Use a channel message exit to substitute the user ID in the message descriptor for a less privileged one. Or set PUTAUT(DEF) and set the MQCD MCAUSER to a low privilege user.

*How do I stop specific user IDs from putting to a cluster queue at the originating end?*

Use existing queue security at the putting queue manager.

---

## Chapter 12. Using Dynamic Queues

There are two kinds of dynamic queues:

- Temporary dynamic queues
- Permanent dynamic queues

In order to create a dynamic queue you need a model queue as a template. MQSeries provides the SYSTEM.DEFAULT.MODEL.QUEUE which is used to create temporary dynamic queues. To create permanent dynamic queues, define your own model. Temporary dynamic queues are deleted from the system when the queue is closed, whereas a permanent dynamic queue is deleted explicitly.

---

### 12.1 Temporary Dynamic Queues

Temporary dynamic queues hold non-persistent messages only. Such a queue is often used as “reply-to queue”. After the application ends it is no longer needed. Also, it is not recoverable.

To create a temporary dynamic queue:

- Specify the model queue in the MQOD ObjectName field.
- Specify its name in the MQOD DynamicQName field.

The queue is deleted:

- When the queue manager is started
- When the application that created the queue closes it
- When the application that created it ends

In this section we will show how to create and use a temporary dynamic queue as a reply-to queue for a COA (confirmation on arrival) report message. You will put a message in a queue and receive a COA report once the message has reached the queue. In order to do this we will specify a dynamic queue to receive the report message.

#### 12.1.1 Creating a Temporary Dynamic Queue

1. In order to create a dynamic queue you need to specify a model queue to be used as a template. You can use the MQSeries-provided model queue, SYSTEM.DEFAULT.MODEL.QUEUE, shown below:

```
define qmodel(SYSTEM.DEFAULT.MODEL.QUEUE) deftype(TEMPDYN)
```

This queue is automatically created when you create the queue manager. Its default attributes are shown in Figure 154.

```

dis qmodel (system.default.model.queue)
  1 : dis qmodel (system.default.model.queue)
AMQ8409: Display Queue details.
DESCR ( )                                PROCESS ( )
BOQNAME ( )                               INITQ ( )
TRIGDATA ( )                             QUEUE (SYSTEM.DEFAULT.MODEL.QUEUE)
CRDATE (1999-09-13)                      CRTIME (13.12.21)
ALTDATE (1999-09-13)                     ALTTIME (13.12.21)
GET (ENABLED)                             PUT (ENABLED)
DEFPRTY (0)                               DEFPST (NO)
MAXDEPTH (5000)                           MAXMSGL (4194304)
BOTHRESH (0)                             NOSHARE
DEFSOFT (EXCL)                            NOHARDENBO
MSGDLVSQ (PRIORITY)                      RETINTVL (999999999)
USAGE (NORMAL)                            NOTRIGGER
TRIGTYPE (FIRST)                         TRIGPTH (1)
TRIGMPRI (0)                             QDEPTHHI (80)
QDEPTHLO (20)                            QDPMAXEV (ENABLED)
QDPHIEV (DISABLED)                       QDPLOEV (DISABLED)
QSVCI (999999999)                        QSVCI (NONE)
DISTL (NO)                                DEFTYPE (TEMPDYN)
TYPE (QMODEL)

```

Figure 154. SYSTEM.DEFAULT.MOEL.QUEUE Attributes

2. The dynamic queue is created when you issue an MQOPEN. Before you open it, move the name of the model queue into the object descriptor, such as:

```

MQOD      odd = {MQOD_DEFAULT}; /* Dynamic queue object descr */
strncpy(odd.ObjectName,
        "SYSTEM.DEFAULT.MODEL.QUEUE",
        (size_t)MQ_Q_NAME_LENGTH);

```

3. You can specify your own name for the dynamic queue to be created or let the system create one for you. You can also do both. If you add an \* at the end of your name the system will create a tag that will be appended to the name you specified, for example:

```

// Provide your own name for the dynamic queue or a prefix
strncpy(odd.DynamicQName, /* default temp dynamic queue */
        "DYNQ4REP.*",
        (size_t)MQ_Q_NAME_LENGTH);

```

The dynamic queue that the queue manager creates using the above specifications will be similar to DYNQ4REP.1999110120035043. You can see that the suffix contains the date and a random number.

4. Next open the queue with the open options you require, for example:

```
O_options = MQOO_INPUT_AS_Q_DEF      /* open queue for input      */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */

MQOPEN    (Hcon,                      /* connection handle        */
           &odd,                      /* object descriptor for queue*/
           O_options,                 /* open options              */
           &Hobjd,                   /* object handle             */
           &OpenCode, &Reason);      /* completion and reason codes*/
```

5. After the queue is created, save its name in a variable for later use. You will need it when you specify the reply-to queue in the request message:

```
// This field will hold the name of the dynamic queue
MQBYTE    DynQName[49];
strncpy (DynQName, odd.ObjectName, 48);
DynQName[48] = '\0';
```

### 12.1.2 Writing to a Temporary Dynamic Queue

Now let us create a request message that requests a COA report message to be placed in the dynamic queue we just created. MQ Series provides an efficient call you can use when you only want to open a queue, put a message in it and then close it, namely MQPUT1.

1. We use the default options and request a COA report by setting a switch in the message header:

```
O_options = MQOO_OUTPUT;      /* open queue for output      */
md.Report = MQRO_COA ;        /* specify that you expect COA */
```

2. Now we have to specify the “return address”, which is the reply-to queue and the reply-to queue manager:

```
mstrcpy(md.ReplyToQ,          /* ReplyToQ is the dynamic queue */
        odd.ObjectName,
        sizeof(odd.ObjectName));

strncpy(md.ReplyToQMgr,      /* ReplyToQM is the owning queue manager*/
        odd.ObjectQMgrName,
        sizeof(odd.ObjectQMgrName));
```

In the above memcpy instruction we can obtain the queue name from both odd.ObjectName and the variable DynQName.

3. Now we can issue a normal MQPUT1 call. Here we put out an empty message.

```
buflen = 0 ; /* put an empty message */
MQPUT1 (Hcon, /* connection handle */
        &odt, /* object descriptor for queue */
        &md, /* message descriptor */
        &pmo, /* default options (datagram) */
        buflen, /* buffer length */
        &buffer, /* message buffer */
        &CompCode, &Reason); /* completion and reason codes */
```

### 12.1.3 Getting from a Temporary Dynamic Queue

The following code describes how to check that the message you received in the dynamic queue is a report message.

1. Wait a few seconds and then issue an MQGET to retrieve the next message in the queue.

```
gmo.Options = MQGMO_WAIT /* wait for new messages */
             + MQGMO_ACCEPT_TRUNCATED_MSG;
gmo.WaitInterval = 5000; /* 5 second limit for waiting */

buflen = sizeof(buffer) - 1; /* buffer size available for GET */

memcpy(md.MsgId, /* reset MsgId to get a new one */
       MQMI_NONE, sizeof(md.MsgId) );
memcpy(md.CorrelId, /* reset CorrelId to get a new one*/
       MQCI_NONE, sizeof(md.CorrelId) );

MQGET (Hcon, /* connection handle */
       Hobjd, /* object handle */
       &md, /* message descriptor */
       &gmo, /* get message options */
       buflen, /* buffer length */
       buffer, /* message buffer */
       &messlen, /* message length */
       &CompCode, &Reason); /* completion and reason codes */
```

2. Following the MQGET we check the return code and then find out of the message we just retrieved is indeed the COA. You can use the following code:

```

if (Reason != MQRC_NONE)
{
    printf("MQGET ended with reason code %ld\n", Reason);
}
else if ( (md.MsgType == MQMT_REPORT)  &&
         (md.Feedback == MQFB_COA) )
{
    printf("Relax ... Message Arrived !\n");
}

```

---

## 12.2 Report Messages

The table below lists report message types and who generates them:

Report Type	Generated by
Exception report	MCA (message channel agent)
Expiry report	MQM (queue manager) when it executes an MQGET and detects that the message that has expired
COA Confirmation on arrival	MQM (queue manager) when the message reached the target queue
COD Confirmation on delivery	MQM (queue manager) when the message has been retrieved by the application
PON Positive action notification	Application (programmer)
NON Negative action notification	Application (programmer)

---

## 12.3 Permanent Dynamic Queues

Permanent dynamic queues hold persistent and non-persistent messages. Those queues are recoverable in case of a system failure.

