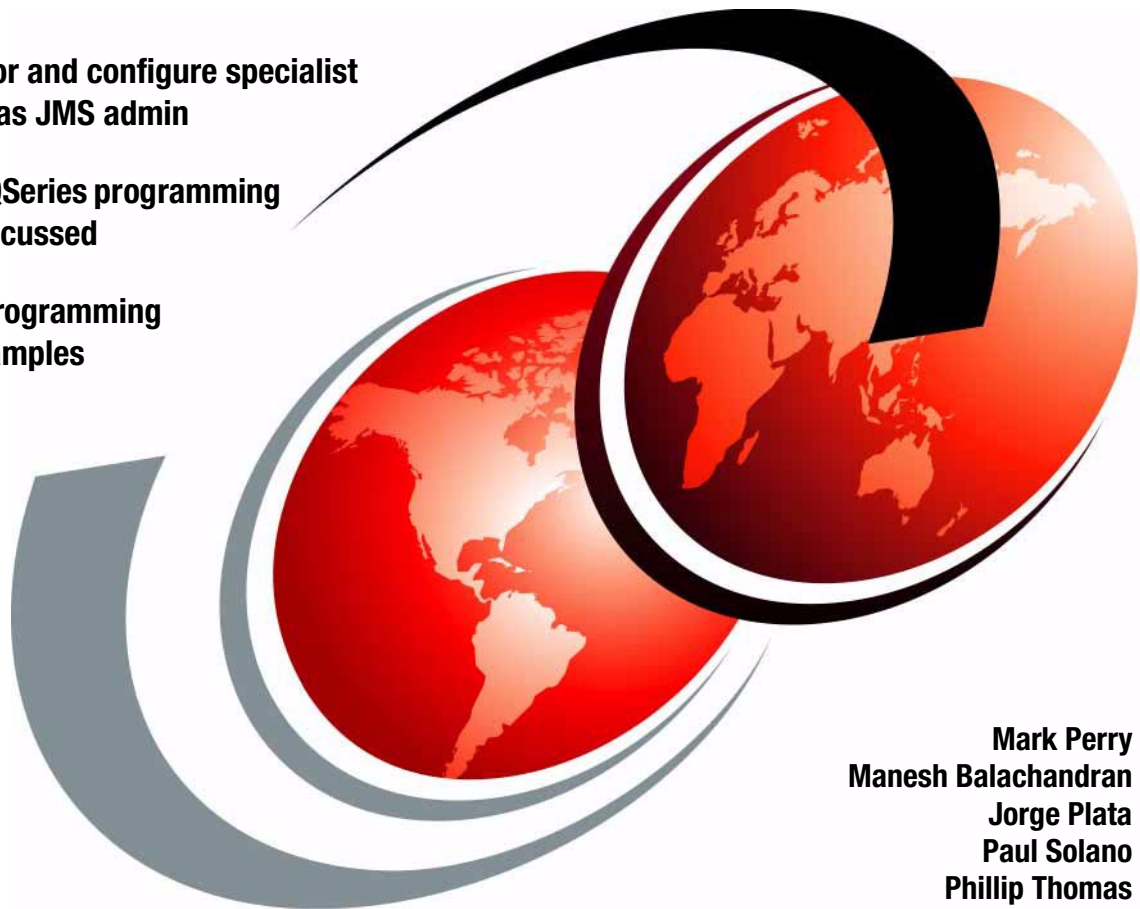


MQSeries Programming Patterns

Install, tailor and configure specialist tools such as JMS admin

Popular MQSeries programming choices discussed

Common programming pattern examples



Mark Perry
Manesh Balachandran
Jorge Plata
Paul Solano
Phillip Thomas



International Technical Support Organization

MQSeries Programming Patterns

April 2002

Take Note! Before using this information and the product it supports, be sure to read the general information in “Notices” on page ix.

First Edition (April 2002)

This edition applies to MQSeries V5.2 for the Windows, UNIX (including AIX, HP-UX, and Sun Solaris), OS/400, z/OS and OS/390 operating systems.

Comments may be addressed to:

International Business Machines Corporation, International Technical Support Organization
MP135
IBM UK Labs Ltd, Hursley
Hampshire, SO21 2JN United Kingdom

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 2002. 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

Notices	ix
Trademarks	x
Preface	xi
The team that wrote this redbook	xi
Special notice	xiii
Comments welcome	xiv
Part 1. Introduction	1
Chapter 1. Introduction and patterns	3
1.1 Programming with MQSeries	4
1.1.1 What are patterns?	4
1.1.2 One-to-one message flows	4
1.1.3 One-to-many message flows	5
1.1.4 Many-to-one message flows	6
1.1.5 Publish/subscribe	7
1.1.6 Request/reply	8
1.1.7 One-way datagram, or send-and-forget pattern	9
Chapter 2. Messaging and the APIs	11
2.1 Messaging, queuing and patterns	12
2.1.1 What is messaging?	12
2.1.2 What is queuing?	12
2.1.3 What is message queuing?	13
2.2 Transaction management	13
2.2.1 Single-phase and two-phase commits	14
2.2.2 XA specification	15
2.2.3 Transactions in MQSeries	15
2.3 Message grouping and segmentation	16
2.4 MQSeries programming interfaces	17
2.4.1 MQI	18
2.4.2 AMI	18
2.4.3 C++	19
2.4.4 MQSeries automation classes for ActiveX	19
2.4.5 Java	19
2.4.6 JMS	20
Part 2. The APIs	21

Chapter 3. Programming with MQI	23
3.1 Overview	24
3.2 Platforms and languages	25
3.3 Libraries and stub programs	26
3.4 Architectural model	28
3.5 Programming with MQI	30
3.5.1 Basic API concepts	32
3.5.2 Connecting to the queue manager	34
3.5.3 Opening MQSeries objects	35
3.5.4 Closing the MQSeries object	40
3.5.5 Disconnecting from the queue manager	40
3.5.6 Putting messages in a queue	41
3.5.7 Getting messages from a queue	45
3.5.8 Advanced topics	48
3.6 Transactions in MQI	55
3.7 Message grouping in MQI	56
3.8 Exploring the patterns	59
3.8.1 The one-to-one, or point-to-point pattern	59
3.8.2 The publish/subscribe pattern	71
Chapter 4. Programming with AMI	89
4.1 Overview	90
4.2 Platforms and languages	94
4.3 Libraries and packages	96
4.4 Architectural model	98
4.5 Programming with AMI	101
4.5.1 Connecting to the queue manager	101
4.5.2 Opening MQSeries objects	102
4.5.3 Basic operations	109
4.5.4 Deleting the session and closing the connection	114
4.6 How AMI compares to MQI	115
4.7 Transaction management	115
4.8 Grouping	117
4.9 Exploring the patterns	118
4.9.1 One-to-one or point-to-point	118
4.9.2 Publish/subscribe	124
Chapter 5. Programming with C++	133
5.1 Overview	134
5.1.1 Key features	134
5.2 Platforms and languages	134
5.3 Libraries	135
5.4 C++ architectural model	135

5.5	Programming with the C++ API	138
5.5.1	Connecting to the queue manager	138
5.5.2	Opening MQSeries objects	139
5.5.3	Closing MQSeries objects	141
5.5.4	Disconnecting from the queue manager	141
5.5.5	Putting messages on a queue	141
5.5.6	Getting messages from a queue	145
5.6	Advance topics	148
5.6.1	Browsing messages on a queue	148
5.6.2	Inquiring about and setting object attributes	149
5.7	Transaction management	151
5.8	Message grouping	152
5.9	Exploring the patterns	154
5.9.1	The one-to-one or point-to-point pattern	154
5.9.2	The publish/subscribe pattern	165
Chapter 6.	Programming with ActiveX	181
6.1	Overview	182
6.2	Platforms and languages	183
6.3	Libraries	184
6.4	Architectural model	184
6.5	Programming with MQSeries automation classes for ActiveX	185
6.5.1	Connecting to the queue manager	185
6.5.2	Opening MQSeries objects	186
6.5.3	Basic operations	189
6.5.4	Closing objects	193
6.5.5	Closing the connection	193
6.6	Transaction management	194
6.7	Grouping	197
6.8	Exploring the patterns	198
6.8.1	Send-and-forget	198
6.8.2	Request/reply	199
Chapter 7.	Programming with Java	203
7.1	Overview	204
7.2	Platforms	204
7.2.1	Obtaining the package	204
7.2.2	Running the MQSeries classes for Java	205
7.3	Using the MQSeries classes for Java	207
7.3.1	Connection modes	207
7.4	Working with MQSeries Java API	210
7.4.1	Setting up the connections	210
7.4.2	Interacting with queues	212

7.4.3 Working with MQSeries messages	213
7.5 Application development	215
7.5.1 Point-to-point pattern.	215
Chapter 8. Programming with JMS	235
8.1 What is JMS?	236
8.2 Overview	237
8.3 JMS messages	247
8.3.1 Mapping JMS messages onto MQSeries messages	247
8.3.2 JMS additional features.	252
8.4 MQSeries JMS implementation.	252
8.4.1 MQSeries JMS installation	252
8.4.2 JMS administered objects - JNDI and JMSAdmin	253
8.4.3 JMSAdmin tool	254
8.4.4 Invoking the administration tool.	254
8.4.5 JMSAdmin tool configuration	255
8.4.6 Using JMSAdmin with the Persistent Name Server	256
8.4.7 Using the Persistent Name Server with VisualAge for Java	256
8.4.8 Configuring VisualAge for Java for use with JMS	258
8.4.9 Administering JMS JNDI objects with VisualAge for Java using JMSAdmin	260
8.4.10 Defining JMS administered objects.	261
8.5 JMS application development	270
8.5.1 JMS point-to-point (PTP) model	270
8.5.2 Programming approach in point-to-point messaging	271
8.5.3 Send-and-forget	272
8.5.4 Request/reply	277
8.5.5 JMS publish/subscribe model	278
8.6 Asynchronous processing	285
8.6.1 Message listeners	285
8.6.2 Exception listeners	287
8.7 Message selectors	289
8.7.1 Working with message selectors.	292
Appendix A. Additional material	293
Locating the Web material	293
Using the Web material	293
System requirements for downloading the Web material	294
How to use the Web material	294
Related publications	295
IBM Redbooks	295
Other resources	295
Referenced Web sites	295

How to get IBM Redbooks 296

 IBM Redbooks collections..... 296

Abbreviations and acronyms 297

Index 299

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.



This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

e (logo)® 	IMS™	OS/400®	SP™
IBM ®	iSeries™	QMF™	SupportPac™
AIX®	MQSeries®	Redbooks™	Tivoli®
AS/400®	MVS™	Redbooks Logo	VisualAge®
C/400®	MVS/ESA™	 ™	WebSphere®
CICS®	Net.Commerce™	RETAIN®	z/OS™
DB2®	OS/2®	RPG/400®	
Encina®	OS/390®	S/390®	

The following terms are trademarks of International Business Machines Corporation and Lotus Development Corporation in the United States, other countries, or both:
Lotus®

The following terms are trademarks of other countries:

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

C-bus is a trademark of Corollary, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

SET, SET Secure Electronic Transaction, 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.

Preface

Today MQSeries offers the programmer more choices than ever in which to write new MQSeries applications, from the most traditional Message Queue Interface API all the way through to the popular and highly portable JMS interface.

Because of the many options available, it can sometimes be difficult for an application programmer new to MQSeries to easily understand the differences and benefits of each, or appreciate the implications of using one programming approach versus another.

Of course, all the information needed to make these decisions is available in the broad selection of manuals provided with the product and available separately with SupportPacs. However, the intention of this redbook is to bring together relevant parts of this information into one place and to describe each of the programming choices in a way that is intended to help guide the MQSeries programmer into making the best choices for particular situations.

This redbook will help you install, tailor and configure specialist tools such as JMS admin, and will help you to design/create MQSeries applications. It gives a broad and general understanding of the currently available MQSeries APIs.

We do this first by describing some of the more common examples and coding patterns, and then explaining each one in turn using the different APIs MQSeries supports to show the merits of each particular programming choice.

Because this book has been written by a team of MQSeries customers and business partners who have had to make all these decisions and choices in the past, they have attempted to illustrate and explain many of the easier routes the programmer can take based on their experience.

This redbook positions the different MQSeries programming choices against each other in such a way as to help the application writer to make a clearer and more informed judgment as to which is the most suitable programming method for a particular situation.

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, Hursley Center.



The authors - Manesh, Mark, Phillip, Paul, and Jorge

Mark Perry is an IT Specialist at the International Technical Support Organization, Hursley Center. He joined IBM in 1977 and was an MVS system and application programmer until joining MQSeries in 1991. He has 10 years of experience within the MQSeries community at Hursley, mostly as a team leader in System Test, as well as spending some time working for MQSeries Level 2 Support at IBM Raleigh, NC, USA, and IFS at IBM Poughkeepsie, NY, USA. Mark has previously worked on other MQSeries-related redbooks.

Manesh Balachandran is a senior System Analyst working as an independent consultant in the United States. He has over six years of experience in programming, data management and system integration. He has worked extensively on IBM technologies and holds IBM Solutions Expert Certification in DB2/UDB, IBM Solutions Expert Certification in Net.Commerce and IBM Specialist certification in MQSeries.

Jorge Plata is an IT Specialist working for Technology Enablers, Inc., an IBM business partner in Dallas. He has four years of experience in middleware. He holds a degree in Computer Science from the Universidad Iberoamericana in Mexico. His areas of expertise includes the development and management of MQSeries and related products.

Paul Solano is an AIM Specialist working for GBM de Costa Rica and IBM Alliance company located in San José, Costa Rica. He has four years experience in Web-based application development and integration. He holds a degree in Computer Science from the Instituto Tecnológico de Costa Rica. His areas of expertise includes the WebSphere family of products and MQSeries application design and development.

Phillip Thomas is a Lecturer in the Computer Information Systems Department at California State University, Los Angeles. He has been working with MQSeries since 1996. His areas of interest include object-oriented languages, distributed systems and databases architecture.

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

Matt Lucas, Alex Russell, Craig Newman, Andy Hickson, Jeremy Weaving,
Steve Hall
WebSphere MQ, IBM Hursley, UK

Kavitha Sakthirajan, Nikhilesh Coss
IBM India

Ian Parkinson, Lee Hollingdale
IBM Hursley, UK

Federico Demi
Primeur Italia Srl, Pisa, Italy

Special notice

This publication is intended to help MQSeries application programmers to make informed programming choices when writing new applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by MQSeries. See the PUBLICATIONS section of the IBM Programming Announcement for MQSeries for more information about what publications are considered to be product documentation.

Comments welcome

Your comments are important to us!

We want our IBM Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

ibm.com/redbooks

- Send your comments in an Internet note to:

redbook@us.ibm.com

- Mail your comments to the address on page ii.



Part 1

Introduction

In the first part of this redbook, we discuss the typical programming flow patterns that a programmer would use and the choices that the application programmer is faced with when deciding upon the programming route to take. We provide brief overviews of the language choices for the programming models, and we attempt to make the process of choosing the right language easier, before we move on to the real detail in Part 2.



Introduction and patterns

In this chapter we will discuss some of the common MQSeries programming flow patterns. We will introduce one-to-one, one-to-many, many-to-one, request/reply, send-and-forget, and publish/subscribe patterns.

It is essential at this point that you have read and fully understand the MQSeries book, *An Introduction to Messaging and Queuing*, GC33-0805, which fully discusses all the MQSeries fundamentals we will be referring to throughout this book.

You may also find it valuable to refer to the following MQSeries books:

Application Programming Reference, SC33-1673

Application Programming Guide, SC33-0807

Application Programming Reference Summary, SX33-6095

All of these books can be downloaded from:

<http://www-3.ibm.com/software/ts/mqseries/library/manuals/>

1.1 Programming with MQSeries

MQSeries applications can be developed using a variety of programming languages and styles. Procedural and object-oriented programming can be performed using Visual Basic, C, C++, Java, and COBOL. Microsoft Windows NT ActiveX/COM technology is also supported.

No matter how large or complex an MQSeries application is, it always has to perform certain common operations just the same as any other MQSeries application. These are the building blocks of any messaging application and at its simplest messages have to be constructed, put onto queues, and be consumed from the queues.

Of course, there will always need to be special code to check for errors and exceptions as well as the code needed to interpret the messages and take the appropriate actions based on the message content. Described below are some of the most common flow patterns that can be used in MQSeries applications, and in the subsequent chapters we show examples of how these can be coded using the different APIs.

1.1.1 What are patterns?

Patterns are programming techniques that are used to address recurring design problems that arise in specific design situations. In general, patterns attempt to present solutions to these recurring problems based upon the experience of those who have come across them many times in the past. This book looks at the request/reply and publish/subscribe patterns within the context of the MQSeries message-oriented middleware product.

1.1.2 One-to-one message flows

A one-to-one or point-to-point application (see Figure 1-1) is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and a second program gets messages from that queue.

Queues retain all the messages sent to them until the messages are either consumed or until the messages expire.

One-to-one messaging has the following characteristics:

- ▶ Each message has only one consumer.
- ▶ There are no timing dependencies between a sender and a consumer of a message. The consumer can get the message whether or not it was running when the sender put the messages onto the queue.

- ▶ The message consumer can, if required, acknowledge the successful processing of a message by sending a reply back to the sender.
- ▶ One-to-one messaging can be used when every message you put on a queue is to be processed successfully by one consumer.

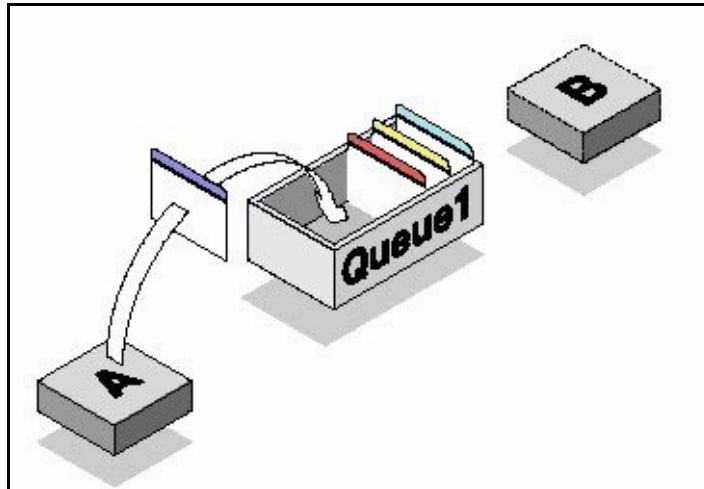


Figure 1-1 One-to-one message flow

A one-to-one message flow that does not require a reply is often referred to as a send-and-forget pattern.

1.1.3 One-to-many message flows

A one-to-many message flow (see Figure 1-2) differs from the one-to-one flow in that the messages are put onto a queue as before but can be read from that queue by multiple processes. This might be for the purposes of load balancing, or simply that the message information needs to be used in different ways by each consuming or browsing process. Again queues retain all the messages sent to them until the messages are either consumed or until the messages expire.

One-to-many messaging has the following characteristics:

- ▶ Each message has more than one possible application that will read or consume it.
- ▶ There are no timing dependencies between a sender and consumers of a message. The consumers can get the messages whether or not they were running when the sender put the messages onto the queue.

- The message consumers can, if required, acknowledge the successful processing of a message by sending a reply back to the sender.

One-to-many messaging can be used when every message you put on a queue is to be processed successfully by multiple processes or consumers.

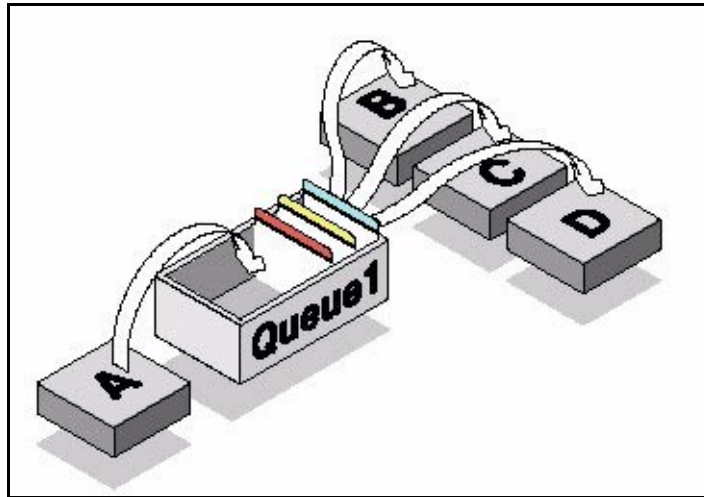


Figure 1-2 One-to-many message flow

1.1.4 Many-to-one message flows

A many-to-one application (see Figure 1-3) can be built around the concept of message queues, senders, and receivers. Messages from multiple processes are addressed to a single specific queue, and a further program gets messages from that queue.

Queues retain all the messages sent to them until the messages are either consumed or until the messages expire.

Many-to-one messaging has the following characteristics:

- There are no timing dependencies between the senders and a consumer of a message. The consumer can get the message whether or not it was running when the sender put the messages onto the queue.
- The message consumer can, if necessary, acknowledge the successful processing of a message by sending a reply back to the sender.

Many-to-one messaging can be used when messages you put on a simple queue from multiple processes are to be processed successfully by one consumer.

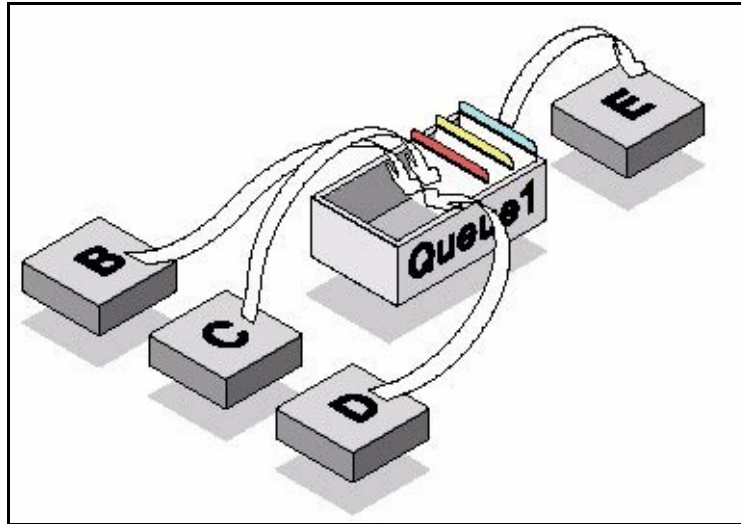


Figure 1-3 Many-to-one message flow

1.1.5 Publish/subscribe

The publish/subscribe messaging pattern is built around the concept of topics. In the publish/subscribe pattern, message-producing (sending applications) applications broadcast messages by publishing to a topic. The message-consuming (receiving) applications subscribe to topics to receive messages published to those topics. Topics follow a tree hierarchy structure.

Figure 1-4 illustrates a simple topic hierarchy with the root level topic being “NEWS”. The topic NEWS contains three child topics, called “Current Affairs”, “Sports” and “Finance”. The topic Sports in turn has two topics under it, namely “Baseball” and “Football”. The topic Finance has one subtopic, called “Stocks”. In the publish/subscribe model, the publishers and subscribers can be decoupled.

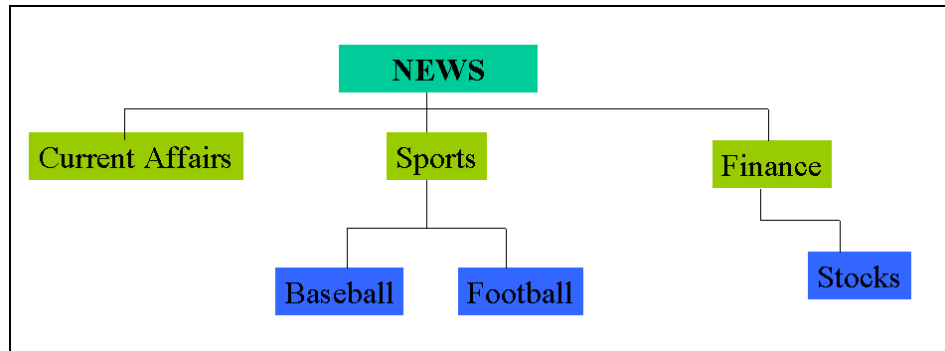


Figure 1-4 Publish/subscribe hierarchy

Any application can publish messages to a particular topic; for example, an application publishing the result of a baseball game to the topic called “Baseball”. Any application in turn can retrieve messages from a topic by subscribing to that topic. In publish/subscribe models, publishers and subscribers can be added dynamically, thus allowing the system to grow or shrink dynamically. Every application that subscribes to a topic gets its own copy of the messages published to that topic. The publish/subscribe model is facilitated by a software component called message broker. The message broker maintains the subscriptions to the topic hierarchy.

1.1.6 Request/reply

The request/reply pattern involves two processes: one consists of sending a message and expecting a reply (in other words sending a request message), and the other sends reply messages upon receiving a request message. The flow of messages is initiated by the system or application when it sends out the request message and then waits for the reply message. The responding side consumes the request message, produces a reply message, and sends it back to the initiating side. The flow is completed when the initiating side receives the reply message.

In the request/reply model, there is a closer coupling of the systems, which requires additional application logic to correlate between the request and reply messages, and additional logic on the requesting side to handle delay and failure in receiving a reply message. The request/reply pattern can be used with either one-to-one (point-to-point) or publish/subscribe patterns. To facilitate the request/reply pattern, MQSeries provides a facility to identify a request message and its reply message with the use of a correlation ID. The correlation ID is set by the application sending the request message. The application generating the reply message copies the correlation ID from the request message on to the reply message sent back to the application which originally sent the request

message. The application that sent the request message can use the correlation ID to map the reply message to the request message it sent earlier. Additionally, you can also effectively use the temporary queues or temporary topics with the request/reply pattern.

1.1.7 One-way datagram, or send-and-forget pattern

In this pattern the sending application sends messages without expecting any reply from the receiving application. With this pattern, the sending application is fully decoupled from the receiving application, allowing for a totally asynchronous relationship between the sending system and the receiving system. The sending system can continue processing without being hindered by the availability of the receiving system.

With the assured delivery capability of MQSeries messaging, the sending system can be sure that the messages would be delivered to the receiving system. In adopting this pattern, business rules of failure of the receiving application should be kept in mind. The one-way datagram or send-and-forget pattern can be used with both point-to-point and publish/subscribe patterns. The send-and-forget pattern is implemented in the one-to-one model by putting messages on a specific queue. In the case of the publish/subscribe pattern, the send-and-forget pattern is implemented by publishing messages to a specific topic.

Now that we have discussed these common message flow patterns, we will move on to discuss the many programming choices available and how each one relates to these patterns. We show how these various options are implemented, along with example code wherever appropriate.

Although we cover some publish/subscribe patterns in each of the following chapters, it is highly recommended that you follow this up by reading *MQSeries Publish/Subscribe Applications*, SG24-6282, which can be downloaded from:

<http://www.ibm.com/redbooks>

This book discusses and fully documents all the steps involved in configuring the programming tools and coding a real publish/subscribe application in detail.



Messaging and the APIs

In this chapter we discuss in general terms the programming choices that the application programmer faces when deciding upon the most suitable programming route to take. From brief overviews of the language choices, to the programming models, we attempt to make the process of choosing the right language easier and more straightforward.

We introduce the different programming interfaces supported by MQSeries for applications development, transaction and system management.

2.1 Messaging, queuing and patterns

Much has been written about MQSeries and MQSeries programming over the period since its introduction. Those documents provide the basis for some excellent background reading for anyone using this book. In fact as a prerequisite you should, as previously mentioned, read and make sure you fully understand the book *An Introduction to Messaging and Queuing*, GC33-0805, which is supplied with the MQSeries product. However, in this chapter we briefly cover the basic concepts of what messaging is and how it fits together.

2.1.1 What is messaging?

Messaging is when systems communicate with each other using messages to convey the information rather than communicating directly through the transport mechanisms.

MQSeries defines four types of messages that can be used:

Datagram	A simple message for which no reply is expected
Request	A message for which a reply is expected
Reply	A reply to a request message
Report	A message that describes an event such as the occurrence of an error

2.1.2 What is queuing?

Queuing is the mechanism by which messages are held until an application is ready to process them. Queuing allows you to:

- ▶ Communicate between programs (which may each be running in different environments) without having to write the communication code.
- ▶ Select the order in which a program processes messages.
- ▶ Balance loads on a system by arranging for more than one program to service a queue when the number of messages exceeds a threshold.
- ▶ Increase the availability of your applications by arranging for an alternative system to service the queues if your primary system is unavailable.

2.1.3 What is message queuing?

Message queuing has been used in data processing for many years and is most commonly used today in electronic mail. Without queuing, sending an electronic message over long distances would require every node on the route to be constantly available for forwarding messages, with the addressees to be logged on and aware of the fact that you are trying to send them a message.

In a queuing system, messages are stored at intermediate nodes until the system is ready to forward them. At their final destination, they are stored in an electronic mailbox until the addressee is ready to read them. Even so, many complex business transactions are processed today without queuing. In a large network, the system might be maintaining many thousands of connections in a ready-to-use state. If one part of the system suffers a problem, many parts of the system become unusable.

Message queuing can be thought of as being electronic mail for programs. In a message-queuing environment, each program from the set that makes up an application suite is designed to perform a well-defined, self-contained function in response to a specific request. To communicate with another program, a program must put a message on a predefined queue. The other program retrieves the message from the queue, and processes the requests and information contained in the message. So message queuing is a style of program-to-program communication.

2.2 Transaction management

In a human context, a transaction is an action, or group of actions, that takes place between two people. In a computer context, transactions relate to a group of activities that may need to access multiple resources and perform some kind of operation and updates on them. These sets of activities must be completed together so that if any of them were to fail, then the whole set would need to be undone or backed out.

Transactions have four main properties, called the ACID properties. ACID stands for *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

Atomicity A transaction must execute completely or not at all. All the activities in the transaction must be completed successfully. If any fails to complete, then the transaction is aborted and all the activities already executed are rolled back. If all the activities complete successfully, then the transaction is complete and all the operations and updates to the resources are committed.

Consistency	Once a transaction has been executed, whether if it has been completed successfully or not, the underlying resources used during the transaction must be consistent.
Isolation	A transaction must be able to execute without the interference of other processes or transactions. Any intermediate states are transparent to other transactions, so multiple transactions can be executed serially.
Durability	Any changes made to the underlying resources during a committed transaction must be stored in a persistent data store and should survive software or hardware failures.

In the simplest case, the whole transaction is committed or backed out once it is finished, but in many cases you might want to synchronize data changes at other points within the transaction. Those points are called *syncpoints*, and the period of processing between two syncpoints is called the *unit of work*. Units of work can contain one or many different operations.

Syncpoint coordination is the process by which units of work are either committed or backed out, thereby maintaining data integrity. This gives the programmer greater control over the transaction, making it much easier during any problem-solving process.

2.2.1 Single-phase and two-phase commits

Transactions can be local to a single specific resource manager or they can involve multiple resource managers. In the latter case, a transaction manager is needed to coordinate the transaction in the different resource managers.

Within a local transaction scope, a *single-phase commit* can be applied. A single-phase commit process is one in which the program can commit changes without coordinating its changes with other resource managers.

On the other hand, if changes made to one resource need to be coordinated with updates to other resource managers, for example with a relational database such as DB2, we want all the resources to be committed or backed out together. This is known as a *two-phase commit*.

When a two-phase commit is needed, the transaction manager must be able to coordinate the transaction with the different resource managers. In order to do that, the resource manager must have a common transaction interface. The XA specification defines a set of functions that allows the coordination between different resource managers.

The two-phase commit protocol operates in two different phases to determine whether the transaction should be committed or rolled back.

In the first phase, called the prepare phase, the transaction manager evaluates the status of each of the resource managers involved in the transaction. All the resource managers must be ready to commit the transaction before the transaction manager can continue with the next phase.

Once the transaction managers have received an answer from all the transaction managers, the second phase or commit phase concludes the transaction. Here the transaction manager instructs the resource managers to commit the transaction if all have agreed, or to roll back if at least one disagrees.

Rollback is where the transaction manager has been unsure about the state of any one of the participating resources. The transaction would have been in an in-doubt state, and so any changes that may have been in the process of being made need to be resolved by returning them to their original status.

2.2.2 XA specification

The XA specification is a portion of the XA Distributed Transaction Processing model of the Open Group organization. The XA interface is a bidirectional interface, which consist of a set of UNIX-type APIs.

Along with the functions it provides, the XA specification provides a switch data structure that contains the resource manager name, non-null pointers to the resource manager's APIs, a flag, and a version word. This way, programs using the resource manager APIs access the function through the pointers provided in this structure rather than using the actual function names, or having to link to the service program that actually contains the functions. This gives an additional level of portability, since the resource manager can be changed without the programmer having to recompile the application.

For more detailed information about this API, please refer to the X/Open publications available. Information can be found at:

<http://www.opengroup.org>

2.2.3 Transactions in MQSeries

In the case of a messaging system such as MQSeries, transaction management allows the programmer to assure that a set of messages have been sent in an all-or-nothing way: if there is a problem delivering any of the messages in a transaction, the whole set of messages can be rolled back and some kind of corrective action can be taken. If messages were retrieved from a queue during that unit of work, those messages are restored to the queue by MQSeries when the transaction is rolled back.

Within a transaction, if messages are put onto a queue, they are only visible to other applications after the whole transaction has been committed. Likewise, if within a transaction messages are destructively read from a queue, they are not be deleted from the queue until the transaction is actually committed so that no other program would be able to retrieve them in the meantime.

Since messages in a unit of work are returned to the queue when a backout occurs, the program processing those messages can get into an infinite loop, processing the same unit of work over and over again and simply finding the same problem every time.

To avoid this situation, MQSeries keeps track of the number of times it has been backed out in the `BackoutCount` attribute. This way if the message repeatedly causes the unit of work to fail, this attribute value will eventually exceed the value in the `BackoutThreshold`, which is set when the queue is defined. The application can decide to remove the message from the unit of work and put it onto another queue, or take some other corrective action so the unit of work can commit as planned.

We can use single-phase or two-phase commit in MQSeries depending on the requirements of the application, which means that MQSeries queuing functions can be brought within the scope of a unit of work, managed by MQSeries or other transaction managers such as CICS, IMS, Encina, Tuxedo and Top End.

If the transaction coordination is going to be made by MQSeries, units of work can either be local to the MQSeries queue manager or global, involving other resource managers such as relational databases.

2.3 Message grouping and segmentation

Message grouping is a facility that allows the programmer to add some grouping logic to the messages that are being sent through a queue without having to introduce that logic in the messages themselves. This way the application processing those messages can concentrate on the actual data to be processed and leave the group management to MQSeries.

There are two main reason for using grouping:

- ▶ The messages in the given group have to be processed in the correct order.
- ▶ Each of the messages in a group need to be processed in a related way.

Logical order means that the messages belonging to a group are read in their correct order even if some other message or messages arrive onto the queue in the middle of the group. That, of course, requires the program to identify the message group it is going to be working with, which is done by giving the MQGET call the message group identification in the MQMO structure.

MQSeries also allows the program to require the whole group to be available before a get operation can be done on the first message of the queue (the first message of the queue must be present to be able to browse the group at all). By doing this, the program developer can concentrate on the business logic and leave the group ordering handling to the queue manager.

On the other hand, even if group ordering is not an issue, the group identification can be helpful in terms of keeping together a conceptual group of information that your system uses, so messages in a given group can be given a specific treatment later on.

MQSeries provides another level of message grouping called segmentation. The purpose of segmentation is to give the programmer the opportunity to split a message into multiple segments. This is mostly used when messages are larger than the maximum configurable size of a message either in MQSeries configuration or in the program itself.

With segmentation, large messages can be split into smaller messages so that they can be read from the queue into smaller buffers. This gives the programmer the opportunity to work with more manageable pieces of information before the whole message has arrived. If the whole message has to be received before some work can be performed on it, you can use a transactional model so that the message can be committed after all the parts are available on the queue. The same thing can be done with grouping. A segment of a message is identified by the GroupID, MsgSeqNumber, and Offset.

2.4 MQSeries programming interfaces

We now move on to give a brief overview of the key features of the many programming choices that an MQSeries application programmer can select from. Each one of these choices are described in much more detail in subsequent chapters.

2.4.1 MQI

The MQSeries Message Queue Interface (MQI) is an extremely rich programming interface that gives the programmer access to all the facilities of the MQSeries messaging platform along with detailed control over the messages and the way they behave.

The MQI calls allow you to:

- ▶ Connect programs to, and disconnect programs from, a queue manager
- ▶ Open and close objects such as queues, queue managers, namelists, and processes
- ▶ Put messages on queues
- ▶ Get (remove) messages from a queue, or browse them (leaving them on the queue)
- ▶ Inquire about the attributes of MQSeries objects, and set some of the attributes of queues
- ▶ Commit and back out changes made within a unit of work, in environments where there is no natural syncpoint support, for example OS/2 and UNIX systems
- ▶ Coordinate queue manager updates and updates made by other resource managers

The MQI API provides data structures and types to be used as the input and output of the calls. It also provides a large set of named constants that can be used to modify the options given in those data structures. The MQI is consistent across all the supported platforms so the applications can be moved to a different platform without code modifications.

2.4.2 AMI

The Application Message Interface (AMI) provides programmers with a very simple interface that can be used to work with queue manager objects. With the AMI, the programmer doesn't need to have in-depth knowledge of all the MQI calls, but instead can concentrate on the business logic of the application. This means fewer programming errors and more flexibility to address business and technology changes. AMI reduces the amount of code that is required to write a new application

The AMI's key features are:

- ▶ Reduced (application) administration overhead
- ▶ Function moved from application to AMI
- ▶ Support for different application programming models

- ▶ Cross-platform and multiple language (object-oriented and procedural) support
- ▶ Simple API calls

2.4.3 C++

The C++ MQSeries interface is an extension of the MQI API. It gives the programmer an object-oriented approach to the messaging interface in MQSeries.

The C++ MQI provides all the features available in the MQI API, such as getting, putting and browsing messages. It also allows you to inquire and set object options. Additionally, it provides the following features:

- ▶ Automatic initialization of MQSeries data structures
- ▶ Just-in-time queue manager connection and queue opening
- ▶ Implicit queue closure and queue manager disconnection
- ▶ Dead-letter header transmission and receipt
- ▶ IMS Bridge header transmission and receipt
- ▶ Reference message header transmission and receipt
- ▶ Trigger message receipt
- ▶ CICS Bridge header transmission and receipt
- ▶ Work header transmission and receipt
- ▶ Client channel definition

2.4.4 MQSeries automation classes for ActiveX

The MQSeries automation classes for ActiveX is a set of ActiveX components that provide classes. The MQSeries automation classes for ActiveX are intended to be used by designers and programmers who want to develop MQSeries applications that are able to run on the Windows platform. The classes can then be easily integrated into any application, because the MQSeries objects that are needed can be coded using the native syntax of the implementation language. The overall design of the application is the same as for any MQSeries application.

2.4.5 Java

The MQSeries classes for Java (MQSeries base Java) allows a programmer to write an application or applet in the Java programming language to:

- ▶ Connect to MQSeries as an MQSeries client
- ▶ Connect directly to an MQSeries server

MQSeries base Java enables Java applets, applications, and servlets to issue calls and queries to MQSeries. The programmer can choose to connect directly to an MQSeries server or to connect to MQSeries as an MQSeries client. This gives access to mainframe and legacy applications, typically over the Internet, without necessarily having any other MQSeries code on the client machine.

In addition, applications written using the MQSeries classes for Java may realize a significant cost savings in an environment where a large number of desktops need to be periodically updated with the latest version of the software. This is because once the byte code is initially downloaded onto the user's machine, the latest version of the software is automatically downloaded each time the software is subsequently used.

2.4.6 JMS

The Java Message Service or JMS is the standard API for messaging in the same way that the JDBC API is for databases. The JMS Specification (1.0.2) was developed by Sun Microsystems with the active involvement of IBM and other enterprise messaging vendors, transaction processing vendors and RDBMS vendors. JMS provides a common model for Java programs to interact with messaging systems performing various operations against the messaging systems objects. The common operations that a program uses against a messaging systems's object are create messages, send messages, receive messages, and read messages from the enterprise messaging system. JMS provides a common way for programs being developed in Java to access these messaging systems.

Now that we have covered the various choices from a high level, we move on to discuss each one in turn, along with real programming examples.



Part 2

The APIs

Having discussed and explained in overall terms what programming choices are available and what the programming models are, we now show how these various options are implemented, providing sample code wherever appropriate.



Programming with MQI

In this chapter, we discuss the Message Queue Interface (MQI) API. This is the basic rich programming interface provided with MQSeries for all its supported platforms and provides the most comprehensive set of operations.

We introduce some of the basic concepts associated with the MQSeries application development process in general, since this API is the source of all the other available APIs.

Later we explain how to perform MQSeries operations such as:

- ▶ Connecting and disconnecting from a queue manager
- ▶ Opening and closing queue objects
- ▶ Sending, browsing and getting messages
- ▶ Inquiring about and setting object attributes
- ▶ Transaction management
- ▶ Message grouping

Finally we explore the implementation of the programming patterns previously discussed and explained in Chapter 1, “Introduction and patterns” on page 3.

3.1 Overview

The Message Queue Interface or MQI is a programming interface that gives the programmer access to all the facilities of the MQSeries messaging platform and gives full and detailed control over the messages and the way they flow.

The MQI calls allow you to:

- ▶ Connect programs to, and disconnect programs from, a queue manager
- ▶ Open and close objects (such as queues, queue managers, namelists, and processes)
- ▶ Put messages on queues
- ▶ Retrieve messages from a queue, or browse them (leaving them on the queue)
- ▶ Inquire about the attributes of MQSeries objects, and set some of the attributes of the queues
- ▶ Commit and back out changes made within a unit of work, in environments where there is no natural syncpoint support, for example, OS/2 and UNIX systems
- ▶ Coordinate queue manager updates and updates made by other resource managers

The MQI provides a set of calls or functions to perform these operations, along with data structures and types to be used as the input and output of the calls. It also provides a large set of named constants that can be used to modify the options given in those data structures. Data structures can also be initialized with the default values provided by the API in the form of named constants.

The MQI is consistent across all the supported platforms so the applications can be moved to a different platform without code modifications, although this might require the programmer to add some logic to assure portability. For example:

```
#ifdef _OS2
    OS/2 specific code
#else
    generic code
#endif
```

MQI programs can run in a server or client environment. Some restrictions apply when the program is to be used in a client environment, since the connection to the queue manager cannot be in binding mode.

Additionally, some environment variables are required in a client configuration so the application can find the MQSeries server. Please refer to *MQSeries Clients*, GC33-1632, for detailed information about how to set up a client environment.

3.2 Platforms and languages

These are the platforms and languages supported by MQI:

- ▶ For MQSeries for MVS/ESA:
 - COBOL
 - Assembler language
 - C
 - PL/I
- ▶ For MQSeries for AS/400:
 - RPG
 - COBOL
 - C
 - C++
- ▶ For MQSeries for AT&T GIS UNIX, MQSeries for Digital OpenVMS, MQSeries for SINIX and DC/OSx, and MQSeries for SunOS:
 - COBOL
 - C
- ▶ For MQSeries for HP-UX and MQSeries for Sun Solaris:
 - COBOL
 - C
 - C++
- ▶ For MQSeries for AIX, MQSeries for OS/2 Warp, and MQSeries for Windows NT:
 - COBOL
 - C
 - C++
 - PL/I
- ▶ For MQSeries for Tandem NonStop Kernel:
 - COBOL
 - C
 - TAL
- ▶ For MQSeries for Windows:
 - C
 - Visual Basic

3.3 Libraries and stub programs

These are the libraries and stub programs supported by MQI.