To create a permanent dynamic queue:

- Define your own model queue.
- Specify the model queue name in the MQOD ObjectName field.
- Specify the name of the dynamic queue in the MQOD DynamicQName field.

A permanent dynamic queue is deleted:

- When an application closes the queue with the MQCO\_DELETE or MQCO\_DELETE\_PURGE option (not necessarily the application that created the queue).
- The purge succeeds when there are committed messages in the queue.
- It can be deleted as normal queues.

A model for a permanent dynamic queue is not automatically created. So you have to define one by yourself. Below are two examples:

```
define qmodel (PERM.MODEL.QUEUE) deftype (PERMDYN)

define qmodel (PERM.MODEL.QUEUE) deftype (PERMDYN) DEFPSIST (YES)
```

Before the open you have to put the names of the model queue and the temporary queue in the object descriptor. This is similar to creating a temporary dynamic queue:

```
strncpy (odd.ObjectName,          /* model queue */
         "PERM.MODEL.QUEUE",
         (size_t)MQ_Q_NAME_LENGTH);

strncpy (odd.DynamicQName,       /* perm. dynamic queue */
         "PERQ4REP.*",
         (size_t)MQ_Q_NAME_LENGTH);
```

When the program ends, the queue may still be there. Only temporary dynamic queues are automatically deleted when the program ends. For permanent dynamic queues, you have to specify the close option as shown below:

```
C_options = MQCO_DELETE ;

MQCLOSE (Hcon,                  /* connection handle      */
         C_options,            /* close option           */
         &CompCode, &Reason); /* completion and reason codes */
```

**Note:** The complete programs discussed in this chapter are on the diskette supplied with this book.



## Appendix A. Sample Configuration Output

```
"CLUSTER CONFIGURE QM_1....."  
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
```

Starting MQSeries Commands.

```
      : * This ALTER QMGR needs to run early. The scripts for QM_1 & QM_3 ore  
      : * therefore need to be RUN first!  
1 : ALTER QMGR REPOS(CL_MQ51)  
AMQ8005: MQSeries queue manager changed.  
      :  
      : * This is the channel which would be created if you checked  
      : * "Create Server Connection Channel to allow remote administration  
      : * of the queue manager over TCP/IP"  
      : * in Step 3 of the Create Queue Manager wizard  
2 : DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)  
      REPLACE  
AMQ8014: MQSeries channel created.  
      :  
3 : DEFINE CHANNEL(TO_QM3) CHLTYPE(CLUSSDR) TRPTYPE(TCP)  
      CONNAME('127.0.0.1(1417)') CLUSTER(CL_MQ51) REPLACE  
AMQ8014: MQSeries channel created.  
      :  
4 : DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)  
      CONNAME('127.0.0.1(1415)') CLUSTER(CL_MQ51) NETPRTY(0)  
      REPLACE  
AMQ8014: MQSeries channel created.  
      :  
5 : DEFINE QLOCAL(TQ_1) REPLACE  
AMQ8006: MQSeries queue created.  
      :  
6 : DEFINE QLOCAL(CLO_1) CLUSTER(CL_MQ51) REPLACE  
AMQ8006: MQSeries queue created.  
      :  
      :  
6 MQSC commands read.  
No commands have a syntax error.  
All valid MQSC commands were processed.  
"CLUSTER CONFIGURE QM_3....."  
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
```

Starting MQSeries Commands.

```
      : * This ALTER QMGR needs to run early. The scripts for QM_1 & QM_3  
      : * therefore need to be RUN first!  
1 : ALTER QMGR REPOS(CL_MQ51)  
AMQ8005: MQSeries queue manager changed.  
      :  
      : * This is the channel which would be created if you checked  
      : * "Create Server Connection Channel to allow remote administration of  
      : * the queue manager over TCP/IP"  
      : * in Step 3 of the Create Queue Manager wizard  
2 : DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
```

```

                REPLACE
AMQ8014: MQSeries channel created.
      :
      3 : DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
        CONNAME('127.0.0.1(1415)') CLUSTER(CL_MQ51) REPLACE
AMQ8014: MQSeries channel created.
      :
      4 : DEFINE CHANNEL(TO_QM3) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
        CONNAME('127.0.0.1(1417)') CLUSTER(CL_MQ51) NETPRTY(0) REPLACE
AMQ8014: MQSeries channel created.
      :
      5 : DEFINE QLOCAL(TQ_3) REPLACE
AMQ8006: MQSeries queue created.
      :
      6 : DEFINE QLOCAL(CLQ_1) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
      :
      7 : DEFINE QLOCAL(CLQ_ACROSS_2_3_4) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
      :
      :
7 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
"CLUSTER CONFIGURE QM_2....."
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.

```

Starting MQSeries Commands.

```

      : * Do not run this script until AFTER the scripts for QM_1 and QM_3
      : * have been run!
      :
      : * This is the channel which would be created if you checked
      : * "Create Server Connection Channel to allow remote administration of
      : * the queue manager over TCP/IP" in Step 3 of the Create Queue
      : * Manager wizard
      1 : DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
        REPLACE
AMQ8014: MQSeries channel created.
      :
      :
      :
      :
etc.

```

---

## Appendix B. Log File Created by crt\_str\_all

```
G:\>crtmqm QM_1
MQSeries queue manager created.
Creating or replacing default objects for QM_1.
Default objects statistics : 29 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.

G:\>crtmqm QM_2
MQSeries queue manager created.
Creating or replacing default objects for QM_2.
Default objects statistics : 29 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.

G:\>crtmqm QM_3
MQSeries queue manager created.
Creating or replacing default objects for QM_3.
Default objects statistics : 29 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.

G:\>crtmqm QM_4
MQSeries queue manager created.
Creating or replacing default objects for QM_4.
Default objects statistics : 29 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.

G:\>strmqm QM_1
MQSeries queue manager 'QM_1' started.

G:\>strmqm QM_2
MQSeries queue manager 'QM_2' started.

G:\>strmqm QM_3
MQSeries queue manager 'QM_3' started.

G:\>strmqm QM_4
MQSeries queue manager 'QM_4' started.

G:\>runmqsc QM_1 < QM_1.cfg
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.

Starting MQSeries Commands.

: * This ALTER QMGR needs to run early. The scripts for QM_1 & QM_3
: * therefore need to be RUN first!
1 : ALTER QMGR REPOS(CL_MQ51)
AMQ8005: MQSeries queue manager changed.
:
: * This is the channel which would be created if you checked
: * "Create Server Connection Channel to allow remote administration of
: * the queue manager over TCP/IP"
: * in Step 3 of the Create Queue Manager wizard
: * DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
```

```

                REPLACE
                :
    2 : DEFINE CHANNEL(TO_QM3) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
        CONNAME('wtr05246.itso.ral.ibm.com(1417)') CLUSTER(CL_MQ51) REPLACE
AMQ8014: MQSeries channel created.
                :
    3 : DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
        CONNAME('wtr05246.itso.ral.ibm.com(1415)') CLUSTER(CL_MQ51)
NETPRTY(0)          REPLACE
AMQ8014: MQSeries channel created.
                :
    4 : DEFINE QLOCAL(TQ_1) REPLACE
AMQ8006: MQSeries queue created.
                :
    5 : DEFINE QLOCAL(CLO_1) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
                :
                :
5 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.

```

```

G:\>runmqsc QM_2 < QM_2.cfg
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.

```

Starting MQSeries Commands.

```

    : * Do not run this script until AFTER the scripts for QM_1 and QM_3
      have been run!
    :
    : * This is the channel which would be created if you checked
    : * "Create Server Connection Channel to allow remote administration of
      the queue manager over TCP/IP"
    : * in Step 3 of the Create Queue Manager wizard
    : * DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
      REPLACE
    :
    1 : DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
        CONNAME('wtr05246.itso.ral.ibm.com(1415)') CLUSTER(CL_MQ51) REPLACE
AMQ8014: MQSeries channel created.
                :
    2 : DEFINE CHANNEL(TO_QM2) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
        CONNAME('wtr05246.itso.ral.ibm.com(1416)') CLUSTER(CL_MQ51)
NETPRTY(0)          REPLACE
AMQ8014: MQSeries channel created.
                :
    3 : DEFINE QLOCAL(TQ_2) REPLACE
AMQ8006: MQSeries queue created.
                :
    4 : DEFINE QLOCAL(CLO_1) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
                :
    5 : DEFINE QLOCAL(CLO_ACROSS_2_3_4) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
                :
                :
5 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.

```

```
G:\>runmqsc QM_3 < QM_3.cfg
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
```

Starting MQSeries Commands.

```
      : * This ALTER QMGR needs to run early. The scripts for QM_1 & QM_3
      : therefore need to be RUN first!
1 : ALTER QMGR REPOS(CL_MQ51)
AMQ8005: MQSeries queue manager changed.
      :
      : * This is the channel which would be created if you checked
      : * "Create Server Connection Channel to allow remote administration of
      : the queue manager over TCP/IP"
      : * in Step 3 of the Create Queue Manager wizard
      : * DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
      : REPLACE
      :
2 : DEFINE CHANNEL(TO_QM1) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
  CONNAME('wtr05246.itso.ral.ibm.com(1415)') CLUSTER(CL_MQ51) REPLACE
AMQ8014: MQSeries channel created.
      :
3 : DEFINE CHANNEL(TO_QM3) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
  CONNAME('wtr05246.itso.ral.ibm.com(1417)') CLUSTER(CL_MQ51)
  NETPRTY(0) REPLACE
AMQ8014: MQSeries channel created.
      :
4 : DEFINE QLOCAL(TQ_3) REPLACE
AMQ8006: MQSeries queue created.
      :
5 : DEFINE QLOCAL(CLO_1) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
      :
6 : DEFINE QLOCAL(CLO_ACROSS_2_3_4) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
      :
      :
6 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

```
G:\>runmqsc QM_4 < QM_4.cfg
04L1830,5639-B43 (C) Copyright IBM Corp. 1994, 1998. ALL RIGHTS RESERVED.
```

Starting MQSeries Commands.

```
      : * Do not run this script until AFTER the scripts for QM_1 and QM_3
      : have been run!
      :
      : * This is the channel which would be created if you checked
      : * "Create Server Connection Channel to allow remote administration of
      : the queue manager over TCP/IP"
      : * in Step 3 of the Create Queue Manager wizard
      : * DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) CHLTYPE(SVRCONN) TRPTYPE(TCP)
      : REPLACE
      :
1 : DEFINE CHANNEL(TO_QM3) CHLTYPE(CLUSSDR) TRPTYPE(TCP)
  CONNAME('wtr05246.itso.ral.ibm.com(1417)') CLUSTER(CL_MQ51) REPLACE
AMQ8014: MQSeries channel created.
```

```
      :
      2 : DEFINE CHANNEL(TO_QM4) CHLTYPE(CLUSRCVR) TRPTYPE(TCP)
          CONNAME('wtr05246.itso.ral.ibm.com(1418)') CLUSTER(CL_MQ51)
          NETPRTY(0) REPLACE
AMQ8014: MQSeries channel created.
      :
      3 : DEFINE QLOCAL(TQ_4) REPLACE
AMQ8006: MQSeries queue created.
      :
      4 : DEFINE QLOCAL(CLQ_1) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
      :
      5 : DEFINE QLOCAL(CLQ_ACROSS_2_3_4) CLUSTER(CL_MQ51) REPLACE
AMQ8006: MQSeries queue created.
      :
      :
5 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

## Appendix C. Source Code for clusput.c

```
/* **** */
/*
/* Original Program name: AMQSPUT0
/*
/* Description: Sample C program that puts messages to
/*             a message queue (example using MQPUT)
/*
/* **** */
/*
/* AMQSPUT0 has 2 parameters
/*             - the name of the target queue (required)
/*             - queue manager name (optional)
/*
/* **** */
/* This has been altered for the MQSeries V5.1 update class
/* MQOO_BIND_NOT_FIXED has been added to the open options.
/* This allows us to have the cluster load balancing spread
/* multiple PUTs (following the MQOO_BIND_NOT_FIXED open)
/* across multiple queues in the cluster (if appropriate
/* queues have been defined).
/* **** */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
    /* includes for MQI */
#include "cmqc.h"

int main(int argc, char **argv)
{
    /* Declare file and character for sample input
    FILE *fp;

    /* Declare MQI structures needed
    MQOD    od = {MQOD_DEFAULT};    /* Object Descriptor
    MQMD    md = {MQMD_DEFAULT};    /* Message Descriptor
    MQPMO   pmo = {MQPMO_DEFAULT};  /* put message options
        /** note, sample uses defaults where it can **/

    MQHCONN Hcon;                    /* connection handle
    MQHOBJ  Hobj;                    /* object handle
    MQLONG  O_options;                /* MQOPEN options
```

```

MQLONG C_options;          /* MQCLOSE options          */
MQLONG CompCode;          /* completion code          */
MQLONG OpenCode;         /* MQOPEN completion code   */
MQLONG Reason;           /* reason code               */
MQLONG CReason;          /* reason code for MQCONN   */
MQLONG messlen;          /* message length            */
char    buffer[100];      /* message buffer            */
char    QMName[50];       /* queue manager name        */

printf("MQ V5.1 Update Class: Workload-Balance enabled PUT\n");
if (argc < 2)
{
    printf("Required parameter missing - queue name\n");
    exit(99);
}

/*****
/*
/* Connect to queue manager
/*
/*
*****/
QMName[0] = 0; /* default */
if (argc > 2)
    strcpy(QMName, argv[2]);
MQCONN(QMName,          /* queue manager          */
        &Hcon,          /* connection handle      */
        &CompCode,     /* completion code        */
        &CReason);     /* reason code            */

/* report reason and stop if it failed */
if (CompCode == MQCC_FAILED)
{
    printf("MQCONN ended with reason code %ld\n", CReason);
    exit( (int)CReason );
}

/*****
/*
/* Use parameter as the name of the target queue
/*
/*
*****/
strncpy(od.ObjectName, argv[1], (size_t)MQ_Q_NAME_LENGTH);
printf("target queue is %s\n", od.ObjectName);

/*****
/*
/* Open the target message queue for output
*****/