MQSeries for AS/400

In MQSeries for AS/400, you must bind your ILE C/400 programs and RPG/400 static calls to the supplied AMQZSTUB service program.

MQSeries for Windows

In MQSeries for Windows, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

- ▶ MQM16.LIB server for 16-bit C
- ▶ MQM.LIB server for 32-bit C
- ▶ MQM16.LIB server for 16-bit Visual Basic
- ▶ MQMSTD.LIB server for 32-bit Visual Basic

MQSeries for Windows NT

In MQSeries for Windows NT, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system:

- ▶ MQM.LIB server for 32-bit C
- ▶ MQIC.LIB client for 16-bit C
- ▶ MQIC32.LIB client for 32-bit C
- ▶ MQMXA.LIB static XA interface for C
- ▶ MQMCICS.LIB CICS for Windows NT V2 exits for C
- ▶ MQMCICS4.LIB Transaction Server for Windows NT, V4 exits for C
- ▶ MQMZFLIB installable services exits for C
- ▶ MQMCBB.LIB server for 32-bit IBM COBOL
- ▶ MQMCB32 server for 32-bit Micro Focus COBOL
- ▶ MQICCB.LIB client for 32-bit IBM COBOL
- ▶ MQICCB32 client for 32-bit Micro Focus COBOL
- ▶ QMENC.LIB dynamic XA interface in C for Encina
- ▶ MQMTUX.LIB dynamic XA interface in C for Tuxedo

MQSeries for AIX

In MQSeries for AIX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application, these are as follows:

- ▶ libmqm.a server for C

- ▶ libmqic.a client for C
- ▶ libmqmzf.a installable service exits for C
- ▶ libmqmxa.a XA interface for C
- ▶ libmqmcbrt.o MQSeries runtime library for Micro Focus COBOL support
- ▶ libmqmcb.a server for COBOL
- ▶ libmqicb.a client for COBOL

In a threaded application, these are as follows:

- ▶ libmqm_r.a server for C
- ▶ libmqmzf_r.a installable service exits for C
- ▶ libmqmxa_r.a XA interface for C
- ▶ libmqmxa_r.a for Encina

MQSeries for HP-UX

In MQSeries for HP-UX, you must link your program to the MQI library files supplied for the environment in which you are running your application, in addition to those provided by the operating system.

In a non-threaded application, these are as follows:

- ▶ libmqm.sl server for C
- ▶ libmqic.sl client for C
- ▶ libmqmzf.sl installable service exits for C
- ▶ libmqmxa.sl XA interface for C
- ▶ libmqmcbrt.o MQSeries runtime library for Micro Focus COBOL support
- ▶ libmqmcb.sl server for COBOL
- ▶ libmqicb.sl client for COBOL

In a threaded application, these are as follows:

- ▶ libmqm_r.sl server for C
- ▶ libmqmzf_r.sl installable service exits for C
- ▶ libmqmxa_r.sl XA interface for C

MQSeries for Sun Solaris

In MQSeries for Sun Solaris, you must link your program to the MQI library files supplied for the environment in which you are running your application in addition to those provided by the operating system. These are as follows:

- ▶ libmqm.so server for C
- ▶ libmqmzse.so for C
- ▶ libmqic.so client for C
- ▶ libmqmcs.so client for C
- ▶ libmqmzf.so installable service exits for C
- ▶ libmqmxa.a XA interface for C

3.4 Architectural model

MQI architecture is a simple, straightforward implementation of the different features available in MQSeries.

The different calls directly refer to a basic MQSeries operation such as getting messages from a queue or putting messages to a queue. These calls can be grouped based on their functionality as follows:

- ▶ Connecting to and disconnecting from a queue manager:
MQCONN, MQCONNX, and MQDISC
- ▶ Open and close an MQSeries object, such as a queue:
MQOPEN and MQCLOSE
- ▶ Put one or multiple messages on a queue:
MQPUT and MQPUT1
- ▶ Browse or remove messages from a queue:
MQGET
- ▶ Inquire about the attributes of an object:
MQINQ
- ▶ Set some of the queue attributes at runtime:
MQSET
- ▶ Manage local or distributed transactions:
MQBEGIN, MQCMIT, and MQBACK

The different options and basic information required to perform these operations can be supplied using the data structures and elementary data types provided by the API.

These are the MQI data structures:

- ▶ MQBO (Begin options)
Specifies options for the MQBEGIN call (MQSeries Version 5 products only).
- ▶ MQCNO (Connect options)
Specifies options for the MQCONNX call (MQSeries Version 5 products only).
- ▶ MQDH (Distribution header)
Describes the data that is present in a message on a transmission queue when that message is a distribution-list message (MQSeries Version 5 products and MQSeries for AS/400 only).

- ▶ MQGMO (Get message options)
Specifies options for the MQGET call.
- ▶ MQMD (Message descriptor)
Provides control information for a message you are putting on (using MQPUT or MQPUT1) or getting from (using MQGET) a queue.
- ▶ MQMDE (Message descriptor extension)
In conjunction with an MQMD Version 1, this contains grouped message and segmentation information that would normally be held in the MQMD Version 2 (MQSeries Version 5 products and MQSeries for AS/400 only).
- ▶ MQOD (Object descriptor)
Identifies the object you want to work with when using MQOPEN.
- ▶ MQOR (Object record)
Identifies the destinations you want to work with in a distribution list (MQSeries Version 5 products and MQSeries for AS/400 V4R2 only).
- ▶ MQPMO (Put message options)
Specifies options for the MQPUT and MQPUT1 calls.
- ▶ MQPMR (Put-message record)
Contains specific information relating to the individual destinations included in a distribution list (MQSeries Version 5 products and MQSeries for AS/400 V4R2 only).

The following structures are used for special purposes:

- ▶ MQDLH (Dead-letter header)
Defines the format of the header of messages put on the dead-letter (undelivered-message) queue (not supported on MQSeries for Windows V2.0).
- ▶ MQRMH (Reference message header)
Defines the format of a reference message (MQSeries Version 5 products and MQSeries for AS/400 only).
- ▶ MQTM (Trigger message)
Defines the format of a trigger message.
- ▶ MQTMC (Trigger message)
Defines the format of a trigger message as a set of character fields (MQSeries for AS/400 only).
- ▶ MQTMC2 (Trigger message)

Defines the format of a trigger message including the queue manager name (MQSeries for MVS/ESA, MQSeries on UNIX systems, MQSeries for OS/2 Warp, and MQSeries for Windows NT only).

- ▶ MQXP (Exit parameter block) structure
Used to communicate with the API-crossing exit (MQSeries for MVS/ESA only).
- ▶ MQXQH (Transmission queue header)
Defines the format of the header that is added to messages put on a transmission queue.

For C and Visual Basic, MQI provides the following elementary data types:

MQBYTE	A single byte of data
MQBYTEn	A string of 16, 24, 32, or 64 bytes
MQCHAR	One single-byte character
MQCHARn	A string of 4, 8, 12, 16, 20, 28, 32, 48, 64, 128, or 256 single-byte characters
MQHCONN	A connection handle (this data is 32 bits long)
MQHOBJ	An object handle (this data is 32 bits long)
MQLONG	A 32-bit signed binary integer
PMQLONG	A pointer to data of type MQLONG

3.5 Programming with MQI

As explained in previous sections, the MQI is a set of functions and data types that allow the programmer to send and receive messages. The different functions accept input/output parameters, in order to be compatible among all the languages listed.

Generally, an MQI program applies a simple flow of operations as shown in Figure 3-1 on page 31.

1. The program must first connect to a queue manager, using the MQCONN call.
2. Once a connection has been successfully established, one or many objects can be opened using the MQOPEN call.
3. Any number of operations, such as get or put operations, can be performed on each object until the object is no longer needed.
4. Then the object can be closed using the MQCLOSE call.

5. The queue manager connection is discarded using the MQDISC call.

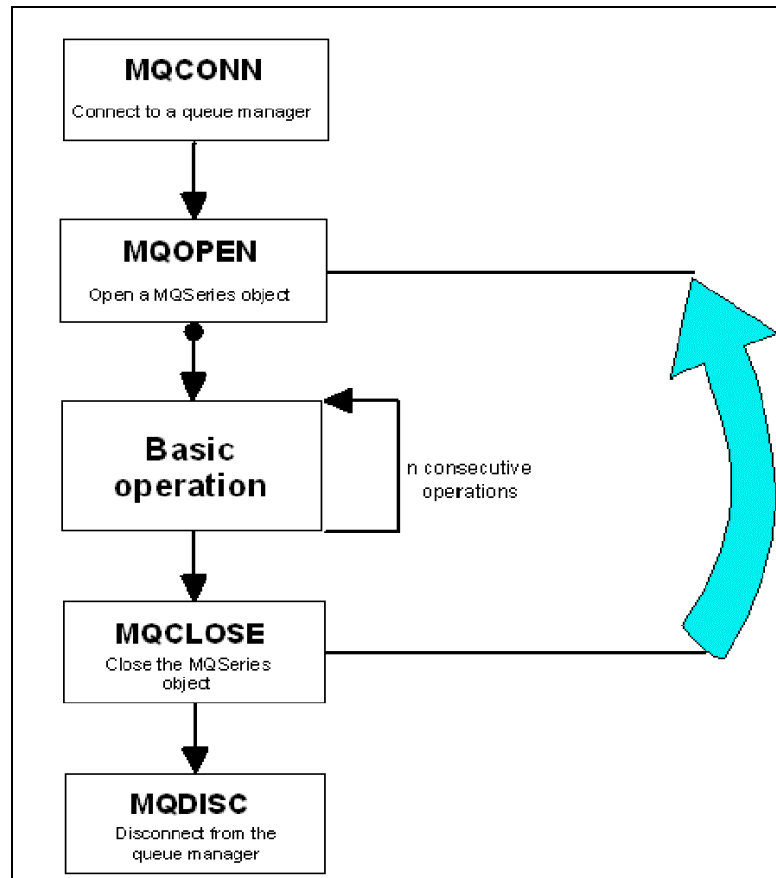


Figure 3-1 MQI program sequence

We first review some of the common elements of all these calls, then we take a look at the calls needed to connect and disconnect from a queue manager and how to open and close the MQSeries objects

Once we have done that, we review the four basic operations that can be performed with the MQI:

- ▶ Putting messages on a queue
- ▶ Getting messages from a queue
- ▶ Browsing messages on a queue
- ▶ Inquiring and setting object attributes

Any number of basic operations can be performed within the open/close calls.

We won't provide the detail specification of each function call, since this is not the purpose of this redbook, but you can refer to the *Application Programming Reference* and *Application Programming Guide* that comes with the MQSeries product. Here we try to present a brief description of the possibilities you have with this API so you can find out if your requirements can be solved with this API and how.

3.5.1 Basic API concepts

The basic concepts of the MQI API are discussed in the following sections.

Parameters common to all calls

There are two types of parameters common to all calls:

- Handles

These are returned by the queue manager connection and open queue calls and are then used as input parameters for the subsequent calls.

- Return codes

Two return codes are common to all the calls: a completion code and a reason code.

- The completion code specifies whether the call was completed successfully (with an MQCC_OK) or if a failure occurred (with an MQCC_FAILED). It can also return an intermediate state, MQCC_WARNING, indicating partial success.
- The reason code is MQCC_NONE if the completion code is MQCC_OK. If not, some other value is returned explaining the cause of the warning or failure reported in the completion code.

Retrieving messages in order

Messages in a queue can be scanned in either physical or logical order. This ordering applies if there is a get or browse request made to the queue where the messages are held.

Physical ordering is related to the way messages are received by the queue. The first message to arrive in the queue is the first message to be presented during a get or browse operation. Physical ordering gives us a FIFO (First-in/First-out) or FIFO within Priority sequence of the messages on the queue.

If FIFO within Priority ordering is used, the messages with the higher priority appears first in the queue, even if they arrive after a lower priority message has arrived.

This can cause some messages to be ignored when a program is getting or browsing messages on a queue. This is because once a message with a lower priority has been reached and another message with a higher priority appears in the queue, that message is put before the current message (pointer) so the application won't be able to see it until the queue is closed and re-opened, or an MQOGET is issued with the MQ00_BROWSE_FIRST option.

For example, let's take the following message sequence sent to a Priority order queue:

1. Message 1, priority 1
2. Message 2, priority 2
3. Message 3, priority 1

Let's say that an application is browsing messages on that queue. The program first sees the messages in this order:

1. Message 2, priority 2
2. Message 1, priority 1 (*the browse cursor is here*)
3. Message 3, priority 1

If the browse cursor of the program is pointing to Message 1 when a Message 4 appears with priority 2, then the order of the queue will be:

1. Message 2, priority 2
2. Message 4, priority 2
3. Message 1, priority 1 (*the browse cursor is here*)
4. Message 3, priority 1

In this case, the program is not able to see the Message 4 until it re-opens the queue or moves the browse cursor to the top of the queue using the MQOO_BROWSE_FIRST option.

The logical order, on the other hand, is mainly related to message grouping and segmentation. In this case, messages are presented in the queue grouped by GroupId and the groups are ordered depending on the physical position of the first message of the group. Message segments have the same behavior.

For example, let's take the following sequence of messages sent to a queue:

1. Logical message 1 (not last) of group 123
2. Logical message 1 (not last) of group 456
3. Logical message 2 (last) of group 456
4. Logical message 2 (not last) of group 123
5. Logical message 3 (last) of group 123

This messages appear to the application in the following sequence:

1. Logical message 1 (not last) of group 123

2. Logical message 2 (not last) of group 123
3. Logical message 3 (last) of group 123
4. Logical message 1 (not last) of group 456
5. Logical message 2 (last) of group 456

Here, the same problem explained with the FIFO within Priority can be found, since parts of a message group can arrive onto the queue after the browse cursor has been positioned to the next message group.

3.5.2 Connecting to the queue manager

To start working with MQSeries using this interface, you should first connect to a queue manager using one of the two available connection calls: MQCONN and MQCONNX. The syntax would be as follows:

```
MQCONN (QMgrName, Hconn, CompCode, Reason)
```

```
MQCONNX (QMgrName, ConnectOpts, Hconn, CompCode, Reason)
```

As input to MQCONN we must supply the name of the queue manager, or a null starting string if we want to open the default queue. Additionally, the MQCONNX call allows you to specify some options regarding the binding method (whether the connection is binding or non-binding).

The output from these calls are:

- ▶ The result codes (completion and reason code).
- ▶ A connection handle to the queue manager.

The scope of a connection call is generally the thread that issues the call. Refer to the *Application Programming Guide* for detailed information about the scope considerations on each platform.

The following code fragment shows how to connect to a queue manager using the C MQI:

```
MQHCONN Hcon;                /* connection handle          */
MQLONG  CompCode;            /* completion code           */
MQLONG  Reason;              /* reason code                */
char     QMName[50];         /* queue manager name         */

strcpy(QMName, "SampleQM");

// Connecting to the Queue Manager
MQCONN(QMName,                /* queue manager              */
        &Hcon,                /* connection handle          */
        &CompCode,            /* completion code            */
        &Reason);
```

```

        &CReason);                                /* reason code          */

if (CompCode == MQCC_FAILED) {
    printf("MQCONN failed with reason code %ld\n", CReason);
}

```

3.5.3 Opening MQSeries objects

There are four types of MQSeries objects that can be opened:

- ▶ Queue
- ▶ Namelists (in MQSeries for MVS/ESA)
- ▶ Process definition
- ▶ Queue manager

To open any of these objects we use the MQOPEN call:

```
MQOPEN (Hconn, ObjDesc, Options, Hobj, CompCode, Reason)
```

This call receives:

- ▶ A connection handle as returned by the MQCONN call.
- ▶ A description of the object we want to open, in the form of an object descriptor (MQOD) structure.
- ▶ One or more options that control the action of the call.

The output from this call is:

- ▶ The result codes (completion and reason code).
- ▶ An object handle that represents your access to the object.
- ▶ A modified object-descriptor structure, if the object opened was a dynamic queue.

We focus on how to open a queue object in the next section. For more information on how to open the other types of objects, please refer to the *Application Programming Guide*.

Opening queues

In the MQI, the only available type of message container is a queue. Queues can be either for output or input messages and apply to all the patterns described in Chapter 1, “Introduction and patterns” on page 3.

There are three types of queue from a programmer perspective:

- ▶ Local queues

- ▶ Remote queues
- ▶ Dynamic queue

Local and remote queues represent actual queues defined in the queue manager from an administrative prospective. They differ mostly in the way the name of the queue can be specified in the ObjectName field of the MQOD data structure.

A local queue's name is the one specified in the local queue manager.

A remote queue's name can be referred to by the name of the remote queue as known by the local queue manager, or by the name in the remote queue manager. If the name in the remote queue manager is used, then the ObjectQMGrName field must specify either:

- ▶ The name of the transmission queue that has the same name as the remote queue manager.
- ▶ The name of an alias queue object that resolves to the transmission queue that has the same name of the remote queue manager.

The following code fragment shows how to open a local or remote queue using the local queue manager name:

```
MQOD      od = {MQOD_DEFAULT};    /* Object Descriptor */
MQLONG    O_options;              /* MQOPEN options */

// Setting Queue Name in the Object Descriptor data structure
strcpy(od.ObjectName, "SampleQueue");

// Setting Open Options
O_options = MQOO_INPUT_AS_Q_DEF  /* open queue for input      */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
// Opening the assigned object
MQOPEN(Hcon,                      /* connection handle        */
       &od,                      /* object descriptor for queue */
       O_options,                /* open options             */
       &Hobj,                    /* object handle            */
       &CompCode,                /* completion code          */
       &Reason);                /* reason code              */

// Here we use the reason code instead of the CompCode to show how
// this parameter can also be used to determine if any problem has occurred.
if (Reason != MQRC_NONE) {
    printf("MQOPEN failed with reason code %ld\n", Reason);
}
```

Dynamic queues are created on demand and are destroyed once they are no longer in use. They are not created at an administrative level in MQSeries. For example, they can be used in a request/response scenario where the requester specifies a “reply-to” queue.

To create a dynamic queue, we use a template known as a model queue that is created administratively, together with an MQOPEN call. The name of the dynamic queue can be specified in the *DynamicQName* field of the MQOD structure. The name can be specified in three ways:

- ▶ Giving a full name, not longer than 33 characters.
- ▶ Giving a partial name, in which case you can specify certain prefixes for the queue name follow by an asterisk (*). The queue manager then generates a unique name using the prefix you specified.
- ▶ Allowing the queue manager to generate a full name, by specifying an asterisk (*) in the first character.

If the dynamic queue is created successfully, the object-descriptor structure is returned with the actual name of the dynamic queue in the ObjectName field. The following code fragment opens a dynamic queue and then prints its name:

```
strcpy(od.ObjectName, "SampleModel");
strcpy(od.DynamicQName, "SampleDQ*");

// Setting Open Options
O_options = MQOO_INPUT_AS_Q_DEF /* open queue for input */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
// Opening the assigned object
MQOPEN(Hcon, /* connection handle */
       &od, /* object descriptor for queue */
       O_options, /* open options */
       &Hobj, /* object handle */
       &CompCode, /* completion code */
       &Reason); /* reason code */

// Here we use the reason code instead of the CompCode to show how
// this parameter can also be used to determine if any problem has occurred.
if (Reason != MQRC_NONE) {
    printf("MQOPEN failed with reason code %ld\n", Reason);
} else {
    printf("THE newly open dynamic queue name is %s", od.ObjectName);
}
```

Opening distribution lists

Distribution lists allow you to put a message to multiple queues in a single MQPUT or MQPUT1 call.

In order to do this, the distribution list must be open using the MQOPEN call with the following input parameters:

- ▶ A connection handle
- ▶ Generic information in the Object Descriptor structure (MQOD)
- ▶ The Object Descriptor structure (MQOD) must specify MQOD_VERSION_2 in the Version field and the number of Object Record structures (which is the same as the number of queues you want to open) must be specified in the RecsPresent field.
- ▶ The name of each queue you want to open, using the Object Record structure (MQOR).

The output of this call is:

- ▶ An object handle that represents your access to the distribution list.
- ▶ A generic completion code.
- ▶ A generic reason code.
- ▶ Response records (MQRR structure), containing the result codes (completion and reason code) for each destination.

One MQOR record must be provided for each destination, as a combination of the queue name and queue manager name.

The address of this array of destination queues can be specified in two ways:

- ▶ By using the offset field ObjectRecOffset, giving the offset of the first element in the array from the start of the MQOD structure. This is common when using COBOL, for example:

```
01 MY-OPEN-DATA.  
  02 MY-MQOD.  
    COPY CMQODV.  
  02 MY-MQOR-TABLE OCCURS 100 TIMES.  
    COPY CMQORV.  
  MOVE LENGTH OF MY-MQOD TO MQOD-OBJECTRECOFFSET.
```

- ▶ By using the pointer field ObjectRecPtr, giving the address of the array of MQOR records. This is the usual approach in C, for example:

```
MQOD MyMqod;  
MQOR MyMqor[NUMBER_OF_DESTINATIONS];  
MyMqod.ObjectRecPtr = MyMqor;
```

The MQRR structure contains the completion and reason code for a specific destination in the distribution list. If any of the destinations fail to open, this MQRR array allows us to recognize the queue with the problem and take some corrective action. See Example 3-1.

Example 3-1 Opening distribution lists

```
// In this example the number of queues is constant but it can be defined as a
//variable value depending on the application needs.
#define NumQueues /* set the number of queues */

MQLONG   Index ;                /* Index into list of queues      */
PMQRR    pRR=NULL;              /* Pointer to response records */
PMQOR    pOR=NULL;              /* Pointer to object records   */

// These two arrays are used to simplify this example. The names of the queues
// and queue manager can be taken from any available resource.
char queueNames[MQ_Q_NAME_LENGTH][NumQueues];
char queueMNames[MQ_Q_MGR_NAME_LENGTH][NumQueues];

// Allocating the response and object records arrays.
pRR = (PMQRR)malloc( NumQueues * sizeof(MQRR));
pOR = (PMQOR)malloc( NumQueues * sizeof(MQOR));

// Setting the queue and queue manager names in the PMQOR record array.
for( Index = 0 ; Index < NumQueues ; Index ++ ) {
    strncpy( (pOR+Index)->ObjectName,
             queueNames[Index],
             (size_t)MQ_Q_NAME_LENGTH);
    strncpy( (pOR+Index)->ObjectQMgrName,
             queueMNames[Index],
             (size_t)MQ_Q_MGR_NAME_LENGTH);
}

// Setting values in the Object Descriptor data structure.
od.Version = MQOD_VERSION_2 ;
od.RecsPresent = NumQueues ;    /* number of object/resp recs */
od.ObjectRecPtr = pOR;          /* address of object records   */
od.ResponseRecPtr = pRR ;       /* Number of object records    */
O_options = MQOO_OUTPUT         /* open queue for output       */
           + MQOO_FAIL_IF QUIESCING; /* but not if MQM stopping    */

// Opening the distribution list. The queue manager connection
// must be established before opening the distribution list.
MQOPEN(Hcon,                    /* connection handle          */
       &od,                     /* object descriptor for queue */
       O_options,               /* open options               */
       &Hobj,                   /* object handle              */
       &CompCode,               /* MQOPEN completion code     */
       &Reason);               /* reason code                */
```

More details about how these structures are used can be found in the *Application Programming Guide*.

Some other options available when opening a queue are considered in their specific context in the following sections.

3.5.4 Closing the MQSeries object

To close an MQSeries object we use the MQCLOSE call.

```
MQCLOSE (Hconn, Hobj, Options, CompCode, Reason)
```

This call receive the following input:

- ▶ A connection handle.
- ▶ The handle of the object we want to close.
- ▶ The close options.

The output of this call is:

- ▶ The result codes (completion and reason code).
- ▶ The object handle, reset to the value MQHO_UNUSABLE_HOBJ.

Unless you are closing a permanent dynamic queue, the close options will be MQCO_NONE.

Typically a dynamic queue is deleted once the program that created it calls an MQCLOSE for that queue, but in the case of permanent dynamic queues, they can be retained by the queue manager or deleted depending on the options used in the MQCLOSE call.

```
MQLONG   C_options;                /* MQCLOSE options */

C_options = 0;                      /* no close options */
MQCLOSE(Hcon,                      /* connection handle */
        &Hobj,                     /* object handle */
        C_options,
        &CompCode,                /* completion code */
        &Reason);                 /* reason code */
```

It is recommended that you close all MQSeries objects before the application finishes.

3.5.5 Disconnecting from the queue manager

The final step in an MQSeries program is to disconnect from the queue manager. This can be done with the MQDISC call.

```
MQDISC (Hconn, CompCode, Reason)
```


For this call you must provide the connection handle to the queue manager. After the execution of this call, the connection handle will have an MQHC_UNUSABLE_HCONN.

```
MQDISC(&Hcon,                /* connection handle */
       &CompCode,            /* completion code */
       &Reason);             /* reason code */
```

3.5.6 Putting messages in a queue

To put a message in a queue the MQI API gives the programmer two options:

- ▶ Put multiple messages to an already open queue.
- ▶ Put a single message to a queue without having to explicitly open it.

To put multiple messages in a queue we can use the MQPUT call:

```
MQPUT (Hconn, Hobj, MsgDesc, PutMsgOpts, BufferLength,  
       Buffer, CompCode, Reason)
```

This call receives:

- ▶ A connection handle, as returned by the MQCONN call (in CICS on MVS/ESA and AS/400 applications, it can also be the MQHC_DEF_HCONN constant).
- ▶ A queue handle, as returned by the MQOPEN call.
- ▶ A description of the message you want to put on the queue, in the form of a message descriptor.
- ▶ Control information, in the form of a put-message options (MQPMO) structure.
- ▶ The length of the data contained in the message.
- ▶ The message itself.

The output of this call is:

- ▶ The result codes (completion and reason codes).
- ▶ Updated message descriptor and options if the call was executed successful.

The queue must be opened with the MQOO_OUTPUT option, before any number of calls to this function can be made.

The updated message descriptor will have the MsgId of the message, if the initial structure requested that the queue manager generate a unique value for the message.

The options structure returns with the name of the queue and the queue manager to which the message was sent.

Example 3-2 shows how to send a message to the SampleQueue used before.

Example 3-2 MQPUT call

```

/* Declare MQI structures needed */
MQOD   od = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD   md = {MQMD_DEFAULT}; /* Message Descriptor */
MQPMO  pmo = {MQPMO_DEFAULT}; /* put message options */

MQHCONN Hcon; /* connection handle */
MQHOBJ  Hobj; /* object handle */
MQLONG  O_options; /* MQOPEN options */
MQLONG  CompCode; /* completion code */
MQLONG  Reason; /* reason code */
MQLONG  messlen; /* message length */
char     buffer[100]; /* message buffer */

// A connection handle must be obtained using the MQCONN call
// as shown in the previous sections.

MQCONN(...);

// Setting the target queue name in the Object Descriptor data structure.
strncpy(od.ObjectName, "SampleQueue");

// Setting Open options, in this case MQOO_OUTPUT is require to execute
// the following MQPUT call.
O_options = MQOO_OUTPUT /* open queue for output */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */

MQOPEN(Hcon,&od,O_options,&Hobj,&CompCode,&Reason);

// Validating the completion and reason codes
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

if (CompCode == MQCC_FAILED) {
    printf("unable to open queue for output\n");
}

// Preparing message data. In this example a simple message data
// will be sent. Message data can be prepared using other resources
// such as files, information in a database, etc.
strcpy(buffer, "Message data");
messlen = strlen(buffer);

```

```

// The following two statements are not required if the
// MQPMO_NEW_MSG_ID and MQPMO_NEW_CORREL_ID options are used.
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) );

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);
}

// The queue must be closed after the last message has been sent, and we must
// disconnect from the queue manager, as explained in previous sections.

```

Putting a single message on a queue

To put a single message without having to explicitly open a queue, we can use the MQPUT1 call.

MQPUT1 (Hconn, ObjDesc, MsgDesc, PutMsgOpts, BufferLength,
Buffer, CompCode, Reason)

The MQPUT1 interface is the same as the MQPUT interface except for the queue handle, where we must instead specify an object descriptor as specified in the MQOPEN call. See Example 3-3.

Example 3-3 MQPUT1 call

```

/* Declare MQI structures needed */
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 */
MQLONG  messlen;              /* message length */
char     buffer[100];         /* message buffer */

```

```

// A connection handle must be obtained using the MQCONN call
// as shown in the previous sections.

MQCONN(...);

// Setting the target queue name in the Object Descriptor data structure.
strncpy(od.ObjectName, "SampleQueue");

// Preparing message data. In this example a simple message data
// will be sent. Message data can be prepared using other resources
// such as files, information in a database, etc.
strcpy(buffer, "Message data");
messlen = strlen(buffer);

// The following two statements are not required if the
// MQPMO_NEW_MSG_ID and MQPMO_NEW_CORREL_ID options are used.

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) );

MQPUT1(Hcon,              /* connection handle */
       &od,               /* object descriptor */
       &md,               /* message descriptor */
       &pmo,              /* default options (datagram) */
       messlen,           /* message length */
       buffer,            /* message buffer */
       &CompCode,         /* completion code */
       &Reason);          /* reason code */

if (Reason != MQRC_NONE) {
    printf("MQPUT ended with reason code %ld\n", Reason);
}

// The queue must be closed after the last message has been sent, and
// we must disconnect from the queue manager, as explained in previous
// sections.

```

Note: It is important to notice that this approach is useful if the amount of operations that are actually going to use this function is small enough not to consider the MQOPEN and MQPUT functions, since the overhead associated with this single put function is much higher.

Putting messages to multiple queue at once

You can also put messages to a distribution list. The message is sent to all the queues in the distribution list with a single MQPUT or MQPUT1 call. In this case the MQPUT call input parameters are:

- ▶ A connection handle.
- ▶ An object handle to the distribution list opened using MQOPEN call, as shown previously in this chapter.
- ▶ A message descriptor structure (MQOD).
- ▶ General control information.
- ▶ Control information in the form of Put Message Records (MQPMR).
- ▶ The length of data contained within the message.
- ▶ The message itself.

The output of this call is:

- ▶ The result codes (completion and reason codes).
- ▶ Response records.

The MQPMR structure gives destination-specific information for some fields that may differ from those already in the MQMD structure.

3.5.7 Getting messages from a queue

To get messages from a queue, we can use the MQGET call.

*MQGET (Hconn, Hobj, MsgDesc, GetMsgOpts, BufferLength,
Buffer, DataLength, CompCode, Reason)*

The input parameters for this call are:

- ▶ A connection handle.
- ▶ A queue handle.
- ▶ A description of the message we want to get from the queue in the form of an MQMD structure.
- ▶ Control information in the form of a Get Message Options (MQGMO) structure.
- ▶ The size of the buffer in which the message is going to be stored.
- ▶ The address of the buffer.

The output of this call is:

- ▶ The result codes (reason and completion codes).
- ▶ The message in the buffer specified, if the call completes successfully.
- ▶ The options structure, modified to show the name of the queue from which the message was retrieved.
- ▶ The message descriptor structure, with the information of the message that is retrieved.
- ▶ The actual length of the message.

To use this, the queue must be opened with the MQOO_INPUT_SHARED or MQOO_INPUT_EXCLUSIVE options.

Messages are retrieved from the queue in either Physical or Logical order, depending on the options used in the MQOPEN and MQGET calls. See Example 3-4.

Example 3-4 MQGET call

```
/* Declare 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 */
MQHOBJ  Hobj;                  /* object handle */
MQLONG  O_options;             /* MQOPEN options */
MQLONG  CompCode;              /* completion code */
MQLONG  Reason;                /* reason code */
MQBYTE  buffer[101];           /* message buffer */
MQLONG  buflen;                /* buffer length */
MQLONG  messlen;               /* message length received */

// A connection handle must be obtained using the MQCONN call
// as shown in the previous sections.

MQCONN(...);

// Setting the target queue name in the Object Descriptor data structure.
strcpy(od.ObjectName, "SampleQueue");

// Setting Open options, in this case MQOO_INPUT_AS_Q_DEF, MQOO_INPUT_SHARE
// or MQOO_INPUT_EXCLUSIVE is required to execute the following MQGET call.
O_options = MQOO_INPUT_AS_Q_DEF /* open queue for input */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon, &od,O_options,&Hobj,&CompCode,&Reason);

// Validating the completion and reason codes
```

```

if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

if (CompCode == MQCC_FAILED) {
    printf("unable to open queue for input\n");
}

// These options cause the MsgId and CorrelId to be replaced, so
// that there is no need to reset them before each MQGET if more than one
// get operation will be performed using the same get message options data
// structure. These options may not be available in all the environments.
gmo.Version = MQGMO_VERSION_2; /* Set Message Options version 2*/

// No match options will be used while scanning the queue
gmo.MatchOptions = MQMO_NONE;
gmo.Options = MQGMO_WAIT /* wait for new messages */
             + MQGMO_CONVERT; /* convert if necessary */
gmo.WaitInterval = MQWI_UNLIMITED; /* unlimited time for waiting */

buflen = sizeof(buffer) - 1; /* buffer size available for GET */

// Setting Encoding and CodedCharSetId in case some conversion is needed.
md.Encoding = MQENC_NATIVE;
md.CodedCharSetId = MQCCSI_Q_MGR;

// Perform the get operation
MQGET(Hcon, Hobj, &md, &gmo, buflen, buffer, &messlen, &CompCode, &Reason);

// report reason, if the call fails.
if (CompCode == MQCC_FAILED) {
    printf("MQGET ended with reason code %ld\n", Reason);
} else {
    // Display the message received
    buffer[messlen] = '\0'; /* add terminator */
    printf("message <%s>\n", buffer);
}

// The queue must be closed after the last message has been received, and
// you must disconnect from the queue manager, as explained in previous
// sections.

```

Getting a specific message from a queue

Messages can also be retrieved based on their description, as provided in the MQMD structure.

The MsgId and CorrelId fields can be used to refer to a specific message, but since they can be set by the application they might not be unique. In these cases the first message that matches all the criteria provided is retrieved, and the call can be repeated to get the rest of the messages.

If the MQMD structure version 2 is used, then the GroupId, MsgSeqNumber, and Offset fields can be used, too.

A no-match value can be specified in any of these fields so the field is not considered while searching for a match. Using the MQMD structure version 2, it is also possible to declare which fields to use during the queue scan.

The following piece of code shows how to set a correlId as the message search key before the MQGET call:

```
gmo.MatchOptions = MQMO_MATCH_CORREL_ID;
memcpy(md.CorrelId, myCorrelId, sizeof(md.CorrelId));

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             */
```

The queues have some indexing capabilities to increase the performance of these operations. The index field can be either MsgId or CorrelId, depending on the value of the indexType attribute of the queue.

3.5.8 Advanced topics

The following sections cover browsing messages on a queue and inquiring about and setting object attributes.

Browsing messages on a queue

To browse messages on a queue, we:

- ▶ Call MQOPEN to open the queue for browsing, specifying the MQOO_BROWSE option.
- ▶ Call MQGET with the MQGMO_BROWSE_FIRST option to retrieve the first message on the queue.

- Repeatedly call MQGET with the MQGMO_BROWSE_NEXT option to step through many messages, setting the MsgId and CorrelId null before any new MQGET call.
- Close the queue using the MQCLOSE call.

When browsing messages, in the same way as when getting messages from a queue, the messages are presented in either a physical or logical order.

Example 3-5 shows how to browse messages in a queue in physical order.

Example 3-5 Browse messages

```

/* Declare 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 */
MQHOBJ   Hobj; /* object handle */
MQLONG   O_options; /* MQOPEN options */
MQLONG   CompCode; /* completion code */
MQLONG   Reason; /* reason code */
MQBYTE   buffer[101]; /* message buffer */
MQLONG   buflen; /* buffer length */
MQLONG   messlen; /* message length received */

// A connection handle must be obtained using the MQCONN call
// as shown in the previous sections.

MQCONN(...);

// Setting the target queue name in the Object Descriptor data structure.
strncpy(od.ObjectName, "SampleQueue");

// Setting Open options, in this case MQOO_INPUT_AS_Q_DEF, MQOO_INPUT_SHARE
// or MQOO_INPUT_EXCLUSIVE is required to execute the following MQGET call.
O_options = MQOO_BROWSE /* open queue for browse, */
           + MQOO_FAIL_IF QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon,&od,O_options,&Hobj,&CompCode,&Reason);

// Validating the completion and reason codes
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

if (CompCode == MQCC_FAILED) {
    printf("unable to open queue for browse\n");
}

```

```

gmo.Version = MQGMO_VERSION_2; /* Set Message Options version 2*/

// No match options will be used while scanning the queue
gmo.MatchOptions = MQMO_NONE;

// Set the get message options. Here is where the MQGET method
// differs a destructive get from a browse get. We use the
// MQGMO_BROWSE_NEXT since the queue has just been opened and then
// the browse cursor is positioned in the beginning of the queue.
// If the queue had been used before, we can use MQGMO_BROWSE_FIRST for
// the first MQGET call and then MQGMO_BROWSE_NEXT for the rest of the calls.
gmo.Options = MQGMO_NO_WAIT      /* don't wait for new messages */
+ MQGMO_BROWSE_NEXT            /* browse messages in order */
+ MQGMO_ACCEPT_TRUNCATED_MSG;  /* truncate longer message */

// Here we are giving a 100 bytes buffer to receive the message. If the
// message data is larger than 100 bytes then the message will be
// truncated since we give the MQGMO_ACCEPT_TRUNCATED_MSG option in the
// get message options structure. In this case a reason code of
// MQRC_TRUNCATED_MSG_ACCEPTED can be expected.
buflen = sizeof(buffer) - 1; /* buffer size available for GET */

// Setting Encoding and CodedCharSetId in case some conversion is needed.
while (CompCode != MQCC_FAILED) {
// Set encoding before each MQGET call in case any data conversion
// is required.
md.Encoding      = MQENC_NATIVE;
md.CodedCharSetId = MQCCSI_Q_MGR;

    MQGET(Hcon,Hobj,&md,&gmo,buflen,buffer,&messlen,&CompCode,&Reason);

// report reason, if the call fails.
if (CompCode == MQCC_FAILED) {
    printf("MQGET ended with reason code %ld\n", Reason);
} else {
    // Display the message received
    buffer[messlen] = '\0';          /* add terminator */
    printf("message <%s>\n", buffer);
}
}

//The queue must be closed after the last message has been browsed, and you
//must disconnect from the queue manager, as explained in previous sections.

```

If you don't know the size of the messages in the queue, you can use an MQGET call with the following options to obtain the size of the message and then browse or get it from the queue:

- ▶ Either the MQGMO_BROWSE_FIRST or MQGMO_BROWSE_NEXT option.
- ▶ The MQGMO_ACCEPT_TRUNCATED_MSG option.
- ▶ Buffer length of zero (0).

It returns the size of the message in the DataLength parameter.

Inquiring about and setting object attributes

To inquire about the attributes of an object, we can use the MQINQ call.

```
MQINQ (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount,
      IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason)
```

As input to this call, we must supply:

- ▶ A connection handle.
- ▶ An object handle, as returned by the MQOPEN call. The object open can be any of the possible objects specified in 3.5.3, “Opening MQSeries objects” on page 35.
- ▶ The number of selectors.
- ▶ An array of attribute selectors. The selectors can represent either integer (MQIA_*) or character (MQCA_*) attributes, and can be specified in any order.
- ▶ The number of integer attributes being inquired.
- ▶ An array of integer variables where the call returns the integer attributes specified.
- ▶ The length of the character attributes buffer. The buffer should be at least the sum of the length of all the character attributes inquired.
- ▶ The character buffer where the call puts the values of the character attributes inquired.

The output for this call is:

- ▶ A set of integer attribute values copied into the array.
- ▶ The buffer in which character attributes are returned.
- ▶ The result codes (completion and reason codes).

A complete list of the attributes selectors can be found in the *Application Programming Reference*.

In the case of character attributes, the resulting buffer is filled with the attributes values, one after the other, with a fixed length. If the actual value of any of these attributes is smaller than the fixed length of the attribute, the rest of the space is filled with blanks.

If any of the attributes requested for the object (in this case it is a queue) does not apply to that type of queue, the space is filled with asterisks (*).

Example 3-6 shows how to inquire about the attributes of a queue:

Example 3-6 Inquire object attributes

```

MQOD      odI = {MQOD_DEFAULT};      /* Object Descriptor (INQUIRE) */

MQHCONN   Hcon;                      /* connection handle          */
MQHOBJ    Hinq;                      /* handle for MQINQ           */
MQLONG    O_options;                 /* MQOPEN options             */
MQLONG    CompCode;                  /* completion code             */
MQLONG    Reason;                    /* reason code                  */
MQLONG    Select[3];                 /* attribute selectors          */
MQLONG    IAV[3];                    /* integer attribute values     */

// A connection handle must be obtained using the MQCONN call
// as shown in the previous sections.

MQCONN(...);

// Open named queue for INQUIRE
// Set the name of the queue object to inquire
strcpy(odI.ObjectName, "SampleQueue");

O_options = MQOO_INQUIRE      /* open to inquire attributes */
           + MQOO_FAIL_IF_QUIESCING;

MQOPEN(Hcon,&odI,O_options,      /* open options                */
       &Hinq,                   /* object handle for MQINQ     */
       &CompCode,                /* completion code              */
       &Reason);                 /* reason code                  */

// Validating the completion and reason codes
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}
if (CompCode == MQCC_FAILED) {
    printf("unable to open queue for inquire\n");
} else {
    Select[0] = MQIA_INHIBIT_GET;    /* attribute selectors */
    Select[1] = MQIA_CURRENT_Q_DEPTH;

```

```

Select[2] = MQIA_OPEN_INPUT_COUNT;
MQINQ(Hcon,          /* connection handle          */
      Hinq,          /* object handle          */
      3L,            /* Selector count         */
      Select,        /* Selector array         */
      3L,            /* integer attribute count */
      IAV,           /* integer attribute array */
      0L,            /* character attribute count */
      NULL,          /* character attribute array */
      /* note - can use NULL because count is zero */
      &CompCode,      /* completion code        */
      &Reason);      /* reason code            */

if (CompCode == MQCC_OK) {
    sprintf(reply, " has %ld messages, used by %ld jobs", IAV[1], IAV[2]);
    strcat(buffer, reply);
    if (IAV[0]) { /* if GET inhibited */
        strcat(buffer, "; GET inhibited");
    }
}

// The queue object must be closed, and we must disconnect
// from the queue manager, as explained in previous sections.

```

Setting object attributes

Only queue objects accept a set operation. To set attributes of a queue we use the MQSET call.

```
MQSET (Hconn, Hobj, SelectorCount, Selectors, IntAttrCount,
      IntAttrs, CharAttrLength, CharAttrs, CompCode, Reason)
```

The parameters for this call are the same of those in the MQINQ call. All of them are input parameters except the completion code and reason code.