```



```

/*                                                                 */
/*****                                                             */
O_options = MQOO_OUTPUT          /* open queue for output  */
          + MQOO_FAIL_IF_QUIESCING /* but not if MQM stopping */
          + MQOO_BIND_NOT_FIXED;  /* put to multiple clustered */
                                   /* queues                    */
MQOPEN(Hcon,                      /* connection handle     */
       &od,                      /* object descriptor for queue */
       O_options,                 /* open options          */
       &Hobj,                    /* object handle         */
       &OpenCode,                /* MQOPEN completion code */
       &Reason);                /* reason code           */

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE)
{
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

if (OpenCode == MQCC_FAILED)
{
    printf("unable to open queue for output\n");
}

/*****                                                             */
/*                                                                 */
/* Read lines from the file and put them to the message queue */
/* Loop until null line or end of file, or there is a failure */
/*                                                                 */
/*****                                                             */
CompCode = OpenCode;          /* use MQOPEN result for initial test */
fp = stdin;

memcpy(md.Format,              /* character string format */
       MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

pmo.Options |= MQPMO_NEW_MSG_ID;

while (CompCode != MQCC_FAILED)
{
    if (fgets(buffer, sizeof(buffer), fp) != NULL)
    {
        messlen = strlen(buffer); /* length without null */
        if (buffer[messlen-1] == '\n') /* last char is a new-line */
        {
            buffer[messlen-1] = '\0'; /* replace new-line with null */
            --messlen;                /* reduce buffer length */
        }
    }
}

```

```

    }
}
else messlen = 0;          /* treat EOF same as null line      */

/*****
/*
/* Put each buffer to the message queue
/*
/*
*****/
if (messlen > 0)
{
    MQPUT(Hcon,          /* connection handle      */
          Hobj,         /* object handle         */
          &md,          /* message descriptor    */
          &pmo,         /* default options (datagram) */
          messlen,     /* message length        */
          buffer,      /* message buffer        */
          &CompCode,   /* completion code       */
          &Reason);    /* reason code           */

    /* report reason, if any */
    if (Reason != MQRC_NONE)
    {
        printf("MQPUT ended with reason code %ld\n", Reason);
    }
}
else /* satisfy end condition when empty line is read */
    CompCode = MQCC_FAILED;
}

/*****
/*
/* Close the target queue (if it was opened)
/*
/*
*****/
if (OpenCode != MQCC_FAILED)
{
    C_options = 0;          /* no close options      */
    MQCLOSE(Hcon,         /* connection handle     */
            &Hobj,       /* object handle         */
            C_options,    /*
            &CompCode,   /* completion code       */
            &Reason);    /* reason code           */

    /* report reason, if any */
    if (Reason != MQRC_NONE)
    {

```

```

        printf("MQCLOSE ended with reason code %ld\n", Reason);
    }
}

/*****
/*
/* Disconnect from MQM if not already connected
/*
/*
*****/
if (CReason != MQRC_ALREADY_CONNECTED)
{
    MQDISC(&Hcon,          /* connection handle
            &CompCode,    /* completion code
            &Reason);     /* reason code

    /* report reason, if any
    if (Reason != MQRC_NONE)
    {
        printf("MQDISC ended with reason code %ld\n", Reason);
    }
}

/*****
/*
/* END OF AMQSPUT0
/*
*****/
printf("MQ V5.1 Update Class: Workload-Balance enabled PUT: end\n");
return(0);
}

```



## Appendix D. Source Code for fastget.c

```
/*
*****
/*
/* Program name: AMQSGET0
/*
/* Description: Sample C program that gets messages from
/* a message queue (example using MQGET)
/*
/* Statement: Licensed Materials - Property of IBM
/*
/* 04L1773, 5765-B73
/* 04L1802, 5639-B42
/* 04L1788, 5765-B74
/* 04L1816, 5765-B75
/* 04L1830, 5639-B43
/* (C) Copyright IBM Corp. 1994, 1998
/*
*****
/*
/* Function:
/*
/* AMQSGET0 is a sample C program to get messages from a
/* message queue, and is an example of MQGET.
/*
/* -- sample reads from message queue named in the parameter
/*
/* -- displays the contents of the message queue,
/* assuming each message data to represent a line of
/* text to be written
/*
/* messages are removed from the queue
/*
/* -- writes a message for each MQI reason other than
/* MQRC_NONE; stops if there is a MQI completion code
/* of MQCC_FAILED
/*
/* Program logic:
/* Take name of input queue from the parameter
/* MQOPEN queue for INPUT
/* while no MQI failures,
/* . MQGET next message, remove from queue
/* . print the result
/* . (no message available counts as failure, and loop ends)
*/
```

```

/*      MQCLOSE the subject queue      */
/*      */
/*      */
/*****/
/*      */
/*      AMQSGET0 has 2 parameters -      */
/*      - the name of the message queue (required)      */
/*      - the queue manager name (optional)      */
/*      */
/*****/
/* This has been altered for the MQSeries V5.1 update class      */
/* MQGMO_ACCEPT_TRUNCATED_MSG has been added to the Get Msg Options */
/* which ensures we can always read messages from the queue */
/* even if they are too long for our buffer.      */
/* USEFUL as a simple way to clear a queue.      */
/* gmo.WaitInterval has been set to 1000, which means the program */
/* now only waits for 1 second after the queue is empty      */
/* before it ends. Good for impatient lab students!!!      */
/*****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
    /* includes for MQI */
#include <cmqc.h>

int main(int argc, char **argv)
{

    /* Declare MQI structures needed      */
    MQOD      od = {MQOD_DEFAULT};      /* Object Descriptor      */
    MQMD      md = {MQMD_DEFAULT};      /* Message Descriptor      */
    MQGMO     gmo = {MQGMO_DEFAULT};    /* get message options      */
    /* note, sample uses defaults where it can */

    MQHCONN   Hcon;                      /* connection handle      */
    MQHOBJ    Hobj;                      /* object handle          */
    MQLONG    O_options;                 /* MQOPEN options        */
    MQLONG    C_options;                 /* MQCLOSE options      */
    MQLONG    CompCode;                 /* completion code       */
    MQLONG    OpenCode;                 /* MQOPEN completion code */
    MQLONG    Reason;                  /* reason code           */
    MQLONG    CReason;                 /* reason code for MQCONN */
    MQBYTE    buffer[101];             /* message buffer        */
    MQLONG    buflen;                  /* buffer length         */
    MQLONG    messlen;                 /* message length received */
    MQCHAR48  QMName;                 /* queue manager name    */

```

```

printf("MQ V5.1 Update Class: fast GET which ALWAYS clears the
queue\n");
if (argc < 2)
{
printf("Required parameter missing - queue name\n");
exit(99);
}

/*****
/*
/* Create object descriptor for subject queue
/*
*****/
strcpy(od.ObjectName, argv[1]);
QMName[0] = 0; /* default */
if (argc > 2)
strcpy(QMName, argv[2]);

/*****
/*
/* Connect to queue manager
/*
*****/
MQCONN(QMName, /* queue manager */
        &Hcon, /* connection handle */
        &CompCode, /* completion code */
        &CReason); /* reason code */

/* report reason and stop if it failed */
if (CompCode == MQCC_FAILED)
{
printf("MQCONN ended with reason code %ld\n", CReason);
exit( (int)CReason );
}

/*****
/*
/* Open the named message queue for input; exclusive or shared
/* use of the queue is controlled by the queue definition here
/*
*****/
O_options = MQOO_INPUT_AS_Q_DEF /* open queue for input */
+ MQOO_FAIL_IF QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon, /* connection handle */
        &od, /* object descriptor for queue */
        O_options, /* open options */

```

```

        &Hobj,                /* object handle          */
        &OpenCode,           /* completion code       */
        &Reason);           /* reason code           */

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE)
{
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

if (OpenCode == MQCC_FAILED)
{
    printf("unable to open queue for input\n");
}

/*****
/*
/*  Get messages from the message queue
/*  Loop until there is a failure
/*
/*
/*****
CompCode = OpenCode;        /* use MQOPEN result for initial test */
gmo.Version = MQGMO_VERSION_2; /* Avoid need to reset Message */
gmo.MatchOptions = MQMO_NONE; /* ID and Correlation ID after */
                               /* every MQGET */
gmo.Options = MQGMO_WAIT     /* wait for new messages */
              + MQGMO_CONVERT; /* convert if necessary */
              + MQGMO_ACCEPT_TRUNCATED_MSG; /* we always want to read the msg */
gmo.WaitInterval = 1000;    /* 1 second limit for waiting */

while (CompCode != MQCC_FAILED)
{
    buflen = sizeof(buffer) - 1; /* buffer size available for GET */

    /*****
    /*
    /*  MQGET sets Encoding and CodedCharSetId to the values in
    /*  the message returned, so these fields should be reset to
    /*  the default values before every call, as MQGMO_CONVERT is
    /*  specified.
    /*
    /*
    /*****

    md.Encoding          = MQENC_NATIVE;
    md.CodedCharSetId = MQCCSI_Q_MGR;

    MQGET(Hcon,          /* connection handle */

```



```

        Hobj,          /* object handle          */
        &md,           /* message descriptor    */
        &gmo,          /* get message options   */
        buflen,       /* buffer length         */
        buffer,       /* message buffer        */
        &messlen,     /* message length        */
        &CompCode,    /* completion code       */
        &Reason);    /* reason code           */

/* report reason, if any */
if (Reason != MQRC_NONE)
{
    if (Reason == MQRC_NO_MSG_AVAILABLE)
    {
        /* special report for normal end */
        printf("no more messages\n");
    }
    else /* general report for other reasons */
    {
        printf("MQGET ended with reason code %ld\n", Reason);

        /* treat truncated message as a failure for this sample */
        if (Reason == MQRC_TRUNCATED_MSG_FAILED)
        {
            CompCode = MQCC_FAILED;
        }
    }
}

/*****
/* Display each message received */
*****/
if (CompCode != MQCC_FAILED)
{
    buffer[messlen] = '\0'; /* add terminator */
    printf("message <%s>\n", buffer);
}

/*****
/*
/* Close the source queue (if it was opened)
/*
*****/
if (OpenCode != MQCC_FAILED)
{
    C_options = 0; /* no close options */
    MQCLOSE(Hcon, /* connection handle */

```

```

        &Hobj,                /* object handle          */
        C_options,
        &CompCode,           /* completion code       */
        &Reason);           /* reason code            */

/* report reason, if any */
if (Reason != MQRC_NONE)
{
    printf("MQCLOSE ended with reason code %ld\n", Reason);
}
}

/*****
/*
/* Disconnect from MQM if not already connected
/*
/*
*****/
if (CReason != MQRC_ALREADY_CONNECTED )
{
    MQDISC(&Hcon,             /* connection handle      */
          &CompCode,         /* completion code        */
          &Reason);         /* reason code            */

/* report reason, if any */
if (Reason != MQRC_NONE)
{
    printf("MQDISC ended with reason code %ld\n", Reason);
}
}

/*****
/*
/* END OF AMQSGET0
/*
/*
*****/
printf("Sample AMQSGET0 end\n");
return(0);
}

```

---

## Appendix E. MQSeries Processes

With MQSeries installed but no queue managers defined, expect to see the following MQSeries related processes in the NT Task Manager list:

amqsrvm.exe	com server for mmc
amqmtbrn.exe	Task bar routine for Services snap-in to MMC
amqsvc.exe	MQSeries NT Service

As each queue manager is started, expect to see the following set of processes:

amqhasmn.exe	Logger for circular logging <i>or</i>
amqharmn	Logger for linear logging
amqpcsea.exe	Command Server
amqrrmfa.exe	Repository process (if cluster)
amqzlaa0.exe	Local queue manager agentt
amqzllp0.exe	Checkpoint process
amqzxma0.exe	Execution controller
runmqchi.exe	Channel Initiator
runmqlsr.exe	Listener
amqxssvn.exe	Shared memory servers -- there may be multiples of these per queue manager



## Appendix F. Diskette Contents

This redbook also contains additional material in diskette format. The diskette that accompanies this redbook contains the following:

Directory \ File Name	Description
<b>\Ch02</b>	<b>Chapter 2, "About Clusters" on page 7</b>
qm1.in	Figure 19 on page 22, definitions for QM_1
qm2.in	Figure 20 on page 22, definitions for QM_2
qm3.in	Figure 21 on page 22, definitions for QM_3
<b>\Ch05</b>	<b>Chapter 5, "Creating a Cluster with Scripts" on page 95</b>
QM_1.cfg	Configuration for QM_1, Figure 107 on page 98
QM_2.cfg	Configuration for QM_2, Figure 108 on page 99
QM_3.cfg	Configuration for QM_3, Figure 109 on page 100
QM_4.cfg	Configuration for QM_4, Figure 110 on page 101
crt_str_all.bat	BAT file that builds cluster CL_MQ51, Figure 111 on page 102
cft_str_all.log	Appendix B, "Log File Created by crt_str_all" on page 217
end_dlt_all.bat	BAT file that stops and destroys cluster CL_MQ51, Figure 112 on page 103
<b>\Ch06</b>	<b>Chapter 6, "Workload Management" on page 107</b>
clusput.c	Source and executable of modified amqsput0.c to put messages with BIND_NOT_FIXED, Appendix C, "Source Code for clusput.c" on page 221
clusput.exe	
fastget.c	Source code and executable of modified amqsget0.c that reads messages even if they are too long for the buffer, Appendix D, "Source Code for fastget.c" on page 227
fastget.exe	
str_all.bat	BAT file that starts four queue managers and their listeners, Figure 114 on page 110
end_all.bat	BAT file that stops for queue managers and their listeners, Figure 115 on page 110
<b>\WLExit</b>	<b>Chapter 6.3, "Writing a Workload Management Exit" on page 116</b>
WLlogger.c	Workload management exit sample described in 6.3.2, "Commented Program Listing for Exit WLlogger.c" on page 118
WLlogger.cfg	Sample to alter queue manager to support workload exit, see page 116

Directory \ File Name	Description
WLlogger_comp.bat	BAT file to compile the exit and move it into the exits directory
<b>Ch08</b>	<b>Chapter 8, "Web Administration" on page 147</b>
Setup_QM_1.mqs	Inner script to set up QM_1
Setup_QM_2.mqs	Inner script to set up QM_2
Setup_QM_3.mqs	Inner script to set up QM_3
Setup_QM_4.mps	Inner script to set up QM_4
Add_cluster_Q.mps	Script to create a cluster queue on three queue managers
Setup_4_Q<grs.mqs	Outer script that calls the above inner scripts
	<b>Visual Basic Files</b>
amqsgen.exe	A VisualBasic Program that generates messages
VB40032.DLL	You need this file to run amqsgen; put it in \WINNT\SYSTEM32
<b>Ch10</b>	<b>Chapter 10, "File Transfer Programs" on page 167</b>
mqfm_defs.h	Header file; see 10.4, "mqfm_defs.h" on page 173
putMsg.c	Puts an "instruction message"; see 10.5, "putMsg.c" on page 174
putFile.c	Puts a file; see 10.6, "putFile.c" on page 178
getFile.c	Gets a file; see 10.7, "getFile.c" on page 192
<b>Ch12</b>	<b>Chapter 12, "Using Dynamic Queues" on page 209</b>
dynq4rep.c	12.1, "Temporary Dynamic Queues" on page 209
perq4rep.c	12.3, "Permanent Dynamic Queues" on page 213

## F.1 Locating the additional material on the Internet

The diskette associated with this redbook is also available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG245849/>

Alternatively, you can go to the IBM Redbooks Web site at:

<ftp://www.redbooks.ibm.com/>

Select the **Additional materials** and open the directory that corresponds with the redbook form number.