These are the attributes that can be set using the MQSET call:

- ▶ InhibitGet (but not for remote queues)
- ▶ DistList
- ▶ InhibitPut
- ▶ TriggerControl
- ▶ TriggerType
- ▶ TriggerDepth
- ▶ TriggerMsgPriority
- ▶ TriggerData

Example 3-7 shows how to inhibit put operations on a queue:

Example 3-7 Set object attributes

```
MQOD    odS = {MQOD_DEFAULT};    /* Object Descriptor for SET    */

MQHCONN Hcon;                    /* connection handle        */
MQHOBJ  Hset;                    /* handle for MQSET         */
MQLONG  O_options;               /* MQOPEN options           */
MQLONG  CompCode;                /* completion code          */
MQLONG  Reason;                  /* reason code               */
MQLONG  Select[1];               /* attribute selector(s)    */
MQLONG  IAV[1];                  /* integer attribute value(s) */

// A connection handle must be obtained using the MQCONN call
// as shown in the previous sections.

MQCONN(...);

// Open named queue for SET
strcpy(odS.ObjectName, "SampleQueue");
O_options = MQOO_SET              /* open to set attributes    */
           + MQOO_FAIL_IF_QUIESCING;
MQOPEN(Hcon,                      /* connection handle        */
       &odS,                      /* object descriptor for queue */
       O_options,                 /* open options              */
       &Hset,                     /* object handle for MQSET   */
       &CompCode,                 /* completion code           */
       &Reason);                 /* reason code               */

/* prepare to report error if it failed */
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}
if (CompCode == MQCC_FAILED) {
    printf("unable to open queue for set\n");
} else {

// Inhibit Put if the queue was opened successfully
Select[0] = MQIA_INHIBIT_PUT;    /* attribute selector        */
IAV[0]     = MQQA_PUT_INHIBITED; /* attribute value           */

MQSET(Hcon,                      /* connection handle        */
      Hset,                      /* object handle            */
      1L,                        /* Selector count           */
      Select,                    /* Selector array           */
      1L,                        /* integer attribute count   */
      IAV,                       /* integer attribute array   */
      0L,                        /* character attribute count */
      NULL,                      /* character attribute array */
      /* note - can use NULL because count is zero */
      );
```

```

        &CompCode,      /* completion code      */
        &Reason);      /* reason code      */

if (CompCode == MQCC_OK) {
    strcat(buffer, " PUT inhibited");
    messlen = strlen(buffer);          /* length of reply */
    md.MsgType = MQMT_REPLY;
} else {
    md.MsgType = MQMT_REPORT;
    md.Feedback = Reason;              /* result of MQSET */
}

// The queue object must be closed, and we must disconnect
// from the queue manager, as explained in previous sections.

```

More details about the use of this function can be found in the *Application Programming Reference*.

3.6 Transactions in MQI

Either local or global units of work can be started using the MQI API. Once started, any unit of work is completed using MQCMIT or MQBACK calls.

MQCMIT (*Hconn, CompCode, Reason*)

MQBACK (*Hconn, CompCode, Reason*)

A local unit of work is started when MQPMO_SYNCPOINT or MQGMO_SYNCPOINT is coded on an MQPUT or MQGET call and an MQBEGIN call has not been made, as follows:

```

MQPMO pmo;
pmo.Options = MQPMO_SYNCPOINT;
MQPUT(...);

```

Or:

```

MQGMO gmo;
gmo.Options = MQGMO_SYNCPOINT;
MQGET(...);

```

Every operation within the unit of work must have the MQPMO or MQGMO options set as shown in the previous code fragment.

A global unit of work starts when an MQBEGIN call is made. If a local unit of work has already been started, the MQBEGIN call fails with an MQRC_UOW_IN_PROGRESS reason.

```
MQBEGIN (Hconn, BeginOptions, CompCode, Reason)
```

The following pseudo-code fragment shows a distributed transaction including basic get and put operation with some relational database operation:

```
MQBEGIN(...);

MQGET(...);

// Execute some relational database update
UPDATE tb11 (f1,f2) VALUES (v1,v2);

MQPUT(...);

// If any operation fails the transaction is backed out.
if (CompCode != MQCC_OK) {
    MQBACK(...);
} else {
    MQCOMMIT(...);
}
```

3.7 Message grouping in MQI

As explained in Chapter 2, “Messaging and the APIs” on page 11, message grouping lets us add some grouping logic to the messages without having to code all that logic into the application. The grouping is managed by the queue manager, and provides basic features such as ordering and group completion control.

The group identification is introduced in the message descriptor structure (MQMD) sent with the message, when using an MQPUT or MQPUT1 call.

Every message in the group must have the MQMF_MSG_IN_GROUP flag but the last one, which has an MQMF_LAST_MSG_IN_GROUP flag instead. The order of the messages in the group is stored in the MsgSeqNumber field of the MQMD structure, which is generated automatically by the queue manager.

The following code fragment shows how to send three messages as part of a message group:

```
// Setting put message options and message descriptor versions.
md.Version = MQMD_VERSION_2;
```



```

pmo.Version = MQPMO_VERSION_2;
// Sets the put message options to generate a new message id for every
// message put into the queue and to put the messages in their logical
// order into the queue.
pmo.Options = MQPMO_LOGICAL_ORDER | MQPMO_NEW_MSG_ID;
md.MsgFlags = MQMF_MSG_IN_GROUP;

// Assign the GroupId in the message descriptor structure
memcpy(md.GroupId,MY_GROUP_ID,sizeof(md.GroupId));

// Puts a first message of the group
strcpy(buffer, "First message");
messlen=strlen(buffer);

MQPUT(...);

// Puts a second message of the group
strcpy(buffer,"Middle Message");
messlen=strlen(buffer);

MQPUT(...);

// Puts the final message of the group. The final message must
// be identified by giving the MQMF_LAST_MSG_IN_GROUP flags in the message
// descriptor structure.
md.MsgFlags = MQMF_LAST_MSG_IN_GROUP;
strcpy(buffer,"Final Message");
messlen=strlen(buffer);

MQPUT(...);

```

If that group of messages be processed in order, you have to use the **MQGMO_LOGICAL_ORDER** flag to retrieve the messages from the group in their logical order.

```

// Setting get message options and message descriptor versions.
md.Version = MQMD_VERSION_2;
gmo.Version = MQMD_VERSION_2;

// Set the get message options required for this operation, especially
// the MQGMO_LOGICAL_ORDER
gmo.Options = MQGMO_LOGICAL_ORDER + MQGMO_WAIT //wait for new messages
              + MQGMO_CONVERT; /* convert if necessary */
gmo.WaitInterval = 1500; /* 15 second limit for waiting */
// We want to get all the messages on the queue so no match options will
// be needed.
gmo.MatchOptions = MQGMO_NONE;

while (CompCode != MQCC_FAILED) {

```

```

buflen = sizeof(buffer) - 1; /* buffer size available for GET */

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;
        }
    }
}

// Shows the message data
if (CompCode != MQCC_FAILED) {
    buffer[messlen] = '\0';          /* add terminator */
    printf("message <%s>\n", buffer);
}
}

```

Additionally, the queue manager can control whether or not a message group has been received completely. If we want only complete message groups to appear in the queue, the MQGMO_ALL_MSGS_AVAILABLE option can be set in the get message options structure.

3.8 Exploring the patterns

In this section, we explore the patterns presented in Chapter 2, “Messaging and the APIs” on page 11 using the C ANSI implementation of the MQI API in a Microsoft Windows environment. All of these samples can be modified to work with any other platform and language available.

The complete code for these examples can be found in the additional materials that may be downloaded from the IBM Redbooks Web site.

3.8.1 The one-to-one, or point-to-point pattern

As explained in Chapter 1, “Introduction and patterns” on page 3, the one-to-one or point-to-point programming pattern can be used for a send-and-forget scenario as well as a request/reply scenario or any combination of those.

Here, we present a simple example of each one of them. The message data used in these examples does not have any business logic, but the example can be easily modified to be applied in a real-world situation.

Send-and-forget

This simple example of the send-and-forget pattern contains two programs. The first one acts as the message sender, while the second one acts as the message consumer. No response or acknowledgment is expected by the sender and nothing is sent back by the consumer.

The sender program shown in Example 3-8 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the PTP.QUEUE.LOCAL queue for output
- ▶ Prepare a message to be sent
- ▶ Send the message to the opened queue
- ▶ Close the queue
- ▶ Disconnect from the queue manager

Example 3-8 Sender program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* includes for MQI */
#include <cmqc.h>

#define QMGR_NAME" SAMPLE.QMGR1.ITSOE"
#define Q_NAME" PTP.QUEUE.LOCAL"

int main(int argc, char **argv) {
```

```

/* Declare MQI structures needed */
MQOD      od = {MQOD_DEFAULT}; /* Object Descriptor */
MQMD      md = {MQMD_DEFAULT}; /* Message Descriptor */
MQPMO     pmo = {MQPMO_DEFAULT}; /* put message options */

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; /* MQCONN reason code */
MQLONG     messlen; /* message length */
char       buffer[100]; /* message buffer */
char       QMName[50]; /* queue manager name */

printf("Send/Forget Sample");

// Connect to queue manager
strcpy(QMName, QMGR_NAME);
MQCONN(QMName, &Hcon, &CompCode, &CReason);

/* report reason and stop if it failed */
if (CompCode == MQCC_FAILED) {
    printf("MQCONN ended with reason code %ld\n", CReason);
    exit( (int) CReason );
}

// Set queue name in the object descriptor
strcpy(od.ObjectName, Q_NAME);
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 */
MQOPEN(Hcon, &od, O_options, &Hobj, &OpenCode, &Reason);

/* 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");
} else {

    memcpy(md.Format, /* character string format */
           MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

```

```

// Set message descriptor and put message options on version 2
md.Version = MQMD_VERSION_2;
pmo.Version = MQPMO_VERSION_2;

// Request a new messageId for this message
pmo.Options |= MQPMO_NEW_MSG_ID;

strcpy(buffer, "This is a simple Send/Forget sample");
messlen = strlen(buffer);

MQPUT(Hcon, Hobj, &md, &pmo, messlen, buffer, &CompCode, &Reason);

/* report reason, if any */
if (Reason != MQRC_NONE) {
    printf("MQPUT ended with reason code %ld\n", Reason);
}

// Close the target queue (if it was opened)
if (OpenCode != MQCC_FAILED) {
    C_options = 0; /* no close options */
    MQCLOSE(Hcon, &Hobj, C_options, &CompCode, &Reason);

    /* report reason, if any */
    if (Reason != MQRC_NONE) {
        printf("MQCLOSE ended with reason code %ld\n", Reason);
    }
}

// Disconnect from MQM if not already disconnected
if (CReason != MQRC_ALREADY_CONNECTED) {
    MQDISC(&Hcon, &CompCode, &Reason);

    /* report reason, if any */
    if (Reason != MQRC_NONE) {
        printf("MQDISC ended with reason code %ld\n", Reason);
    }
}

printf("Send/Forget Sample end\n");
return(0);
}

```

The consumer program shown in Example 3-9 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the PTP.QUEUE.LOCAL queue for output

- ▶ Set the data buffer for the incoming message
- ▶ Get the message from the opened queue
- ▶ Print the message received
- ▶ Close the queue
- ▶ Disconnect from the queue manager

Example 3-9 Consumer program

```
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 */
    char     QMName[50]; /* queue manager name */

    printf("Consumer sample\n");

    // Create object descriptor for subject queue
    strcpy(od.ObjectName, Q_NAME);
    strcpy(QMName, QMGR_NAME);

    // Connect to queue manager
    MQCONN(QMName, &Hcon, &CompCode, &CReason);

    /* 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 shared
    O_options = MQOO_INPUT_SHARED /* open queue for input */
        + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */

    MQOPEN(Hcon, &od, O_options, &Hobj, &OpenCode, &Reason);
}
```

```

/* 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");
}

// These options cause the MsgId and CorrelId to be replaced, so
// that there is no need to reset them before each MQGET
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 */
gmo.WaitInterval = 15000;          /* 15 second limit for waiting */

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,Hobj,&md,&gmo,buflen,buffer,&messlen,&CompCode,&Reason);

/* 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,&Hobj,C_options,&CompCode,&Reason);

    /* report reason, if any      */
    if (Reason != MQRC_NONE) {
        printf("MQCLOSE ended with reason code %ld\n", Reason);
    }
}

//   Disconnect from MQM if not already disconnected
if (CReason != MQRC_ALREADY_CONNECTED ) {
    MQDISC(&Hcon,&CompCode,&Reason);

    /* report reason, if any      */
    if (Reason != MQRC_NONE) {
        printf("MQDISC ended with reason code %ld\n", Reason);
    }
}

printf("Consumer sample end\n");
return(0);
}

```

Request/reply

Just as in the send-and-forget pattern sample, this request/reply sample contains two programs. The first one sends a request message to a queue (the PTP.QUEUE.LOCAL queue) and waits for a response in another queue (the PTP.REPLY.QUEUE.LOCAL queue). The second program acts as the replier and it starts reading messages from a queue (the PTP.QUEUE.LOCAL queue). Whenever a message is put onto that queue, it sends a generic response to the PTP.REPLY.QUEUE.LOCAL queue.

The request program shown in Example 3-10 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the request (PTP.QUEUE.LOCAL) queue for output
- ▶ Open the reply (PTP.REPLY.QUEUE.LOCAL) queue for input
- ▶ Prepare the request message to be sent
- ▶ Send the request message to the opened queue
- ▶ Set the data buffer for the incoming message
- ▶ Assign the correlId used to identify the reply message
- ▶ Wait for the reply message in the reply queue
- ▶ Show the received reply message data.
- ▶ Close the queues
- ▶ Disconnect from the queue manager

Example 3-10 Request program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* includes for MQI */
#include <cmqc.h>

#define QMGR_NAME" SAMPLE.QMGR1.ITSOE"
#define REQUEST_Q_NAME" PTP.QUEUE.LOCAL"
#define REPLY_Q_NAME" PTP.REPLY.QUEUE.LOCAL"

int main(int argc, char **argv) {
    /* Declare MQI structures needed */
    MQOD      od = {MQOD_DEFAULT};    /* Object Descriptor */
    MQMD      md = {MQMD_DEFAULT};    /* Message Descriptor */
    MQMD      gmd = {MQMD_DEFAULT};    /* Message Descriptor */
    MQPMO     pmo = {MQPMO_DEFAULT};  /* put message options */
    MQGMO     gmo = {MQGMO_DEFAULT};  /* get message options */

    MQHCONN   Hcon;                   /* connection handle */
    MQHOBJ     Hobj_request;           /* request object handle */
    MQHOBJ     Hobj_reply;             /* reply 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;                /* MQCONN reason code */
    MQLONG     buflen;                 /* buffer length */
    MQLONG     messlen;                /* message length */
    char       buffer[100];            /* message buffer */
    char       QMName[50];             /* queue manager name */

    printf("Request/Reply Sample");
```

```

// Connect to queue manager
strcpy(QMName, QMGR_NAME);
MQCONN(QMName,&Hcon,&CompCode,&CReason);

/* report reason and stop if it failed */
if (CompCode == MQCC_FAILED) {
    printf("MQCONN ended with reason code %ld\n", CReason);
    exit( (int) CReason );
}

// Set the request queue name in the object descriptor.
strcpy(od.ObjectName, REQUEST_Q_NAME );
printf("The request queue is %s\n", od.ObjectName);

// Open the request message queue for output
O_options = MQOO_OUTPUT /* open queue for output */
            + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon,&od,O_options,&Hobj_request,&CompCode,&Reason);

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

OpenCode = CompCode;

// Set the reply queue name in the object descriptor.
strcpy(od.ObjectName, REPLY_Q_NAME );
printf("The reply queue is %s\n", od.ObjectName);

// Open the reply message queue for input
O_options = MQOO_INPUT_SHARED /* open queue for output */
            + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon,&od,O_options,&Hobj_reply,&CompCode,&Reason);

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

// If any of them fail to open then the program exits
OpenCode |= CompCode;

if (OpenCode == MQCC_FAILED) {
    printf("Unable to open queue for output\n");
} else {

```

```

memcpy(md.Format,          /* character string format          */
       MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

// Set message descriptor and put message options on version 2
md.Version = MQMD_VERSION_2;
pmo.Version = MQPMO_VERSION_2;
gmo.Version = MQMD_VERSION_2;

// Request a new messageId and correlID for this message
pmo.Options |= MQPMO_NEW_MSG_ID;
pmo.Options |= MQPMO_NEW_CORREL_ID;

strcpy(buffer, "This is a simple Request/Reply sample");
messlen = strlen(buffer);

MQPUT(Hcon, Hobj_request, &md, &pmo, messlen, buffer, &CompCode, &Reason);

/* report reason, if any */
if (Reason != MQRC_NONE) {
    printf("MQPUT ended with reason code %ld\n", Reason);
}

if (OpenCode != MQCC_FAILED) {

    gmo.Options = MQGMO_WAIT          /* wait for new messages          */
                 + MQGMO_CONVERT;    /* convert if necessary          */
    gmo.WaitInterval = MQWI_UNLIMITED; /* 15 second limit for wait */
    gmo.MatchOptions = MQMO_MATCH_CORREL_ID;

    buflen = sizeof(buffer)-1;
    memcpy(gmd.CorrelId, md.CorrelId, sizeof(md.CorrelId));

    MQGET(Hcon, Hobj_reply, &gmd, &gmo, buflen, buffer,
          &messlen, &CompCode, &Reason);

    // Close the target queue (if it was opened)
    C_options = 0;                      /* no close options          */
    MQCLOSE(Hcon, &Hobj_request, C_options, &CompCode, &Reason);

    /* report reason, if any          */
    if (Reason != MQRC_NONE) {
        printf("MQCLOSE ended with reason code %ld\n", Reason);
    }
}

}

// Disconnect from MQM if not already disconnected
if (CReason != MQRC_ALREADY_CONNECTED) {

```

```

MQDISC(&Hcon,&CompCode,&Reason);

    /* report reason, if any */
    if (Reason != MQRC_NONE) {
        printf("MQDISC ended with reason code %ld\n", Reason);
    }
}

printf("Request/Reply Sample end\n");
return(0);
}

```

The reply program shown in Example 3-11 on page 68 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the request (PTP.QUEUE.LOCAL) queue for input
- ▶ Open the reply (PTP.REPLY.QUEUE.LOCAL) queue for output
- ▶ Set the data buffer for the incoming message
- ▶ Wait for the request message in the request queue
- ▶ Show the received request message data.
- ▶ Prepare the reply message to be sent
- ▶ Assign the correlId used to identify the reply message
- ▶ Send the reply message to the opened queue
- ▶ Close the queues
- ▶ Disconnect from the queue manager

Example 3-11 Reply program

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* includes for MQI */
#include <cmqc.h>

#define QMGR_NAME" SAMPLE.QMGR1.ITSOE"
#define REQUEST_Q_NAME" PTP.QUEUE.LOCAL"
#define REPLY_Q_NAME" PTP.REPLY.QUEUE.LOCAL"

int main(int argc, char **argv) {
    /* Declare MQI structures needed */
    MQOD    od = {MQOD_DEFAULT};    /* Object Descriptor */
    MQMD    md = {MQMD_DEFAULT};    /* Message Descriptor */
    MQMD    gmd = {MQMD_DEFAULT};    /* Message Descriptor */
    MQPMO    pmo = {MQPMO_DEFAULT};    /* put message options */
    MQGMO    gmo = {MQGMO_DEFAULT};    /* get message options */

    MQHCONN Hcon;    /* connection handle

```

```

MQHOBJ  Hobj_request;          /* request object handle      */
MQHOBJ  Hobj_reply;           /* reply 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;              /* MQCONN reason code        */
MQLONG  buflen;               /* buffer length              */
MQLONG  messlen;              /* message length             */
char     buffer[100];          /* message buffer             */
char     QMName[50];           /* queue manager name         */

printf("Request/Reply Sample");

// Connect to queue manager
strcpy(QMName, QMGR_NAME );
MQCONN(QMName,&Hcon,&CompCode,&CReason);

/* report reason and stop if it failed */
if (CompCode == MQCC_FAILED) {
    printf("MQCONN ended with reason code %ld\n", CReason);
    exit( (int) CReason );
}

// Set the request queue name in the object descriptor.
strcpy(od.ObjectName, REQUEST_Q_NAME );
printf("The request queue is %s\n", od.ObjectName);

// Open the request message queue for input
O_options = MQOO_INPUT_SHARED /* open queue for output      */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon,&od,O_options,&Hobj_request,&CompCode,&Reason);

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

OpenCode = CompCode;

// Set the reply queue name in the object descriptor.
strcpy(od.ObjectName, REPLY_Q_NAME );
printf("The request queue is %s\n", od.ObjectName);

// Open the reply message queue for output
O_options = MQOO_OUTPUT /* open queue for output      */
           + MQOO_FAIL_IF_QUIESCING; /* but not if MQM stopping */
MQOPEN(Hcon,&od,O_options,&Hobj_reply,&CompCode,&Reason);

```

```

/* report reason, if any; stop if failed */
if (Reason != MQRC_NONE) {
    printf("MQOPEN ended with reason code %ld\n", Reason);
}

// If any of them fail to open then the program exits
OpenCode |= CompCode;

if (OpenCode == MQCC_FAILED) {
    printf("Unable to open queue for output\n");
} else {

    // Set message descriptor and put message options on version 2
    md.Version = MQMD_VERSION_2;
    gmd.Version = MQMD_VERSION_2;
    pmo.Version = MQPMO_VERSION_2;
    gmo.Version = MQMD_VERSION_2;

    gmo.Options = MQGMO_WAIT          /* wait for new messages */
                + MQGMO_CONVERT;    /* convert if necessary */
    gmo.WaitInterval = MQWI_UNLIMITED; /* 15 second limit for wait */
    gmo.MatchOptions = MQMO_MATCH_CORREL_ID;

    buflen = sizeof(buffer)-1;

MQGET(Hcon,Hobj_request,&gmd,&gmo,buflen,buffer,&messlen,&CompCode,&Reason);

/* report reason, if any */
if (Reason != MQRC_NONE) {
    printf("MQPUT ended with reason code %ld\n", Reason);
}

if (OpenCode != MQCC_FAILED) {

    // Terminating the message buffer
    buffer[messlen+1] = 0;
    printf("The request message is:%s",buffer);

    // Request a new messageId and correlID for this message
    memcpy(md.Format,          /* character string format */
           MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

    pmo.Options |= MQPMO_NEW_MSG_ID;

    // Setting the correlId as the message selector

```

```

memcpy(md.CorrelId, gmd.CorrelId, sizeof(md.CorrelId));
strcpy(buffer, "This is a simple reply message");
messlen = strlen(buffer);

MQPUT(Hcon, Hobj_reply, &md, &pmo, messlen, buffer, &CompCode, &Reason);

// Close the target queue (if it was opened)
C_options = 0; /* no close options */
MQCLOSE(Hcon, &Hobj_request, C_options, &CompCode, &Reason);

/* report reason, if any */
if (Reason != MQRC_NONE) {
    printf("MQCLOSE ended with reason code %ld\n", Reason);
}
}

// Disconnect from MQM if not already disconnected
if (CReason != MQRC_ALREADY_CONNECTED) {
    MQDISC(&Hcon, &CompCode, &Reason);

    /* report reason, if any */
    if (Reason != MQRC_NONE) {
        printf("MQDISC ended with reason code %ld\n", Reason);
    }
}

printf("Request/Reply Sample end\n");
return(0);
}

```

3.8.2 The publish/subscribe pattern

The publish/subscribe pattern has two major components, as explained in 1.1.5, “Publish/subscribe” on page 7.

- ▶ The publisher: This component is the one that actually publishes the topics in the broker stream.
- ▶ The subscriber: This component represent a client of the publisher or publishers. It subscribes to one or many topics, and waits for any publication on these topics to be sent to it by the broker.

We explore these two components with a simple example where a publisher program publishes some data in a given topic and any number of subscribers receive that data and show it in the standard output.

Publisher

The publisher program presented here is divided into three functions:

- ▶ The BuildMQRFHeader function
This function constructs an MQRFH data structure and appends the required value/pair at the end of this structure.
- ▶ The PutPublication function
This function is responsible for sending the actual publication commands to the broker using the stream queue.
- ▶ The main function
This function constructs the publication message and sends it to the stream queue.

Example 3-12 shows the BuildMQRFHeader function described above. This code was taken from the C samples that come with the Publish/Subscribe SupportPac.

Example 3-12 The BuildMQRFHeader function

```
void BuildMQRFHeader( PMQBYTE   pStart
                     , PMQLONG   pDataLength
                     , MQCHAR     TopicType[] )
{
    PMQRFH   pRFHeader = (PMQRFH)pStart;
    PMQCHAR   pNameValueString;
    /*****
    /* Clear the buffer before we start (initialize to nulls).    */
    /*****
    memset((PMQBYTE)pStart, 0, *pDataLength);

    /*****
    /* Copy the MQRFH default values into the start of the buffer.    */
    /*****
    memcpy( pRFHeader, &DefaultMQRFH, (size_t)MQRFH_STRUC_LENGTH_FIXED);

    /*****
    /* Set the format of the user data to be MQFMT_STRING. Even though */
    /* some of the publications use a structure to pass user data, the */
    /* data within this structure is entirely MQCHAR and can be        */
    /* treated as MQFMT_STRING by the data conversion routines.        */
    /*****
    memcpy( pRFHeader->Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

    /*****
    /* As we have user data following the MQRFH we must set the CCSID */
    /* of the user data in the MQRFH for data conversion to be able to */
    /* be performed by the queue manager. As we do not currently know */
```



```

/* the CCSID that we are running in we can tell MQSeries that the */
/* data that follows the MQRFH is in the same CCSID as the MQRFH. */
/* The MQRFH will default to the CCSID of the queue manager      */
/* (MQCCSI_Q_MGR), so the user data will also inherit this CCSID. */
/*****
pRFHeader->CodedCharSetId = MQCCSI_INHERIT;

/*****
/* Start the NameValueString directly after the MQRFH structure. */
/*****
pNameValueString = (MQCHAR *)pRFHeader + MQRFH_STRUC_LENGTH_FIXED;

/*****
/* Add the command to the start of the NameValueString, this must */
/* always be the first MQPS name token in the string.              */
/*****
strcpy(pNameValueString, MQPS_COMMAND_B);
strcat(pNameValueString, MQPS_PUBLISH);

/*****
/* Add the publication options and topic to the NameValueString. */
/* We specify 'no registration' because neither sample application */
/* is concerned with who is currently publishing, it also allows */
/* us not to specify an identity queue (we are also publishing */
/* datagrams so no replies will be sent either) which means that */
/* we do not have to define a queue for this sample to use.      */
/*****
strcat(pNameValueString, MQPS_PUBLICATION_OPTIONS_B);
strcat(pNameValueString, MQPS_NO_REGISTRATION);

strcat(pNameValueString, MQPS_TOPIC_B);
strcat(pNameValueString, TOPIC_PREFIX);
strcat(pNameValueString, TopicType);

/*****
/* Any user data that follows the NameValueString should start on */
/* a word boundary. To ensure all platforms are satisfied we align */
/* to a 16 byte boundary.                                          */
/* As the NameValueString has been null terminated (by using */
/* strcat) any characters between the end of the string and the */
/* next 16 byte boundary will be ignored by the broker, but if the */
/* message is to be data converted we advise any extra characters */
/* are set to nulls ('\0') or blanks (' '). In this sample we have */
/* initialized the whole message block to nulls before we started */
/* so all extra characters will be nulls by default.              */
/*****
*pDataLength = MQRFH_STRUC_LENGTH_FIXED
               + ((strlen(pNameValueString)+15)/16)*16;
pRFHeader->StrucLength = *pDataLength;

```

```
}
```

The PutPublication function shown in Example 3-13 follows the logical flow below:

- ▶ Set the message format to MQFMT_RF_HEADER
- ▶ Set the message type to MQMT_DATAGRAM
- ▶ Set the message persistence
- ▶ Set the MQPMO_NEW_MSG_ID option to request a new messageId for the message that is being sent
- ▶ Set the message buffer using the useFullBuffer method
- ▶ Send the message and returns true if successful

Example 3-13 The PutPublication function

```
void PutPublication( MQHCONN  hConn
                   , MQHOBJ   hObj
                   , PMQBYTE   pMessage
                   , MQLONG     messageLength
                   , PMQLONG    pCompCode
                   , PMQLONG    pReason )
{
    MQPMO  pmo = { MQPMO_DEFAULT };
    MQMD    md  = { MQMD_DEFAULT };

    /*****
    /* Set the md for a datagram MQRFH message.
    *****/
    memcpy(md.Format, MQFMT_RF_HEADER, (size_t)MQ_FORMAT_LENGTH);
    md.MsgType = MQMT_DATAGRAM;
    md.Persistence = MQPER_PERSISTENT;
    pmo.Options |= MQPMO_NEW_MSG_ID;

    // MQPUT the message to the queue.
    MQPUT( hConn, hObj, &md, &pmo, messageLength, pMessage, pCompCode, pReason );
}
```

The main function shown in Example 3-14 follows the logical flow below:

- ▶ Connect to the queue manager
- ▶ Open the stream queue for output
- ▶ Allocate the message data buffer
- ▶ Build the MQRFH Header structure using the BuildMQRFHeader function
- ▶ Append the publication data to the end of the MQRFH Header structure
- ▶ Put the publication in the stream queue using the PutPublication function

```
int main(int argc, char **argv) {
    MQHCONN      hConn = MQHC_UNUSABLE_HCONN;
    MQHOBJ       hObj  = MQHO_UNUSABLE_HOBJ;
    MQLONG       CompCode;
    MQLONG       Reason;
    MQOD         od = { MQOD_DEFAULT };
    MQLONG       Options;
    PMQBYTE      pMessageBlock = NULL;
    MQLONG       messageLength;
    char         QMName[MQ_Q_MGR_NAME_LENGTH+1] = "";
    char         text[] = "HELLO WORLD";
    MQLONG       ConnReason;

    strcpy(QMName, QMGR_NAME );

    /******
    /* Connect to the queue manager.
    /******
    MQCONN( QMName
        , &hConn
        , &CompCode
        , &ConnReason );
    if( CompCode == MQCC_FAILED ) {
        printf("MQCONN failed with CompCode %d and Reason %d\n",
            CompCode, ConnReason);
    }

    if( CompCode == MQCC_OK ) {
        strncpy(od.ObjectName, STREAM, (size_t)MQ_Q_NAME_LENGTH);
        Options = MQOO_OUTPUT + MQOO_FAIL_IF_QUIESCING;
        MQOPEN( hConn
            , &od
            , Options
            , &hObj
            , &CompCode
            , &Reason );
        if( CompCode != MQCC_OK )
        {
            printf("MQOPEN failed to open \"%s\"\nwith CompCode %d and RC %d\n",
                od.ObjectName, CompCode, Reason);
        }
    }

    if( CompCode == MQCC_OK ) {
        messageLength = DEFAULT_MESSAGE_SIZE;
        pMessageBlock = (PMQBYTE)malloc(messageLength);
```

```

        BuildMQRFHeader( pMessageBlock
                        , &messageLength
                        , "TEST");

        strcpy(pMessageBlock+messageLength,text);
        messageLength += strlen(text);
        PutPublication( hConn
                        , hObj
                        , pMessageBlock
                        , messageLength
                        , &CompCode
                        , &Reason );
    }
}

```

Subscriber

The subscriber program presented here is basically divided into four functions:

- ▶ The BuildMQRFHeader function

This function constructs an MQRFH data structure and appends the required value/pair at the end of this structure.

- ▶ The CheckForResponse function

This function waits for a subscription acknowledgment from the broker, and validates that the subscription has been successfully accepted.

- ▶ The PubSubCommand function

This function creates and sends a publish/subscribe command to the broker and checks for the broker response using the CheckForResponse function.

- ▶ The main function

This function provides two possible behaviors, depending on the number of parameters used when calling this program:

- If two parameters are specified, the first parameter is taken as the client queue where the broker will send the publications for the client, and the second parameter is used as the correlationId used to identify the publication of that specific client.

The main function registers the client to the topic and starts an infinite loop getting publications.

- If a third parameter is specified, the main function de-registers the client from the topic and finishes.

The CheckResponse function uses the PrintNameValueString function, which basically prints the name/value pairs to the screen. This function is not shown here but is available in the additional materials that may be downloaded from the IBM Redbooks Web site.

Example 3-15 shows the BuildMQRFHeader function described above. This code was taken from the C samples that come with the Publish/Subscribe SupportPac.

Example 3-15 The BuildMQRFHeader (Subscriber example)

```

void BuildMQRFHeader( PMQBYTE   pStart
                     , PMQLONG  pDataLength
                     , PMQCHAR   pCommand
                     , MQLONG    regOptions
                     , MQLONG    pubOptions
                     , PMQCHAR   pTopic ) {
    PMQRFH   pRFHeader = (PMQRFH)pStart;
    PMQCHAR   pNameValueString;

    /******
    /* Clear the buffer before we start (initialize to nulls).      */
    /******
    memset((PMQBYTE)pStart, '\0', *pDataLength);

    /******
    /* Copy the MQRFH default values into the start of the buffer.  */
    /******
    memcpy( pRFHeader, &DefaultMQRFH, (size_t)MQRFH_STRUC_LENGTH_FIXED);

    /******
    /* Start the NameValueString directly after the MQRFH structure. */
    /******
    pNameValueString = (MQCHAR *)pRFHeader + MQRFH_STRUC_LENGTH_FIXED;

    /******
    /* Add the command to the start of the NameValueString, this must */
    /* always be the first MQPS name token in the string.              */
    /******
    strcpy(pNameValueString, MQPS_COMMAND_B);
    strcat(pNameValueString, pCommand);

    /******
    /* If registration options were supplied, add them to the string. */
    /* For ease of implementation we insert the decimal representation */
    /* of the options into the string as opposed to the character      */
    /* strings supplied for each option.                                */
    /******
    if( regOptions != 0 ) {
        strcat(pNameValueString, MQPS_REGISTRATION_OPTIONS_B);

```

```

        sprintf(pNameValueString, "%s %d", pNameValueString, regOptions);
    }

    /*****
    /* If publication options were supplied add them to the string.
    /* For ease of implementation we insert the decimal representation
    /* of the options into the string as opposed to the character
    /* strings supplied for each option.
    *****/
    if( pubOptions != 0 ) {
        strcat(pNameValueString, MQPS_PUBLICATION_OPTIONS_B);
        sprintf(pNameValueString, "%s %d", pNameValueString, pubOptions);
    }

    /*****
    /* Add the stream name to the NameValueString (optional for
    /* publications).
    *****/
    strcat(pNameValueString, MQPS_STREAM_NAME_B);
    strcat(pNameValueString, STREAM_QUEUE);

    /*****
    /* Add the topic to the NameValueString.
    *****/
    strcat(pNameValueString, MQPS_TOPIC_B);
    strcat(pNameValueString, pTopic);

    /*****
    /* Any user data that follows the NameValueString should start on
    /* a word boundary. To ensure all platforms are satisfied we align
    /* to a 16 byte boundary.
    /* As the NameValueString has been null terminated (by using
    /* strcat) any characters between the end of the string and the
    /* next 16 byte boundary will be ignored by the broker, but if the
    /* message is to be data converted we advise any extra characters
    /* are set to nulls ('\0') or blanks (' '). In this sample we have
    /* initialized the whole message block to nulls before we started
    /* so all extra characters will be nulls by default.
    *****/
    *pDataLength = MQRFH_STRUC_LENGTH_FIXED
        + ((strlen(pNameValueString)+15)/16)*16;
    pRFHeader->StrucLength = *pDataLength;
}

```

The CheckForResponse function shown in Example 3-16 follows the logical flow below:

- ▶ Prepare the message buffer
- ▶ Set the message options such as correlId and wait interval options
- ▶ Wait for a response from the broker
- ▶ If the response is not received, return a fail value
- ▶ If the response is received, validate the format of the response message
- ▶ Then extract the MQRFH Header structure from the message and validate the completion code
- ▶ Show the response to the user if the subscription was not accepted

Example 3-16 The CheckForResponse function

```

void CheckForResponse( MQHCONN  hConn
                      , MQHOBJ   hObj
                      , PMQMD    pMd
                      , PMQBYTE  pMessageBlock
                      , MQLONG    blockSize
                      , PMQLONG  pCompCode
                      , PMQLONG  pReason )
{
    MQGMO    gmo = { MQGMO_DEFAULT };
    MQLONG    messageLength;
    PMQRFH    pMQRFHeader;
    PMQCHAR    pNameValueString;
    PMQCHAR    pInputNameValueString;
    ULONG      stringLength;

    /*****
    /* Wait for a response message to arrive on our subscriber queue, */
    /* the response's correlId will be the same as the messageId that */
    /* the original message was sent with (returned in the md from the */
    /* MQPUT) so match against this. */
    *****/
    gmo.Options = MQGMO_WAIT + MQGMO_CONVERT;
    gmo.WaitInterval = MAX_RESPONSE_TIME;
    gmo.Version = MQGMO_VERSION_2;
    gmo.MatchOptions = MQMO_MATCH_CORREL_ID;
    memcpy( pMd->CorrelId, pMd->MsgId, sizeof(MQBYTE24));
    memset( pMd->MsgId, '\0', sizeof(MQBYTE24));

    MQGET( hConn, hObj, pMd, &gmo, blockSize, pMessageBlock, &messageLength,
           pCompCode, pReason );

    if( *pCompCode != MQCC_OK ) {
        printf("MQGET failed with CompCode %d and Reason %d\n",
               *pCompCode, *pReason);
        if( *pReason == MQRC_NO_MSG_AVAILABLE )
            printf("No response sent by broker, check broker is running.\n");
    } else {
        /*****

```

```

/* Check that the message is in the MQRFH format. */
/*****
if( memcmp(pMd->Format, MQFMT_RF_HEADER, MQ_FORMAT_LENGTH) == 0 ) {
    /*****
    /* Locate the start of the NameValueString and its length. */
    /*****
    pMQRFHeader = (PMQRFH)pMessageBlock;
    pNameValueString = (PMQCHAR)(pMessageBlock
                                + MQRFH_STRUC_LENGTH_FIXED);
    stringLength = pMQRFHeader->StrucLength
                  - MQRFH_STRUC_LENGTH_FIXED;

    /*****
    /* The start of a response NameValueString is always in the */
    /* same format: */
    /* MQPSCCompCode x MQPSReason y MQPSReasonText string ... */
    /* We can scan the start of the string to check the CompCode */
    /* and reason of the reply. */
    /*****
    sscanf(pNameValueString, "MQPSCCompCode %d MQPSReason %d ",
           pCompCode, pReason);

    if( *pCompCode != MQCC_OK ) {
        /*****
        /* One possible error is acceptable, MQRCCF_NO_RETAINED_MSG, */
        /* which is returned from a Request Update when there is no */
        /* retained message on the broker. This is an allowable */
        /* error so we can continue as before. */
        /*****
        if( *pReason == MQRCCF_NO_RETAINED_MSG ) {
            *pCompCode = MQCC_OK;
            *pReason = MQRC_NONE;
        } else {
            /*****
            /* Otherwise, display the error message supplied with the */
            /* user data that was returned. This will be the original */
            /* commands NameValueString. */
            /*****
            /*****
            /* A response NameValueString is ALWAYS NULL terminated. */
            /* Therefore, we can use printf to display it (as it is a */
            /* string in the true sense of the word). We do not */
            /* necessarily generate NULL terminated NameValueStrings */
            /* so we use the PrintNameValueString function to display */
            /* the NameValueString returned with the message, if any */
            /* (most error responses do return the original */
            /* NameValueString as user data). */
            /*****
            printf("Error response returned :\n");
            printf(" %s\n",pNameValueString);

```



```

        if( messageLength != pMQRFHeader->StrucLength ) {
            printf("Original Command String:\n");
            pInputNameValueString =
                (PMQCHAR)(pMessageBlock + pMQRFHeader->StrucLength);
            PrintNameValueString(pInputNameValueString,
                                (messageLength - pMQRFHeader->StrucLength));
        }
    }
} else {
    /*****
    /* If the message is not in the MQRFH format we have the wrong */
    /* message. */
    *****/

    printf("Unexpected message format: %.8s\n", pMd->Format );
    *pCompCode = MQCC_FAILED;
}
}
}

```

The PubSubCommand function shown in Example 3-17 follows the logical flow listed below:

- ▶ Build the MQRFH Header structure using the BuildMQRFHeader function
- ▶ Set message format and type
- ▶ Specify the replyTo queue for this message
- ▶ Request a new messageId for this message
- ▶ Set the message buffer using the useFullBuffer method
- ▶ Assign the correlId for the request message (this correlId is going to be used by the broker to send the message back to the subscriber)
- ▶ Put the command in the broker control queue
- ▶ Check for the broker response using the CheckForResponse function

Example 3-17 The PubSubCommand function

```

void PubSubCommand( MQHCONN      hConn
                   , MQHOBJ      hBrokerObj
                   , MQHOBJ      hReplyObj
                   , MQCHAR       Command[]
                   , PMQCHAR      pTopic
                   , MQLONG       topicLength
                   , const MQBYTE *pCorrelId
                   , MQLONG       regOptions
                   , PMQLONG      pCompCode

```

```

        , PMQLONG      pReason )
{
    MQPMO    pmo = { MQPMO_DEFAULT };
    MQMD     md  = { MQMD_DEFAULT };
    MQLONG   messageLength;
    PMQBYTE  pMessageBlock = NULL;

    /******
    /* Allocate a block of storage to hold the Command message.      */
    /******
    messageLength = DEFAULT_MESSAGE_SIZE;
    pMessageBlock = (PMQBYTE)malloc(messageLength);
    if( pMessageBlock == NULL ) {
        printf("Unable to allocate storage\n");
        *pCompCode = MQCC_FAILED;
    } else {
        /******
        /* Define an MQRFH structure at the start of the allocated    */
        /* storage, fill in the required fields and generate the      */
        /* NameValueString that follows it.                          */
        /******
        BuildMQRFHeader( pMessageBlock
                        , &messageLength
                        , Command
                        , regOptions
                        , MQPUBO_NONE
                        , pTopic );

        /******
        /* Send the command as a request so that a reply is returned to */
        /* us on completion at the broker.                               */
        /******
        memcpy(md.Format, MQFMT_RF_HEADER, (size_t)MQ_FORMAT_LENGTH);
        md.MsgType = MQMT_REQUEST;
        /******
        /* Specify the subscriber's queue in the ReplyToQ of the MD.    */
        /* We have not put the subscriber's queue in the MQRFH          */
        /* NameValueString so the one in the ReplyToQ of the MD will be  */
        /* used as the identity of the subscriber.                      */
        /******
        memcpy( md.ReplyToQ, ReplyToQueueName, MQ_Q_NAME_LENGTH);
        pmo.Options |= MQPMO_NEW_MSG_ID;
        /******
        /* All commands sent use the correlId as part of their identity. */
        /******
        memcpy( md.CorrelId, pCorrelId , sizeof(MQBYTE24));

        /******
        /* Put the command message to the broker control queue.          */
        /******

```

```

/*****
MQPUT( hConn
    , hBrokerObj
    , &md
    , &pmo
    , messageLength
    , pMessageBlock
    , pCompCode
    , pReason );

if( *pCompCode != MQCC_OK )
    printf("MQPUT failed with CompCode %d and Reason %d\n",
        *pCompCode, *pReason);
else {
    /*****
    /* The put was successful; now wait for a response from the */
    /* broker to inform us if the command was accepted by the */
    /* broker. */
    /* We use our command storage block to receive the response */
    /* into to save on allocating extra storage. */
    /*****
    CheckForResponse( hConn
        , hReplyObj
        , &md
        , pMessageBlock
        , DEFAULT_MESSAGE_SIZE
        , pCompCode
        , pReason );
    }
    /*****
    /* Free the storage. */
    /*****
    free( pMessageBlock );
}
}

```

Example 3-18 shows the subscriber main function. This function follows the logical flow below:

- ▶ Connect to the queue manager
- ▶ Open the three required queues (Control, Stream and Client queues)
- ▶ If only two arguments are received:
 - Send the registration command to the queue using the PubSubCommand function
 - Go into a infinite loop waiting for publications to arrive in the client queue

- If a third argument is received, then it sends the de-registration command to the broker using the PubSubCommand function and finishes.

Example 3-18 The main function (Subscriber example)

```

int main(int argc, char **argv) {
    MQHCONN      hConn = MQHC_UNUSABLE_HCONN;
    MQHOBJ       hControlObj = MQHO_UNUSABLE_HOBJ;
    MQHOBJ       hStreamObj = MQHO_UNUSABLE_HOBJ;
    MQHOBJ       hSubscriberObj = MQHO_UNUSABLE_HOBJ;
    MQLONG       CompCode;
    MQLONG       Reason;
    MQLONG       ConnReason;
    MQOD         od = { MQOD_DEFAULT };
    MQGMO        gmo = { MQGMO_DEFAULT };
    MQMD         md = { MQMD_DEFAULT };
    MQLONG       Options;
    PMQBYTE      pMessageBlock = NULL;
    MQLONG       messageLength;
    MQCHAR32     OpenQueue[3];
    PMQHOBj      pHObj[3];
    MQLONG       queueCounter;
    MQCHAR32     subscriptionTopic;
    PMQRFH       pMQRFHeader;
    PMQCHAR      pNameValueString;
    PMQBYTE      pUserData;
    MQLONG       nameValueStringLength;
    MQBYTE24     EventCorrelId;
    char          QMName[MQ_Q_MGR_NAME_LENGTH+1] = "";

    strcpy(OpenQueue[0], CONTROL_QUEUE);
    pHObj[0] = &hControlObj;
    strcpy(OpenQueue[1], STREAM_QUEUE);
    pHObj[1] = &hStreamObj;
    strcpy(OpenQueue[2], argv[1]);
    pHObj[2] = &hSubscriberObj;

    strcpy(EventCorrelId,argv[2]);
    ReplyToQueueName = argv[1];
    strcpy(QMName, "SAMPLE.QMGR1.ITSOE");
    //strcpy(QMName, "ITSOG.QMGR1");

    MQCONN( QMName, &hConn, &CompCode, &ConnReason );

    if( CompCode == MQCC_FAILED ) {
        printf("MQCONN failed with CompCode %d and Reason %d\n",
            CompCode, ConnReason);
        printf("Usage: amqsres <QManager>\n");
    }
}

```

```

if( CompCode == MQCC_OK ) {
    for( queueCounter = 0
        ; (queueCounter < 3) && (CompCode == MQCC_OK)
        ; queueCounter++ ) {

        strncpy(od.ObjectName, OpenQueue[queueCounter],
            (size_t)MQ_Q_NAME_LENGTH);

        Options = MQOO_FAIL_IF QUIESCING;

        if( strcmp(OpenQueue[queueCounter], argv[1]) == 0 )
            Options += MQOO_INPUT_AS_Q_DEF;
        else
            Options += MQOO_OUTPUT;

        MQOPEN(hConn, &od, Options, pHObj[queueCounter], &CompCode, &Reason);
        if( CompCode != MQCC_OK ) {
            printf("MQOPEN failed to open \"%s\"\\nwith CompCode %d and RC %d\\n",
                od.ObjectName, CompCode, Reason);
            printf("Usage: amqsres <QManager>\\n");
        }
    }
}

if (argc > 3) {
    strcpy( subscriptionTopic, TOPIC_PREFIX);
    strcat( subscriptionTopic, "*");
    PubSubCommand( hConn
        , hControlObj
        , hSubscriberObj
        , MQPS_DEREGISTER_SUBSCRIBER
        , subscriptionTopic
        , strlen(subscriptionTopic)
        , EventCorrelId
        , MQREGO_CORREL_ID_AS_IDENTITY
        , &CompCode
        , &Reason );
} else {
    strcpy( subscriptionTopic, TOPIC_PREFIX);
    strcat( subscriptionTopic, "*");
    PubSubCommand( hConn
        , hControlObj
        , hSubscriberObj
        , MQPS_REGISTER_SUBSCRIBER
        , subscriptionTopic
        , strlen(subscriptionTopic)
        , EventCorrelId
        , MQREGO_CORREL_ID_AS_IDENTITY

```

```

        , &CompCode
        , &Reason );

if( CompCode == MQCC_OK ) {
    /******
    /* Allocate a block of memory for the publications to be
    /* loaded into by MQGET. We know the maximum size of a
    /* publication published by amqsgam so we can allocate a
    /* block large enough for any message we will receive.
    /******
    messageLength = DEFAULT_MESSAGE_SIZE;
    pMessageBlock = (PMQBYTE)malloc(DEFAULT_MESSAGE_SIZE);

    gmo.Options = MQGMO_WAIT + MQGMO_CONVERT;
    gmo.WaitInterval = MAX_WAIT_TIME;
    gmo.Version = MQGMO_VERSION_2;
    gmo.MatchOptions = MQMO_MATCH_CORREL_ID;
    memcpy( md.CorrelId, EventCorrelId,
            (size_t)MQ_CORREL_ID_LENGTH);

    while( CompCode == MQCC_OK ) {
        MQGET( hConn
            , hSubscriberObj
            , &md
            , &gmo
            , DEFAULT_MESSAGE_SIZE
            , pMessageBlock
            , &messageLength
            , &CompCode
            , &Reason );
        if( memcmp(md.Format, MQFMT_RF_HEADER, MQ_FORMAT_LENGTH) == 0 ) {
            /******
            /* Split the message data into the three important
            /* areas: the MQRFH header, the NameValueString that
            /* follows it and any user data following that.
            /******
            pMQRFHeader = (PMQRFH)pMessageBlock;
            pNameValueString = (PMQCHAR)(pMessageBlock
                + MQRFH_STRUC_LENGTH_FIXED);
            nameValueStringLength = pMQRFHeader->StrucLength
                - MQRFH_STRUC_LENGTH_FIXED;
            pUserData = pMessageBlock + pMQRFHeader->StrucLength;
            *(pMessageBlock + messageLength) = 0;
            printf("The publication received is: %s\n",pUserData);
        }
    }
}
}
}
}

```

In this chapter we have seen how the basic MQI can be used by application programmers to build applications. In the next chapter we discuss the use of the AMI.



Programming with AMI

This chapter is an overview of the Application Message Interface, what it is and how it can be used. Please refer to the *Application Message Interface Reference*, SC34-5604 for more detailed information.

4.1 Overview

The Application Message Interface (AMI) is a new addition to the existing MQSeries APIs. It provides programmers with a very simple interface that can be used to work with queue manager objects. With AMI, the programmer doesn't need to have in-depth knowledge of all the MQI calls, but can instead concentrate on the business logic of the application. This means fewer programming errors and more flexibility to address business and technology changes. The AMI reduces the amount of code required to write a new application.

There are separate calls for each programming pattern. For example, if the programmer wants to code a publish/subscribe application, he will have to use different calls from those used by someone who is coding a request/reply application. There are fewer structures but more verbs within a single function. Many functions are now part of the middleware layer where a set of policies defined by the enterprise are applied on the application's behalf. AMI has bindings for standard programming languages including Java, C, and C++.

AMI allows centralized control and flexible change management, high-level interface for point-to-point models, and publish/subscribe. A key feature of AMI is that it is independent from the transport and messaging infrastructure.

AMI can be used to send and receive messages in the following ways:

- ▶ Send-and-forget, where no reply is needed.
- ▶ Distribution list, where a message is sent to multiple destinations.
- ▶ Request/response, where a sending application needs a response to the request message.
- ▶ Publish/subscribe, where a broker manages the distribution of messages.

An example of a simple send-and-forget application is shown in Figure 4-1 on page 91.

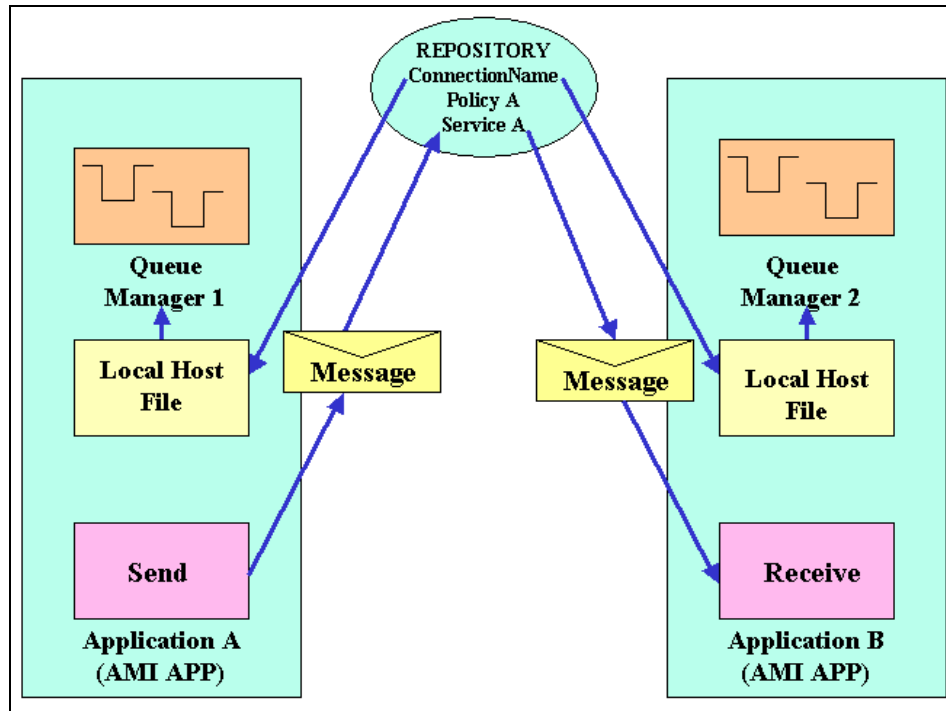


Figure 4-1 Example of an AMI send-and-forget application

Using AMI, the programmer generally deals with just three concepts:

- ▶ Message, or “what” is being exchanged. A message consists of:
 - Header information that identifies the message and its properties.
 - The body of the message, which contains the application data. This application data can be generated by the application or a message service API.

In an MQSeries application where MQI is being used, the message attributes are set up explicitly using the MQI, so the application programmer must understand their purpose. With AMI they are contained in the message object, or defined in a policy that is set up by the system administrator, so the programmer is not concerned with these details.

- ▶ Policy, or “how” the message is going to be handled. It contains information such as priority or confirmation of delivery. Any number of applications can use the same policy and any application can use more than one policy. IBM provides a suite of common or default policies and also provides an open policy handler framework that allows additional policies to be created by

third-party software vendors. If a policy has to be changed there is, in general, no need to change the applications that use it.

- Services, or “where” the message is going to be sent to or received from. In MQSeries terms, these represent queues, distribution lists, etc. A service may represent a single destination serviced by a single application, but it can also represent a list of destinations or a message broker. The service is normally defined in the repository, which specifies the mapping to real resources in the messaging network. Different types of services can be defined in the repository. Services are implicitly opened and closed by the AMI.

A repository provides definitions for services and policies. If the name of a service or policy is not found in the repository, or an AMI application does not have a repository, the definitions built into the AMI are used. Repository definitions are stored in a repository file in XML format. These definitions can be changed via an administration tool, which is only available on the Windows platform.

We can share the repository file between different platforms either by using standard file sharing facilities or by simply transferring the file. It is important to clarify that AMI applications can be run with or without a repository. If there is no repository, then the default values will be used.

In the administration tool, sender and receiver definitions are represented in the repository by a single definition called a service point. The following objects can be defined from the administration tool:

- Service points
- Distribution lists
- Publishers
- Subscribers
- Policies

Policies and services other than distribution lists can be created with or without a corresponding repository definition. Distribution lists can be created only after a service point has been created. To create a service or policy using the repository, it must contain a definition of the appropriate type with a name that matches the name specified by the application.

For example, to create a sender object named “ORDERS”, the repository must have a service point definition named “ORDERS”. Policies and services created with a repository have their contents initialized from the named repository definition. Policies and services created without a repository have their contents initialized from values defined in the default system definitions. The administration tool is only part of the MQSeries AMI SupportPac (MA0F) for

Windows NT and can only be installed on a machine running Windows NT 4.0 or Windows 2000. To start the administration tool, select **IBM MQSeries AMI -> IBM MQSeries AMI Administration Tool** or from Windows Explorer double-click the file `\amt\AMITool\amitool.bat`. Figure 4-2 shows the AMI administration tool.

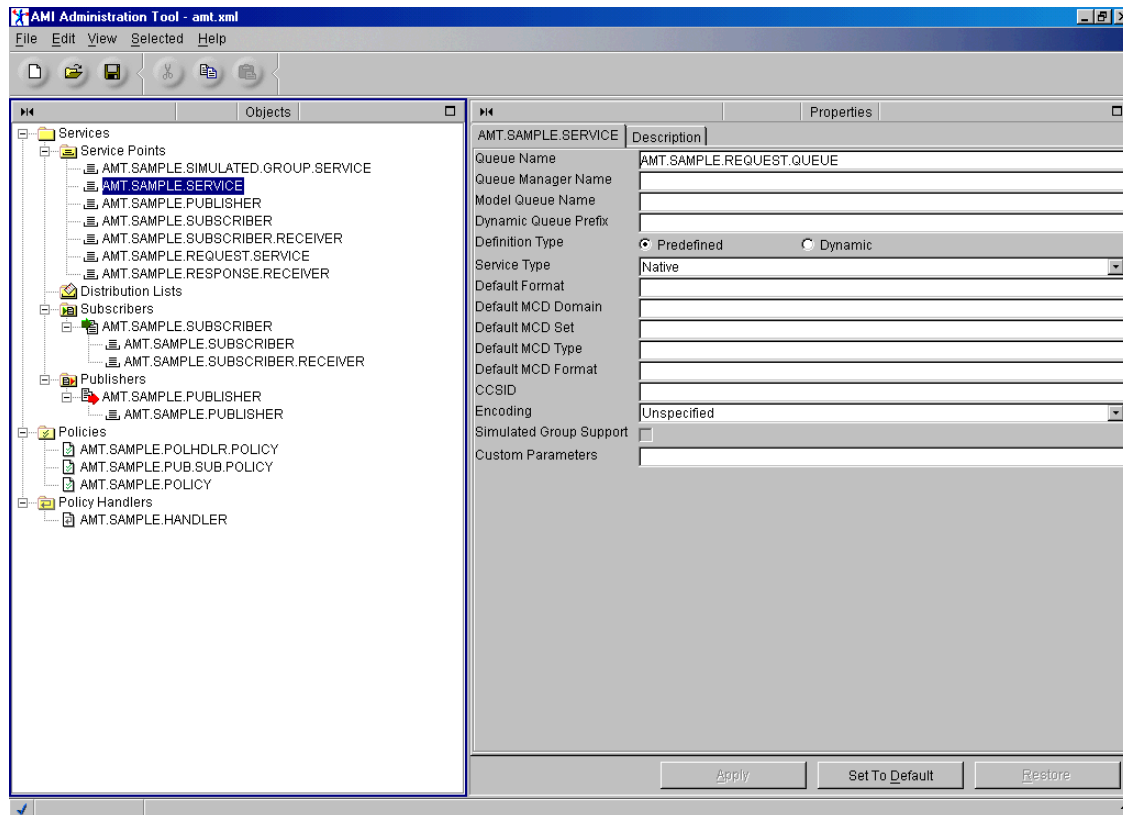


Figure 4-2 AMI administration tool

There are two separate roles within AMI: the administration role and the development role.

In the development role, the developer focuses on what information is going to be sent by using the resources provided by the administrator. This role doesn't require in-depth knowledge of MQSeries.

The administrator role is responsible for creating and defining services and policies in the repository. In other words, the administrator defines "where" and "how" the information is going to be sent. This role does require in-depth knowledge of MQSeries.

With AMI, connectivity code is reduced to specifying a service and a policy to be used when sending or receiving messages. Policies and services are defined in a repository using the administration tool, so they can be modified by the administrator without having any impact in the application. Using AMI, it is possible to exchange messages with one or more of the following:

- ▶ Another application that is using AMI.
- ▶ An application that is using any of the different MQSeries APIs (MQI, MQSeries Classes for Java, ActiveX, etc.).
- ▶ A message broker (MQSeries Publish/Subscribe or WebSphere MQ Integrator).

AMI simplifies the creation of publish/subscribe applications. Much of the configuration of the publish/subscribe environment is held in the repository that is referenced by the applications. AMI is recommended when the programmer is looking for a simple API that doesn't require any in-depth knowledge of MQSeries. AMI is available via a SupportPac (MA0F) and it can be downloaded from IBM's Web site at:

<http://www.ibm.com/software/ts/mqseries/txppacs/>

Note: MQSeries Publish/Subscribe SupportPac (MA0C) must be installed before attempting to use AMI's publish/subscribe functions.

4.2 Platforms and languages

As mentioned in 4.1, "Overview" on page 90, AMI is available for C, C++ and Java. It can be used on the following platforms:

- ▶ Windows NT and Windows 2000
- ▶ AIX V4.3, or later
- ▶ Sun Solaris 2.6 or 2.7
- ▶ HP-UX V11.0
- ▶ AS/400 V4R4, or later

AMI is also available for COBOL but this is only limited to OS/390 V2R6 or later, CICS 4.1 and IMS V5.1.

AMI has two levels of procedural application programming:

- ▶ High-level: procedural C and COBOL (on OS/390 interface). The number of AMI functions is reduced because operations are implicit.
- ▶ Object level: Java and C++ class interface/object style C interface.

Note: The C high-level interface contains functions that cover the requirements of most applications. However, if extra functionality is needed, we can use the C object interface in combination with the high-level interface.

In addition to providing a simple interface to work with queue manager objects, AMI has a natural style for each programming language. Example 4-1 shows a send-and-forget application using the C API.

Example 4-1 Simple send-and-forget application written in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <amtc.h>
#include <time.h>

int main(void)
{

    /* Create a Session */
    hSession = amSesCreate( SAMPLE_SESSION_NAME, &compCode, &reason);
    hPol = amSesCreatePolicy( hSession, SAMPLE_POLICY_NAME, &compCode, &reason);
    hSender = amSesCreateSender( hSession, SAMPLE_SENDER_NAME, &compCode, &reason);
    success = amSesOpen( hSession, hPol, &compCode, &reason);
    success = amSndOpen( hSender, hPol, &compCode, &reason);
    success = amSndSend( hSender, hPol, AMH_NULL_HANDLE, AMH_NULL_HANDLE,
        strlen(sampleMsg), (unsigned char *)sampleMsg, AMH_NULL_HANDLE, &compCode,
        &reason );

    success = amSesDelete( &hSession, &compCode, &reason );

    endSample(EXIT_SUCCESS);
}
```

Example 4-2 shows the same send-and-forget application shown in Example 4-1, but using the Java API. Notice the natural style for each of the two programming languages.

Example 4-2 Simple send-and-forget written in Java

```
import java.util.*;
import com.ibm.mq.amt.*;

.
.
.
public void main()
{
    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession(SAMPLE_SESSION_NAME);
    myPolicy = mySession.createPolicy(SAMPLE_POLICY_NAME);
}
```

```

mySender = mySession.createSender(SAMPLE_SENDER_NAME);
mySendMSG = mySession.createMessage(SAMPLE_MESSAGE_NAME);
mySession.open(myPolicy);
mySender.open(myPolicy);
String sampleMessage = new String("Sample message");
mySendMSG.writeBytes(sampleMessage.getBytes());
mySender.send(mySendMSG);
mySender.close(myPolicy);
mySession.close(myPolicy);
}

```

4.3 Libraries and packages

Now that we know the different programming languages that can be used to write AMI applications, we must know the different libraries that are used to write AMI applications. If C is used to write the application, AMI provides a header file called `amtc.h`. This file contains all the functions, structures and constants used by AMI. This header file must be included in the application, which can be achieved by using the following statement:

```
#include <amtc.h>
```

Figure 4-1 shows the location of the `amtc.h` file on the different platforms where AMI is supported.

Table 4-1 Location of the AMI C header file

Operating System Platform	Location
AS/400	QMQAMI/H
UNIX (including AIX, HP-UX, and Solaris)	{MQSeries Directory}/amt/inc
Windows	{MQSeries Directory}\amt\include
OS/390	hlq.SCSQC370

Note: During compilation time, the file `amtc.h` must be accessible to the program.

For C++, AMI also provides another header file called `amtcpp.hpp`. This header file contains the functions, structures and constants for C++, just as the header file for C, `amtcpp.hpp`, must be included in the application. This can also be achieved by using the following statement:

```
#include<amtcpp.hpp>
```


Table 4-2 shows the location of amtcpp.hpp file on the different platforms where AMI is supported.

Table 4-2 Location of the AMI C++ header file

Operating System Platform	Location
AS/400	QM/QAMI/H
UNIX (including AIX, HP-UX, and Solaris)	{MQSeries Base Directory}/amt/inc
Windows	{MQSeries Base Directory}\amt\include

Note: During compilation time, the files amtc.h and amtcpp.hpp must be accessible to the program, even if amtcpp.hpp is the only header file in use.

If the developer wants to use the Java API, AMI provides a JAR file that contains all the classes comprising the AMI package for Java.

- ▶ Java package: com.ibm.mq.amt
- ▶ Java JAR file: com.ibm.mq.amt.jar

In order to use the AMI package for Java, it must be imported to the Java application by using the import statement:

```
import com.ibm.mq.amt.*;
```

Note: The JAR file must be part of the CLASSPATH environment variable (this must be done in both the environment in which the Java application is compiled and in the environment in which it is run).

Table 4-3 shows the location of the AMI JAR file on the different platforms where AMI is supported.

Table 4-3 Location of the AMI Java JAR file

Operating System Platform	Location
AS/400	/QIBM/ProdData/mqm/amt/Java/lib
UNIX (including AIX, HP-UX, and Solaris)	{MQSeries Base Directory}/java/lib
Windows	{MQSeries Base Directory}\java\lib

If COBOL is going to be used, AMI provides the following copybooks to assist the programmer with the coding of applications:

- ▶ AMTV: This copybook contains constants and return codes.
- ▶ AMTELEML: This copybook contains the definition of the AMELEM data structure that is used to pass name and value element information across AMI. This copybook provides the definitions without any initial values.

- **AMTELEMV:** This copybook has the same contents as the AMTELEML copybook but the only difference is that it provides the definitions with initial values.

These copybooks are installed in the MQSeries for OS/390 library hlq.SCSQCOBC. It is recommended that you use the copybook AMTELEMV to define an AMELEM structure. This provides default initial values, which ensures that the structId and version fields have valid values. If the values passed for these fields are not valid, AMI will reject them.

Table 4-4 shows the language compilers for C, COBOL, C++, and Java that are supported by AMI:

Table 4-4 Language compilers.

Operating system platform	Supported compilers
AIX	VisualAge for C++ Version 5.0 JDK 1.1.7 and later
OS/400	AS/400 Developer Kit for Java (5769JV1) ILE C for AS/400 (5769CX2) ILE C++ for AS/400 (5799GDW) Visual Age for C++ for OS/400 (5716CX4)
HP-UX	HP aC++ B3910B A.03.10 HP aC++ B3910B A.03.04 (970930) Support library JDK 1.1.7, and later
OS/390	OS/390 C/C++ Version 2 Release 6 or later IBM COBOL for OS/390 & VM Version 2 Release 1 or later IBM COBOL for MVS and VM Version 1 Release 2 or later
Sun Solaris	Workshop Compiler 4.2 (with Solaris 2.6) Workshop Compiler 5.0 (with Solaris 7) JDK 1.1.7, or later
Windows	Microsoft Visual C++ Version 6 JDK 1.1.7, or later

For further information on how to prepare and run the AMI applications in any of the supported languages, refer to the MQSeries Application Messaging Interface documentation.

4.4 Architectural model

AMI provides the following objects:

- ▶ Session: Used to create a new AMI session and also to control transactional support. This is the first object that needs to be initialized before creating and managing all other objects (messages, policies, receivers, etc.).
- ▶ Message: This object contains the message data and the message descriptor structure used when sending or receiving messages.
- ▶ Sender: This is a service that represents a destination to which messages are sent. The MQOD structure is encapsulated within this object.
- ▶ Receiver: This is a service that represents a source from which messages are received. The MQOD structure is also encapsulated within this class.
- ▶ Distribution List: Contains a list of sender services to provide a list of destinations.
- ▶ Publisher: Contains a sender service where the destination is a publish/subscribe broker.
- ▶ Subscriber: Contains a sender service to send subscribe and unsubscribe messages to a publish/subscribe broker, and a receiver service to receive publications from the broker.
- ▶ Policy: Defines how the message should be handled, including items such as priority, persistence and whether it is included in a unit of work.

Sender, receiver, distribution list, publisher and subscriber are all services. The only objects that are connected directly to the message transport layer are the senders and receivers. Therefore, distribution list and publisher objects contain senders while subscriber objects contain a sender and a receiver. Figure 4-3 shows the AMI architectural model and how each object interacts with each other.

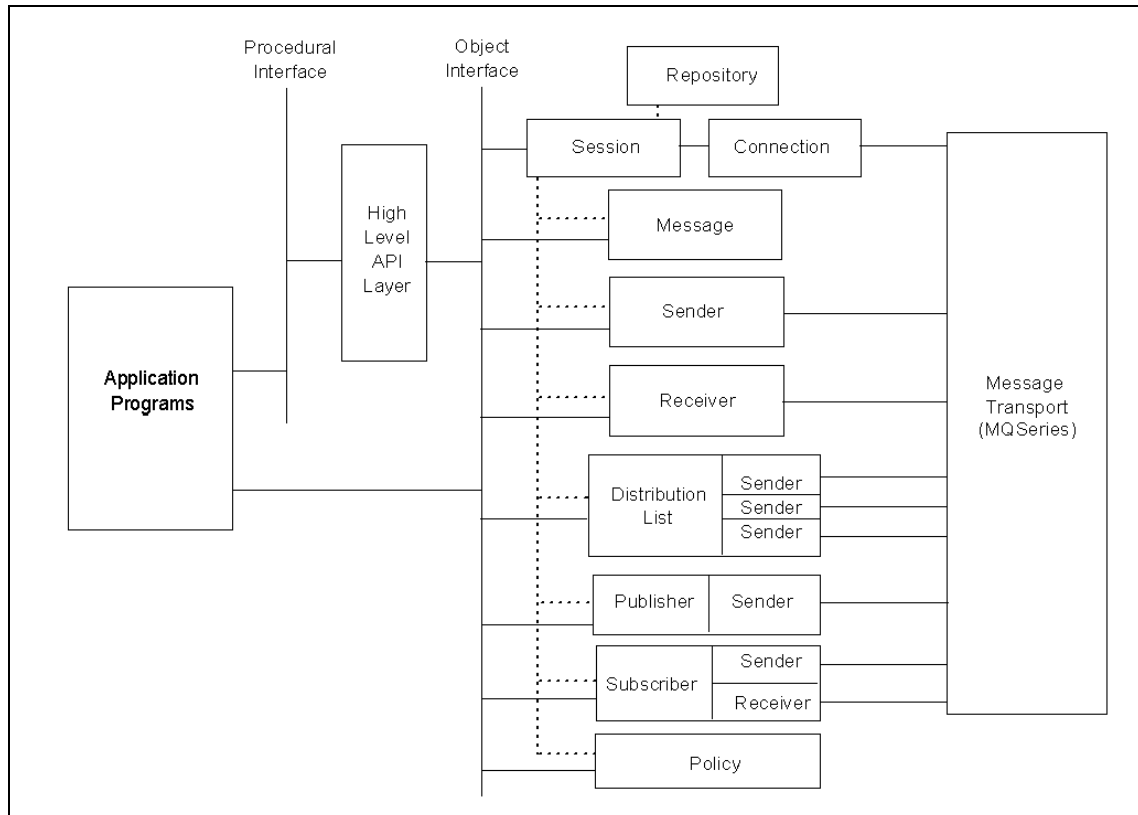


Figure 4-3 AMI architectural model

Message, service and policy objects are created and managed by a session object. This session object also provides the scope for a unit of work. The combination of a connection, sender and receiver objects provides the transport for the message.

The programmer can either use the attributes for message, service and policy objects provided with the system defaults, or he can use the attributes defined by the administrator that are stored in the repository. In addition to the objects mentioned above, AMI provides additional object definitions for C++ and Java.

These objects are:

- ▶ Session factory: This object is used to create session objects. Without a session factory, a session object can't be created.
- ▶ Helper and Exception objects.

4.5 Programming with AMI

In the following sections we will see how AMI applies the common steps used when working with queue manager objects that were discussed in Chapter 1, “Introduction and patterns” on page 3. The examples provided in this section were written using the Java interface. For information on how to issue the calls in any other programming language where AMI is supported, please refer to the Application Messaging Interface documentation.

4.5.1 Connecting to the queue manager

Just as in the other MQSeries APIs, we *must* connect to a queue manager before attempting to access any queue manager object. This can be achieved by using two AMI Java calls. First we need to create a new session factory object. Remember that in order to create a session object, there must be at least one session factory object (this only applies to C++ and Java). The command used to create a session factory is:

```
SessionFactoryObject = new AmSessionFactory(String factoryName)
```

factoryName is an optional parameter. This string is actually the directory where the repository and host files are located, and it can be the fully qualified directory that includes the path under which the files are located, for example C:\Program Files\MQSeries\amt. If this parameter is not specified, the value specified in the AMT_DATA_PATH environment variable is going to be used. This environment variable is normally set during the installation of the MQSeries AMI SupportPac (MA0F).

After creating the session factory object, we can define a new session object. This is done by using the createSession() method of the AmSessionFactory class.

```
mySessionFactory.createSession(String sessionName)
```

The sessionName parameter is a name that we can use to identify the session object. Once the session factory is initialized and the session object is created, the application is ready to work with MQSeries objects. In Chapter 3, “Programming with MQI” on page 23, we saw that in order to connect to a queue manager using the MQI API, the queue manager must be specified in the MQCONN call. In AMI, the queue manager name is read from the host file

(amthost.xml by default) located in {MQSeries Base Directory}/amt. Example 4-3 shows a sample host file used to connect to a queue manager called ITSOH. If no queue manager is specified in the defaultConnection tag, then the default queue manager will be used.

Example 4-3 Sample host file

```
<?xml version="1.0" encoding="UTF-8" ?>
<queueManagersNames defaultConnection="ITSOH"
connectionName1="queueManagerName1" connectionName2="queueManagerName2" />
```

Example 4-4 shows how to create a AmSessionFactory object and an AmSession object called ITSO.

Example 4-4 Creating the new objects

```
private AmSessionFactory      mySessionFactory = null;
private AmSession             mySession = null;

// -----
// Since we are not specifying a name for the AmSessionFactory,
// we expect the AMI files to be stored in the default
// location.
// -----
mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITSO");
```

Once the session is created, the order in which we create the other AMI objects does not matter. However, it is recommended that we initialize the objects in the following order:

1. Session
2. Policy
3. Sender/Receiver/Publisher/Subscriber/Distribution List
4. Message

4.5.2 Opening MQSeries objects

After creating session factory and session objects, the next step is to create/initialize the other AMI objects.

Creating a policy

First we need to create a policy object. This can be done by using the createPolicy() method of the AmSession class.

```
sessionObject.createPolicy(String policyName)
```

If the policyName that we specified matches a policy name that already exists in the repository, then the policy will be created using that repository definition. If it doesn't exist, then the policy is going to be created using the default values. In Example 4-5 we can see how to create AMT.SAMPLE.POLICY. Since AMT.SAMPLE.POLICY is already defined in the repository, then the policy is going to be created using the values already defined in the repository.

Example 4-5 Creating the policy object

```
public static void main()
{
    AmSessionFactory      mySessionFactory = null;
    AmSession             mySession = null;
    AmPolicy              myPolicy = null;

    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
    myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
}
```

After we have created the policy, the next step is to create the type of object depending on the design pattern that the application is going to use. This type of object can be:

- ▶ Senders and/or receivers (also known as service points)
- ▶ Publishers
- ▶ Subscribers
- ▶ Distribution lists

Creating a sender

In order to create a sender object, we use the createSender() method of the session object.

```
sessionObject.createSender(String senderName);
```

If the senderName that is specified matches a name that already exists in the repository, then the sender object will be created using the definition found in the repository. If it doesn't exist, then the sender will be created using the default values. Example 4-6 shows how to create a sender called AMT.SENDER.NAME.

Example 4-6 Creating a sender

```
AmSessionFactory      mySessionFactory = null;
AmSession             mySession = null;
AmPolicy              myPolicy = null;
```

```
mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
mySender = mySession.createSender("AMT.SENDER.NAME");
```

Creating a receiver

To create a receiver object, we use the `createReceiver()` method of the session object.

```
sessionObject.createReceiver(String receiverName);
```

If the `receiverName` that is specified matches a name that already exists in the repository, then the receiver object will be created using the definition found in the repository. If it doesn't exist, then the receiver will be created using the default values. Example 4-7 shows how to create a receiver called `AMT.RECEIVER.NAME`.

Example 4-7 Creating a receiver

```
AmSessionFactory mySessionFactory = null;
AmSession mySession = null;
AmPolicy myPolicy = null;
AmReceiver myReceiver = null;
```

```
mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
myReceiver = mySession.createReceiver("AMT.RECEIVER.NAME");
```

Creating a publisher

To create a publisher object, we use the `createPublisher()` method of the session object.

```
sessionObject.createPublisher(String publisherName);
```

If the `publisherName` that is specified matches a name that already exists in the repository, then the publisher object will be created using the definition found in the repository. If it doesn't exist, then the sender will be created using the default values. The service type for the sender and receiver points used by the publisher and subscriber must be defined in the repository as `MQRFH`. This causes an

MQRFH header containing publish/subscribe name/value elements to be added to a message when it is sent. For more information, please refer to the Application Messaging Interface documentation. Example 4-8 shows how to create a publisher called AMT.PUBLISHER.NAME.

Example 4-8 Creating a publisher

```
AmSessionFactory      mySessionFactory = null;
AmSession             mySession = null;
AmPolicy              myPolicy = null;
AmPublisher myPublisher = null;

mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
myPublisher = mySession.createPublisher("AMT.PUBLISHER.NAME");
```

Creating a subscriber

To create a subscriber object, we use the createSubscriber() method of the session object.

```
sessionObject.createSubscriber(String subscriberName);
```

If the subscriberName that we specified matches a name that already exists in the repository, then the subscriber will be created using the definition found in the repository. If it doesn't exist, then it will be created using the default values. Example 4-9 shows how to create a subscriber called AMT.SAMPLE.SUBSCRIBER.

Example 4-9 Creating a subscriber

```
public static void main()
{
    .
    .
    .
    AmSessionFactory      mySessionFactory = null;
    AmSession             mySession = null;
    AmPolicy              myPolicy = null;

    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
    myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
    mySubscriber = mySession.createSubscriber("AMT.SAMPLE.SUBSCRIBER");
}
```

Creating a distribution list

To create a distribution list object, we use the `createDistributionList()` method of the session object.

```
sessionObject.createDistributionList(String distributionlistName);
```

If the distribution list name that we specified matches a name that already exists in the repository, then the object will be created using the definition found in the repository. If it doesn't exist, then the distribution list will be created using the default values. Before we can use a distribution list, the administrator must first define sender services and then define these services as part of the distribution list. Example 4-10 shows how to create a distribution list.

Example 4-10 Creating a distribution list

```
public static void main()
{
    AmSessionFactory      mySessionFactory = null;
    AmSession             mySession = null;
    AmPolicy              myPolicy = null;

    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
    myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
    mySender = mySession.createDistributionList("AMT.DISTRIBUTION.LIST");
}
```

Figure 4-4 shows how a distribution list called `AMT.DISTRIBUTION.LIST` was defined in the repository. Notice that the objects that are part of the distribution list are also defined in the service points section.

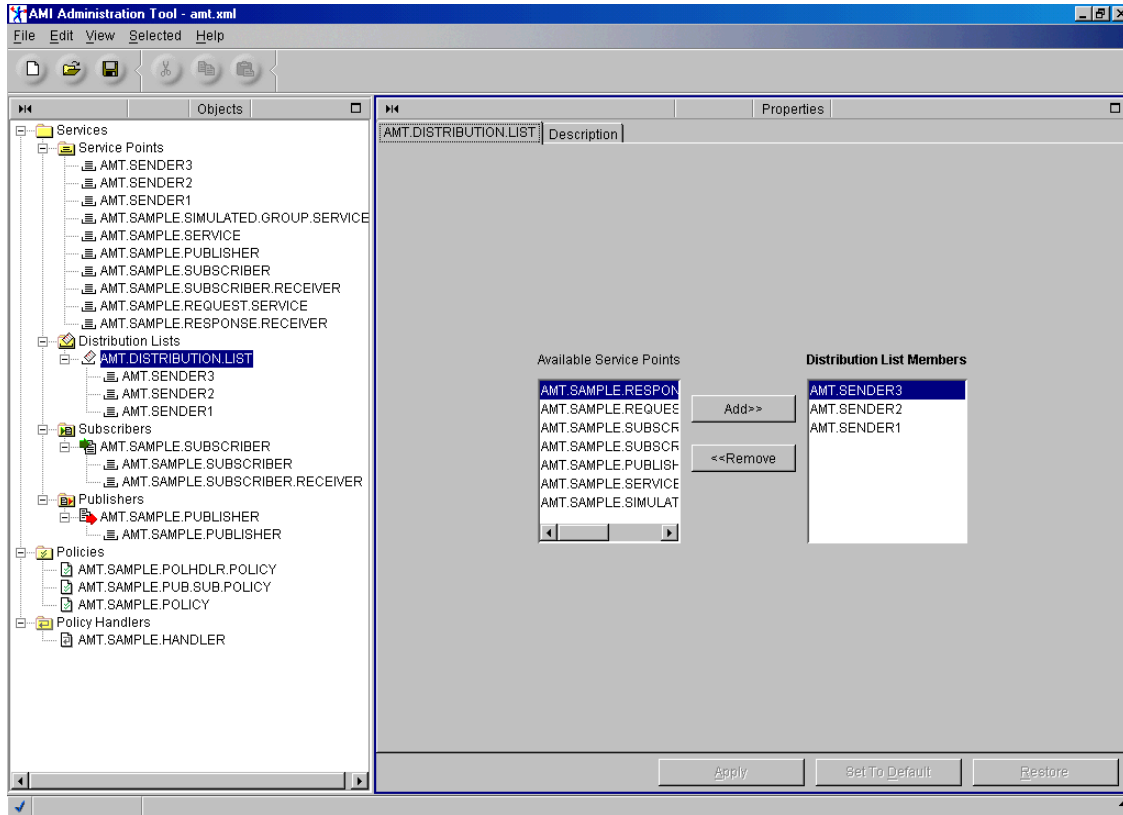


Figure 4-4 Definition of a distribution list called AMT.DISTRIBUTION.LIST

Creating a message

To create a message object, we use the `createMessage()` method of the `AmSession` class as follows.

```
AmSession.createMessage(String messageName)
```

Where `messageName` is a name that has some meaning to the application. Example 4-11 shows a sample code of how to create a message called `ITSO.SAMPLE.MESSAGE.NAME`.

Example 4-11 Creating a message object

```
public static void main()
{
    AmSessionFactory    mySessionFactory = null;
    AmSession           mySession = null;
```

```

AmPolicy          myPolicy = null;
AmPolicy          myPolicy = null;
AmMessage         mySendMSG = null;

mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITS0.SESSION.NAME");
myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
mySender = mySession.createSender("AMT.SENDER_NAME");
mySendMSG = mySession.createMessage("ITS0.SAMPLE.MESSAGE.NAME");
}

```

Now that we created the objects, the next step is to open the objects that were defined. This is achieved by using the `open()` method. For instance, if a sender object needs to be opened, then the `open()` method of the sender object will be used. All the open calls share one common parameter, which is the reference to the policy object. This reference was created using the `createPolicy()` method of the `AmSession` class as we saw in “Creating a policy” on page 102. If the policy is not specified, then the system default policy is used. The syntax for the open method is:

```

AmSession.open(AmPolicy policyObject);

AmSender.open(AmPolicy policyObject);

AmReceiver.open (AmPolicy policyObject);

AmPublisher.open(AmPolicy policyObject);

AmSubscriber.open(AmPolicy policyObject);

```

Example 4-12 shows how to open session, publisher, and receiver objects.

Example 4-12 Opening different types of objects

```

Open the session factory
Create the required objects (session, publisher, message, etc.)
/....

myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");

mySession.open(myPolicy);
myPublisher.open(myPolicy);
myRespReceiver.open(myPolicy);

```

4.5.3 Basic operations

In 4.5.2, “Opening MQSeries objects” on page 102, we saw how to create and open the different objects that are available. Now that the objects are created and initialized, we can proceed with the basic operations that can be performed against the MQSeries objects. These basic operations include getting messages, sending messages, publishing, and subscribing messages.

Before we can send or receive a message in MQI, we have to open a queue for either input or output. In MQI terms this is known as the open options (MQOO_INPUT_SHARED, MQOO_OUTPUT). In AMI these definitions are defined by the administrator and stored in the repository, so the programmer doesn’t have to explicitly define them when the queue manager objects are opened or as part of the call.

Sending messages

To send a message, we use the `send()` method of the sender class. Remember that before attempting to send a message, at least four objects must be created (session, policy, message, and sender). If a response is expected, then a receiver object and an additional message object will have to be created.

The message content is always sent in byte form (Java’s native form), so it is recommended that you use the `getBytes()` method (Java native method) to convert the message into an array of bytes before sending it. Once the message is converted, it then needs to be written to the message object by using the `writeBytes()` method of the sender class.

```
senderObject.send(AmMessage messageObject, AmReceiver receiverObject/  
AmMessage receivedMessage, AmPolicy policyObject)
```

Additionally, we can specify a policy object and/or a receiver or another message object. Remember that the policy is “how” the message is going to be handled. In other words, with a policy we can specify the priority, or what action should be taken if there is an error while attempting to deliver the message, etc.

If we require some type of response, such as a confirmation of delivery report or an acknowledgment from the remote application, we have to specify a receiver object or a message object, depending on what we want to do with the response. Example 4-13 shows how to send a sample message.

Example 4-13 Sending a sample message

```
public static void main()  
{  
    AmSessionFactory    mySessionFactory = null;  
    AmSession            mySession = null;
```

```

AmPolicy          myPolicy = null;
AmSender          mySender = null;
AmMessage         mySendMSG = null;

mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
mySender = mySession.createSender("AMT.SENDER_NAME");
mySendMSG = mySession.createMessage("ITSO.SAMPLE.MESSAGE.NAME");
String sampleMessage = new String("Sample message");
mySendMSG.writeBytes(sampleMessage.getBytes());
mySender.send(mySendMSG);
}

```

We can also send messages to multiple destinations simultaneously. This can be achieved by using the `send()` method of the `distributionList` object. Refer to 4.5.2, “Opening MQSeries objects” on page 102 for further information on how to create and open a distribution list. Example 4-14 shows an example of how to send a sample message to multiple destinations.