---

## Appendix G. Special Notices

This publication is intended to help application designers, programmers and system administrators to utilize the functions of MQSeries Version 5.1. The information in this publication is not intended as the specification of any programming interfaces that are provided by MQSeries Version 5.1. See the PUBLICATIONS section of the IBM Programming Announcement for MQSeries Version 5.1 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

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 Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have

been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX	AS/400
CICS	DB2
FFST	IBM
IMS	MQ
MQSeries	MVS/ESA
Netfinity	Operating System/2
OS/2	OS/390
OS/400	Parallel Sysplex
QMF	RS/6000
S/390	SP
SP1	SP2
SupportPac	System/390
VisualAge	

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

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.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel 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.



SET and the SET logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.



---

## Appendix H. Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

---

### H.1 International Technical Support Organization Publications

For information on ordering these ITSO publications see “How to Get IBM Redbooks” on page 243.

- *MQSeries Version 5 Programming Examples*, SG24-5214
- *MQSeries Backup and Recovery*, SG24-5222
- *MQSeries Security: Example of Using a Channel Security Exit, Encryption and Decryption*, SG24-5306
- *Using MQSeries on the AS/400*, SG24-5236
- *MQSeries for Windows Version 2.1 in a Mobile Environment*, SG24-2103
- *Connecting the Enterprise to the Internet with MQSeries and VisualAge for Java*, SG24-2144
- *Internet Application Development with MQSeries and Java*, SG24-4896
- *Application Development with VisualAge for Smalltalk and MQSeries*, SG24-2117
- *Examples of Using MQSeries on WWW*, SG24-4882
- *Using the MQSeries Integrator Version 1.0*, SG24-5386

---

### H.2 Redbooks on CD-ROMs

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at <http://www.redbooks.ibm.com/> for information about all the CD-ROMs offered, updates and formats.

<b>CD-ROM Title</b>	<b>Collection Kit Number</b>
System/390 Redbooks Collection	SK2T-2177
Networking and Systems Management Redbooks Collection	SK2T-6022
Transaction Processing and Data Management Redbooks Collection	SK2T-8038
Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044
AS/400 Redbooks Collection	SK2T-2849
Netfinity Hardware and Software Redbooks Collection	SK2T-8046
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

---

### H.3 Other Publications

These publications are also relevant as further information sources:

- *MQSeries Queue Manager Clusters*, SC34-5349
- *MQSeries Command Reference*, SC33-1369
- *MQSeries System Administration*, SC33-1873
- *MQSeries Intercommunication*, SC33-1872
- *MQSeries Application Programming Guide*, SC33-0807
- *MQSeries Application Programming Reference*, SC33-1673
- *MQSeries Using C++*, SC33-1877
- *MQSeries Using Java*, SC34-5456

---

## How to Get IBM Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** <http://www.redbooks.ibm.com/>

Search for, view, download, or order hardcopy/CD-ROM redbooks from the redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this redbooks site.

Redpieces are redbooks in progress; not all redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

Send orders by e-mail including information from the redbooks fax order form to:

	<b>e-mail address</b>
In United States	usib6fpl@ibmmail.com
Outside North America	Contact information is in the "How to Order" section at this site: <a href="http://www.elink.ibm.com/pbl/pbl">http://www.elink.ibm.com/pbl/pbl</a>

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the "How to Order" section at this site: <a href="http://www.elink.ibm.com/pbl/pbl">http://www.elink.ibm.com/pbl/pbl</a>

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the "How to Order" section at this site: <a href="http://www.elink.ibm.com/pbl/pbl">http://www.elink.ibm.com/pbl/pbl</a>

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the redbooks Web site.

### IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and redbooks by accessing the IBM Intranet Web site at <http://w3.itso.ibm.com/> and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at <http://w3.ibm.com/> for redbook, residency, and workshop announcements.

---

## IBM Redbooks Fax Order Form

Please send me the following:

Title	Order Number	Quantity
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

First name \_\_\_\_\_ Last name \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ Postal code \_\_\_\_\_ Country \_\_\_\_\_

Telephone number \_\_\_\_\_ Telefax number \_\_\_\_\_ VAT number \_\_\_\_\_

Invoice to customer number \_\_\_\_\_

Credit card number \_\_\_\_\_

Credit card expiration date \_\_\_\_\_ Card issued to \_\_\_\_\_ Signature \_\_\_\_\_

**We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.**

---

## List of Abbreviations

<b>ADSI</b>	Active Directory Service Interface	<b>MQM</b>	MQ Queue Manager
<b>API</b>	Application Programming Interface	<b>MQMD</b>	MQ Message Descriptor (message header)
<b>BMP</b>	Batch Message Processing, region (IMS)	<b>MQSI</b>	MQSeries Integrator
<b>COA</b>	Confirmation on Arrival	<b>MVS</b>	Multiple Virtual Storage
<b>COD</b>	Confirmation on Delivery	<b>OAM</b>	Object Access Manager
<b>COM</b>	Component Object Model	<b>ODBC</b>	Open Database Connectivity
<b>CSD</b>	Correctional Service Diskette	<b>OS/2</b>	Operating System/2
<b>DHCP</b>	Dynamic Host Configuration Protocol	<b>OS/400</b>	Operating System for AS/400
<b>DNS</b>	Domain Name System	<b>PC</b>	Personal Computer
<b>EOF</b>	End of File	<b>PCF</b>	Programmable Command Format
<b>FIFO</b>	first in, first out	<b>PDF</b>	Portable Document Format
<b>GIF</b>	Graphic Interchange Format	<b>PSID</b>	Page Set Identifier
<b>GUI</b>	Graphical User Interface	<b>SAM</b>	Security Account Manager
<b>HPFS</b>	High Performance File System	<b>SID</b>	Security Identifier
<b>HTTP</b>	HyperText Markup Language	<b>TCP</b>	Transmission Control Protocol
<b>IBM</b>	International Business Machines Corporation	<b>UDB</b>	Universal Database
<b>IP</b>	Internet Protocol	<b>UR</b>	Unit of Recovery
<b>ITSO</b>	International Technical Support Organization	<b>URL</b>	Uniform Resource Locator
<b>IVP</b>	Installation Verification Program	<b>VSAM</b>	Virtual Storage Access Method
<b>JDK</b>	Java Development Kit	<b>WWW</b>	World Wide Web
<b>LSX</b>	Lotus Script Extension		
<b>MCA</b>	Message Channel Agent		
<b>MMC</b>	Microsoft Management Console		
<b>MPP</b>	Message Processing Program		
<b>MQ</b>	Message Queuing		
<b>MQI</b>	Message Queuing Interface		