Example 4-14 Send a message to multiple destinations

```

public static void main()
{
AmSessionFactory    mySessionFactory = null;
AmSession           mySession = null;
AmPolicy            myPolicy = null;
AmDistributionList   myDistributionLst = null;
AmMessage           mySendMSG = null;

mySessionFactory = new AmSessionFactory();
mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
myDistributionLst = mySession.createDistributionList("ITSO.DISTRIBUTION.LIST");
mySendMSG = mySession.createMessage("ITSO.SAMPLE.MESSAGE.NAME");
String sampleMessage = new String("Sample message");
mySendMSG.writeBytes(sampleMessage.getBytes());
myDistributionList.send(mySendMSG);
}

```

Getting messages

To get a message we use the `receive()` method of the receiver class.

```

amReceiver.receive(AmMessage messageObject, AmSender senderObject,
AmMessage selectionmessageObject, AmPolicy policyObject);

```

There are four parameters in this call but only the `messageObject` is always required. The received message is going to be stored in this parameter. If a response is requested by the sending application, then we will have to specify a `senderObject`. The `policyObject` is used to specify things such as the wait interval, or whether or not data conversion is required. If we need to retrieve a specific message based on the correlation ID, then we will have to specify the `selectionmessageObject`. We mentioned before that when a message is sent, it has to be converted to an array of bytes, so when receiving a message we need to read that array of bytes.

In order to read the array of bytes we use the `readBytes()` method of the message class. Example 4-15 shows how to retrieve a message.

Example 4-15 Retrieving a message

```
public static void main()
{
    AmSessionFactory      mySessionFactory = null;
    AmSession              mySession = null;
    AmPolicy               myPolicy = null;
    AmReceiver             myReceiver = null;
    AmMessage              myReceiveMSG = null;
    AmMessage              mySendMSG = null;

    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
    myPolicy = mySession.createPolicy("ITSO.SAMPLE.POLICY");
    myReceiveMSG = mySession.createMessage("ITSO.SAMPLE.MESSAGE.NAME");
    myReceiver.receive(MyReceiveMSG);
    String sampleMessage = new
    String(MyReceiveMSG.readbytes(myReceiveMSG.getDataLength()));
}
```

Publishing messages

There are some steps that need to be done to publish a message. Once message and publisher objects are opened, we need to create a new topic for the message. This topic is just a word that describes the data that will be published. We create a topic using the `addTopic()` method of the message class.

```
messageObject.addTopic(String topicName)
```

The only parameter required is the `topicName`, which as we mentioned before is a word that describes the data that's going to be published.

After adding the topic, we can publish the message. To publish the message we use the `publish()` method of the publisher class.

```
publisherObject.publish(AmMessage messageObject, AmReceiver  
receiverObject, AmPolicy policyObject)
```

The `messageObject` is the object that contains the message that will be published. We can omit the `receiverObject` only if the policy doesn't specify implicit registration of the publisher. The `messageObject` should always be specified. If the `policyObject` is not specified, then the default definition will be used.

Example 4-16 shows sample code of how to publish a message. In this example the `policyObject` is not specified; therefore the default definition will be used. The default definition doesn't require an implicit registration of the publisher, so no `receiverObject` is required.

Example 4-16 Publishing a message

Create and open required objects (session, policy, message and publisher)

```
string pubMessage="SUNNY";  
  
mySendMSG.addTopic("Weather");  
mySendMSG.writeBytes(pubMessage.getBytes());  
  
myPublisher.publish(mySendMSG);
```

Subscribing messages

For subscribe messages, the application needs to send a request for the information that it is interested in. This is done by including the topic(s) within the request. Once that request is done, the subscriber can receive information from many different publishers, and the information received can also be sent to other subscribers. To include a topic in a request, we use the `addTopic()` method of the message class.

```
messageObject.addTopic(String topicName)
```

Where `topicName` is a name that identifies the data that the subscriber application is interested in. This call will have to be used for each topic that the application wishes to subscribe to. Once the application has defined all the topics, we send the subscription request using the `subscribe()` method of the subscriber class.


```
subscriberObject.subscribe(AmMessage messageObject, AmReceiver  
receiverObject, AmPolicy policyObject)
```

The messageObject is the object that we added topics to using the addTopic() method. This parameter is optional. Publications matching the subscription request are sent to the receiverObject. However, this parameter is not always required. Another way to receive the publications that match the request is by issuing the receive() method of the subscriber class.

The policyObject may or may not be specified. If specified, the policy contains options such as anonymous registration, retrieve only new publications, etc. If is not specified, then the default policy will be used. If the receiverObject was not specified during the subscription request, the publications can be retrieved using the receive() method.

```
subscriberObject.receive(AmMessage messageObject, AmMessage  
selectionmessageObject, AmPolicy policyObject)
```

The messageObject will contain all the messages that have been published, and gets reset implicitly before the messages are received. The selectionMessageObject is used only if we want to select a specific message based on the correlation ID. If it is not specified, then the first available message is received. The policyObject may or may not be specified.

Once we have received all the messages that met the criteria specified in the subscription request or if no more messages are needed, the application can unsubscribe or in other words send a request, so no more messages are sent.

The application can send a request to unsubscribe from all the topics that it initially requested or just from a particular topic. This is done by issuing the unsubscribe() method of the subscribe class.

```
subscribeObject.unsubscribe(AmMessage messageObject, AmReceiver  
receiverObject, AmPolicy policyObject)
```

The messageObject contains the topic(s) to which the unsubscribe request applies. If a confirmation to the unsubscribe request is expected, a receiverObject will have to be passed. Just as in previous calls, the policy may or may not be specified. Example 4-17 shows how the calls that we saw in this section are applied.

Example 4-17 Subscribing messages

```
int iCounter = 0;
```

```
String topic = "Weather";

mySendMSG.addTopic("Weather");
mySubscriber.subscribe(mySendMSG, myPolicy);
// Only 5 messages are expected
for (iCounter = 0; iCounter < 5; iCounter++)
    mySubscriber.receive(myReceiveMSG, myPolicy);
    String myRequest = new
        String(myReceiveMSG.readBytes(myReceiveMSG.getDataLength()));
    System.out.println(myRequest);
}

// The application has received all the messages that it wanted so it proceeds
// to send an unsubscribe request.
mySubscriber.unsubscribe(mySendMSG, myPolicy);
```

4.5.4 Deleting the session and closing the connection

After we have finished working with the queue manager applications, we can proceed to close the objects. This is achieved by using the `close()` method of each of the objects that were used. Another alternative is to close only the session object. Once this object is closed, the rest of the objects will be closed. However, it is strongly recommended that you explicitly close all of the objects that were opened.

```
subscriberObject.close(AmPolicy policy Object);

sessionObject.close(AmPolicy policy Object);

receiverObject.close(AmPolicy policy Object);

publisherObject.close(AmPolicy policy Object);

distributionlistObject.close(AmPolicy policy Object);
```

The only parameter that is expected by all the `close()` methods is the `policyObject`. Remember that the `policyObject` also contains close options, such as delete dynamic queue on close, among others. For more information, refer to the MQSeries Application Messaging Interface documentation.

Note: The last object that must be closed is the `sessionObject`. Once the `sessionObject` is closed, the rest of the references will be invalid.

4.6 How AMI compares to MQI

Now that we have talked about the different APIs that MQSeries offers, we can talk about how they compare with each other. In this section we compare AMI with MQI.

With MQI both the message destination and the message options for send/receive are managed by the application, while with AMI, they are managed by the policies. MQI offers full MQSeries function support and is concerned exclusively with message transport. AMI offers reduced MQSeries function and provides additional functionality.

The programming interface for MQI is low level (fewer verbs but many structures) and the APIs are similar across the different languages (C, C++, Java, and COBOL). AMI has a high-level programming interface, which means that there are more verbs but fewer structures and the API has a natural style to each individual language. MQI is transport specific, while AMI is transport independent.

The MQI is IBM proprietary while a subset of the AMI complies with the Open Applications Group/Open Applications Middleware API standard (C++ and Java).

4.7 Transaction management

In order for the messages sent and received by AMI to be part of a transactional unit of work, the syncpoint attribute of the policy must be specified by the administrator using the AMI administration tool. By default, the syncpoint attribute is set to off. This attribute can be found in the General tab of the policy definition.

The API calls used to control the transaction depend on the type of transaction that is being used. There are two different scenarios:

- When the only resource is the MQSeries message:

In this scenario, the transaction is started by the first message sent or received under syncpoint control, as specified in the policy for the send or receive objects. Multiple messages can be included in the same unit of work. The transaction can be committed by using the `commit()` method of the session object, or it can be backed out by using the `rollback()` method of the session object.

Example 4-18 shows an example of syncpoint control where the MQSeries messages are the only resource.

Example 4-18 Syncpoint control

```
public static void main()
{
    AmSessionFactory      mySessionFactory = null;
    AmSession              mySession = null;
    AmPolicy               myPolicy = null;
    AmReceiver             myReceiver = null;
    AmMessage              myReceiveMSG = null;
    AmMessage              mySendMSG = null;

    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
    myPolicy = mySession.createPolicy("ITSO.SAMPLE.POLICY");
    myReceiveMSG = mySession.createMessage("ITSO.SAMPLE.MESSAGE.NAME");
    myReceiver.receive(myReceiveMSG);
    // If no failures were found commit the action
    mySession.commit(myPolicy);
    String sampleMessage = new
    String(MyReceiveMSG.readbytes(myReceiveMSG.getDataLength()));
    // If some problems were found, don't retrieve the message
    mySession.rollback(myPolicy);
}
```

► **MQSeries as an XA transaction coordinator.**

The transaction must be started explicitly using the `begin()` method of the `AmSession` class before the first recoverable resource (such as a relational database) is changed. The unit of work can then be committed by using the `commit()` method of the `AmSession` class or backed out by using the `rollback()` method of the `AmSession` class. Example 4-19 shows an example where MQSeries is an XA transaction coordinator.

Example 4-19 MQSeries as an XA transaction coordinator

```
public static void main()
{
    AmSessionFactory      mySessionFactory = null;
    AmSession              mySession = null;
    AmPolicy               myPolicy = null;
    AmSender              mySender = null;
    AmMessage              mySendMSG = null;

    // Connect to the database

    mySessionFactory = new AmSessionFactory();
    mySession = mySessionFactory.createSession("ITSO.SESSION.NAME");
    myPolicy = mySession.createPolicy("AMT.SAMPLE.POLICY");
    mySender = mySession.createSender("AMT.SENDER_NAME");
```

```

mySendMSG = mySession.createMessage("ITS0.SAMPLE.MESSAGE.NAME");

mySession.Begin(myPolicy);

// Update a table.

// If the update was successful then commit the action and send a message to
// another application
String sampleMessage = new String("Sample message");
mySendMSG.writeBytes(sampleMessage.getBytes());
mySender.send(mySendMSG);
mySession.commit(myPolicy);
// If problems occurred during the update, backout the changes
mySession.rollback(myPolicy);
}

```

Note: An additional scenario can be when an external transaction coordinator, such as Tuxedo, is used. In this case, the transaction is controlled using the API calls of an external transaction coordinator. Even though the AMI calls are not used, the syncpoint attribute must still be specified in the policy used on the call.

4.8 Grouping

AMI allows a sequence of related messages to be included in, and sent as, a message group. Group context information is sent with each message to allow the message sequence to be preserved and made available to a receiving application. In order to include messages in a group, the group status information of the first and subsequent messages in the group must be set as follows:

- ▶ AMGRP_FIRST_MSG_IN_GROUP for the first message
- ▶ AMGRP_MIDDLEMSG_IN_GROUP for all the messages other than the first and last
- ▶ AMGRP_LAST_MSG_IN_GRP for the last message

This can be achieved by using the `setGroupStatus()` method of the message class.

```
messageObject.setGroupStatus(int groupStatus)
```

The `groupStatus` can be any of the following values:

- ▶ AMGRP_MSG_NOT_IN_GROUP
- ▶ AMGRP_FIRST_MSG_IN_GRP
- ▶ AMGRP_MIDDLEMSG_IN_GRP

- ▶ AMGRP_LAST_MSG_IN_GROUP
- ▶ AMGRP_ONLY_MSG_IN_GROUP

If AMGRP_FIRST_MSG_IN_GROUP is out of sequence, the behavior of this message is the same as for AMGRP_MIDDLE_MSG_IN_GRP.

Note: Once the application starts sending messages in a group, the group must be completed before attempting to send any other messages that are not part of the group.

4.9 Exploring the patterns

In this section we give examples of how AMI can be used to build applications using the different programming patterns.

4.9.1 One-to-one or point-to-point

Send-and-forget

For a send-and-forget application, the following objects will have to be created:

- ▶ Session
- ▶ Policy
- ▶ Sender
- ▶ Message

The sample program that we discuss in this section follows the logic below:

- ▶ Create a session factory object.
- ▶ From the session factory, create a session object.
- ▶ From the session object, create a policy object, a message object, and a sender object.
- ▶ Open the session and the sender objects.
- ▶ Populate the message object with data.
- ▶ Send the message object on the sender object using the send method.
- ▶ Close the sender object and then close the session object.

The first thing that we need to do before even attempting to connect to a queue manager is to inform the Java virtual machine (JVM) to include the MQSeries AMI Java classes. This can be achieved by using the import statement as shown in Example 4-20. Keep in mind that the JAR file must be included in the CLASSPATH environment variable on both the system where the application is going to be compiled and on the one where it is going to be running.

Example 4-20 Import statement

```
import com.ibm.mq.amt.*;
```

Once we have included the AMI JAR file, the next step is to define and initialize the objects that are needed for a send-and-forget application. As we mentioned before, when using this type of pattern we need session factory, session, sender, message, and policy objects (one of each). It is recommended that the names of the objects are descriptive so that it is easy to remember what they are used for.

Example 4-21 shows how to define and initialize the objects.

Example 4-21 Define and initialize

```
private AmSessionFactory mySessionFactory = null;  
private AmSession mySession = null;  
private AmSender mySender = null;  
private AmMessage mySendMSG = null;  
private AmPolicy myPolicy = null;
```

When creating an instance of a class, it is necessary to allocate memory to store its data. When we define the instance at the beginning, the compiler is being told that a variable with a certain name will be used in the class. Therefore, it is still necessary to allocate memory for the variable. This can be achieved by using the new operator. We need to allocate memory for the session factory object.

Example 4-22 shows how to define a reference to the session factory object mySessionFactory that we defined in Example 4-21.

Example 4-22 Define reference

```
mySessionFactory = new AmSessionFactory();
```

Once there is memory allocated for the session factory, we can create the rest of the objects needed for this type of pattern. We will use the createSession(), createPolicy(), createSender() and createMessage() methods. Example 4-23 shows how to create the following objects:

- ▶ A session object called SAMPLE_SESSION_NAME
- ▶ A policy object called SAMPLE_POLICY_NAME
- ▶ A sender object called SAMPLE_SENDER_NAME
- ▶ A message object called SAMPLE_MESSAGE_NAME

Example 4-23 Create sample object

```
mySession = mySessionFactory.createSession("SAMPLE_SESSION_NAME");  
myPolicy = mySession.createPolicy("SAMPLE_POLICY_NAME");  
mySender = mySession.createSender("SAMPLE_SENDER_NAME");  
mySendMSG = mySession.createMessage("SAMPLE_MESSAGE_NAME");
```

After the objects are created, the session and the sender objects must be opened.

Example 4-24 shows how the session object `mySession` and the sender object `mySender` are opened.

Example 4-24 Open objects

```
mySession.open(myPolicy);  
mySender.open(myPolicy);
```

With the objects opened, we can send the message using the `send()` method. As we mentioned in 4.5.3, “Basic operations” on page 109, that the message content is always sent in byte form (Java’s native form), so the `getBytes()` method (Java native method) will have to be used to convert the message into an array of bytes before sending it. Once the message is converted, it then needs to be written to the message object by using the `writeBytes()` method of the sender class. Example 4-25 shows how to prepare and send a message using the coded character set identifier 819.

Example 4-25 Prepare and send

```
String sampleMessage = new String("Sample message");  
mySendMSG.setCCSID(819);  
mySendMSG.writeBytes(sampleMessage.getBytes());  
mySender.send(mySendMSG, myPolicy);
```

If no more messages are going to be sent, the objects will have to be closed by issuing the `close()` method of the sender and session classes. Example 4-26 shows how to close the objects that were opened in Example 4-24 on page 120.

Example 4-26 Close objects

```
mySender.close(myPolicy);  
mySession.close(myPolicy);
```

Request/Reply

For a request/reply application, the following objects will have to be created:

- ▶ Session
- ▶ Policy
- ▶ Sender
- ▶ Receiver
- ▶ Message

The sample program that we are going to discuss in this section follows the logic listed below:

- ▶ Create a session factory object.

- ▶ From the session factory object, create a session object.
- ▶ From the session object, create a policy object, a message object for sending, a message object for receiving, a sender object, and a receiver object.
- ▶ Open the session object, receiver object, and the sender object.
- ▶ Populate the message object for sending with data using the `writeBytes()` method.
- ▶ Send the message object for sending, on the sender object using the `send` method passing the receiver object as the response service.
- ▶ Set the `waitTime` in the policy object to two seconds prior to issuing a receive on the receiver object.
- ▶ Extract and display the data from the message object used for receiving using the `readBytes` method, and use the message for sending as the selection message.
- ▶ Close the receiver, the sender, and the session objects.

The first thing that we must do before even attempting to connect to a queue manager is to inform the Java virtual machine (JVM) to include the MQSeries AMI Java classes. This can be achieved by using the import statement as shown in Example 4-27. Keep in mind that the JAR file must be included in the CLASSPATH environment variable on both the system where the application is going to be compiled and on the one where it is going to be running.

Example 4-27 Import statement

```
import com.ibm.mq.amt.*;
```

Once we have included the AMI JAR file, the next step will be to define and initialize the objects that are needed for a request/reply application. As we mentioned before, with this type of pattern we need a session factory, a session object, a sender object, a receiver object, two message objects (one for sending the message and another for the reply message) and a policy object. It is recommended that the names of the objects be descriptive so that it is easier to remember what they are used for.

Example 4-28 shows how to define and initialize the objects.

Example 4-28 Define and initialize

<code>private AmSessionFactory</code>	<code>mySessionFactory = null;</code>
<code>private AmSession</code>	<code>mySession = null;</code>
<code>private AmSender</code>	<code>mySender = null;</code>
<code>private AmReceiver</code>	<code>myReceiver = null;</code>
<code>private AmMessage</code>	<code>mySendMSG = null;</code>
<code>private AmMessage</code>	<code>myReceiveMSG = null;</code>

```
private AmPolicy          myPolicy = null;
```

When creating an instance of a class, it is necessary to allocate memory to store its data. When we define the instance at the beginning, the compiler is being told that a variable with a certain name will be used in the class. Therefore, it is still necessary to allocate memory for the variable. This can be achieved by using the new operator. We need to allocate memory for the session factory object.

Example 4-29 shows how to define a reference to the session factory object mySessionFactory that we defined in Example 4-28.

Example 4-29 Define reference

```
mySessionFactory = new AmSessionFactory();
```

Once there is memory allocated for the session factory, we can create the rest of the objects needed for this type of pattern. We will use createSession, createPolicy, createSender, createReceiver, and createMessage methods.

Example 4-30 shows how to create the following objects:

- ▶ A session object called SAMPLE_SESSION_NAME
- ▶ A policy object called SAMPLE_POLICY_NAME
- ▶ A sender object called SAMPLE_SENDER_NAME
- ▶ A receiver object SAMPLE_RECEIVER_NAME
- ▶ Two message objects: SAMPLE_SEND_MESSAGE_NAME (to send messages) and SAMPLE_RECEIVE_MESSAGE_NAME (to receive the reply)

Example 4-30 Creat objects

```
mySession = mySessionFactory.createSession("SAMPLE_SESSION_NAME");  
myPolicy = mySession.createPolicy("SAMPLE_POLICY_NAME");  
mySender = mySession.createSender("SAMPLE_SENDER_NAME");  
myReceiver = mySession.createReceiver("SAMPLE_RECEIVER_NAME");  
mySendMSG = mySession.createMessage("SAMPLE_SEND_MESSAGE_NAME");  
myReceiveMSG = mySession.createMessage("SAMPLE_RECEIVE_MESSAGE_NAME");
```

After the objects are created, the session, the sender and the receiver objects must be opened. Example 4-31 shows how the session object mySession, the sender object mySender and the receiver object myReceiver are opened.

Example 4-31 Open objects

```
mySession.open(myPolicy);  
mySender.open(myPolicy);
```

```
myReceiver.open(myPolicy);
```

With the objects opened, we can send the message using the `send()` method. As we mentioned in 4.5.3, “Basic operations” on page 109, the message content is always sent in byte form (Java’s native form), so the `getBytes()` method (Java native method) will have to be used to convert the message into an array of bytes before sending it. Once the message is converted, it then needs to be written to the message object by using the `writeBytes()` method of the sender class.

Example 4-32 shows how to prepare and send a message using the coded character set identifier 819. Notice that this time we pass the receiver object to indicate that we are expecting a reply message.

Example 4-32 Prepare and send

```
String sampleMessage = "Sample message";

mySendMSG.setCCSID(819);
mySendMSG.writeBytes(sampleMessage.getBytes());
mySender.send(mySendMSG, myReceiver, myPolicy);
```

To specify a wait time for the reply, the `setWaitTime()` method of the policy class must be used. Once the message arrives, we use the `receive()` method of the receiver class. To read the contents of the message object, the `readBytes()` method of the receiver class must be used.

Example 4-33 shows how to receive a message. It also shows how to set up the wait time to two seconds (two thousand milliseconds).

Example 4-33 Receive a message

```
myPolicy.setWaitTime(2000);
mySendMSG.setCCSID(AMCP_819);
myReceiver.receive(myReceiveMSG, mySendMSG, myPolicy);
replyMessage = new
String(myReceiveMSG.readBytes(myReceiveMSG.getDataLength()), "ISO8859_1");
```

If no more messages are going to be sent, then the objects must be closed by issuing the `close()` method of the sender, receiver and session classes. The `close()` method of the sender and receiver classes can be omitted because once the session object is closed, the rest of the references are going to be invalid. Example 4-34 shows how to close the objects that were opened in Example 4-31.

Example 4-34 Close objects

```
mySender.close(myPolicy);
myReceiver.close(myPolicy);
```

```
mySession.close(myPolicy);
```

4.9.2 Publish/subscribe

Publisher

For a publisher application, the following objects have to be created:

- ▶ Session
- ▶ Policy
- ▶ Publisher
- ▶ Receiver
- ▶ Two message objects (one for the message that is going to be published and another for the response)

The publisher sample program that we discuss in this section follows the logic listed below:

- ▶ Create a session factory object.
- ▶ From the session factory, create a session object.
- ▶ From the session object, create a policy object, a message object for publications, and a publisher object.
- ▶ Open the session object and the publisher.
- ▶ Add the topic to the publication message using the addTopic method.
- ▶ Select from topics and write the data into the outgoing message object using the writeBytes method.
- ▶ Publish the information by calling the publish method of the publisher object.
- ▶ Reset the publication message.
- ▶ Wait for two seconds.
- ▶ Retrieve the publish response generated by the broker.
- ▶ Perform the previous six steps repeatedly until the maximum number of publications specified at the beginning has been reached.
- ▶ Close the publisher and the session object.

The first thing that we need to do before even attempting to connect to a queue manager is to inform the Java virtual machine (JVM) to include the MQSeries AMI Java classes. This can be achieved by using the import statement as shown in Example 4-35. Keep in mind that the JAR file must be included in the CLASSPATH environment variable on both the system where the application is going to be compiled and on the one where it is going to be running.

Example 4-35 Import statement

```
import com.ibm.mq.amt.*;
```

Once we have included the AMI JAR file, the next step will be to define and initialize the objects that are needed for a publisher application. As we mentioned before, for this type of pattern we need a session factory, a session object, a publisher object, a receiver object, two message objects (one for the message that will be published and the other one for the response) and a policy object. It is recommended that the names of the objects be descriptive so that it is easier to remember what they are used for.

Example 4-36 shows how to define and initialize the objects.

Example 4-36 Define and initialize objects

```
private AmSessionFactory      mySessionFactory = null;
private AmSession             mySession = null;
private AmPublisher           myPublisher = null;
private AmMessage             mySendMSG = null;
private AmMessage             myRespMSG = null;
private AmPolicy              myPolicy = null;
private AmReceiver            myRespReceiver = null;
```

When creating an instance of a class, it is necessary to allocate memory to store its data. When we define the instance at the beginning, the compiler is being told that a variable with a certain name will be used in the class. Therefore, it is still necessary to allocate memory for the variable. This can be achieved by using the new operator. We need to allocate memory for the session factory object.

Example 4-37 shows how to define a reference to the session factory object mySessionFactory that we defined in Example 4-36.

Example 4-37 Define reference

```
mySessionFactory = new AmSessionFactory();
```

Once there is memory allocated for the session factory, we can create the rest of the objects needed for this type of pattern. We will use the createSession(), createPolicy(), createSender(), createReceiver() and createMessage() methods.

Example 4-38 shows how to create the following objects:

- ▶ A session object called SAMPLE_SESSION_NAME
- ▶ A policy object called SAMPLE_POLICY_NAME
- ▶ A publisher object called SAMPLE_PUBLISHER_NAME
- ▶ A receiver object called SAMPLE_RESPONSE_NAME

- Two message objects: `SAMPLE_MESSAGE_NAME` (for the message that will be published) and `SAMPLE_RESP_MESSAGE_NAME` (for the response)

Example 4-38 Create objects

```
mySession = mySessionFactory.createSession("SAMPLE_SESSION_NAME");
myPolicy = mySession.createPolicy("SAMPLE_POLICY_NAME");
myPublisher = mySession.createPublisher("SAMPLE_PUBLISHER_NAME");
myRespReceiver = mySession.createReceiver("SAMPLE_RESPONSE_NAME");
mySendMSG = mySession.createMessage("SAMPLE_MESSAGE_NAME");
myRespMSG = mySession.createMessage("SAMPLE_RESP_MESSAGE_NAME");
```

After the objects are created, the session, the publisher, and the receiver objects must be opened.

Example 4-39 shows how the session object `mySession`, the publisher object `myPublisher`, and the receiver object `myRespReceiver` are opened.

Example 4-39 Open objects

```
mySession.open(myPolicy);
myPublisher.open(myPolicy);
myRespReceiver.open(myPolicy);
```

The next step is to add the topic to the message object. This is achieved by using the `addTopic()` method of the message object. Example 4-40 shows how to add the topic "Weather" to the publisher object created in Example 4-38. It then sets the message attribute `CCSID` and adds the data to the message object `mySendMSG`, by converting the message string into bytes in the specified code page.

Once the data is converted from a string to an array of byte, it then publishes the contents of the message object. In the publish call, the receiver object `myRespReceiver` is sent with the data, with the purpose of getting an acknowledgment sent by the broker.

Example 4-40 Adding a topic

```
String sunny = "SUNNY again";
String showers = "WIND and SCATTERED showers";
String rain = "HEAVY RAIN";
String outlook[] = {sunny, showers, rain};
for (int i = 0, j = 0; i < SAMPLE_MAX_PUBLICATIONS; i++, j++)
{
    try
    {
        mySendMSG.addTopic("Weather");
        if (j == 3)
    }
```

```

{
    j = 0;
}
String sampleMessage = new String(outlook[j]);
mySendMSG.setCCSID(819);
mySendMSG.writeBytes(sampleMessage.getBytes("ISO8859_1"));
myPublisher.publish(mySendMSG, myRespReceiver, myPolicy);
myRespReceiver.receive(myRespMSG, mySendMSG, myPolicy);
}
}

```

If no more messages are going to be published, the objects will have to be closed by issuing the `close()` method of the publisher, receiver and session classes. The `close()` method of the publisher and receiver classes can be omitted because once the session object is closed, the rest of the references are going to be invalid.

Example 4-41 shows how to close the objects that were opened in Example 4-39.

Example 4-41 Closing objects

```

myPublisher.close(myPolicy);
myReceiver.close(myPolicy);
mySession.close(myPolicy);

```

Subscriber

For a subscriber application, the following objects should be created:

- ▶ Session
- ▶ Policy
- ▶ Subscriber
- ▶ Two message objects (one to send the subscription and another to receive the publications)

The subscriber sample program that we discuss in this section follows the logic listed below:

- ▶ Create a session factory object.
- ▶ From the session factory, create an session object.
- ▶ From the session, create a policy object, a message object for subscribing, a message object for receiving the publications, and a subscriber object.
- ▶ Open the session object and the subscriber object.
- ▶ Add the topic to the subscription message using the `addTopic` method and pass it on the `subscribe` method of the subscriber class.

- ▶ Call the receive method of the subscriber class.
- ▶ Read and display the publication data in the incoming message using the readBytes method of the message object.
- ▶ Reset the message object used for receiving publications.
- ▶ Perform the previous three steps repeatedly until the maximum number of publications specified has been reached.
- ▶ Unsubscribe from the topic.
- ▶ Close the subscriber and the session object.

The first thing we must do before even attempting to connect to a queue manager is to inform the Java virtual machine (JVM) to include the MQSeries AMI Java classes. This can be achieved by using the import statement as shown in Example 4-42. Keep in mind that the JAR file must be included in the CLASSPATH environment variable on both the system where the application will be compiled and on the other system where it is going to be running.

Example 4-42 Importing

```
import com.ibm.mq.amt.*;
```

Once we have included the AMI JAR file, the next step will be to define and initialize the objects that are needed for a subscriber application.

As we mentioned before, in this type of pattern we need a session factory, session object, a subscriber object, two message objects (one to send the subscription and another to receive the publications), and a policy object. It is recommended that the names of the objects be descriptive so that it's easier to remember what they are used for.

Example 4-43 shows how to define and initialize the objects.

Example 4-43 Define and initialize

```
private AmSessionFactory    mySessionFactory = null;
private AmSession           mySession = null;
private AmSubscriber        mySubscriber = null;
private AmMessage           mySendMSG = null;
private AmMessage           myReceiveMSG = null;
private AmPolicy            myPolicy = null;
```

When creating an instance of a class, it is necessary to allocate memory to store its data. When we define the instance at the beginning, the compiler is being told that a variable with a certain name will be used in the class. Therefore, it is still necessary to allocate memory for the variable. This can be achieved by using the new operator. We need to allocate memory for the session factory object.

Example 4-44 shows how to define a reference to the session factory object `mySessionFactory` that we defined in Example 4-43.

Example 4-44 Define a reference

```
mySessionFactory = new AmSessionFactory();
```

Once there is memory allocated for the session factory, we can create the rest of the objects needed for this type of pattern. We will use the `createSession`, `createPolicy`, `createSender`, and `createMessage` methods.

Example 4-45 shows how to create the following objects:

- ▶ A session object called `SAMPLE_SESSION_NAME`
- ▶ A policy object called `SAMPLE_POLICY_NAME`
- ▶ A subscriber object called `SAMPLE_SUBSCRIBER_NAME`
- ▶ Two message objects, one called `SAMPLE_RECEIVE_MESSAGE_NAME` (to receive the publications) and one called `SAMPLE_SEND_MESSAGE_NAME` (to send the subscription request).

Example 4-45 Create objects

```
mySessionFactory = new AmSessionFactory();  
mySession = mySessionFactory.createSession("SAMPLE_SESSION_NAME");  
myPolicy = mySession.createPolicy("SAMPLE_POLICY_NAME");  
mySubscriber = mySession.createSubscriber("SAMPLE_SUBSCRIBER_NAME");  
myReceiveMSG = mySession.createMessage("SAMPLE_RECEIVE_MESSAGE_NAME");  
mySendMSG = mySession.createMessage("SAMPLE_SEND_MESSAGE_NAME");
```

Once the objects are created, the session and the subscriber objects must be opened.

Example 4-46 shows how the session object `mySession` and the subscriber object `mySubscriber` are opened.

Example 4-46 Open session

```
mySession.open(myPolicy);  
mySubscriber.open(myPolicy);
```

Example 4-47 shows how to reset the message object to its initial state before getting the publications, the topic name will have to be added to the message object using the `addTopic()` method of the message class. Once the topic is added to `mySendMSG`, the subscription request is sent using the `subscribe()` method of the subscriber object `mySubscriber`.

Example 4-47 Reset message object

```
String topic = "Weather";  
try
```

```

{
    mySendMSG.reset();
    mySendMSG.addTopic(topic);
    mySubscriber.subscribe(mySendMSG, myPolicy);
}

```

Now we need to retrieve some publications that have the topic Weather. To accomplish this the receive() call of mySubscriber is used. Once received, the publication has to be converted to an array of bytes. The readBytes() method of the receiver object,

Example 4-48 is used to convert the message data from string to an array of bytes. The message is then displayed on the screen and then the message object is reset.

Example 4-48 Convert message

```

for (i = 0; i < SAMPLE_MAX_PUBLICATIONS; i++)
{
    try
    {
        mySubscriber.receive(myReceiveMSG, myPolicy);
        String myRequest = new
        String(myReceiveMSG.readBytes(myReceiveMSG.getDataLength()), "ISO8859_1");
        System.out.println(myRequest);
        myReceiveMSG.reset();
    }
}

```

In Example 4-49, the unsubscribe() method of the subscriber object mySubscriber is used to send a message to the broker to deregister the subscription.

Example 4-49 De-register

```

String topic = "Weather";

try
{
    mySendMSG.reset();
    mySendMSG.addTopic(topic);
    // -----
    // Send the request.
    // No response is expected so no receiver object is passed.
    // -----
    mySubscriber.unsubscribe(mySendMSG, myPolicy);
}

```

If no more messages are to be retrieved, the objects must be closed by issuing the `close()` method of the subscriber and session classes. The `close()` method of the subscriber class can be omitted because once the session object is closed, the rest of the references are going to be invalid.

Example 4-50 shows how to close the objects that were opened in Example 4-46 on page 129.

Example 4-50 Close objects

```
mySubscriber.close(myPolicy);  
mySession.close(myPolicy);
```

In this chapter we have seen how AMI can be used by application programmers to build applications without needing to understand all the details of the MQSeries Message Queue Interface.



Programming with C++

This chapter covers the MQSeries C++ API. This API is an object-oriented extension of the MQI API explained in Chapter 3, “Programming with MQI” on page 23. Here, we discuss the basic concepts of this API, the architectural model, and the API availability.

Then we introduce some of the basic operations that can be performed using this API, such as:

- ▶ Connecting and disconnecting from a queue manager
- ▶ Opening and closing MQSeries object (Queue object for example)
- ▶ Sending and getting messages from a queue
- ▶ Transaction management
- ▶ Message grouping

Finally we explore the implementation of the programming patterns explained in Chapter 1, “Introduction and patterns” on page 3 using this API.

5.1 Overview

The MQSeries C++ interface is an extension of the MQI API presented in Chapter 3, “Programming with MQI” on page 23. It gives the programmer a object-oriented approach to the messaging interface in MQSeries.

Since this API is based on an object-oriented model, attributes and methods are inherited to child classes as shown in Figure 5-1 on page 136 and Figure 5-2 on page 137. In the following sections, we specify the methods as members of the parent class so we can easily find them in the *Using C++*, SC33-1877, which comes with the product.

5.1.1 Key features

The C++ MQI provides all the features available in the MQI API, such as getting, putting, and browsing messages, it also allows users to inquire and set object options.

Additionally it provides the following features:

- ▶ Automatic initialization of MQSeries data structures
- ▶ Just-in-time queue manager connection and queue opening
- ▶ Implicit queue closure and queue manager disconnection
- ▶ Dead-letter header transmission and receipt
- ▶ IMS Bridge header transmission and receipt
- ▶ Reference message header transmission and receipt
- ▶ Trigger message receipt
- ▶ CICS Bridge header transmission and receipt
- ▶ Work header transmission and receipt
- ▶ Client channel definition

5.2 Platforms and languages

The MQSeries C++ is available in the following server environments:

- ▶ MQSeries for AIX Version 5
- ▶ MQSeries for AS/400 Version 4 Release 2
- ▶ MQSeries for HP-UX Version 5
- ▶ MQSeries for OS/2 Warp Version 5
- ▶ MQSeries for Sun Solaris Version 5
- ▶ MQSeries for Windows NT Version 5

It is also available in the following client environments:

- ▶ AIX
- ▶ HP-UX
- ▶ OS/2
- ▶ Sun Solaris
- ▶ Windows 3.1
- ▶ Windows 95
- ▶ Windows NT

5.3 Libraries

Table 5-1 shows the libraries required to compile the C++ program developed using this API for each available platform.

Table 5-1 Libraries

Platform	Library
MQSeries for Windows NT	IMQ*.LIB
MQSeries for AIX	In a non-threaded application: libimq*.a In a threaded application: libimq*_r.a
MQSeries for Sun Solaris	imq*.so
MQSeries for HP-UX	imq*.so

The imqi.hpp header contains all the declarations require to use this API.

5.4 C++ architectural model

All classes in the API are inherited from the ImqError class, which allows an error condition to be associated to each object. Figure 5-1 on page 136 and Figure 5-2 on page 137 show UML class diagrams of classes available in the API:

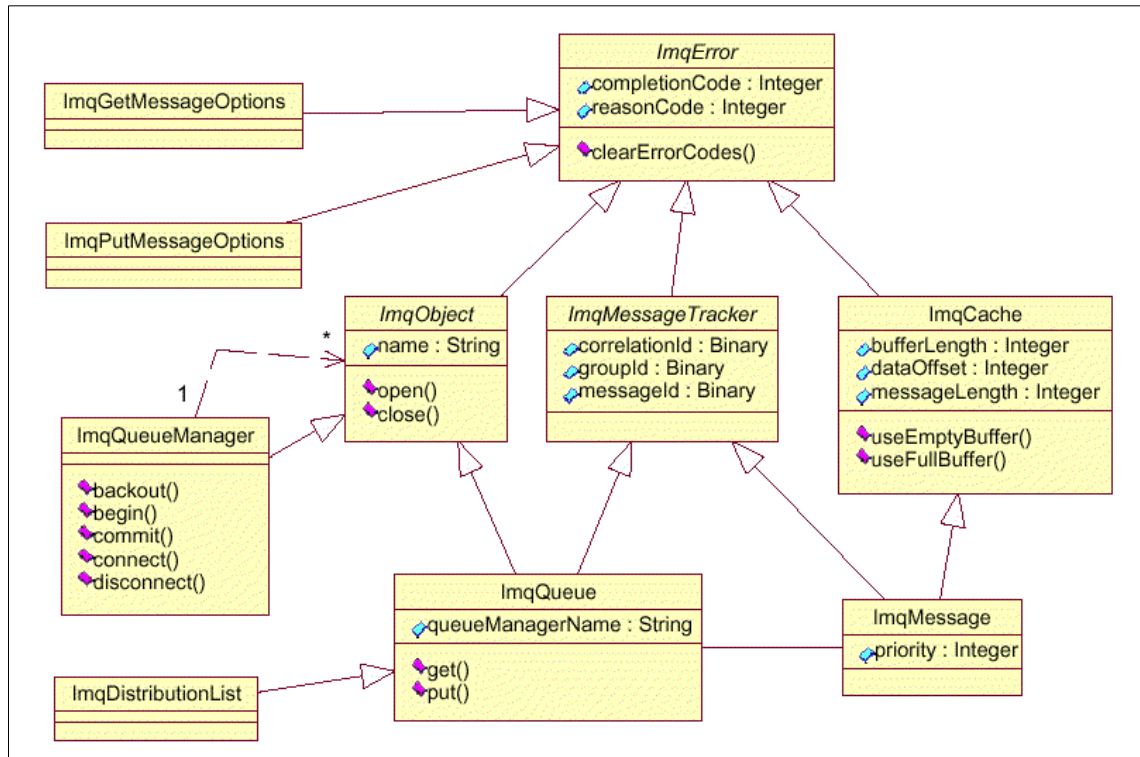


Figure 5-1 Queue management classes

Figure 5-2 on page 137 shows the item-related classes, these classes encapsulate the message header structures provided in MQI such as the IMS bridge header and the dead-letter header.

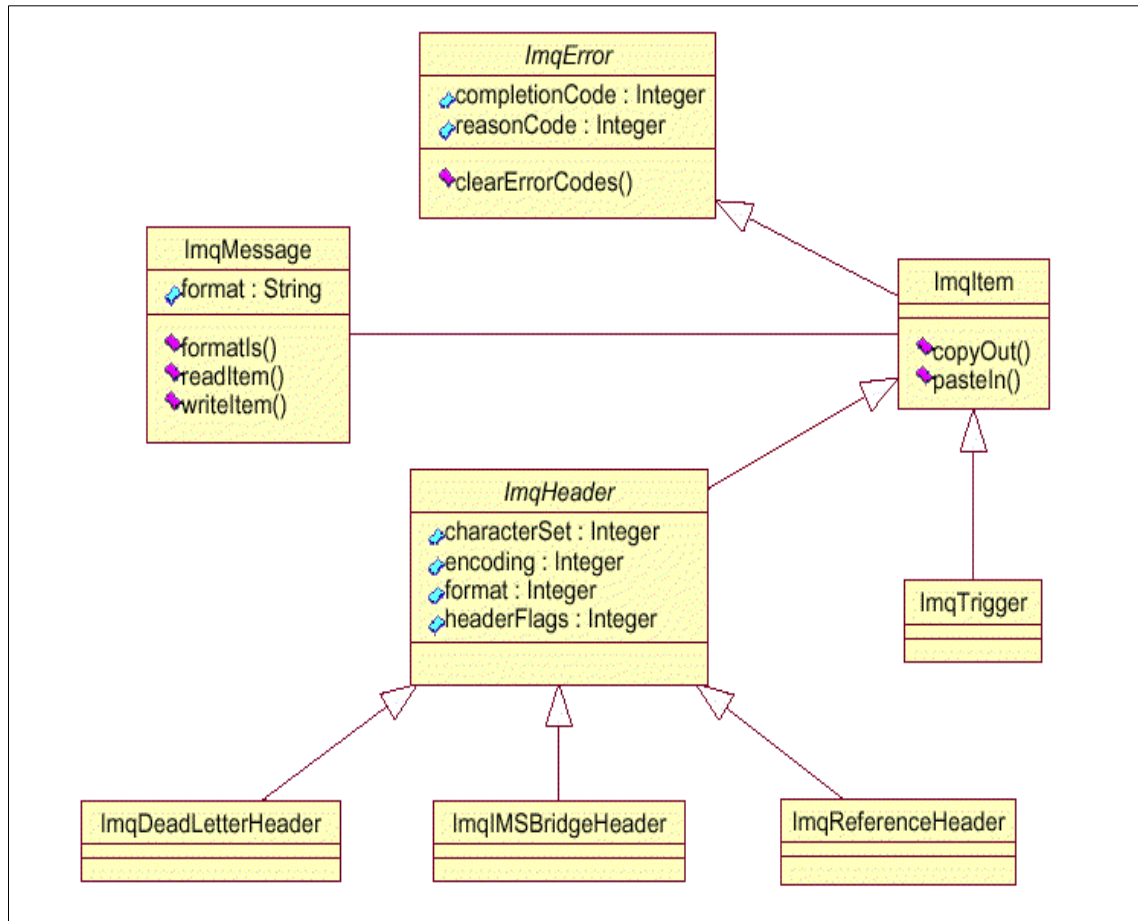


Figure 5-2 Item handling classes

Both the queue management classes and the item handling classes use the following classes and data types:

- ▶ The `ImqBinary` class which encapsulates byte arrays such as `MQBYTE24`.
- ▶ The `ImqBoolean` data type which is defined as `typedef unsigned char ImqBoolean`.
- ▶ The `ImqString` class which encapsulates character arrays such as `MQCHAR64`.

This API is an object oriented version of the MQSeries Interface presented in Chapter 3, “Programming with MQI” on page 23. We will explain the API architecture related to the features in the MQI API.

All the MQI data structure entities are subsumed within the appropriate object classes, giving us access methods to the individual data structure fields.

The entities with handles inherited from the `ImqObject` class or one of its descendants, and gives us encapsulated interfaces to the MQI. Additionally, these objects exhibit intelligent behavior that can reduce the amount of method calls compared to those required in a procedural MQI implementation. For example, connections to a queue manager are created and discarded as required.

The `ImqMessage` class encapsulates the MQMD data structure and also acts as a holding point for user data and items by providing cached buffer facilities. The cached buffer can either be supplied by the application or created automatically by the system.

The `ImqItem` class represents items in a message body. Items are pieces of a message that need to be processed sequentially and separately. Apart from the regular user data, items might be a dead-letter header or a trigger message. There is a class object for each item that corresponds to a recognizable MQSeries message format. There is no class of object for user data, but it can be written by specializing the `ImqItem` class. If not, user data processing is left to the application.

The `ImqChannel` class encapsulates the channel definition (MQCD) structure and lets us specify the connection options to be used in a client environment when a `ImqQueueManager::connect()` is executed. More information about the automatic channel definition options of MQSeries can be found in the MQSeries Intercommunication book.

5.5 Programming with the C++ API

We will now explain how the basic MQSeries operation, such as connecting to a queue manager, opening a queue and sending or receiving messages, can be achieved using this API.

5.5.1 Connecting to the queue manager

To connect to a queue manager we will use the `ImqQueueManager` class, which encapsulates the MQSeries queue manager object.

The name of the queue manager can be provided in the constructor call, or by using the `setName` method of the `ImqQueueManager` class.

```
ImqQueueManager qmanager;  
qmanager.setName(name);
```

or

```
ImqQueueManager *pmanager = new ImqQueueManager(name);
```

Note: We will be using the pmanager object throughout the rest of this chapter.

Then the connection can be established using the connect method of the ImqQueueManager.

```
pmanager.connect();
```

Information about the queue manager can be accessed using the ImqQueueManager interface.

5.5.2 Opening MQSeries objects

MQSeries objects can be opened using the ImqObject or ImqQueue classes depending on if the object is a queue, or another type of object.

Generally, the ImqQueue class will be used, unless some object attributes have to be inquired or set.

Opening queues

The ImqQueue class encapsulates the MQSeries queue object and adds some intelligence to the queue object behavior.

Before any put or get operation can be performed in a queue, the queue manager holding the queue must be assigned to the ImqQueue object using the setConnectionReference method of the ImqQueue class.

```
ImqQueue pqueue;
```

```
pqueue.setConnectionReference(pmanager);
```

The queue name can be provided during the object construction or using the setName method of the ImqObject class.

```
pqueue.setName(queueName);
```

Queues will be automatically opened with the required options when a put or get method invocation is issued, meaning that no explicit open operations are required. The ImqQueue object will close and reopen the queue if the actual open options do not match the requirement to perform an operation on the queue.

In some cases this can cause some additional overhead, or some problems, depending on the type of queue that is being opened.

To avoid the automatic closing and reopening of the queue, we must directly set the opening options by using the `setOpenOptions`, or the `openFor` methods of the `ImqObject` class. A queue can also be opened explicitly using the `open` method of the `ImqObject` class, but if the open options have been specified it will not give any major advantage over the implicit open provided by this interface.

```
pqueue.setOpenOptions(MQOO_OUTPUT | MQOO_INPUT_SHARED);
```

or

```
pqueue.openFor(MQOO_OUTPUT | MQOO_INPUT_SHARED);
```

The `openFor` method incrementally adds the open options specified to those actually assign to the object. The default open option for a `ImqQueue` object is `MQOO_INQUIRE`.

Opening dynamic queues

Dynamic queues cannot be closed by a `reopen` automatically because a `close` operation on a dynamic queue will destroy the queue. Therefore to open a dynamic queue we must specify the open options.

The name of the model queue is specified with the `setName` method of the `ImqObject` class and the dynamic queue name or its prefix (in the same way it would be specified using the MQI API , “Opening queues” on page 35) can be specified with the `setDynamicQueueName` method of the `ImqQueue` class.

The actual name of the dynamic queue can be obtained with the `dynamicQueueName` method after the queue has been opened.

```
pqueue.setDynamicQueueName(dynamicqueueName);
```

Opening distribution lists

Distribution lists are managed by the `ImqDistributionList` class, which inherits from the `ImqQueue` class.

Any number of `ImqQueue` objects can be associated with a `ImqDistributionList` object using the `setDistributionListReference` method of the `ImqQueue` class.

Before opening the distribution list the associated queues must be assigned the name of the queue and the queue manager that holds it, as shown in Example 5-1 on page 140.

Example 5-1 Opening distribution lists

```
ImqDistributionList dlist;  
ImqQueue queueA, queueB;  
ImqString queueManagerName(pmanager.name());
```

```
// Set queue manager connection reference
queueA.setConnectionReference(pmanager);
queueB.setConnectionReference(pmanager);

// Set target queue names
queueA.setName(queueName1);
queueB.setName(queueName2);

// Assign the queue manager name
queueA.setQueueManagerName( queueManagerName);
queueB.setQueueManagerName( queueManagerName);

// Associate the queues with the distribution list.
queueA.setDistributionListReference(dlist);
queueB.setDistributionListReference(dlist);
```

Once the queues of the distribution lists have been set, the distribution list can be opened and operated just as any other `ImqQueue` object.

5.5.3 Closing MQSeries objects

MQSeries objects are automatically closed when the corresponding `ImqObject` is deleted.

5.5.4 Disconnecting from the queue manager

A disconnect operation is implicitly performed when the `ImqQueueManager` object is deleted.

5.5.5 Putting messages on a queue

Messages can be put to a `ImqQueue` or `ImqDistributionList` object using the `put` method of the `ImqQueue` class.

The `put` method provides two interfaces:

```
ImqBoolean put(ImqMessage & msg);

ImqBoolean put(ImqMessage & msg, ImqPutMessageOptions & pmo);
```

The message data is managed by the `ImqMessage` class. The `ImqMessage` class inherits from the `ImqMessageTracker` class, which encapsulates the MQMD data structure, and the `ImqCache` class which handles the message data buffer.

The message identification can be set using the `setMessageId` method of the `ImqMessageTracker` class.

```
ImqMessage msg;  
  
msg.setMessageId(msgId);
```

In the same way, there are methods to access the correlation ID and the group ID.

The `messageId`, `correlationId` and `groupId` variables must be created using the `ImqBinary` class. This class encapsulates the `BYTExx` data types that are used in the MQI which provide some methods to perform basic operations.

Example 5-2 shows how to create a binary object using the `ImqBinary` class.

Example 5-2 Creating a `ImqBinary` object

```
ImqBinary correlationId ;  
MQBYTE24 byteId = "BYTEID1234";  
  
// Set the value of the ImqBinary object  
correlationId.set(byteId,sizeof(byteId ));
```

Preparing message data

This API differs from the MQI API in the way message data is prepared and handled.

In MQI message data is managed completely by the application, from allocating the correct buffer to store the data, to processing the different possible headers in the message when it is read.

In the C++ API some buffered cache functionality has been added, and this buffer is managed by the `ImqCache` object. A buffer is associated to each message (`ImqMessage` object), by inheritance.

The buffer is, by default, provided by the `ImqCache` object automatically or it can be provided by the application using any of the following `ImqCache` object methods:

- **useEmptyBuffer:** This method allows the application to assign a fixed-length empty buffer to the `ImqMessage` object. The message length will be set to zero automatically and the buffer will be clear, unless the actual message length is assigned.

```
ImqMessage msg;  
char pszBuffer[24]= "Hello World";  
  
msg.useEmptyBuffer(pszBuffer, sizeof(pszBuffer));
```

```

msg.setFormat(MQFMT_STRING);
msg.setMessageLength(12);
or
char pszBuffer[12];

msg.useEmptyBuffer(pszBuffer, sizeof(pszBuffer));
msg.setFormat(MQFMT_STRING);

```

- **useFullBuffer:** This method allows the application to assign an already prepared message buffer to the `ImqMessage` object. The buffer will **not** be clear and the message length will be set to the length provided in the method invocation.

```

ImqMessage msg;
char pszBuffer[] = "Hello world";

msg.useFullBuffer(pszBuffer, sizeof(pszBuffer));
msg.setFormat(MQFMT_STRING);

```

The message buffer can be reused and the number of bytes transmitted can be varied by setting the message length using the `setMessageLength` method of the `ImqCache` class.

The advantage of an application supplying the message buffer is that no data copying is required, since the data can be prepared directly in the buffer.

To set the `ImqCache` object back to the automatic buffer facility, the application can call the `useEmptyBuffer` with a null buffer pointer and a length of zero (0).

When supplied automatically, the buffer extends to accommodate the message as it grows. This gives more flexibility when the message length is unknown before it is prepared. The message (data) can be copied into the buffer using the `ImqCache` write method.

```
msg.write(12, "Hello world");
```

Items can be copied to the buffer using the `ImqMessage` `writeItem` method. For example you might want to add a dead-letter header to a message and put it in the dead-letter queue.

Example 5-3 shows how to create a `ImqDeadLetterHeader` and then insert it at the beginning of an existing message.

Example 5-3 Creating a dead-letter header

```

ImqDeadLetterHeader header;

// Set up the dead-letter header information.
header.setDestinationQueueManagerName(pmanager.name());
header.setDestinationQueueName(pqueue.name());

```

```

header.setPutApplicationName(/?*?*/);
header.setPutApplicationType(/?*?*/);
header.setPutDate(/* TODAY */);
header.setPutTime(/* NOW */);
header.setDeadLetterReasonCode(/* REASON */);

// Insert the dead-letter header information. This will vary
// the encoding, character set and format of the message.
// Message data is moved along, past the header.
msg.writeItem(header);

```

Using additional options in the put method

The put method of the `ImqQueue` class provides two interfaces as shown at the beginning of this section.

Additional options must frequently be specified when putting a message onto a queue. These options can be specified by calling the put method with a second parameter in the form of a `ImqPutMessageOptions` object.

The `ImqPutMessageOptions` class encapsulates the MQPMO data structure, which allows the application to specify some additional options such as sync-point control or message context.

Example 5-4 on page 144 shows how to start and set the syncpoint participation option to true. This will start a local queue manager transaction that can be ended by using the commit or backout methods of the `ImqQueueManager` class. The transactional management with C++ will be further explained in the following section.

Example 5-4 Putting a message with syncPoint participation

```

ImqQueue pqueue;
ImqMessage msg;
ImqPutMessageOptions pmo;

// This sets the put message options to participate
// in a syncpoint transaction.
pmo.setSyncPointParticipation(TRUE);

pqueue.put(msg, pmo);

```

Please refer to the *Using C++* manual for more information about the options available with the `ImqQueue` class.

5.5.6 Getting messages from a queue

It is possible to get messages from a `ImqQueue` object using the `get` method provided by this class.

The `ImqQueue` `get` method provides four interfaces:

```
ImqBoolean get(ImqMessage & msg, ImqGetMessageOptions & options);  
ImqBoolean get(ImqMessage & msg);  
ImqBoolean get(ImqMessage & msg, ImqGetMessageOptions & options,  
               const size_t buffer-size);  
ImqBoolean get(ImqMessage & msg, const size_t buffer-size);
```

The message information is contained in the `ImqMessage` object after the method invocation.

By default, the message buffer is provided by the system and can be obtained by using the `dataPointer` or `bufferPointer` methods. The message data length can be obtained by using the `dataLength` method of the `ImqCache` class.

```
pqueue.get(msg);
```

```
char *pszDataPointer = msg.dataPointer(); /* Address. */  
int iDataLength = msg.dataLength(); /* Length. */
```

Note: After each `get` method invocation, the physical location of the data buffer can be changed, so it is not recommended to use the actual buffer pointer to access data. Instead the data pointer should be re-assigned using either the `dataPointer` or `bufferPointer` methods.

If the application wants to provide a fixed-length buffer to receive the message data, the `ImqCache` `useEmptyBuffer` method can be used before using the `ImqQueue` `get` method. The message length will be restricted to that of the given buffer, so long messages must be considered during the application design.

```
char pszBuffer[BUFFER_LENGTH];
```

```
pqueue.useEmptyBuffer(pszBuffer, BUFFER_LENGTH);  
pqueue.get(msg);
```

In this case, the actual buffer pointer `pszBuffer` can always be used, but it is recommendable to use the `dataPointer` method to assure portability.

Reading message data

Once the message has been received, message data can be in the form of items or raw user data, depending on the format of the message. Items are pieces of data that have to be processed separately and sequentially.

The message format can be validated using the `ImqMessage formatIs` method.

If the message format represents any known message header data structure, that structure can be retrieved from the message buffer using the `ImqMessage readItem` method. There are three predefined message headers in this API:

- ▶ Dead letter header (`ImqDeadLetterHeader` class).
- ▶ IMS bridge header (`ImqIMSBridgeHeader`).
- ▶ Reference header (`ImqReferenceHeader`).

Each of these corresponds to an MQSeries defined message format. Other types of format can be defined by the user by specifying the `ImqItem` class.

```
if (msg.formatIs(MQFMT_DEAD_LETTER_HEADER)) {
    ImqDeadLetterHeader header;

    // The readItem method must be called with the
    // right class of object pointer.

    if (msg.readItem(header)) {
        // Perform the corresponding operation for this item type.
    }
}
```

If the message format is unknown, then the message data can be directly access using the `dataPointer` method as explained before.

Additional get method options

The `ImqGetMessageOptions` class provides additional information for the message retrieval process, such as:

- ▶ Wait interval for the get operation
- ▶ Match options
- ▶ Message options
- ▶ Syncpoint participation
- ▶ Group status
- ▶ Segmentation status

The `ImqGetMessageOptions` `setOptions` method can be used to specify any of the MQI available message get options. One of the options more commonly used is the `MQGMO_WAIT` option which will enable a wait interval for the get operation to finish. This way if the expected message hasn't yet arrived on the queue, the get method will wait for the amount of time specified using the `setWaitInterval` method of the `ImqGetMessageOptions` class, before returning an error.

Example 5-5 shows how to get a message from a queue with an unlimited wait option.

Example 5-5 Getting a message with a wait option

```
ImqGetMessageOptions gmo;
ImqMessage msg;

gmo.setOptions(MQGMO_WAIT);

// Set the wait interval to unlimited meaning that the get operation
// will wait until one message appears in the queue.
gmo.setWaitInterval(MQWI_UNLIMITED);

pqueue.get(msg,gmo);
```

Getting a specific message from the queue

A specific message can be identified by any combination of this message attributes as found in the `ImqMessageTracker` class:

- ▶ `MsgId`
- ▶ `CorrelationId`
- ▶ `GroupId`

These options have to be specified in the `ImqMessage` object passed in the get method invocation.

The `ImqGetMessageOptions` class gives the application a way to specify which options will be used during the message search.

```
gmo.setMatchOptions(MQMO_MATCH_MSG_ID);
msg.setMessageId(msgId);
If (pqueue.get(msg,gmo)) {
    // Perform any operation with this message
}
```

If more than one message matches the given criteria, the first message of that set will be returned, and consecutive calls to the get method will give access to all the messages.

The message object information will change after a get method call if a message matching the criteria is found, otherwise the function will return false.

5.6 Advance topics

Here we present some of the advance functionality of the C++ API, specifically the browsing messages functionality, and the inquiring and setting object attributes functionality.

5.6.1 Browsing messages on a queue

Messages on a queue can be browsed using an `ImqQueue` get method. The `ImqQueue` object must be open using the `MQOO_BROWSE` open option. That can be done using the `setOpenOptions` or the `openFor` method as described in “Opening queues” on page 139.

```
pqueue.setOpenOptions(MQOO_BROWSE);
```

or

```
pqueue.openFor(MQOO_BROWSE);
```

After the queue object has been opened for browse, the `ImqQueue` get method has to be called with an `ImqGetMessageOptions` object with the following options:

- ▶ `MQGMO_BROWSE_FIRST` message option, if you want the browse cursor to be positioned at the first message that matches the criteria specified in the `ImqMessage` object.
- ▶ `MQGMO_BROWSE_NEXT` message option, if you want the browse cursor to move to the next message that matches the criteria specified in the `ImqMessage` object.

The get method will return an updated version of the `ImqMessage` object with the information of the current message pointed to by the browse cursor, without removing it from the queue.

When the queue object has just been opened, the browse cursor points to the first message in the queue, so the `MQGMO_BROWSE_NEXT` option will have the same behavior as the `MQGMO_BROWSE_FIRST`.

```
gmo.setOptions(MQGMO_BROWSE_NEXT | MQGMO_WAIT);
```

```
// Browsing all the messages in the queue in sequential order
while (pqueue.get(msg,gmo)) {
    // Perform some operation with the message.
    ...
}
```

```

// The MessageId and CorrelationId must be set to null before
// the next get method call.
msg.setMessageId(MQMI_NONE);
msg.setCorrelId(MQCI_NONE);
}

```

Messages can be browsed in either physical or logical order.

Physical ordering can be FIFO (First-in/First-out) or FIFO within Priority ordering, depending on the value of the Message Delivery Sequence (MsgDeliverySequence) for the queue.

Logical order means that messages belonging to a group will be presented sequentially in their correct position in the queue, even if any message of a different group appears in the queue before the last message of that group is received.

To browse messages in a logical order, we must specify the MQGMO_LOGICAL_ORDER option when calling the get method.

```
gmo.setOptions(MQGMO_BROWSE_NEXT | MQGMO_WAIT | MQGMO_LOGICAL_ORDER);
```

For additional information about this topic, please refer to the *Application Programming Guide* that comes with MQSeries.

5.6.2 Inquiring about and setting object attributes

Inquiring about attributes

Inquiring about, and setting object attributes with this API is a straight forward operation when compared with the MQI API. Here, the ImqObject class offers two inquire methods that inquire any integer or character attribute indicated.

```

ImqBoolean inquire(const MQLONG int-attr, MQLONG & value );
ImqBoolean inquire(const MQLONG char-attr, char * & buffer, const size_t
length);

```

The int-attr and char-attr parameters give the attribute MQIA_* and MQCA_* indexes.

The integer object attributes value is returned in the value parameter, as shown in the following code fragment:

```

MQLONG depth;

pqueue.inquire(MQIA_CURRENT_Q_DEPTH, depth);

```

```
printf("The current queue depth is: %d",depth);
```

The character object attributes value is returned in the buffer parameter, as shown in the following code fragment:

```
char qname[MQCA_Q_MGR_NAME_LENGTH];

pqueue.inquire(MQCA_Q_MGR_NAME, qname, MQCA_Q_MGR_NAME_LENGTH);
printf("The current queue depth is: %s",qname);
```

The buffer must be large enough to hold the attribute value. The length of the buffer must be specified in the length parameter.

Setting object attributes

To set queue attributes, the `ImqObject` provides two set methods just like those provided to inquire attributes that we have already explained.

```
ImqBoolean set(const MQLONG int-attr, MQLONG & value);

ImqBoolean set(const MQLONG char-attr, char * buffer, const size_t length);
```

The following code fragment shows the possible usage of these functions:

```
// This instruction inhibits any put operation on any type of queue.
pqueue.set(MQIA_INHIBIT_PUT,MQQA_PUT_INHIBITED);
```

or

```
// This instruction inhibits any get operation on any local queue.
pqueue.set(MQIA_INHIBIT_GET,MQQA_GET_INHIBITED);
```

Only the following queue attributes value can be modified using these functions:

- ▶ Available for all types of queue
 - MQIA_INHIBIT_PUT
- ▶ Available for local queues
 - MQCA_TRIGGER_DATA
 - MQIA_DIST_LISTS
 - MQIA_INHIBIT_GET
 - MQIA_TRIGGER_CONTROL
 - MQIA_TRIGGER_DEPTH
 - MQIA_TRIGGER_MSG_PRIORITY
 - MQIA_TRIGGER_TYPE
 - MQIA_DIST_LISTS

- ▶ Available for alias queues
 - MQIA_INHIBIT_GET

5.7 Transaction management

Local resource manager transactions can be started by setting the syncpoint participation in the `ImqPutMessages` or `ImqGetMessageOptions` classes.

```
ImqPutMessageOptions pmo;  
  
//This starts a local resource manager transaction  
pmo.setSyncPointParticipation(TRUE);  
pqueue.put(msg,pmo);
```

or

```
ImqGetMessageOptions gmo;  
  
//This starts a local resource manager transaction  
gmo.setSyncPointParticipation(TRUE);  
pqueue.get(msg,gmo);
```

The `ImqQueueManager` object provides the transaction management interfaces required to begin, commit or rollback distributed transactions with this API.

A distributed transaction starts with a `ImqQueueManager` begin method call. Any operation within a transaction begin and end call is part of the transaction.

```
// This call starts a distributed transaction  
pmanager.begin();
```

Distributed transaction can only be started if there are no other local or distributed transactions.

Both, local and distributed transaction can be terminated with a `ImqQueueManager` commit method call if the transaction was successful, or with a `ImqQueueManager` backout method call.

```
pmanager.commit();
```

or

```
pmanager.backout();
```

5.8 Message grouping

Messages can be grouped using the `ImqMessageTracker` `setGroupId` method. We also have to identify messages in the group by giving the `MQMF_MSG_IN_GROUP` or `MQMF_LAST_MSG_IN_GROUP` flags using the `ImqMessage` `setMessageFlags` method.

Example 5-6 on page 152 shows how to send three messages as a group. The first two messages will be sent using the `MQMF_MSG_IN_GROUP` flag while the third message will use the `MQMF_LAST_MSG_IN_GROUP`.

Example 5-6 Message grouping

```
// Setting put message options and message descriptor versions.
BYTE24 MY_GROUP_ID = "123456";
ImqPutMessageOptions pmo;
ImqMessage message;
ImqBinary grpId;

// Set the grpId binary object value
grpId.set(MY_GROUP_ID, sizeof(MY_GROUP_ID));

// Sets the put message options to generate a new message ID for every
// message put into the queue and to put the messages in their logical
// order into the queue.
pmo.setOptions(MQPMO_LOGICAL_ORDER | MQPMO_NEW_MSG_ID);
message.setMessageFlags(MQMF_MSG_IN_GROUP);

// Assign the GroupId
message.setGroupId(grpId);

// Puts a first message of the group
message.write("First message");
pqueue.put(message, pmo);

// Puts a second message of the group
message.write("Middle message");
pqueue.put(message, pmo);

// Puts the final message of the group. The final message must
// be identified by giving the MQMF_LAST_MSG_IN_GROUP flags in the message
// descriptor structure.
message.setMessageFlags(MQMF_LAST_MSG_IN_GROUP);
message.write("Last message");
pqueue.put(message, pmo);
```

These messages can then be retrieved from the queue in their logical order using the `MQGMO_LOGICAL_ORDER` option in the `setOptions` method of the `ImqGetMessageOptions` class.

```
ImqPutMessageOptions pmo;
ImqMessage message;
char buffer[101];

message.setEmptyBuffer(buffer, sizeof(buffer)-1);

// Set the get message options required for this operation. Specially
// the MQGMO_LOGICAL_ORDER
gmo.setOptions(MQGMO_LOGICAL_ORDER + MQGMO_WAIT + MQGMO_CONVERT);
gmo.setWaitInterval(1500);      /* 15 second limit for waiting */

// We want to get all the messages on the queue so no match options will
// be needed.
gmo.setMatchOptions(MQGMO_NONE);

while (pqueue.completionCode() != MQCC_FAILED) {
    md.setEncoding(MQENC_NATIVE);
    md.setCharacterSet(MQCCSI_Q_MGR);

    if (pqueue.get(message)) {

        // Shows the message data
        buffer[message.dataLength] = 0;
        printf("message <%s>\n", buffer);

        /* report reason, if any */
        if (queue.reasonCode() != MQRC_NONE) {
            // general report for other reasons
            printf("MQGET ended with reason code %ld\n", Reason);
        } else {
            if (queue.reasonCode() == MQRC_NO_MSG_AVAILABLE) {
                // special report for normal end
                printf("no more messages\n");
            }
        }
    }
}
```

Additionally, the queue manager can control whether or not a message group has been received completely. If we want only complete message groups to appear in the queue, then the `MQGMO_ALL_MSGS_AVAILABLE` option can be set in the `ImqGetMessageOptions` `setOptions` method, along with the others.

5.9 Exploring the patterns

All the patterns explained in Chapter 1, “Introduction and patterns” on page 3 can be developed using this API. In the following section we will show simple examples of each one of these pattern. These examples were developed based upon the MQI C examples in Chapter 3, “Programming with MQI” on page 23, so the user can compare the examples and appreciate how the MQI API maps to the C++ API.

All these examples were developed using the Microsoft Visual Studio environment.

5.9.1 The one-to-one or point-to-point pattern

As explained in Chapter 1, “Introduction and patterns” on page 3, the one-to-one or point-to-point programming pattern can be used for a send-and-forget scenario as well as a request/reply scenario or any combination of those.

Here, we present a simple example of each one of them. The message data used in these examples do not have any business logic, but the examples can be easily modified to be applied in a real world situation.

Send-and-forget

This simple example of the send-and-forget pattern contains two programs. The first one acts as the message sender while the second one acts as the message consumer. No response or acknowledgment is expected by the sender and nothing is sent back by the consumer.

The sender program shown in Example 5-7 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the PTP.QUEUE.LOCAL queue for output
- ▶ Prepare a message to be sent
- ▶ Send the message to the opened queue
- ▶ Close the queue
- ▶ Disconnect from the queue manager

Example 5-7 Sender program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define QMGR_NAME "SAMPLE.QMGR1"
#define Q_NAME "PTP.QUEUE.LOCAL"
```

```

#include <imqi.hpp> // MQSeries MQI

int main ( int argc, char * * argv ) {
    ImqQueueManager mgr;           // Queue manager
    ImqQueue queue;               // Queue
    ImqMessage msg;               // Data message
    char    buffer[ 256 ];        // Message buffer

    printf( "Send/Forget Sample\n" );

    // Sets the queue manager name
    mgr.setName( QMGR_NAME );

    if ( ! mgr.connect( ) ) {
        /* stop if it failed */
        printf( "ImqQueueManager::connect ended with reason code %d\n",
            (int)mgr.reasonCode( ) );
        exit( (int)mgr.reasonCode( ) );
    }

    // Associate queue with queue manager.
    queue.setConnectionReference( mgr );

    // Sets the name of the target queue
    queue.setName( Q_NAME );
    printf( "target queue is %s\n", Q_NAME );

    // Open the target message queue for output
    queue.setOpenOptions( MQOO_OUTPUT/* open queue for output      */
        + MQOO_FAIL_IF_QUIESCING ); /* but not if MQM stopping */
    // Explicitly opens the queue
    queue.open( );

    /* report reason, if any; stop if failed */
    if ( queue.reasonCode( ) ) {
        printf( "ImqQueue::open ended with reason code %d\n",
            (int)queue.reasonCode( ) );
    }

    if ( queue.completionCode( ) == MQCC_FAILED ) {
        printf( "unable to open queue for output\n" );
    }

    // Prepare the message to be sent
    msg.useEmptyBuffer( buffer, sizeof( buffer ) );
    msg.setFormat( MQFMT_STRING ); /* character string format */
    strcpy(buffer,"This is a simple Send/Forget sample");
    msg.setMessageLength( strlen(buffer) );

```

```

if ( ! queue.put( msg ) ) {
    /* report reason, if any */
    printf( "ImqQueue::put ended with reason code %d\n",
        (int)queue.reasonCode( ) );
}

// Close the target queue (if it was opened)
if ( ! queue.close( ) ) {
    /* report reason, if any */
    printf( "ImqQueue::close ended with reason code %d\n",
        (int)queue.reasonCode( ) );
}

// Disconnect from MQM if not already disconnected (the
// ImqQueueManager object handles this situation automatically)
if ( ! mgr.disconnect( ) ) {
    /* report reason, if any */
    printf( "ImqQueueManager::disconnect ended with reason code %d\n",
        (int)mgr.reasonCode( ) );
}

printf( "Send/Forget Sample end\n" );
return( 0 );
}

```

The consumer program shown in Example 5-8 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the PTP.QUEUE.LOCAL queue for output
- ▶ Set the data buffer for the incoming message
- ▶ Get the message from the opened queue
- ▶ Print the message received
- ▶ Close the queue
- ▶ Disconnect from the queue manager

Example 5-8 Consumer program

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define QMGR_NAME "SAMPLE.QMGR1.ITSOE"
#define Q_NAME "PTP.QUEUE.LOCAL"

#include <imqi.hpp> // MQSeries MQI

int main ( int argc, char * * argv ) {
    ImqQueueManager mgr;          // Queue manager

```

```

ImqQueue queue;                // Queue
ImqMessage msg;                // Data message
ImqGetMessageOptions gmo;      // Get Message Options
char    buffer[ 256 ];         // Message buffer

printf( "Send/Forget Sample\n" );

// Sets the queue manager name
mgr.setName( QMGR_NAME );

if ( ! mgr.connect( ) ) {
    /* stop if it failed */
    printf( "ImqQueueManager::connect ended with reason code %d\n",
            (int)mgr.reasonCode( ) );
    exit( (int)mgr.reasonCode( ) );
}

// Associate queue with queue manager.
queue.setConnectionReference( mgr );

// Sets the name of the target queue
queue.setName( Q_NAME );
printf( "target queue is %s\n", Q_NAME);

// Open the target message queue for input
queue.setOpenOptions( MQOO_INPUT_SHARED/* open queue for input      */
    + MQOO_FAIL_IF_QUIESCING ); /* but not if MQM stopping */
// Explicitly opens the queue
queue.open( );

/* report reason, if any; stop if failed */
if ( queue.reasonCode( ) ) {
    printf( "ImqQueue::open ended with reason code %d\n",
            (int)queue.reasonCode( ) );
}

if ( queue.completionCode( ) == MQCC_FAILED ) {
    printf( "unable to open queue for output\n" );
}

// Sets the get message options wait interval
gmo.setOptions(MQGMO_WAIT);/* Enables wait for this get operation*/
gmo.setWaitInterval(MQWI_UNLIMITED);/* Sets wait interval to unlimited */

// Prepare the message to be sent
msg.useEmptyBuffer( buffer, sizeof( buffer ) );

if ( ! queue.get( msg, gmo ) ) {
    /* report reason, if any */

```

```

        printf( "ImqQueue::put ended with reason code %d\n",
                (int)queue.reasonCode( ) );
    }

    // Close the target queue (if it was opened)
    if ( ! queue.close( ) ) {
        /* report reason, if any */
        printf( "ImqQueue::close ended with reason code %d\n",
                (int)queue.reasonCode( ) );
    }

    // Disconnect from MQM if not already disconnected (the
    // ImqQueueManager object handles this situation automatically)
    if ( ! mgr.disconnect( ) ) {
        /* report reason, if any */
        printf( "ImqQueueManager::disconnect ended with reason code %d\n",
                (int)mgr.reasonCode( ) );
    }

    printf( "Send/Forget Sample end\n" );
    return( 0 );
}

```

Request/reply

As in the send-and-forget pattern sample, this request/reply sample contains two programs. The first one sends a request message to a queue called PTP.QUEUE.LOCAL and waits for a response in another queue PTP.REPLY.QUEUE.LOCAL. The second program acts as the reply program. It starts reading messages from a queue called PTP.QUEUE.LOCAL and whenever a message is put onto that queue it sends a generic response to the PTP.REPLY.QUEUE.LOCAL queue.

The request program shown in Example 5-9 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the request (PTP.QUEUE.LOCAL) queue for output
- ▶ Open the reply (PTP.REPLY.QUEUE.LOCAL) queue for input
- ▶ Prepare the request message to be sent
- ▶ Send the request message to the opened queue
- ▶ Set the data buffer for the incoming message
- ▶ Assign the correlId used to identify the reply message
- ▶ Wait for the reply message in the reply queue
- ▶ Show the received reply message data.
- ▶ Close the queues
- ▶ Disconnect from the queue manager

Example 5-9 Request program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define QMGR_NAME "SAMPLE.QMGR1.ITSOE"
#define REQUEST_Q_NAME "PTP.QUEUE.LOCAL"
#define REPLY_Q_NAME "PTP.REPLY.QUEUE.LOCAL"

#include <imqi.hpp> // MQSeries MQI

int main ( int argc, char * * argv ) {
    ImqQueueManager mgr;           // Queue manager
    ImqQueue requestQueue;         // Request Queue
    ImqQueue replyQueue;          // Reply Queue
    ImqGetMessageOptions gmo;      // Get Message Options
    ImqMessage pmsg;               // Data message
    ImqMessage gmsg;               // Data message
    char    buffer[ 256 ];         // Message buffer

    printf( "Request/Reply Sample\n" );

    mgr.setName( QMGR_NAME );

    if ( ! mgr.connect( ) ) {
        // stop if it failed
        printf( "ImqQueueManager::connect ended with reason code %d\n",
            (int)mgr.reasonCode( ) );
        exit( (int)mgr.reasonCode( ) );
    }

    // Associate queues with queue manager.
    requestQueue.setConnectionReference( mgr );
    replyQueue.setConnectionReference( mgr );

    // set the name of the request and reply queues
    requestQueue.setName( REQUEST_Q_NAME );
    printf( "Request queue is %s\n", REQUEST_Q_NAME );
    replyQueue.setName( REPLY_Q_NAME );
    printf( "Request queue is %s\n", REPLY_Q_NAME );

    // Open the request message queue for output
    requestQueue.setOpenOptions( MQOO_OUTPUT/* open queue for output      */
        + MQOO_FAIL_IF_QUIESCING );/* but not if MQM stopping */
    requestQueue.open( );

    // report reason, if any; stop if failed
    if ( requestQueue.reasonCode( ) ) {
```

```

        printf( "ImqQueue::open ended with reason code %d\n",
                (int)requestQueue.reasonCode( ) );
    }

    if ( requestQueue.completionCode( ) == MQCC_FAILED ) {
        printf( "unable to open queue for output\n" );
    } else {

        // Open the reply message queue for input
        replyQueue.setOpenOptions( MQOO_INPUT_SHARED/* open queue for input
*/
                                + MQOO_FAIL_IF_QUIESCING ); /* but not if MQM stopping */
        replyQueue.open( );

        /* report reason, if any; stop if failed */
        if ( replyQueue.reasonCode( ) ) {
            printf( "ImqQueue::open ended with reason code %d\n",
                    (int)replyQueue.reasonCode( ) );
        }

        if ( replyQueue.completionCode( ) == MQCC_FAILED ) {
            printf( "unable to open queue for output\n" );
        } else {
            // Sets the request message buffer and type
            pmsg.useEmptyBuffer( buffer, sizeof( buffer ) );
            pmsg.setFormat( MQFMT_STRING ); /* character string format */
            strcpy(buffer,"This is a simple Request message");
            pmsg.setMessageLength( strlen(buffer) );

            if ( ! requestQueue.put( pmsg ) ) {
                // report reason, if any
                printf( "ImqQueue::put ended with reason code %d\n",
                        (int)requestQueue.reasonCode( ) );
            } else {
                // waits for reply

                // Setting the get message options
                gmo.setOptions(MQGMO_WAIT);
                // Enables wait for this get operation
                gmo.setWaitInterval(MQWI_UNLIMITED);
                // Sets wait interval to unlimited
                gmo.setMatchOptions(MQMO_MATCH_CORREL_ID);
                // Sets the message match options to correlId

                // Sets the reply message buffer and type
                msgsg.useEmptyBuffer( buffer, sizeof( buffer ) );
                msgsg.setFormat( MQFMT_STRING ); /* character string format */

                // Sets correlId use to selected reply message from the queue

```



```

        gmsg.setCorrelationId(pmsg.correlationId());

        if ( ! replyQueue.get( gmsg, gmo ) ) {
            // report reason, if any
            printf( "ImqQueue::put ended with reason code %d\n",
                (int)replyQueue.reasonCode( ) );
        } else {
            // Add terminator to message string
            buffer[gmsg.dataLength()] = 0;
            printf("The reply received is: %s\n",buffer);
        }
    }
    // Close the target queue (if it was opened)
    if ( ! replyQueue.close( ) ) {
        /* report reason, if any */
        printf( "ImqQueue::close ended with reason code %d\n",
            (int)replyQueue.reasonCode( ) );
    }
}

// Close the target queue (if it was opened)
if ( ! requestQueue.close( ) ) {
    /* report reason, if any */
    printf( "ImqQueue::close ended with reason code %d\n",
        (int)requestQueue.reasonCode( ) );
}
}

// Disconnect from MQM if not already disconnected (the
// ImqQueueManager object handles this situation automatically)
if ( ! mgr.disconnect( ) ) {
    /* report reason, if any */
    printf( "ImqQueueManager::disconnect ended with reason code %d\n",
        (int)mgr.reasonCode( ) );
}

printf( "Request/Reply Sample end\n" );
return( 0 );
};

```

The reply program shown in Example 5-10 follows the logical flow below:

- ▶ Connect to a queue manager
- ▶ Open the request (PTP.QUEUE.LOCAL) queue for input
- ▶ Open the reply (PTP.REPLY.QUEUE.LOCAL) queue for output
- ▶ Set the data buffer for the incoming message
- ▶ Wait for the request message in the request queue
- ▶ Show the received request message data

- ▶ Prepare the reply message to be sent
- ▶ Assign the correlId used to identify the reply message
- ▶ Send the reply message to the opened queue
- ▶ Close the queues
- ▶ Disconnect from the queue manager

Example 5-10 Reply program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define QMGR_NAME "SAMPLE.QMGR1.ITSOE"
#define REQUEST_Q_NAME "PTP.QUEUE.LOCAL"
#define REPLY_Q_NAME "PTP.REPLY.QUEUE.LOCAL"

#include <imqi.hpp> // MQSeries MQI

int main ( int argc, char * * argv ) {
    ImqQueueManager mgr;           // Queue manager
    ImqQueue requestQueue;         // Request Queue
    ImqQueue replyQueue;          // Reply Queue
    ImqGetMessageOptions gmo;      // Get Message Options
    ImqMessage pmsg;              // Data message
    ImqMessage gmsg;              // Data message
    char    buffer[ 256 ];        // Message buffer

    printf( "Request/Reply Sample\n" );

    mgr.setName( QMGR_NAME );

    if ( ! mgr.connect( ) ) {
        // stop if it failed
        printf( "ImqQueueManager::connect ended with reason code %d\n",
            (int)mgr.reasonCode( ) );
        exit( (int)mgr.reasonCode( ) );
    }

    // Associate queues with queue manager.
    requestQueue.setConnectionReference( mgr );
    replyQueue.setConnectionReference( mgr );

    // set the name of the request and reply queues
    requestQueue.setName( REQUEST_Q_NAME );
    printf( "Request queue is %s\n", REQUEST_Q_NAME );
    replyQueue.setName( REPLY_Q_NAME );
    printf( "Reply queue is %s\n", REPLY_Q_NAME );

    // Open the request message queue for input
```

```

requestQueue.setOpenOptions( MQOO_INPUT_SHARED
//open queue for input
    + MQOO_FAIL_IF_QUIESCING );/* but not if MQM stopping */
requestQueue.open( );

/* report reason, if any; stop if failed */
if ( requestQueue.reasonCode( ) ) {
    printf( "ImqQueue::open ended with reason code %d\n",
        (int)requestQueue.reasonCode( ) );
}

if ( requestQueue.completionCode( ) == MQCC_FAILED ) {
    printf( "unable to open queue for output\n" );
} else {

    // Open the reply message queue for output
    replyQueue.setOpenOptions( MQOO_OUTPUT/* open queue for output */
        + MQOO_FAIL_IF_QUIESCING );/* but not if MQM stopping */
    replyQueue.open( );

    /* report reason, if any; stop if failed */
    if ( replyQueue.reasonCode( ) ) {
        printf( "ImqQueue::open ended with reason code %d\n",
            (int)replyQueue.reasonCode( ) );
    }

    if ( replyQueue.completionCode( ) == MQCC_FAILED ) {
        printf( "unable to open queue for output\n" );
    } else {
        // Sets the get message options wait interval
        gmo.setOptions(MQGMO_WAIT);/* Enables wait for this get operation*/
        gmo.setWaitInterval(MQWI_UNLIMITED);
        // Sets wait interval to unlimited

        // Sets the reply message buffer and type
        gmsg.useEmptyBuffer( buffer, sizeof( buffer ) );
        gmsg.setFormat( MQFMT_STRING ); /* character string format */

        // Sets the correlId use to select the reply message from the queue

        if ( ! requestQueue.get( gmsg, gmo ) ) {
            // report reason, if any
            printf( "ImqQueue::put ended with reason code %d\n",
                (int)requestQueue.reasonCode( ) );
        } else {
            // Add terminator to message string
            buffer[gmsg.dataLength()] = 0;
            printf("The request received is: %s\n",buffer);
        }
    }
}

```

```

        // Set the message buffer in the ImqMessage object
        pmsg.useEmptyBuffer( buffer, sizeof( buffer ) );
        pmsg.setFormat( MQFMT_STRING ); /* character string format */
        strcpy(buffer,"This is a simple Reply message");
        pmsg.setMessageLength( strlen(buffer) );

        // Sets the correlationId of the message being sent
        pmsg.setCorrelationId(gmsg.correlationId());

        if ( ! replyQueue.put( pmsg ) ) {
            // report reason, if any
            printf( "ImqQueue::put ended with reason code %d\n",
                (int)replyQueue.reasonCode( ) );
        }
    }

    // Close the target queue (if it was opened)
    if ( ! replyQueue.close( ) ) {
        /* report reason, if any */
        printf( "ImqQueue::close ended with reason code %d\n",
            (int)replyQueue.reasonCode( ) );
    }
}

// Close the target queue (if it was opened)
if ( ! requestQueue.close( ) ) {
    /* report reason, if any */
    printf( "ImqQueue::close ended with reason code %d\n",
        (int)requestQueue.reasonCode( ) );
}
}

// Disconnect from MQM if not already disconnected (the
// ImqQueueManager object handles this situation automatically)
if ( ! mgr.disconnect( ) ) {
    /* report reason, if any */
    printf( "ImqQueueManager::disconnect ended with reason code %d\n",
        (int)mgr.reasonCode( ) );
}

printf( "Send/Forget Sample end\n" );
return( 0 );
};

```

5.9.2 The publish/subscribe pattern

The publish/subscribe pattern has two major components, as explained in Chapter 1, “Introduction and patterns” on page 3:

- ▶ The Publisher: This component is the one that actually publish the topics in the broker stream.
- ▶ The Subscriber: This component represent a client of the publisher or publishers. It subscribes to one or many topics, and waits for any publication on these topics to be sent to it by the broker.

We will explore these two components with a simple example where a publisher program publishes some data on a given topic and any number of subscribers will receive that data and show it in the standard output.

Publisher

The publisher program presented here is divided into three functions:

- ▶ The BuildMQRFHeader function
This function constructs an MQRFH data structure and appends the required value/pair at the end of this structure.
- ▶ The PutPublication function
This function is responsible for sending the actual publication commands to the broker using the stream queue.
- ▶ The main function
This function constructs the publication message and sends it to the stream queue.

Example 5-11 shows the BuildMQRFHeader function described above. This code was taken from the C samples that come with the publish/subscribe SupportPac.