---

## Index

### Numerics

127.0.0.1 26  
1414 39  
2042 128  
8081 149  
90 days 17

### A

AccountingToken 203  
ActiveX 3, 33  
Add to Chart 159  
adding a trigger monitor 139  
administration  
    remote 145  
administration interfaces 123  
    MQSeries Explorer 45  
    MQSeries Services 47  
    RUNMQSC 95  
    Web administration 150  
administrative error 10  
Adobe Acrobat 34  
ADSI 34  
advertise queue 28  
advertise queue manager 18, 27  
alert message 142  
Alert Monitor 48, 132, 142  
ALTER QMGR 27  
AMQ7077 127  
AMQ8118 127  
AMQ8441 31  
AMQ9509 128  
amqerr01.log 128  
amqpcard.exe 42  
amqsgbr 82  
amqsgen 162, 164  
amqsget 163  
amqsput 82, 108, 113, 115  
amqsput.exe 107  
API calls 49  
API Exerciser 38, 49  
arrival rate 159  
AS/400 2  
automatic definition  
    channels 12, 19  
    CLUSSDRA 32

    cluster xmit queue 8  
    communication channels 7  
    dynamic channels 12  
    message channels 14  
    sender channels 76  
automatic start  
    of channel initiator 128, 144  
    of command server 132  
    of listener 60, 109  
    of MQ components 124  
    of processes 47, 124  
    of queue manager 57  
automatic startup 139  
automatically create cluster 101  
autonomous queue manager 8  
availability 10

### B

BAT files 95  
bind 15, 108  
bind not fixed 107, 108, 114, 223  
bind on open 111  
boot time 47, 57, 124, 133  
browse  
    message browser 82  
browse messages  
    MQExplorer 82, 89, 113, 162  
    using MQExplorer 45  
browser 147  
    interface 33  
business integration 1  
business strategies 2

### C

channel  
    CLUSRCVR 18  
    CLUSSDR 18  
    cluster 32  
    cluster default 105  
    cluster receiver 105  
    cluster sender 105  
    communication 7  
    CONNNAME 97  
    default configuration 39  
    define 20  
    define CLUSSDR 98

- define CUSRCVR 98
- define SVRCONN 98
- definition 55
- display 61
- dynamic 12
- in MQExplorer 130
- inbound 127
- manage 124
- matching pair 24
- name cluster channel 67
- repository 129
- server connection 39
- state 121
- status 31
- stop receiver 164
- types 17, 61
- channel initiator 48, 127, 144, 147
  - start 128
- channel names 39
- channel pairs 77
- channel program 29
- circular logging 57
- client identification 203
- clusput 108, 114
- clusput.c 109
  - source code 221
- clusput.exe 107
- CLUSRCVR 17, 18, 19, 27, 77, 105
  - define 98, 215
  - definition 20
- CLUSSDR 17, 18, 19, 27, 105
  - define 20, 98, 215
- CLUSSDRA 32
- CLUSSDRB 32
- cluster 3, 7, 47, 55, 95
  - administration 3
  - behavior 164
  - channel 32
  - commands 26
  - concept 11
  - default configuration 39
  - DHCP 39
  - example 18, 20, 21, 24
  - fewer definitions 24
  - how it works 17
  - summary 12
  - transmission queue 8, 105
  - why? 7, 8
  - with two networks 24
  - workload exit 15
- cluster objects 104
- cluster receiver channel 17, 18, 129
- cluster sender channel 17, 18, 129
- cluster transmission queue 14
- clustering 14
- clusters
  - benefits for administration 7
  - technology 12
- cluster-wide administration 7
- CLWLDATA 116
- CLWLEXIT 116
- COA 213
- COD 213
- command line 144, 167
- command queue 61, 105
- command server 47, 48, 127, 132
- commands 123
  - clear display 163
  - clusters 26, 27
  - compile 116
  - crtmqm 126
  - display CLUSQMGR 30
  - endmqm 147
  - remote administration 147
  - runmqsc 29, 97
  - scmmqm 47, 124
  - start channel initiator 128
  - start listener 127
  - strmqm 126
  - Web administration 147, 151
- communication
  - between queue managers 12, 28
  - channel 7
- compile
  - workload exit 116
- Component Object Model 3
- components 127
  - failing 3
  - of a cluster 13
  - start a boot time 133
- configuration scripts 95
- CONNAME 26
  - cluster channels 27
  - display 31
  - example 20
  - loopback 26
- connect
  - applications 1

- computers using default configuration 38
- machines 7
- MQCONN 33
- MQM to cluster 129
- to queue manager 175
- using LAN 16
- using SNA 24
- connection handle 107, 174
- connection name 24, 31
  - cluster receiver 67, 73
  - loop back 97
- construct scripts 147
- control commands 124
- controlling the workflow 108
- create
  - channels automatically 7
  - cluster 63
  - cluster automatically 3
  - cluster objects automatically 19
  - cluster queue 83
  - cluster with MQExplorer 55
  - cluster with runmqsc 95
  - default configuration 39, 40
  - instances of queue 86
  - listener 60, 104
  - local queue 79
  - MQSeries objects 104
  - queue manager 28, 57, 97, 126
  - queue manager (MQ Services) 137
  - queue manager (MQExplorer) 57
  - server connection channel 57
  - trigger monitor service 139
  - Web administration script 155
- Create Cluster Wizard 64
- crtmqm 126
  - fails 127
- custom install 34, 148

**D**

- data stores 124
- dead-letter queue 14, 17, 57
- default
  - configuration 33, 38
  - installation 34
  - installation directory 34
  - message distribution 15
  - objects 105
  - open option 108
- Performance Monitor 162
- port 27
- port for Web admin 149
- public script 155
- queue manager 148, 152
  - checkbox 57
- queue manager required 147
- routing 17
- trigger queue 139
- Default Configuration 38, 39
  - objects created 39
  - will fail 39
- DEFBIND 108
- define
  - CLUSRCVR 27
  - CLUSSDR 27
  - cluster channels 19, 130
  - connections between queue managers 77
  - dynamic channels 13
  - fewer resources 7
  - QLOCAL 28
  - queue instances 87
  - recovery options 139
  - recovery procedure 48
  - remote queue 7
  - what starts at boot time 47
- definitions reduced 24
- DEFTYPE 32
- delete cluster 90
- dependencies (installation) 34
- DestinationChosen 116
- DHCP 39
- disconnect
  - from queue manager 177
- diskette contents 235
- display
  - CLUSQMGR 30
  - queues 29
- Display Cluster Information (1) 31
- distributed messaging 13
- DNS 39, 46
- documentation 34
- domain name
  - default configuration 39
- dspmqaout 208
- dynamic channels 12, 13
- dynamic queues 209
- DynamicQName 209

## E

Edit Chart Line 162  
environment variables 203  
Event Services 4  
exception report 213  
expiry report 213  
explicit remote queues 7

## F

failure 10  
fastget 109, 110  
fastget.c  
    source code 227  
file transfer mechanism 167  
File Transfer Programs 167  
full repository 17

## G

getFile.c 167, 192  
getting messages 89  
graphical tools 3  
graphs 159

## H

help 33  
    HTML 34  
    Web Administration 151  
hhupd.exe 34  
hide system queues 81  
HTML Help 34  
http //hostname 8081 149

## I

inbound channels 127  
inconsistencies 145  
Information Center 33, 38  
inner script 155  
installation 34  
    verify 42  
interactive mode 45, 47  
Internet Explorer 34  
IP port 67