Example 5-11 The BuildMQRFHeader function

```
void BuildMQRFHeader( PMQBYTE  pStart,
                     PMQLONG  pDataLength,
                     MQCHAR    TopicType[] ) {
    PMQRFH  pRFHeader = (PMQRFH)pStart;
    PMQCHAR pNameValueString;

    /* Clear the buffer before we start (initialise to nulls). */
    memset((PMQBYTE)pStart, 0, *pDataLength);

    /* Clear the buffer before we start (initialise to nulls). */
```

```

/* Copy the MQRFH default values into the start of the buffer. */
/*****
memcpy( pRFHeader, &DefaultMQRFH, (size_t)MQRFH_STRUC_LENGTH_FIXED);

/*****
/* Set the format of the user data to be MQFMT_STRING, even though */
/* some of the publications use a structure to pass user data the */
/* data within this structure is entirely MQCHAR and can be */
/* treated as MQFMT_STRING by the data conversion routines. */
/*****
memcpy( pRFHeader->Format, MQFMT_STRING, (size_t)MQ_FORMAT_LENGTH);

/*****
/* As we have user data following the MQRFH we must set the CCSID */
/* of the user data in the MQRFH for data conversion to be able to */
/* be performed by the queue manager. As we do not currently know */
/* the CCSID that we are running in we can tell MQSeries that the */
/* data that follows the MQRFH is in the same CCSID as the MQRFH. */
/* The MQRFH will default to the CCSID of the queue manager */
/* (MQCCSI_Q_MGR), so the user data will also inherit this CCSID. */
/*****
pRFHeader->CodedCharSetId = MQCCSI_INHERIT;

/*****
/* Start the NameValueString directly after the MQRFH structure. */
/*****
pNameValueString = (MQCHAR *)pRFHeader + MQRFH_STRUC_LENGTH_FIXED;

/*****
/* Add the command to the start of the NameValueString, this must */
/* always be the first MQPS name token in the string. */
/*****
strcpy(pNameValueString, MQPS_COMMAND_B);
strcat(pNameValueString, MQPS_PUBLISH);

/*****
/* Add the publication options and topic to the NameValueString. */
/* We specify 'no registration' because neither sample application */
/* is concerned with who is currently publishing, it also allows */
/* us not to specify an identity queue (we are also publishing */
/* datagrams so no replies will be sent either) which means that */
/* we do not have to define a queue for this sample to use. */
/*****
strcat(pNameValueString, MQPS_PUBLICATION_OPTIONS_B);
strcat(pNameValueString, MQPS_NO_REGISTRATION);

strcat(pNameValueString, MQPS_TOPIC_B);
strcat(pNameValueString, TOPIC_PREFIX);
strcat(pNameValueString, TopicType);

```

```

/*****
/* Any user data that follows the NameValueString should start on */
/* a word boundary, to ensure all platforms are satisfied we align */
/* to a 16 byte boundary. */
/* As the NameValueString has been null terminated (by using */
/* strcat) any characters between the end of the string and the */
/* next 16 byte boundary will be ignored by the broker, but if the */
/* message is to be data converted we advise any extra characters */
/* are set to nulls ('\0') or blanks (' '). In this sample we have */
/* initialised the whole message block to nulls before we started */
/* so all extra characters will be nulls by default. */
*****/
*pDataLength = MQRFH_STRUC_LENGTH_FIXED
               + ((strlen(pNameValueString)+15)/16)*16;
pRFHeader->StrucLength = *pDataLength;
}

```

The PutPublication function shown in Example 5-12 follows the logical flow below:

- ▶ Sets the message format to MQFMT_RF_HEADER
- ▶ Sets the message type to MQMT_DATAGRAM
- ▶ Sets the message persistence
- ▶ Set the MQPMO_NEW_MSG_ID option to request a new message Id for the message that is being sent
- ▶ Set the message buffer using the useFullBuffer method
- ▶ Sends the message and returns true if successful

Example 5-12 The PutPublication function

```

bool PutPublication( ImqQueue& queue, PMQBYTE pMessage, MQLONG
messageLength) {
    ImqPutMessageOptions pmo;
    ImqMessage msg;

    // Sets message format, type, persistence and options
    msg.setFormat(MQFMT_RF_HEADER);
    msg.setMessageType(MQMT_DATAGRAM);
    msg.setPersistence(MQPER_PERSISTENT);
    pmo.setOptions(MQPMO_NEW_MSG_ID);

    // Sets the message buffer as a full buffer
    msg.useFullBuffer((char *) pMessage,messageLength);

    // Puts the message in the queue and return true if it was successful

```

```

        if (queue.put(msg,pmo))
            return true;
        else
            return false;
    }

```

The main function shown in Example 5-13 follows the logical flow below:

- ▶ Connect to the queue manager
- ▶ Open the stream queue for output
- ▶ Allocate the message data buffer
- ▶ Build the MQRFH Header structure using the BuildMQRFHHeader function
- ▶ Append the publication data to the end of the MQRFH Header structure
- ▶ Put the publication in the stream queue using the PutPublication function

Example 5-13 The main function (Publisher example)

```

int main(int argc, char **argv) {
    ImqQueueManager mgr;           // Queue manager
    ImqQueue streamQueue;         // Request Queue

    PMQBYTE      pMessageBlock = NULL;
    MQLONG       messageLength;
    char  defaultText[] = "HELLO WORLD";
    char  text[256];

    printf( "Publisher Sample\n" );

    if (argc > 1) {
        strcpy(text,argv[1]);
    }

    mgr.setName( QMGR_NAME );

    if ( ! mgr.connect( ) ) {
        // stop if it failed
        printf( "ImqQueueManager::connect ended with reason code %d\n",
            (int)mgr.reasonCode( ) );
        exit( (int)mgr.reasonCode( ) );
    } else {
        // Associate queues with queue manager.
        streamQueue.setConnectionReference( mgr );
        streamQueue.setName( STREAM_Q_NAME );

        streamQueue.setOpenOptions( MQOO_OUTPUT
            // open queue for output
            + MQOO_FAIL_IF QUIESCING );/* but not if MQM stopping */
        streamQueue.open( );
    }
}

```



```

    if ( streamQueue.reasonCode( ) ) {
        printf( "ImqQueue::open ended with reason code %d\n",
                (int)streamQueue.reasonCode( ) );
    }

    if ( streamQueue.completionCode( ) == MQCC_FAILED ) {
        printf( "unable to open queue for output\n" );
    } else {
        messageLength = DEFAULT_MESSAGE_SIZE;
        pMessageBlock = (PMQBYTE)malloc(messageLength);

        BuildMQRFHeader( pMessageBlock
                        , &messageLength
                        , "TEST");

        strcpy((char *)pMessageBlock+messageLength,text);
        messageLength += strlen(text);

        if (PutPublication( streamQueue, pMessageBlock, messageLength)) {
            printf("The message has been publish successfully.\n");
        } else {
            printf("The message could not be publish.\n");
        }
    }
}

return (0);
}

```

Subscriber

The subscriber program presented here is basically divided into four functions:

- ▶ The BuildMQRFHeader function

This function constructs an MQRFH data structure and appends the required value/pair at the end of this structure.
- ▶ The CheckForResponse function

This function waits for a subscription acknowledgment from the broker, and validates that the subscription has been successfully accepted.
- ▶ The PubSubCommand function

This function creates and send a publish/subscribe command to the broker and checks for the broker response using the CheckForResponse function
- ▶ The main function

This function provides two possible behaviors depending on the number of parameters used when calling this program:

- If two parameters are specified, the first parameter will be taken as the client queue where the broker will send the publications for the client, and the second parameter will be used as the correlationId used to identify the publication of that specific client.

The main function will register the client to the topic and will start an infinite loop getting publications.

- If a third parameter is specified the main function will de-register the client from the topic and finish.

The CheckResponse function uses the PrintNameValueString function which basically prints the name/value pairs to the screen. This function is not shown here but may be found in the additional materials that be downloaded from the IBM Redbooks Web site.

Example 5-14 shows the BuildMQRHeader function described above. This code was taken from the C samples that come with the publish/subscribe SupportPac.

Example 5-14 The BuildMQRHeader (Subscriber example)

```
void BuildMQRHeader( PMQBYTE   pStart
                    , PMQLONG  pDataLength
                    , PMQCHAR   pCommand
                    , MQLONG    regOptions
                    , MQLONG    pubOptions
                    , PMQCHAR   pTopic ) {
    PMQRFH  pRFHeader = (PMQRFH)pStart;
    PMQCHAR pNameValueString;

    /******
    /* Clear the buffer before we start (initialise to nulls).    */
    /******
    memset((PMQBYTE)pStart, '\0', *pDataLength);

    /******
    /* Copy the MQRFH default values into the start of the buffer.    */
    /******
    memcpy( pRFHeader, &DefaultMQRFH, (size_t)MQRFH_STRUC_LENGTH_FIXED);

    /******
    /* Start the NameValueString directly after the MQRFH structure.    */
    /******
    pNameValueString = (MQCHAR *)pRFHeader + MQRFH_STRUC_LENGTH_FIXED;

    /******
    /* Add the command to the start of the NameValueString, this must    */
    /******
```

```

/* always be the first MQPS name token in the string. */
/*****
strcpy(pNameValueString, MQPS_COMMAND_B);
strcat(pNameValueString, pCommand);

/*****
/* If registration options were supplied add them to the string, */
/* for ease of implementation we insert the decimal representation */
/* of the options into the string as opposed to the character */
/* strings supplied for each option. */
/*****
if( regOptions != 0 ) {
    strcat(pNameValueString, MQPS_REGISTRATION_OPTIONS_B);
    sprintf(pNameValueString, "%s %d", pNameValueString, regOptions);
}

/*****
/* If publication options were supplied add them to the string, */
/* for ease of implementation we insert the decimal representation */
/* of the options into the string as opposed to the character */
/* strings supplied for each option. */
/*****
if( pubOptions != 0 ) {
    strcat(pNameValueString, MQPS_PUBLICATION_OPTIONS_B);
    sprintf(pNameValueString, "%s %d", pNameValueString, pubOptions);
}

/*****
/* Add the stream name to the NameValueString (optional for */
/* publications). */
/*****
strcat(pNameValueString, MQPS_STREAM_NAME_B);
strcat(pNameValueString, STREAM_QUEUE);

/*****
/* Add the topic to the NameValueString. */
/*****
strcat(pNameValueString, MQPS_TOPIC_B);
strcat(pNameValueString, pTopic);

/*****
/* Any user data that follows the NameValueString should start on */
/* a word boundary, to ensure all platforms are satisfied we align */
/* to a 16 byte boundary. */
/* As the NameValueString has been null terminated (by using */
/* strcat) any characters between the end of the string and the */
/* next 16 byte boundary will be ignored by the broker, but if the */
/* message is to be data converted we advise any extra characters */
/* are set to nulls ('\0') or blanks (' '). In this sample we have */

```

```

/* initialised the whole message block to nulls before we started */
/* so all extra characters will be nulls by default. */
/*****
*pDataLength = MQRFH_STRUC_LENGTH_FIXED
               + ((strlen(pNameValueString)+15)/16)*16;
pRFHeader->StrucLength = *pDataLength;
*/
}

```

The CheckForResponse function shown in Example 5-15 follows the logical flow below:

- ▶ Prepares the message buffer
- ▶ Set the message options such as correlationId and wait interval options
- ▶ Wait for a response from the broker
- ▶ If the response is not received return a fail value
- ▶ If the response is received it validates the format of the response message.
- ▶ Then extract the MQRFH Header structure from the message and validates the completion code
- ▶ Show the response to the user if the subscription was not accepted

Example 5-15 The CheckForResponse function

```

MQLONG CheckForResponse( ImqQueue& queue,
                        ImqMessage msg,
                        PMQBYTE pMessageBlock,
                        MQLONG  blockSize) {

    ImqGetMessageOptions gmo;
    MQLONG  CompCode;
    MQLONG  Reason;
    PMQRFH  pMQRFHHeader;
    PMQCHAR  pNameValueString;
    PMQCHAR  pInputNameValueString;
    ULONG   stringLength;

    /*****
    /* Wait for a response message to arrive on our subscriber queue, */
    /* the response's correlId will be the same as the messageId that */
    /* the original message was sent with (returned in the message */
    /* object from the ImqQueue::put method call) so match against */
    /* this. */
    *****/
    gmo.setOptions(MQGMO_WAIT + MQGMO_CONVERT);
    gmo.setWaitInterval(MAX_RESPONSE_TIME);

```

```

gmo.setMatchOptions(MQMO_MATCH_CORREL_ID);
msg.useEmptyBuffer((char *) pMessageBlock, blockSize);
msg.setCorrelationId();
msg.setMessageId();

queue.get(msg,gmo);

if( queue.completionCode() != MQCC_OK ) {
    printf("MQGET failed with CompCode %d and Reason %d\n",
           queue.completionCode(),
queue.reasonCode());
    if( queue.reasonCode() == MQRC_NO_MSG_AVAILABLE )
        printf("No response sent by broker, check broker is running.\n");
} else {
    /******
    /* Check that the message is in the MQRFH format.          */
    /******
    if( msg.formatIs(MQFMT_RF_HEADER) ) {
        /******
        /* Locate the start of the NameValueString and its length.    */
        /******
        pMQRFHeader = (PMQRFH)pMessageBlock;
        pNameValueString = (PMQCHAR)(pMessageBlock
                                     + MQRFH_STRUC_LENGTH_FIXED);
        stringLength = pMQRFHeader->StrucLength
                       - MQRFH_STRUC_LENGTH_FIXED;

        /******
        /* The start of a response NameValueString is always in the    */
        /* same format:          */
        /* MQPSCompCode x MQPSReason y MQPSReasonText string ...    */
        /* We can scan the start of the string to check the CompCode  */
        /* and reason of the reply.          */
        /******
        sscanf(pNameValueString, "MQPSCompCode %d MQPSReason %d ",
               &CompCode, &Reason);

        if( CompCode != MQCC_OK ) {
            /******
            /* One possible error is acceptable, MQRCCF_NO_RETAINED_MSG, */
            /* which is returned from a Request Update when there is no  */
            /* retained message on the broker. This is an allowable      */
            /* error so we can continue as before.          */
            /******
            if( Reason == MQRCCF_NO_RETAINED_MSG ) {
                CompCode = MQCC_OK;
                Reason = MQRC_NONE;
            } else {
                /******
                /* Otherwise, display the error message supplied with the    */

```

```

        /* user data that was returned, this will be the original */
        /* commands NameValueString. */
        /******
        /* A response NameValueString is ALWAYS NULL terminated, */
        /* therefore, we can use printf to display it (as it is a */
        /* string in the true sense of the word). We do not */
        /* necessarily generate NULL terminated NameValueStrings */
        /* so we use the PrintNameValueString function to display */
        /* the NameValueString returned with the message, if any */
        /* (most error responses do return the original */
        /* NameValueString as user data). */
        /******
        printf("Error response returned :\n");
        printf(" %s\n",pNameValueString);
        if( msg.dataLength() != (unsigned) pMQRFHeader->StrucLength ) {
            printf("Original Command String:\n");
            pInputNameValueString =
                (PMQCHAR)(pMessageBlock + pMQRFHeader->StrucLength);
            PrintNameValueString(pInputNameValueString,
                                (msg.dataLength() - pMQRFHeader->StrucLength));
        }
    }
} else {
    /******
    /* If the message is not in the MQRFH format we have the wrong */
    /* message. */
    /******

    printf("Unexpected message format: %.8s\n", msg.format() );
    CompCode = MQCC_FAILED;
}
}

return CompCode;
}

```

The PubSubCommand function shown in Example 5-16 on page 175 follows the logical flow below:

- ▶ Build the MQRFH Header structure using the BuildMQRFHeader function
- ▶ Set message format and type
- ▶ Specify the replyTo queue for this message
- ▶ Request a new messgeld for this message
- ▶ Set the message buffer using the useFullBuffer method.

- Assign the correlationId for the request message (this correlationId is going to be used by the broker to send the message back to the subscriber).
- Put the command in the broker control queue
- Check for the broker response using the CheckForResponse function

Example 5-16 The PubSubCommand function

```

MQLONG PubSubCommand( ImqQueue &brokerQueue,
                      ImqQueue &replyQueue,
                      MQCHAR Command[],
                      PMQCHAR pTopic,
                      MQLONG topicLength,
                      ImqBinary &pCorrelId,
                      MQLONG regOptions
                      ) {

    ImqPutMessageOptions pmo;
    ImqMessage msg;
    MQLONG messageLength;
    PMQBYTE pMessageBlock = NULL;
    MQLONG CompCode = MQCC_FAILED;

    /******
    /* Allocate a block of storage to hold the Command message.          */
    /******
    messageLength = DEFAULT_MESSAGE_SIZE;
    pMessageBlock = (PMQBYTE)malloc(messageLength);

    if( pMessageBlock == NULL ) {
        printf("Unable to allocate storage\n");
        CompCode = MQCC_FAILED;
    } else {
        /******
        /* Define an MQRFH structure at the start of the allocated      */
        /* storage, fill in the required fields and generate the          */
        /* NameValueString that follows it.                               */
        /******
        BuildMQRFHHeader( pMessageBlock
                          , &messageLength
                          , Command
                          , regOptions
                          , MQPUBO_NONE
                          , pTopic );

        /******
        /* Send the command as a request so that a reply is returned to */
        /* us on completion at the broker.                                */
        /******

```

```

msg.setFormat(MQFMT_RF_HEADER);
msg.setMessageType(MQMT_REQUEST);
/*****
/* Specify the subscriber's queue in the ReplyToQ of the MD.      */
/* We have not put the subscriber's queue in the MQRFH           */
/* NameValueString so the one in the ReplyToQ of the MD will be  */
/* used as the identity of the subscriber.                        */
*****/
msg.setReplyToQueueName(ReplyToQueueName);
pmo.setOptions(MQPMO_NEW_MSG_ID);
/*****
/* All commands sent use the correlId as part of their identity. */
*****/
msg.useFullBuffer((char *) pMessageBlock, messageLength);
msg.setCorrelationId(pCorrelId);

/*****
/* Put the command message to the broker control queue.          */
*****/
brokerQueue.put(msg, pmo);

if( brokerQueue.completionCode() != MQCC_OK )
    printf("MQPUT failed with CompCode %d and Reason %d\n",
        brokerQueue.completionCode(), brokerQueue.reasonCode());
else {
    /*****
    /* The put was successful, now wait for a response from the    */
    /* broker to inform us if the command was accepted by the      */
    /* broker.                                                      */
    /* We use our command storage block to receive the response    */
    /* into to save on allocating extra storage.                  */
    *****/
    CompCode = CheckForResponse( replyQueue, msg, pMessageBlock,
        DEFAULT_MESSAGE_SIZE );
}
/*****
/* Free the storage.                                              */
*****/
free( pMessageBlock );
}

return CompCode;
}

```

Example 5-17 on page 177 shows the subscriber main function. This function follows the logical flow below:

- Connect to the queue manager

- ▶ Open the three required queues (control, stream and client queues)
- ▶ If only two arguments are received:
 - Send the registration command to the queue using the PubSubCommand function
 - Go into a infinite loop waiting for publications to arrive in the client queue.
- ▶ If a third argument is received, then it sends the de-registration command to the broker using the PubSubCommand function and finishes

Example 5-17 The main function (Subscriber example)

```
int main(int argc, char **argv) {
    ImqQueueManager mgr;
    ImqQueue    controlQueue;
    ImqQueue    streamQueue;
    ImqQueue    subscriberQueue;
    ImqGetMessageOptions gmo;
    ImqMessage   msg;

    PMQBYTE      pMessageBlock = NULL;
    MQLONG        messageLength;
    MQCHAR32      subscriptionTopic;
    PMQRFH        pMQRFHeader;
    PMQCHAR        pNameValueString;
    PMQBYTE        pUserData;
    MQLONG        nameValueStringLength;
    ImqBinaryEventCorrelId;
    MQBYTE24 byteId;

    if (argc < 3) {
        printf("Unexpected number of parameters\n");
        return (-1);
    }

    controlQueue.setName(CONTROL_QUEUE);
    streamQueue.setName(STREAM_QUEUE);
    subscriberQueue.setName(argv[1]);

    memset(byteId, 0, 24);
    memcpy(byteId, argv[2], strlen(argv[2]));
    EventCorrelId.set(byteId, sizeof(byteId));
    ReplyToQueueName = (char *) argv[1];
    mgr.setName(QMGR_NAME);

    if (!mgr.connect()) {
        printf("MQCONN failed with CompCode %d and Reason %d\n",
            mgr.completionCode(), mgr.reasonCode());
    }
}
```

```

        printf("Usage: amqsres <QManager>\n");
    }

    if( mgr.completionCode() == MQCC_OK ) {

        controlQueue.setOpenOptions(MQOO_OUTPUT);
        streamQueue.setOpenOptions(MQOO_OUTPUT);
        subscriberQueue.setOpenOptions(MQOO_INPUT_SHARED +
        MQOO_FAIL_IF_QUIESCING);

        controlQueue.setConnectionReference( mgr );
        streamQueue.setConnectionReference( mgr );
        subscriberQueue.setConnectionReference( mgr );

        if ( !controlQueue.open() ) {
            printf("MQOPEN failed to open with CompCode %d and Reason %d\n",
                controlQueue.completionCode(), controlQueue.reasonCode());
            printf("Usage: amqsres <QManager>\n");
        }

        if ( !streamQueue.open() ) {
            printf("MQOPEN failed to open with CompCode %d and Reason %d\n",
                streamQueue.completionCode(), streamQueue.reasonCode());
            printf("Usage: amqsres <QManager>\n");
        }

        if ( !subscriberQueue.open() ) {
            printf("MQOPEN failed to open with CompCode %d and Reason %d\n",
                subscriberQueue.completionCode(), subscriberQueue.reasonCode());
            printf("Usage: amqsres <QManager>\n");
        }

    }

    if (argc > 3) {
        strcpy( subscriptionTopic, TOPIC_PREFIX);
        strcat( subscriptionTopic, "");
        PubSubCommand( controlQueue, subscriberQueue,
            MQPS_DEREGISTER_SUBSCRIBER
            , subscriptionTopic
            , strlen(subscriptionTopic)
            , EventCorrelId
            , MQREGO_CORREL_ID_AS_IDENTITY);
    } else {
        strcpy( subscriptionTopic, TOPIC_PREFIX);
        strcat( subscriptionTopic, "");
        if(PubSubCommand( controlQueue, subscriberQueue,
            MQPS_REGISTER_SUBSCRIBER
            , subscriptionTopic

```

```

        , strlen(subscriptionTopic)
        , EventCorrelId
        , MQREGO_CORREL_ID_AS_IDENTITY) == MQCC_OK ) {
    /******
    /* Allocate a block of memory for the publications to be
    /* loaded into by MQGET. We know the maximum size of a
    /* publication published by amqsgam so we can allocate a
    /* block large enough for any message we will receive.
    /******
    messageLength = DEFAULT_MESSAGE_SIZE;
    pMessageBlock = (PMQBYTE)malloc(DEFAULT_MESSAGE_SIZE);

    gmo.setOptions(MQGMO_WAIT + MQGMO_CONVERT);
    gmo.setWaitInterval(MAX_WAIT_TIME);
    gmo.setMatchOptions(MQMO_MATCH_CORREL_ID);
    msg.useEmptyBuffer((char *) pMessageBlock,messageLength);
    msg.setCorrelationId(EventCorrelId);

    while ( true ) {

        subscriberQueue.get(msg,gmo);

        if( msg.formatIs(MQFMT_RF_HEADER) ) {
            /******
            /* Split the message data into the three important
            /* areas, the MQRFH header, the NameValueString that
            /* follows it and any user data following that.
            /******
            pMQRFHeader = (PMQRFH)pMessageBlock;
            pNameValueString = (PMQCHAR)(pMessageBlock
                                         + MQRFH_STRUC_LENGTH_FIXED);
            nameValueStringLength = pMQRFHeader->StrucLength
                                   - MQRFH_STRUC_LENGTH_FIXED;
            pUserData = pMessageBlock + pMQRFHeader->StrucLength;
            *(pMessageBlock + msg.dataLength()) = 0;
            printf("The publication received is: %s\n",pUserData);
        }
    }
}
return (0);
}

```

In this chapter we've seen how using C++ differs from using the basic MQI and how it can be used by applications programmers to build applications. In the next chapter we discuss the use of the MQSeries automation classes for ActiveX.



Programming with ActiveX

This chapter is an overview of the MQSeries Automation Classes for ActiveX, what they are and how they can be used to work with MQSeries queue manager objects. Please refer to *Using the Component Object Model Interface*, SC34-5387 for more detailed information.

6.1 Overview

MQSeries Automation Classes for ActiveX is another API set that allows the programmer to work with queue manager objects. The MQSeries Automation Classes for ActiveX components provide classes that are intended to be used by designers and programmers who want to develop MQSeries applications that are able to run on the Windows platform. The classes can then be easily integrated into any application, because the MQSeries objects that are needed can be coded using the native syntax of the implementation language. The overall design of the application will be the same as for any MQSeries application.

It is important to mention a couple of concepts that are related to ActiveX. ActiveX components are based on the Component Object Model, also known as COM, which is an object-based programming model defined by Microsoft. This model specifies how software components can be provided in a way that allows them to locate and communicate with each other regardless of the computer language in use or their physical location.

COM is the underlying architecture that forms the foundation for higher-level software services, such as those provided by OLE. COM makes it easy to develop powerful component-based applications. From its original application on a single machine, COM has expanded to allow access to components from other systems. This new model is known as Distributed COM or simply DCOM.

DCOM makes it possible to create networked applications built from components. It extends COM to support communication among different computers - on a local network, a wide area network, or even the Internet. With DCOM, the application can be distributed at locations that make the most sense to the customers and to the application.

COM+ is another extension of COM. It provides a runtime environment and services that are used from any programming language or tool, and enables extensive interoperability between components regardless of how there are implemented. COM+ provides a simple, powerful model for building software systems from interacting objects. All communication with an object must occur through interfaces, and all communications must look like simple method calls - even if the destination object is located in another process or on another machine.

When designing an ActiveX application that uses MQSeries Automation Classes for ActiveX, the most important item of information is the message that is sent or received from the remote MQSeries system. For an MQSeries Automation Classes script to work, both the sending side and the receiving application must know the message structure. Also, when considering how to structure the

implementation of an application that is being designed, keep in mind that the MQSeries Automation Classes for ActiveX scripts run on the same machine as the one on which either the MQSeries queue manager or the MQSeries client is installed.

MQSeries Automation Classes for ActiveX key features are:

- ▶ Gives access to all the functions and features of the MQSeries API. This allows full interconnectivity to the other non-Windows MQSeries platforms.
- ▶ Complies with the ActiveX conventions.
- ▶ Complies with the MQSeries object model.
- ▶ It gives the ActiveX application that is using them the ability to run transactions and access data on any of enterprise systems that can be accessed through MQSeries.

MQSeries Automation Classes for ActiveX implement a free-threading model where objects can be used between threads. While the classes allow the use of MQQueue and MQQueueManager objects, MQSeries does not currently permit the sharing of handles between different threads.

Even though there are some advantages to using these classes, there are also some limitations. Just to mention a few of them:

- ▶ If the application is going to be accessed using a Web browser, the browser must support ActiveX controls (for example Microsoft Internet Explorer 3 or later). If MQSeries is used to send data outside the machine on which it is executed, a script can take advantage of this and cause security problems.
- ▶ The Automation Classes constants are not available to VBScript and JavaScript programs, so the programmer will have to hard code them (these constants can be found in the cmqc.h header file provided with MQSeries products).

6.2 Platforms and languages

MQSeries Automation Classes for ActiveX can only be used on a 32-bit ActiveX scripting client. Therefore, the only platforms where the classes are available are:

- ▶ Windows 98/95
- ▶ Windows NT
- ▶ Windows 2000

An application that uses the MQSeries Automation Classes for ActiveX can be written using a language that supports the creation and use of COM objects; for example, Visual Basic, Java and other ActiveX scripting clients. The classes can be easily integrated into the application because the MQSeries objects can be coded using the native syntax of the implementation language.

To run the ActiveX components in an MQSeries server environment, you must have Windows NT 4.0 (Service Pack 6 and Option Pack 4 if MTS is going to be used as the transaction coordinator) or Windows 2000 and MQSeries version 5.1 or later.

6.3 Libraries

If an application is using MQSeries Automation Classes for ActiveX, it requires the MQAX200.dll link library. This library can be found on *MQSeries Base Directory\bin*. When writing applications, this library needs to be included in the program. In Visual Basic, this can be achieved by adding the mqax200.dll to the program references. For more information, please refer to the Visual Basic product documentation.

6.4 Architectural model

MQSeries Automation Classes for ActiveX provide the following objects:

- ▶ **MQSession:** This is the main class for MQSeries Automation Classes for ActiveX. It contains the status of the last action performed on any of the other classes. There is only one MQSession object per ActiveX process. An attempt to create a second object of this type creates a second reference to the original object.
- ▶ **MQQueueManager:** This class provides access to the queue manager. Calling the methods and properties of this object results in calls being made across the MQI. When an object of this class is destroyed, it is automatically disconnected from its queue manager. Some of the properties that can be accessed through this call are alternate user ID, completion code, connection status, and reason code. Most of these properties can be accessed only if the object is connected to the queue.
- ▶ **MQQueue:** This class provides access to the queues and its properties, such as current depth, depth high limit, etc.
- ▶ **MQMessage:** This class represents an MQSeries message. This class not only includes properties to access the message descriptor, but it also provides a buffer to hold the message data. It includes write methods to copy data from an ActiveX application to an MQMessage object, and provides read

methods to copy data from an MQMessage object to an ActiveX application. This class manages the allocation and deallocation of memory for the buffer automatically. The application does not have to declare the size of the buffer when an MQMessage object is created, because the buffer grows to accommodate data written to it.

- ▶ MQPutMessageOptions: This class encloses all the different options that control the action of putting a message.
- ▶ MQGetMessageOptions: This class encloses all the different options that control the action of getting a message.
- ▶ MQDistributionList: This class encloses a collection of queues - local, remote, or alias for output.
- ▶ MQDistributionListItem: This class encloses the MQOR, MQRR and MQPMR structures and associates them with an owning distribution list.

6.5 Programming with MQSeries automation classes for ActiveX

In the following section we see how the MQSeries Automation Classes for ActiveX can be used to connect and manipulate queue manager objects. The examples shown in this section are written in Visual Basic. Please refer to the MQSeries book *Using the Component Object Model Interface*, SC34-5387, for more detailed information.

6.5.1 Connecting to the queue manager

MQSeries Automation Classes for ActiveX follow the same structure as AMI. You first have to create a session object in order to connect to the queue manager. To create an instance of the MQSession class, we use the New keyword as follows:

```
Set SessionObject = New MQSession
```

Once the session object is created, a connection to the queue manager must be established. This is achieved by using the AccessQueueManager() method of the MQSession class. This method is actually going to connect to the queue manager, open the queue manager, and set the initial property values.

```
Set QueueManagerObject =  
SessionObject.AccessQueueManager(QmgrName as string)
```

The only parameter that this method expects is the queue manager name. If the default queue manager is going to be accessed, then use "" instead of the queue manager name. The connection can be established by either using an MQSeries client or connecting directly to an MQSeries server. If the connection object fails, an error event is raised, the object's reason code and completion code are set and the MQSession object's reason code and completion code are set.

Example 6-1 shows how to connect to the default queue manager.

Example 6-1 Connecting to default Queue Manager

```
Dim MQSess As MQSession          '* session object  
Dim QMgr As MQQueueManager       '* queue manager object  
  
Set MQSess = New MQSession  
Set QMgr = MQSess.AccessQueueManager("")
```

Another way that can be used to connect to the queue manager is using the Connect() method of the MQQueueManager class. Before calling this method, the queue manager name has to be set. This is achieved by using the Name property of the MQQueueManager class.

Example 6-2 shows how these calls are used to connect to a queue manager called SAMPLE.QMGR1.

Example 6-2 Connect to Queue Manager

```
Dim MQSess As MQSession          '* session object  
Dim QMgr As MQQueueManager       '* queue manager object  
  
Set MQSess = New MQSession  
Set QMgr = New MQQueueManager  
QMgr.Name="SAMPLE.QMGR1"  
QMgr.Connect
```

6.5.2 Opening MQSeries objects

Once we have successfully established the connection to the queue manager, we can then open the MQSeries objects. These objects can be:

- ▶ Queues - remote, local, dynamic.
- ▶ Distribution lists
- ▶ Messages

To open a queue, we use the `AccessQueue()` method of the `MQQueueManager()` class:

```
Set QueueObject = QueueManagerObject.AccessQueue(QueueName as
string, OpenOption as Long, QueueManagerName as string,
DynamicQueueName as string, AlternateUserId as string)
```

`QueueName` is the name of the queue. This queue must be created before attempting to connect to the queue manager. The `OpenOption` parameter is used to specify the options that control the action of opening the queue. The open options that are most commonly used are:

- ▶ `MQOO_INPUT_SHARED`: Used when the application needs to get messages from a queue. It opens the queue in a shared access mode, so multiple applications can retrieve messages from the queue simultaneously. This option can only be used for local, alias, and model queues.
- ▶ `MQOO_INPUT_EXCLUSIVE`: Used when the application needs to get messages from a queue. It opens a queue in exclusive mode. This option is valid only for local, alias, and model queues.
- ▶ `MQOO_OUTPUT`: This option should be used if the application is going to put messages in the queue. It is valid for all types of queues, including remote queues and distribution lists.

For more information on the `MQOPEN - Options` parameter, refer to the *MQSeries Application Programming Reference*.

The `QueueManagerName` is also used to specify the name of the queue manager that owns the queue. This parameter is normally omitted since we already created an `MQQueueManager` object referencing a queue manager. If `QueueName` is a model queue, `DynamicQueueName` will be used to assign a name to the dynamic queue that is going to be created.

Example 6-3 shows how to open a queue called `PTP.QUEUE.LOCAL`, which is defined in the default queue manager.

Example 6-3 Open queue

```
Dim MQSess As MQSession          '* session object
Dim QMgr As MQQueueManager        '* queue manager object
Dim ITSQueue As MQQueue           '* input queue object

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("")
Set ITSQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL",MQOO_OUTPUT)
' If PTP.QUEUE.LOCAL needs to be opened to get and receive messages, the
' call will look like this:
```

```
' Set ITSQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL", _  
' MQOO_OUTPUT + MQOO_INPUT_SHARED)
```

Note: Do not set the QueueManagerName, or set it to "" if the queue to be opened is local. If it is set to the name of the remote queue manager that owns the queue, an attempt is made to open a local definition of the remote queue.

When it is necessary to send a message to multiple destinations, the MQDistributionList object can be used. Similar to the MQQueue object, we need to create an object of this type. This can be achieved by using the New keyword as follows:

```
Set DistributionListObject = New MQDistributionList
```

After creating a reference to the MQDistributionList object, we need to specify the MQQueueManager object that has a reference to the queue manager where the distribution lists are defined. This is achieved by using the ConnectionReference property of the MQDistributionList object.

```
DistributionListObject.ConnectionReference = QueueManagerObject
```

Once the reference to the queue manager is specified, the last thing that needs to be done is adding queues to the distribution list. Each queue must be associated with an MQDistributionListItem object. This can be done with the AddDistributionListItem() method of the MQDistributionList class.

```
Set DistributionListItemObject =  
DistributionListObject.AddDistributionListItem(QueueName as string,  
QueueMgrName as string)
```

The QueueName is the name of the queue that needs to be part of the distribution list, and QueueMgrName is the queue manager that owns the queue.

Example 6-4 shows how to send a message to two queues, PTP.QUEUE.LOCAL and PTP.QUEUE2.LOCAL, which are defined in the queue manager SAMPLE.QMGR1:

Example 6-4 Send a message

```
Dim MQSess As MQSession           '* session object  
Dim QMgr As MQQueueManager        '* queue manager object  
Dim DistListItem1 As MQDistributionListItem  
Dim DistListItem2 As MQDistributionListItem
```

```

Dim SampleMsg As MQMessage

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("SAMPLE.QMGR1")
Set DistList = New MQDistributionList

DistList.ConnectionReference = QMgr

Set DistListItem1 = DistList.AddDistributionListItem("PTP.QUEUE.LOCAL")
Set DistListItem2 = DistList.AddDistributionListItem("PTP.QUEUE2.LOCAL")

Set SampleMsg = MQSess.AccessMessage()
SampleMsg.MessageData = "Sample Message"
DistList.OpenOptions = MQOO_OUTPUT
DistList.Open
DistList.Put SampleMsg

End Sub

```

Another object that needs to be created before sending or receiving messages is the message object. As we mentioned before, the message object class contains the message data and properties to access the message descriptor. This type of object can be created using the `AccessMessage()` of the `MQSession` class:

```
Set MessageObject = SessionObject.AccessMessage()
```

The message data is assigned by the `MessageData` property. For example, if you need to send the message "Sample Message", this string will have to be referenced to the message object. This is shown in Example 6-5.

Example 6-5 Assignment

```
MessageObject.Message = "Sample Message"
```

Binary data can be passed to an `MQSeries` message simply by setting the `CharacterSet` property to the coded character set identifier of the queue manager (`MQCCSI_Q_MGR`), and passing it a string.

6.5.3 Basic operations

In 6.5.2, "Opening `MQSeries` objects" on page 186, we saw how to create and open the different queue manager objects. Now that the objects are created, we can proceed with the basic operations that can be performed against the objects. The basic operations include getting messages and sending messages.

Sending messages

Before sending a message, the queue will have to be opened with the MQOO_OUTPUT option. In order to send a message, the Get() method of the MQQueue object will have to be used.

```
QueueObject.Put(MessageObject, PutMsgOptionsObject)
```

MessageObject is the object that represents the message that is going to be sent. Therefore, an object of this type will have to be created prior to this call. Additionally, we can specify the PutMsgOptions object, which contains options to control the put operation. If the PutMsgOptions object is not specified, the default PMOs are used. If this option is going to be used, then an MQPutMessageOptions object should be created using the AccessPutMessageOptions() method of the MQSession class.

Example 6-6 shows how to put a message on a queue called PTP.QUEUE.LOCAL with the PMO option MQPMO_NO_SYNCPOINT (put messages without syncpoint control).

Example 6-6 Put message

```
Dim MQSess As MQSession           '* session object
Dim QMgr As MQQueueManager        '* queue manager object
Dim ITSQueue As MQQueue           '* input queue object
Dim PutOptions As MQPutMessageOptions '* put message options
Dim SampleMsg As MQMessage        '* message object for put

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("")
Set SampleQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL",MQOO_OUTPUT)
Set SampleMsg = MQSess.AccessMessage()
Set PutOptions = MQSess.AccessPutMessageOptions()

SampleMsg.MessageData = "Sample Message"
PutOptions.Options = MQPMO_NO_SYNCPOINT
SampleQueue.put SampleMsg PutOptions
```

Remember that the MQPutMessageOptions object encapsulates the various options that control the action of putting a message onto an MQSeries queue.

To send multiple messages simultaneously, the put() method of the MQDistributionListObject class is used. Just like sending a message to a single queue, before we begin sending multiple messages we need to open the distribution list with the MQOO_OUTPUT option using the OpenOptions property of the MQDistributionList class. Then the open() method of the MQDistributionList class is used to open the objects defined in the distribution list.

Example 6-7 shows how to send a message to two queues, PTP.QUEUE.LOCAL and PTP.QUEUE2.LOCAL, that are defined in the queue manager SAMPLE.QMGR1.

Example 6-7 Send message

```
Dim MQSess As MQSession           '* session object
Dim QMgr As MQQueueManager        '* queue manager object
Dim DistListItem1 As MQDistributionListItem
Dim DistListItem2 As MQDistributionListItem
Dim SampleMsg As MQMessage

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("SAMPLE.QMGR1")
Set DistList = New MQDistributionList

DistList.ConnectionReference = QMgr

Set DistListItem1 = DistList.AddDistributionListItem("PTP.QUEUE.LOCAL")
Set DistListItem2 = DistList.AddDistributionListItem("PTP.QUEUE2.LOCAL")

Set SampleMsg = MQSess.AccessMessage()
SampleMsg.MessageData = "Sample Message"
DistList.OpenOptions = MQOO_OUTPUT
DistList.Open
DistList.Put SampleMsg
```

Receiving messages

To receive or get a message, we use the get() method of the MQQueue class. Remember that before a message can be retrieved, the queue must be opened with either MQOO_INPUT_EXCLUSIVE or MQOO_INPUT_SHARED.

```
QueueObject.Get(MessageObject, GetMsgOptionsObject, MsgLength)
```

MessageObject is the object that represents the message that is going to be retrieved. Therefore an object of this type will have to be created prior to this call. Additionally, we can specify the GetMsgOptionsObject, which contains options to control the get operation. However if it is not specified, the default GMOs are used. If this option is going to be used, then an MQGetMessageOptions object should be created using the AccessGetMessageOptions() method of the MQSession class.

There are two ways of receiving a message from MQSeries:

- ▶ Polling by issuing a get() followed by a wait, using the Visual Basic Timer function.
- ▶ Issuing a get() with the Wait option. Specify the wait duration by setting the WaitInterval property. This is recommended when the software running at the time is running single-threaded, while the system is multi-threaded. This prevents your system from locking up indefinitely.

When an MQSeries application is the originator of a message and MQSeries generates the AccountToken, CorrelationId, GroupId and MessageId, it is recommended that you use the AccountingTokenHex, CorrelationIdHex, GroupIdHex and MessageIdHex properties if you want to look at their value or manipulate them in any way, including passing them back in a message to MQSeries. The reason for this is that the MQSeries generated values are strings of bytes that have any value from 0 through 255 inclusive. They are not strings of printable characters.

If this method succeeds, then the MQMD and Message Data portions of the MQMessage object are completely replaced with the MQMD and Message Data from the incoming message. Otherwise, the MQMessage object doesn't change.

If the contents of the message buffer are undefined, the total message length is set to the full length of the message that would have been retrieved. If the message length parameter is not specified, the length of the message buffer is automatically adjusted to at least the size of the incoming message.

Example 6-8 shows how to get a message from a queue called PTP.QUEUE.LOCAL and then display the message in a text box called txtMessageData.

Example 6-8 Get message

```
Dim MQSess As MQSession          '* session object
Dim QMgr As MQQueueManager       '* queue manager object
Dim ITSQueue As MQQueue          '* input queue object
```



```

Dim GetOptions As MQGetMessageOptions    '* get message options
Dim SampleMsg As MQMessage               '* message object for put

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("")
Set SampleQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL",MQOO_INPUT_SHARED)
Set SampleMsg = MQSess.AccessMessage()
Set GetOptions = MQSess.AccessGetMessageOptions()

GetOptions.Options = GetOptions.Options Or MQGMO_NO_SYNCPOINT
SampleQueue.get SampleMsg GetOptions
txtMessageData.text = SampleMsg.MessageData

```

6.5.4 Closing objects

Once the messages are sent or received and the data has been processed, the objects can be closed. This is done by using the close() method of the object that needs to be closed (queue manager, queue or distribution list).

```

QueueManagerObject.Close

QueueObject.Close

DistributionListObject.Close

```

If a dynamic queue is going to be closed and we are the ones who created it, then either one of the following options should be specified in the close options (MQCO) using the CloseOptions property of the MQQueue class:

- ▶ MQCO_DELETE: Deletes the queue.
- ▶ MQCO_DELETE_PURGE: Deletes the queue, but only after purging all the messages.

6.5.5 Closing the connection

To disconnect from the queue manager, the Disconnect() method of the MQQueueManager class can be used:

```

QueueManagerObject.Disconnect

```

All queue objects associated with the MQQueueManager object are made unusable and cannot be re-opened. Any uncommitted changes (message puts and gets) are committed.