## J

Java 4  
    AWT upgrade 148

    programming interface 4  
Join Cluster Wizard 71

## L

linear logging 57  
listener 28, 127  
    comments 104  
    managed by MQExplorer 104  
    start 127  
load balancing 8  
local cluster queue 115  
local queue 14  
log on  
    Web Administration 149  
loop back address 26, 31, 97

## M

machine name 67  
manual startup 133  
MC73 55  
message  
    broker 1  
    browser 45, 82, 124  
    channel 13, 14  
    channel agent 13  
    queue size 3  
message descriptor 52  
Microsoft  
    Internet Explorer 124, 148  
    Management Console 33, 34, 124  
    V5 compiler 33  
    Virtual Machine 34  
MMC 33, 34, 47  
monitor queue depth 159  
MQ client 19  
MQCO\_DELETE 214  
MQCO\_DELETE\_PURGE 214  
mqfm\_defs.h 167, 173  
MQGET 15, 212  
mqm security group 148  
MQOO\_BIND\_AS\_Q\_DEF 108  
MQOO\_BIND\_NOT\_FIXED 15, 107, 108  
MQOO\_BIND\_ON\_OPEN 15, 108  
MQOPEN 14  
    bindings 108  
MQPUT 14  
MQPUT1 211  
MQS.INI 135

- MQSeries 1
  - API Exerciser 49
  - for AIX 2
  - for AS/400 2, 4
  - for HP-UX 2
  - for OS/2 Warp 2
  - for OS/390 2
  - for Sun Solaris 2
  - for Windows NT 2
  - manuals 38
  - Menu 35
  - objects 45, 124
  - processes 47, 124
  - security 203
  - software 1
  - Version 2.1 1
  - Version 5.1 1, 2
    - features 2
- MQSeries Explorer 38, 45, 123, 124
- MQSeries Family 1
- MQSeries First Steps 38
- MQSeries for Windows NT 33
- MQSeries Information Center 33
- MQSeries Integrator 1
- MQSeries Postcard 42
- MQSeries processes 233
- MQSeries Services 47, 123, 124
  - described 132
  - objects 48
- MQSeries Workflow 2
- MQWXP structure 116
- multiple NT threads 33
- multiple queues 10
- MVS/ESA 1, 24

## N

- naming problems 10
- Netscape Navigator 124, 148
- NON 213
- non-clustering 24
- novice programmers 49

## O

- OAM 203
- Object Access Manager 203
- object in use 128
- ObjectName 209
- open option 15

- OS/390 2
- outbound channels 127
- outer script 154

## P

- partial repository 17
- path length 33
- PCF 131
- PCF commands 47
- Performance Monitor 3, 33, 159
- permanent channels 13
- permanent dynamic queues 209
- PERMDYN 214
- platforms 2
- PON 213
- port 67
  - UNIX 63
- port numbers 63
- postcard 38, 42
  - animated 43
- pre-requisite software 34
- private data stores 124
- process flow 2
- properties 48
- public data stores 124
- public script 155
- Publish/Subscribe 3, 4
- put message options 52
- put messages 82
- put test message 82, 124
- putFile.c 167, 178
- putMsg.c 167, 174
- putting messages 89

## Q

- QM.INI 135
- QMGr.INI 36
- QMTYPE 32
- queue
  - exists 14, 17
  - instance 14
  - monitor depth 159
  - multiple instances 8, 10
- queue manager
  - check status 127
  - components 127
  - create 126
  - properties 135

- start 126
- Web administration 148
- queue name 39, 79
- Quick Tour 38

## R

- recommendations 145
- recovery 48, 142
- reduced administration 7
- refresh 55
- refresh button 55
- refresh repository 17
- REGEDT32 36
- regedt32 137
- registry 36, 48, 137, 160
- remote administration 145
- remote queue 14
- reply-to queue 209
- report messages 213
- repository 12, 13
  - queue 61, 105
  - queue manager 71
  - time 17
- restart
  - a service 142
  - after failure 47
- robustness 134
- round-robin 15, 108
- routing example 16
- runmqchi 128
- runmqslr 127
- RUNMQSC 26, 28, 124
  - and clusters 125
  - fails 127

## S

- scmmqm 47, 124, 125, 133
- script 45
  - file 124
  - management 154
  - statements 151
  - statements(Web Administration) 151
- security account manager 204
- security database 206
- Security Identifier 203
- send notification 144
- server connection channel
  - create 57

- Service Pack 3 34
- SETMQAUT 203
- setmqaut 208
- shared cluster queue 83
- show cluster 91
- show system objects 61, 81
- SID 203
- SNA 24
- snap-in 47, 125
- sophisticated scripts 124
- stanzas 35
- start
  - a cluster 92
  - channel initiator 128
  - listener 127
  - queue manager 28, 133, 144
  - queue manager using MQServices 134
- startup types 133
- stop a cluster 90
- stop Explorer 91
- strmqm 126, 144
- supermarket checkout 11
- SupportPac 55
- system objects 61
- SYSTEM.ADMIN.SVRCONN 39
- SYSTEM.CHANNEL.INITQ 39, 128
  - open 128
- SYSTEM.CLUSTER.COMMAND.QUEUE 105
- SYSTEM.CLUSTER.REPOSITORY.QUEUE 105
- SYSTEM.CLUSTER.TRANSMIT.QUEUE 17, 105
- SYSTEM.DEAD.LETTER.QUEUE 57
- SYSTEM.DEF.CLUSRCVR 105
- SYSTEM.DEF.CLUSSDR 105
- SYSTEM.DEFAULT.INITIATION.QUEUE 139
- SYSTEM.DEFAULT.MODEL.QUEUE 209

## T

- tabular reports 159
- target queue 14
- task bar tray 132
- TCP/IP 39, 46
  - loop back 26
- TEMPDYN 209
- temporary dynamic queues 209
- throughput 10
- trace 48
- track cluster queues 160
- transmission header 14

transmission queue 13  
trigger message 172  
Trigger Monitor 139

## **U**

update the registry 48  
user ID 203  
using control commands 144

## **V**

verify installation 42  
Version 5.1 33  
virtual memory 34  
Visual Basic 33

## **W**

Web Administration 48, 124, 132, 147  
    authority 148  
    custom install 34  
    enabling 148  
    log on 149  
    scripts 147  
    Server 34  
Web Browser 124  
Web server 124  
what's new 2  
Windows NT 147  
    user ID 33  
WLogger.c 118  
workgroup productivity 2  
working with clusters 89  
workload  
    balancing 15, 55  
    distribution example 109  
    exit 15, 108  
workload management 8  
    exit 116  
    exit routines 8  
workload partitioning 12

## **X**

xmit queue 13





---

# IBM Redbooks Evaluation

MQSeries Version 5.1 Administration and Programming Examples  
SG24-5849-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at <http://www.redbooks.ibm.com/>
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to [redbook@us.ibm.com](mailto:redbook@us.ibm.com)

Which of the following best describes you?

**Customer**    **Business Partner**    **Solution Developer**    **IBM employee**  
 **None of the above**

**Please rate your overall satisfaction** with this book using the scale:  
**(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)**

Overall Satisfaction \_\_\_\_\_

**Please answer the following questions:**

Was this redbook published in time for your needs?      Yes\_\_\_ No\_\_\_

If no, please explain:

---

---

---

---

What other redbooks would you like to see published?

---

---

---

**Comments/Suggestions:      (THANK YOU FOR YOUR FEEDBACK!)**

---

---

---

---

SG24-5849-00  
Printed in the U.S.A.

MQSeries Version 5.1 Administration and Programming Examples

SG24-5849-00