6.6 Transaction management

The API calls used to control the transaction depend on the type of transaction that is being used. There are three different scenarios:

- When the only resources are the MQSeries messages (local unit of work):

In this scenario, the transaction is started by the first message sent or received under syncpoint control, as specified using the MQPMO_SYNCPOINT or MQGMO_SYNCPOINT option of the MQPutMessageOptions or MQGetMessageOptions objects. Multiple messages can be included in the same unit of work. The transaction can be committed by using the Commit() method or backed out by using the Backout() method of the MQQueueManager object. Example 6-9 shows how to put a message under syncpoint control.

Example 6-9 Put message under syncpoint

```
Dim MQSess As MQSession           '* session object
Dim QMgr As MQQueueManager        '* queue manager object
Dim ITSQueue As MQQueue          '* input queue object
Dim PutOptions As MQPutMessageOptions '* put message options
Dim SampleMsg As MQMessage        '* message object for put

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("")
Set SampleQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL",MQOO_OUTPUT)
Set SampleMsg = MQSess.AccessMessage()
Set PutOptions = MQSess.AccessPutMessageOptions()

SampleMsg.MessageData = "Sample Message"
PutOptions.Options = MQPMO_SYNCPOINT
SampleQueue.put SampleMsg PutOptions

'Perform any actions before the message is put on the queue

QMgr.Commit
```

Example 6-10 shows how to get a message under syncpoint control.

Example 6-10 Get message under syncpoint

```
Dim MQSess As MQSession           '* session object
Dim QMgr As MQQueueManager        '* queue manager object
Dim ITSQueue As MQQueue          '* input queue object
Dim GetOptions As MQGetMessageOptions '* get message options
Dim SampleMsg As MQMessage        '* message object for put

Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("")
```

```

Set SampleQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL",MQOO_INPUT_SHARED)
Set SampleMsg = MQSess.AccessMessage()
Set GetOptions = MQSess.AccessGetMessageOptions()

GetOptions.Options = GetOptions.Options Or MQGMO_SYNCPOINT
SampleQueue.get SampleMsg GetOptions
'Perform any validations before the message is physically removed
'from the queue.
....
QMGr.Commit

txtMessageData.text = SampleMsg.MessageData
...

'If an error occurred during the validation, don't retrieve the message and
'display the error message
If MQSess.CompletionCode <> MQCC_OK Then
QMGr.Backout
ErrMsg = Err.Description
StrPos = InStr(ErrMsg, " ")          ' * search for first blank
If StrPos > 0 Then
Print Left(ErrMsg, StrPos)
Else
Print Error(Err)                    '* print complete error object
End If
Print ""
Print "MQSeries Completion Code = " & MQSess.CompletionCode
Print "MQSeries Reason Code = " & MQSess.ReasonCode
Print "(" & MQSess.ReasonName & ")"
End If

```

- When MQSeries acts as an XA transaction coordinator (global unit of work).

The transaction must be started explicitly using the begin() method of the MQQueueManager class before the first recoverable resource (such as a relational database) is changed. The unit of work can then be committed by using the commit() method or backed out by using the backout() method of the MQQueueManager object.

Example 6-11 shows how the MQSeries Automation Classes for ActiveX can be used when MQSeries is an XA transaction coordinator.

Example 6-11 Classes

```

Dim MQSess As MQSession           '* session object
Dim QMgr As MQQueueManager        '* queue manager object
Dim ITSQueue As MQQueue           '* input queue object
Dim PutOptions As MQPutMessageOptions '* put message options
Dim SampleMsg As MQMessage        '* message object for put

```

```

'Connect to the database
...
Set MQSess = New MQSession
Set QMgr = MQSess.AccessQueueManager("")
Set SampleQueue = QMgr.AccessQueue("PTP.QUEUE.LOCAL",MQOO_OUTPUT)
Set SampleMsg = MQSess.AccessMessage()
Set PutOptions = MQSess.AccessPutMessageOptions()

QMGr.Begin

SampleMsg.MessageData = "Sample Message"
PutOptions.Options = MQPMO_SYNCPOINT
SampleQueue.put SampleMsg PutOptions

'Perform any validations and update the table
...

QMGr.Commit

'If an error occurred during the validation, don't put the message on the
queue 'and display the error message
If MQSess.CompletionCode <> MQCC_OK Then
QMGr.Backout
ErrMsg = Err.Description
StrPos = InStr(ErrMsg, " ")          ' * search for first blank
If StrPos > 0 Then
Print Left(ErrMsg, StrPos)
Else
Print Error(Err)                    '* print complete error object
End If
Print ""
Print "MQSeries Completion Code = " & MQSess.CompletionCode
Print "MQSeries Reason Code = " & MQSess.ReasonCode
Print "(" & MQSess.ReasonName & ")"
End If

```

- When an external transaction coordinator, such as Microsoft Transaction Server, is used.

In this case, the transaction is controlled using the API calls of an external transaction coordinator. Briefly put, Microsoft Transaction Server is essentially a management tool for COM/ActiveX components on local and remote computers.

Microsoft Transaction Server is typically used with front-end code that is a COM client to the objects held within Microsoft Transaction Server, and back-end services such as databases. The front-end code may be a fat stand-alone program, or an Active Server Page hosted by the Internet Information Server. The front-end code may reside on the same machine as Microsoft Transaction Server and its business objects with connection via COM.

Alternatively the front-end code may reside on a different machine, with connection via DCOM. Different clients may be used to access the same Microsoft Transaction Server business object in different situations. It should not make any difference to the operation of Microsoft Transaction Server and its business object which client is used or how the client connects to Microsoft Transaction Server. The client objects may be written in any language and environment that support the coding of COM client. The most common are Visual Basic for a fat client and VBScript or JavaScript for a thin client. The business objects may be written in any language that supports the coding of COM servers. The most common are Visual Basic, C++, and Java (Microsoft's).

Microsoft Transaction Server is designed to help users run business logic applications in a typical middle-tier server. It divides work up into activities that are typically short, independent chunks of business logic. Microsoft Transaction Server introduced many features to make writing scalable and distributed applications easier. In Windows 2000, a new product called COM+ was introduced. This product is an evolution of Microsoft Transaction Server.

6.7 Grouping

MQSeries Automation Classes for ActiveX allow a sequence of related messages to be included in, and sent as, a message group. Group context information is sent with each message to allow the message sequence to be preserved and made available to a receiving application. The group identification is defined in the message descriptor structure, which can be accessed through the MQMessage class.

Every message in the group must have the MQMF_MSG_IN_GROUP option except for the last one, which needs to have the MQMF_LAST_MSG_IN_GROUP option. The order of the messages that are part of the group is stored in the MsgSeqNumber field of the MQMD structure, which is generated automatically by the queue manager.

Additionally, the queue manager can control whether or not a message group has been received completely. If only completed message groups need to be displayed, the MQGMO_ALL_MSGS_AVAILABLE option can be set in the get message options structure.

6.8 Exploring the patterns

In this section we give examples of how the MQSeries Automation Classes for ActiveX can be used to build applications using the different messaging patterns.

6.8.1 Send-and-forget

The sample program that we discuss in this section follows the logical flow below:

- ▶ Create an MQSession object.
- ▶ From the MQSession object, create an MQQueueManager object, MQQueue object, MQPutMessageOptions object, and MQMessage object.
- ▶ Open the MQQueue object.
- ▶ Populate the MQMessage with data using the MessageData property.
- ▶ Put the MQMessage on the MQQueue using the put method.
- ▶ Read and display all of the data in the MQMessage using the MessageData property. Also, display the format and the MessageId using the Format and MessageId properties.
- ▶ Close the MQQueue, MQQueueManager and MQSession objects.

The first thing that we need to do before even attempting to connect to a queue manager is to add a reference to the MQSeries library MQAX200, which is located on the MQSeries installation directory under the bin directory. Once the reference to the MQSeries Automation Classes for ActiveX library is created, the next step will be to define the objects that are needed for a send-and-forget application.

Example 6-12 shows how to define the objects needed for this type of application. It is recommended that the names of the objects be descriptive, so that it is easier to remember what they are used for.

Example 6-12 Define objects

```
Dim MQ_Manager As MQQueueManager '* queue manager object
Dim MQ_Queue As MQQueue '* queue object
Dim MQ_Session As MQSession '* session object
Dim MQ_Message As MQMessage '* message object for put
```

```
Dim PutOptions As MQPutMessageOptions '* get message options
Dim MsgSr As String '* put message data string
```

In Example 6-13, the MQ_Session object is created and the default queue manager is accessed. Also the PTP.QUEUE.LOCAL queue is opened with the MQOO_INPUT option, which means that messages can be put on the queue.

Example 6-13 Create object

```
Set MQ_Session = New MQSession
Set MQ_QManager = MQ_Session.AccessQueueManager("")
Set MQ_Queue = MQ_QManager.AccessQueue("PTP.QUEUE.LOCAL", MQOO_OUTPUT)
```

Example 6-14 shows how to access a new MQMessage object, add some data, create an MQPutMessageOptions object, and finally put the message on the PTP.QUEUE.LOCAL queue that was opened in Example 6-13.

Example 6-14 Access object

```
Set MQ_Message = MQ_Session.AccessMessage()
MQ_Message.MessageData = "Simple Send And Forget Application Using MQAX "
Set PutOptions = MQ_Session.AccessPutMessageOptions()
MQ_Queue.Put MQ_Message
```

Finally, if no more messages are going to be sent, the queue manager objects have to be closed. In Example 6-15 the PTP.QUEUE.LOCAL queue is going to be closed using the close method of the MQQueue class. After closing, we must disconnect from the local queue manager that was opened in Example 6-13.

Example 6-15 Close objects

```
MQ_Queue.Close
MQ_QManager.Disconnect
```

6.8.2 Request/reply

The sample program that we are going to discuss in this section follows the logical flow below:

- ▶ Create an MQSession.
- ▶ From the MQSession, create an MQQueueManager, two MQQueue objects (one for the sending queue and one for the reply to queue), an MQPutMessageOptions object, an MQGetMessageOptions object, and two MQMessage objects (one for the sending message and one for the reply message).
- ▶ Open the MQQueue.

- Put a message on the queue and wait for a reply on the reply-to queue.
- Get the MQMessage from the MQQueue object using the get method.
- Populate the MQMessage with data that was retrieved using the MessageData property.
- Read and display all of the data in the MQMessage using the MessageData property.
- Close the MQQueue, MQQueueManager and MQSession objects.

The first thing that we need to do before even attempting to connect to a queue manager is to add a reference to the MQSeries library MQAX200, which as we mentioned in 6.3, “Libraries” on page 184 is located in the MQSeries installation directory under the \bin folder. Once the reference to the MQSeries Automation Classes for ActiveX library is created, the next step will be to define the objects that are needed for the request/reply application.

Example 6-16 shows how to define the objects needed for this type of application. It is recommended that the names of the objects be descriptive, so that it is easier to remember what they are used for.

Example 6-16 Define objects

```
Dim MQ_QManager As MQQueueManager '* queue manager object
Dim MQ_SendQueue As MQQueue '* queue object for sending
Dim MQ_ReplyQueue As MQQueue '* queue object for replies
Dim MQ_Session As MQSession '* session object
Dim MQ_PutOptions As MQPutMessageOptions '* put message options
Dim MQ_GetOptions As MQGetMessageOptions '* get message options
Dim MQ_PutMsg As MQMessage '* message object for put
Dim MQ_GetMsg As MQMessage '* message object for get
```

In Example 6-17, the MQ_Session object is created and the default queue manager is accessed. Also the PTP.QUEUE.LOCAL queue is opened with the MQOO_INPUT option and the PTP.REPLY.QUEUE.LOCAL is opened with the MQOO_INPUT_SHARED.

Example 6-17 Create session object

```
Set MQ_Session = New MQSession
Set MQ_QManager = MQ_Session.AccessQueueManager("")
Set MQ_SendQueue = MQ_QManager.AccessQueue("PTP.QUEUE.LOCAL", MQOO_OUTPUT)
Set MQ_ReplyQueue = MQ_QManager.AccessQueue("PTP..REPLY.QUEUE.LOCAL",
MQOO_INPUT_SHARED)
```

Example 6-18 shows how to access a new MQMessage object, add some data, then create an MQPutMessageOptions object and finally put the message on the PTP.QUEUE.LOCAL queue that was opened in Example 6-13.

Example 6-18 Access object

```
Set MQ_Message = MQ_Session.AccessMessage()  
MQ_Message.MessageData = "Simple Request and Reply Application Using MQAX "  
Set MQ_PutOptions = MQ_Session.AccessPutMessageOptions()  
MQ_Queue.Put MQ_Message
```

Once the message is sent, a reply is expected on the PTP.REPLY.QUEUE.LOCAL queue. We use the message ID to retrieve the reply from the queue. This is shown in Example 6-19.

Example 6-19 Retrieve reply

```
MQ_Message.MessageIdHex = MQMessageId  
Set MQ_GetOptions = MQ_Session.AccessGetMessageOptions()  
MQ_GetOptions.Options = GetOptions.Options Or MQGMO_NO_SYNCPOINT  
MQ_Queue.Get MQ_Message  
Message = MQ_Message.MessageData
```

Finally if no more messages are being sent, the queue manager objects have to be closed. In Example 6-20 the PTP.QUEUE.LOCAL queue is going to be closed using the close() method of the MQQueue class. After closing, disconnect from the local queue manager that was opened in Example 6-17.

Example 6-20 Close objects

```
MQ_Queue.Close  
MQ_QManager.Disconnect
```



Programming with Java

This chapter covers programming using the MQSeries classes for Java. It provides the information required by programmers who want to write Java applications or applets that interface with MQSeries queues.

7.1 Overview

The MQSeries classes for Java allow a program written in the Java programming language to access MQSeries objects. The MQSeries Java API provides methods to put messages onto, and get messages from, MQSeries queues.

7.2 Platforms

The MQSeries classes for Java product is available for:

- ▶ AIX
- ▶ iSeries and OS/400
- ▶ HP-UX
- ▶ Linux
- ▶ Sun Solaris
- ▶ z/OS and OS/390 V2R9 or higher
- ▶ Windows platforms

It contains:

- ▶ MQSeries classes for Java (MQSeries base Java) Version 5.2.0
- ▶ MQSeries classes for Java Message Service (MQSeries JMS) Version 5.2

7.2.1 Obtaining the package

The MQSeries classes for Java are supplied as compressed files and are available from the MQSeries Web site at:

<http://www-4.ibm.com/software/ts/mqseries/txppacs/txpsumm.html>

The files are supplied as part of SupportPac MA88.

MQSeries base Java is composed of the following Java .jar files:

- com.ibm.mq.jar** This code includes support for all the connection options.
- com.ibm.mq.iiop.jar** This code supports only the VisiBroker connection.
- com.ibm.mqbind.jar** This code supports only the bindings connection and is not supplied or supported on all platforms.

For z/OS and OS/390, a separate SupportPac, MA1G, is also available . The difference between MA1G and MA88 is that MA88 does not support CICS and MA1G does.

7.2.2 Running the MQSeries classes for Java

In order to run the MQSeries classes for Java, the following software is required:

- ▶ MQSeries for the server platform you wish to use.
- ▶ Java Development Kit (JDK) for the server platform.
- ▶ Java Development Kit, or Java Runtime Environment (JRE), or Java-enabled Web browser for client platforms.
- ▶ VisiBroker for Java (only if running on Windows with a VisiBroker connection).
- ▶ For z/OS and OS/390, OS/390 Version 2 Release 9 or higher, or z/OS, with UNIX System Services.
- ▶ For OS/400, the AS/400 Developer Kit for Java (5769-JV1) and the Qshell Interpreter, OS/400 (5769-SS1) Option 30.

Installation directories

A list of the installation directories is shown in Table 7-1.

Table 7-1 Installation directories

Platform	Directory
AIX	usr/mqm/java
z/OS and OS/390	install_dir/mqm/java
iSeries and OS/400	/QIBM/ProdData/mqm/java/
HP-UX and Sun Solaris	opt/mqm/java/
Linux	install_dir/mqm/java/
Windows 95, 98, 2000 and NT	install_dir\

Note: install_dir is the directory in which the product is installed. On Linux, this is likely to be /opt, and on z/OS and OS/390 it is likely to be /usr/lpp.

Environment variables

A summary of the environment variables needed for each platform is provided in Table 7-2.

Table 7-2 Environment variables

Platform	Sample Classpath
AIX	CLASSPATH = /usr/mqm/java/lib/com.ibm.mq.jar: /usr/mqm/java/lib/connector.jar: /usr/mqm/java/lib: /usr/mqm/java/samples/base:
HP-UX and Sun Solaris	CLASSPATH= /opt/mqm/java/lib/com.ibm.mq.jar: /opt/mqm/java/lib/connector.jar: /opt/mqm/java/lib: /opt/mqm/java/samples/base:
Windows 95, 98, 2000 and NT	CLASSPATH= install_dir\lib\com.ibm.mq.jar; install_dir\lib\com.ibm.mq.iiop.jar; install_dir\lib\connector.jar; install_dir\lib; install_dir\samples\base\;
z/OS and OS/390	CLASSPATH= install_dir/mqm/java/lib/com.ibm.mq.jar: install_dir/mqm/java/lib/connector.jar: install_dir/mqm/java/lib: install_dir/mqm/java/samples/base:
iSeries and OS/400	CLASSPATH= /QIBM/ProdData/mqm/java/lib/com.ibm.mq.jar: /QIBM/ProdData/mqm/java/lib/connector.jar: /QIBM/ProdData/mqm/java/lib: /QIBM/ProdData/mqm/java/samples/base:
Linux	CLASSPATH= install_dir/mqm/java/lib/com.ibm.mq.jar: install_dir/mqm/java/lib/connector.jar: install_dir/mqm/java/lib: install_dir/mqm/java/samples/base:

For actual installation instructions for a particular platform, see *MQSeries Using Java*, SC34-5456. This book is particularly important if the software is not yet installed and operational, because it contains very specific environmental variable information.

7.3 Using the MQSeries classes for Java

An application program can interact with MQSeries objects such as MQSeries queues upon establishing a connection to MQSeries queue manager. The queue manager provides the messaging services for the MQSeries objects that it owns.

There are two connection modes that can be used to establish a connection with the queue manager. The way you would program depends on the connection mode you choose. We first look at the two connection modes, binding mode and client connection mode.

7.3.1 Connection modes

The connection modes that can be used in MQSeries JMS are shown in Figure 7-1.

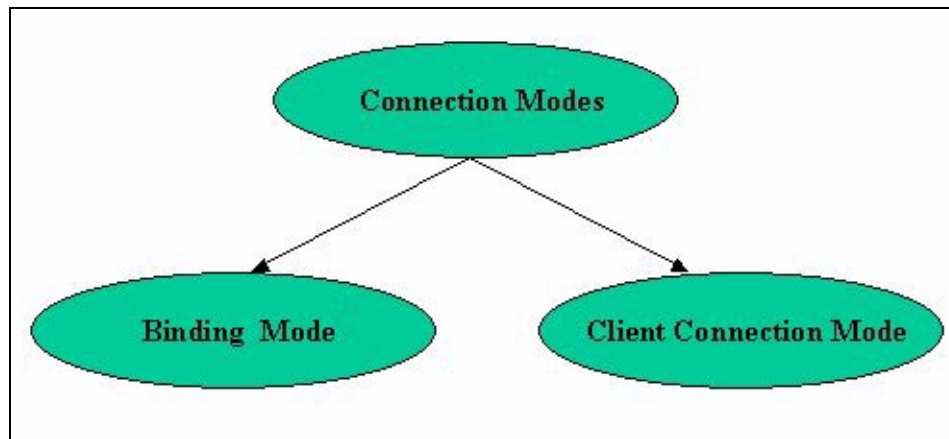


Figure 7-1 Connection modes

Binding mode

In binding mode, also known as server connection, the communication to the queue manager utilizes inter-process communications. One of the key factors that should be kept in mind is that binding mode is available only to programs running on the MQSeries server that hosts the queue manager. A program using binding mode will not run from an MQSeries client machine. In other words, the application is tied to the same machine the queue manager is on.

Binding mode is a fast and efficient way to interact with MQSeries. Certain facilities, such as XA transaction co-ordination by queue manager, are available only in binding mode.

Client connection

Client connection uses a TCP/IP connection to the MQSeries Server and enables communications with the queue manager. Programs using client connections can run on an MQSeries client machine as well as on an MQSeries server machine. Client connections use client channels on the queue manager to communicate with the queue manager. The client connection does not support XA transaction co-ordination by the queue manager. From an API perspective, the MQBEGIN call (begin() method of the MQQueueManager class) is not supported in client connection mode.

When using client connection, you would have to specify a few additional environment properties to establish connection with the queue manager. These are namely the *host name*, which is the name of the MQSeries server machine that hosts the queue manager, and the *channel name*, which is the name of the channel for client connection. Additionally, you can specify the port number on which the MQSeries server listens. If the port number is not specified, the default port number of 1414 would be used.

MQSeries classes for Java

In general, however, the MQSeries classes for Java consist of the following classes and interfaces:

MQChannelDefinition	This class is used to pass information concerning the connection to the queue manager to the send, receive and security exits. This class does not apply when connecting directly to MQSeries in binding mode.
MQChannelExit	This class defines context information passed to the send, receive, and security exits when they are invoked. The exitResponse attribute of this class should be set by the exit to indicate what action the MQSeries Client for Java should take next.
MQDistributionList	This class represents a set of open queues to which messages can be sent using a single call to the put() method. This class is instantiated by using the MQDistributionList constructor or by using the accessDistributionList() method for the MQQueueManager class.
MQDistributionListItem	This class represents a single item (single queue) within a distribution list. This class inherits the MQMessageTracker class.
MQEnvironment	This class contains static member variables that control the environment in which an

	MQQueueManager object (and its corresponding connection to MQSeries) is constructed. Since the values set in this class take effect when the MQQueueManager constructor is called, the values in the MQEnvironment class should be set before an MQQueueManager instance is constructed.
MQException	This class contains the definitions of the MQSeries completion code and error code constants. Constants beginning with MQCC_ are MQSeries completion codes and constants beginning with MQRC_ are MQSeries reason codes. An MQException is thrown whenever an MQSeries error occurs.
MQGetMessageOptions	This class contains the options that control the behavior of the MQQueue.get() method.
MQManagedObject	This class is a superclass for the MQQueueManager, MQQueue, and MQProcess classes. It provides the ability to inquire and set the attributes of these resources.
MQMessage	This class represents the message descriptor and the data for an MQSeries message.
MQMessageTracker	This class is used to tailor message parameters for a given destination in a distribution list. It is inherited by MQDistributionListItem.
MQPoolServices	This class can be used by implementations of ConnectionManager that are intended for use as the default ConnectionManager for MQSeries connections.
MQPoolServicesEvent	This class is used to generate an event whenever an MQPoolToken is added to, or removed from, the set of tokens that MQEnvironment controls. An MQPoolServicesEvent is also generated when the default ConnectionManager is changed.
MQPoolToken	This class can be used to enable the default connection pool.
MQProcess	This class provides inquiry operations for MQSeries processes.
MQPutMessageOptions	This class contains the options that control the behavior of the MQQueue.put() method.

MQueue	This class provides inquiry, set, put, and get operations for MQSeries queues. The inquire and set capabilities are inherited from MQManagedObject.
MQueueManager	This class represents the queue manager for MQSeries.
MQSimpleConnectionManager	This class provides basic connection pooling functionality.

Interfaces

The MQSeries classes for Java have the following interfaces:

MQReceiveExit	This interface allows for examination and possible alteration of the data received from the queue manager by the MQSeries classes for Java. This interface does not apply when connecting to MQSeries in binding mode.
MQSecurityExit	This interface allows customizing of security flows that occur when an attempt is made to connect to a queue manager. This interface does not apply when connecting directly to MQSeries in binding mode.
MQSendExit	This interface allows for the examination and possible alteration of the data sent to the queue manager by the MQSeries Client for Java. This interface does not apply when connecting directly to MQSeries in binding mode.

7.4 Working with MQSeries Java API

We explore the methodology of programming with MQSeries Java API in this section.

7.4.1 Setting up the connections

In this section we examine the implementation of the binding and client connection modes. Assuming that from a design perspective you have chosen to implement either binding or client connection mode, we explain how we can implement them.

Connection to the queue manager is obtained by the constructor calls of the MQueueManager class. At this time, the type of connection obtained is determined by some static fields in the MQEnvironment class. The static field settings that differentiate the connection modes are host, channel, userId, and password. Of these MQEnvironment fields that are used for connecting to the

queue manager, it is the host and channel field settings that mainly differ between binding mode and client connection mode. In binding mode you do not have to set a value for any of these fields except for the `userId` and `password` fields. You can choose to set them in binding mode also.

► **MQEnvironment.hostName**

For client connections, set this to the name of the host that hosts the queue manager. Since this host name is used for a TCP/IP connection to the machine on which the queue manager is running, the value is not case sensitive. For example:

```
MQEnvironment.host = "machinename.dmain.com" ;
```

► **MQEnvironment.channel**

This is the name of the channel for client connections. The value of this field is case sensitive. Typically it is the name of the Server Connection Channel under the queue manager. It is a bidirectional link that enables MQI calls and responses between the client and the queue manager.

For client connections, set it to be the name of the server connection channel under the queue manager to which the application is attempting to connect. For example:

```
MQEnvironment.channel = "JAVA.CLIENT.CHNL";
```

► **MQEnvironment.port**

The port number is an optional field. By default the client would attempt to connect to the queue manager on the port number 1414 of the host. Port number 1414 is the port number used by MQSeries listeners by default. If the port number is different from the default, you can specify the port number using the `MQEnvironment.port` field. For example:

```
MQEnvironment.port = nnnn;
```

► **MQEnvironment.userId, MQEnvironment.password**

The `userId` and `password` fields are blank by default. You can specify a user ID and password by setting the values of the `userId` and `password` fields. For example:

```
MQEnvironment.userId = "userXYZ" ;  
MQEnvironment.password = "password" ;
```

► **MQEnvironment.properties**

This is a hash table of the key value pairs that define the MQSeries environment. Unless you are using VisiBroker connections, set the value of this field for both binding as well as client connection to:

```
MQC.TRANSPORT_PROPERTY, MQC.TRANSPORT_MQSERIES
```

The variables in the MQEnvironment class control the behavior of the connection call to the queue manager. The first step in setting up a connection to the queue manager is to set the MQEnvironment fields depending on the type of connection mode, as explained in the 7.4.1, “Setting up the connections” on page 210.

Connection to the queue manager is obtained through the constructor call of the MQQueueManager class by creating a new instance of the MQQueueManager class. The MQQueueManager class has overloaded constructors. Appropriate constructors are invoked to establish the connection depending on the parameters supplied while creating the new instance of the MQQueueManager class. In the simplest case you can create the new instance of the QueueManager class by supplying the queue manager name as a string. The queue manager name is case sensitive. For example:

```
MQQueueManager qmgr = new MQQueueManager("ITSOG.QMGR1") ;
```

Here ITSOG.QMGR1 is the name of the Queue Manager.

The above approach is valid in both binding and client connection modes.

In the second approach you can create a new instance of the MQQueueManager class by supplying the queue manager name string, and a hash table with key value pairs for setting the Environment options. In this approach, the properties supplied override the values set in the MQEnvironment class. This approach can be used if you want to set the environment values on queue manager to queue manager basis. For example:

```
MQQueueManager qmgr = new MQQueueManager(queueManagerName ,  
propertiesHashTable)
```

The third approach involves creating a new instance of the MQQueueManager class by supplying queue manager name string and queue manager open options, which is an integer field. This approach can be used only with binding mode. The options field lets you choose between fast or normal bindings. For example:

```
MQQueueManager qmgr = new MQQueueMager(queueManagerName ,  
MQC.MQCNO_FASTPATH_BINDING ) ;
```

7.4.2 Interacting with queues

In order to perform any operations on the queue, first we should get a queue handle or queue object by opening the queue. There are two ways to open the queue. We can use the accessQueue method of the MQQueueManager object or through the constructor call of the MQQueue class.

The two different call are of the form:

```
MQQueue queue = qmgr.accessQueue("qName", openOption, "qMgrName" ,  
"dynamicQName", "alternateUserId");
```

The second approach of using the constructor of the MQQueue class is much the same, with an added queue manager parameter.

```
MQQueue queue = new MQQueue(qmgr, "qName", openOption, "qMgrName" ,  
"dynamicQName", "alternateUserId");
```

MQSeries will validate the openOption against the user authorization during the process of opening the queue.

The object of the MQQueue class represents a queue. It has methods to facilitate messaging (namely put, get, set, inquire) and properties that correspond to the attributes of a queue.

7.4.3 Working with MQSeries messages

An object of the MQMessage class represents the message to be put on or got from a queue. It encapsulate both the application data and the MQMD. It has properties corresponding to the MQMD fields and methods to write or read different application data of different data types to and from the message. Within the application, the MQMessage represents a buffer. An application does not have to declare the buffer size as it resizes itself to accommodate the data being written to it. However, if the message size is more than the `MaximumMessageLength` property of the queue, you wouldn't be able to put the message on to the queue.

To create a message, you create a new instance of the class `MQMessage`. Application data is written to the message using the `writeXXX` methods for the specific application data type. The format of data types such as numbers and strings can be controlled by the MQMD properties `characterSet` and `encoding`. The MQMD fields can be set before the message is put on the queue and can be read the MQMD fields upon getting the message from the queue. When a message is instantiated, the MQMD fields are set to their default values.

Applications control the way messages are put on the queue or got from the queue by setting appropriate options with put or get operation. The way the message is put on the queue is controlled by setting appropriate put message option values. Similarly, the way messages are retrieved from the queue is controlled by setting appropriate get message options.

Put message options

The way messages are put on the queue is determined by the value of the options field of the instance of the `MQPutMessageOptions` class. The value of the options can be set using the `MQPMO` structure of `MQSeries Constants MQC`. For example:

```
MQPutMessageOptions pmo = new MQPutMessageOption();
```

The instance of the `MQPutMessageOptions` class has the value for the options property set to the default value. This may be sufficient in most of the simple messaging scenarios. You can set any specific options using the `MQPMO` structure from the `MQSeries constants` interface `MQC`. For example:

```
pmo.options = pmo.options + MQC.MQPMO_NEW_MSG_ID
```

The example sets the value of the options field to instruct the queue manager to generate a new message ID for the message and set it the `MsgId` field of the `MQMD`.

Get message options

The way messages are retrieved from the queue is determined by the value of the options field of the instance of the `MQGetMessageOptions` class. The value of the options can be set using the `MQOO` structure of `MQSeries Constants MQC`. For example:

```
MQGetMessageOptions gmo = new MQGetMessageOption();
```

The new instance of the `MQGetMessageOptions` class has the value of the options property set to the default. You can set the appropriate get message options using the `MQPOO` structure. For example:

```
gmo.options = gmo.options + MQC.MQGMO_NO_WAIT;
```

The above option specifies that the get message call is to return immediately if there are no messages on the queue.

Sending messages

Messages are sent using the `put(MQMessage message)` or `put(MQMessage message, MQPutMessageOptions pmo)` methods of the `MQQueue` class. The `put` method call places the message on the `MQSeries` queue.

The `put` message options control the way the messages are placed on the queue.

Receiving messages

Messages are retrieved from the MQSeries queue using the `get(MQMessage message)` or `get(MQMessage, MQGetMessageOptions gmo)`, `get(MQMessage, MQGeMessageOptions gmo, int maxMessageSize)` methods of the `MQQueue` class.

All calls to `MQSeries` from a given `MQQueueManager` are synchronized.

Important: If you perform a `get` with `wait`, all other threads using the same `MQQueueManager` are blocked from making further `MQSeries` calls until the `get` completes. If you need multiple threads to access `MQSeries` simultaneously, each thread must create its own `MQQueueManager` object.

When the `MaxMessageSize` is not specified, the length of the message buffer is automatically adjusted to the message size of the incoming message. If you use the `MaxMessageSize` with the `get` method call, the largest message this call will be able to receive. If the message on the queue is larger than this size, one of two things can occur:

1. If the `MQC.MQGMO_ACCEPT_TRUNCATED_MSG` flag is set in the options member variable of the `MQGetMessageOptions` object, the message is filled with as much of the message data as will fit in the specified buffer size, and an exception is thrown with completion code `MQException.MQCC_WARNING` and reason code `MQException.MQRC_TRUNCATED_MSG_ACCEPTED`.
2. If the `MQC.MQGMO_ACCEPT_TRUNCATED_MSG` flag is not set, the message is left on the queue and an `MQException` is raised with completion code `MQException.MQCC_WARNING` and reason code `MQException.MQRC_TRUNCATED_MSG_FAILED`.

7.5 Application development

In this section, we explore the implementation of the send-and-forget, request/reply and message grouping point-to-point messaging patterns.

7.5.1 Point-to-point pattern

In this section we illustrate the point-to-point pattern with simple programs that implement this pattern.

In the point-to-point pattern, applications act in pairs. A sending application that we call as a sender puts messages on to an MQSeries application queue on the sending side. On the destination system or receiving side, an application that we call as a receiver retrieves messages from the MQSeries application queue. The sender and receiver applications thus act in pairs to achieve the data movement or messaging between the source and destination systems.

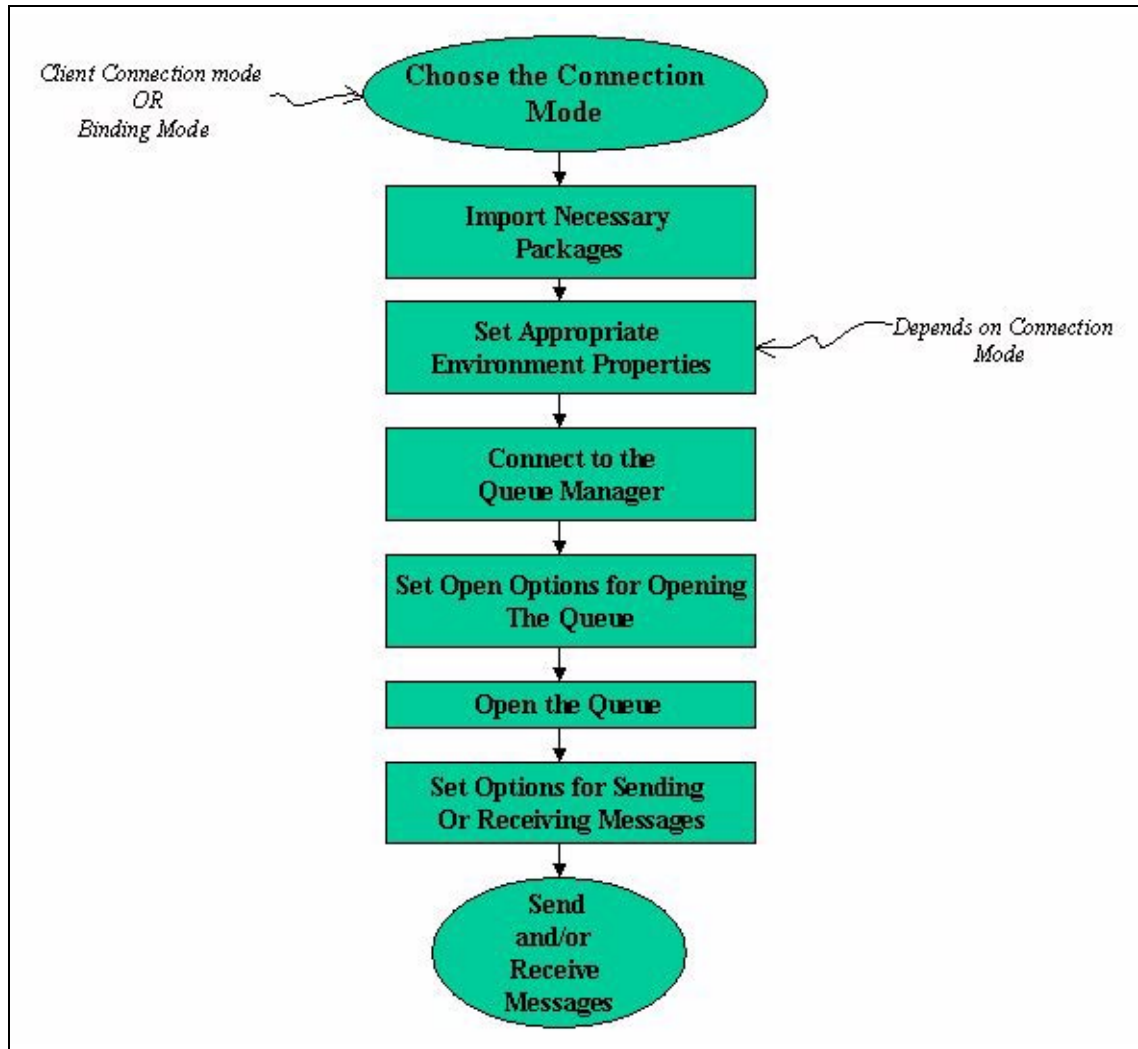


Figure 7-2 Point-to-point programming approach

In the examples, client connection to the queue manager is used. The MQSeries object used in the examples is a queue manager named ITSOG.QMGR1 on host ITSOG. The channel used for client connection is JAVA.CLIENT.CHNL and the port is the default port of 1414. The application queue used is SAMPLE.QUEUE.

Simple message sender application

Our first point-to-point client program creates a simple message and sends it to an MQSeries queue. We also illustrate the receiver program that processes the message sent by the sender.

The steps involved are:

- ▶ Import the MQSeries Java API package.
- ▶ Set up the environment properties for client connection.
- ▶ Connect to the queue manager.
- ▶ Set the options for opening the MQSeries queue.
- ▶ Open the application queue for sending messages.
- ▶ Set the options to put messages on to the application queue.
- ▶ Create a message buffer.
- ▶ Prepare the message with user data and any message descriptor fields.
- ▶ Put the message on the queue.

The following program PtpSender.java is the sender application that would send messages on the application queue:

Example 7-1 PtpSender.java

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

public class Typesetter {

    public static void main(String args[]) {

        try
        {

            String hostName = "ITSOG" ;
            String channel = "JAVA.CLIENT.CHNL" ;
            String qManager = "ITSOG.QMGR1" ;
            String qName = "SAMPLE.QUEUE" ;

            //Set up the MQEnvironment properties for Client Connections
            MQEnvironment.hostname = hostName ;
            MQEnvironment.channel = channel ;
            MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                         MQC.TRANSPORT_MQSERIES);

            //Connection To the Queue Manager
```

```

MQQueueManager qMgr = new MQQueueManager(qManager) ;

/* Set up the open options to open the queue for out put and
   additionally we have set the option to fail if the queue manager is
   quiescing.
*/
int openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;

//Open the queue
MQQueue queue = qMgr.accessQueue(qName,
                                openOptions,
                                null,
                                null,
                                null);

// Set the put message options , we will use the default setting.
MQPutMessageOptions pmo = new MQPutMessageOptions();

/* Next we Build a message The MQMessage class encapsulates the data buffer
   that contains the actual message data, together with all the MQMD
parameters
   that describe the message.
   To Build a new message, create a new instance of MQMessage class and use
   writxxx (we will be using writeString method). The put() method of
MQQueue also
   takes an instance of the MQPutMessageOptions class as a parameter.
*/

MQMessage outMsg = new MQMessage(); //Create The message buffer
outMsg.format = MQC.MQFMT_STRING ; // Set the MQMD format field.

//Prepare message with user data
String msgString = "Test Message from PtpSender program ";
outMsg.writeString(msgString);

// Now we put The message on the Queue
queue.put(outMsg, pmo);

//Commit the transaction.
qMgr.commit();

System.out.println(" The message has been Sussesfully put\n\n#####");
// Close the the Queue and Queue manager objects.
queue.close();
qMgr.disconnect();

}
catch (MQException ex)

```

```

{
    System.out.println("An MQ Error Occurred: Completion Code is :\t" +
ex.completionCode + "\n\n The Reason Code is :\t" + ex.reasonCode );
    ex.printStackTrace();
}

catch(Exception e) {
    e.printStackTrace();
}
}
}

```

Simple message receiver application

Our next point-to-point client program is the message receiver application, which gets the message that was sent by the PtpSender application and prints out the message on the console.

The steps involved are:

- ▶ Import the MQSeries Java API package.
- ▶ Set up the environment properties for client connection.
- ▶ Connect to the queue manager.
- ▶ Set the options for opening the MQSeries queue.
- ▶ Open the application queue for getting messages.
- ▶ Set the options to get messages from the application queue.
- ▶ Create a message buffer.
- ▶ Get the message from the queue on to the message buffer.
- ▶ Read the user data from the message buffer and display on the console.

Example 7-2 PtpReceiver.java

```

import com.ibm.mq.* ;
public class PtpReceiver {

    public static void main(String args[]) {

        try
        {

            String hostName = "ITSOG" ;
            String channel = "JAVA.CLIENT.CHNL" ;
            String qManager = "ITSOG.QMGR1" ;
            String qName = "SAMPLE.QUEUE" ;

            //Set up the MQEnvironment properties for Client Connections
            MQEnvironment.hostname = hostName ;
            MQEnvironment.channel = channel ;

```

```

MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                             MQC.TRANSPORT_MQSERIES);

//Connection To the Queue Manager
MQQueueManager qMgr = new MQQueueManager(qManager) ;

/* Set up the open options to open the queue for out put and
   additionally we have set the option to fail if the queue manager is
   quiescing.
*/
int openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;

//Open the queue
MQQueue queue = qMgr.accessQueue(qName,
                                openOptions,
                                null,
                                null,
                                null);

// Set the put message options.
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ; //Get messages unde sync
point control
gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait if no messages on the
Queue
gmo.options = gmo.options + MQC.MQGMO_FAIL_IF_QUIESCING ; // Fail if Queue
Manager Quiescing
gmo.waitInterval = 3000 ; // Sets the time limit for the wait.

/* Next we Build a message The MQMessage class encapsulates the data buffer
that contains the actual message data, together with all the MQMD parameters
that describe the message.
*/

MQMessage inMsg = new MQMessage(); //Create the message buffer

// Get the message from the queue on to the message buffer.
queue.get(inMsg, gmo) ;

// Read the User data from the message.
String msgString = inMsg.readString(inMsg.getMessageLength());

System.out.println(" The Message from the Queue is : " + msgString);

//Commit the transaction.
qMgr.commit();
// Close the the Queue and Queue manager objects.
queue.close();

```

```

        qMgr.disconnect();

    }
    catch (MQException ex)
    {
        System.out.println("An MQ Error Occurred: Completion Code is :\t" +
            ex.completionCode + "\n\n The Reason Code is :\t" + ex.reasonCode );
        ex.printStackTrace();
    }

    catch(Exception e) {
        e.printStackTrace();
    }
}
}

```

Request/reply

In a request/reply messaging pattern, one application sends a message (request message) to another application (reply producer) responding to the request message. The reply-producing application gets the request message, processes the request, and sends a response back to the requesting application. The reply is sent to the queue specified by the request message header property of `replyToQueue` under the queue manager specified by the `replyToQueueManager` message header property of the request message. The requesting application sets these message header properties on the request message before putting the message on the queue.

The requesting application lets the queue manager generate a unique `messageId`, and the replying application copies the `messageId` of the request message on to the `correlationId` of the reply message. The requesting application uses the `correlationId` value of the reply message to map a response back to the original request.

We illustrate the request reply pattern with a pair of simple applications. The first application, which we call the requester, puts a simple message on to the queue (request queue). The requester sets the `replyToQueue` and `replyToQueueManager` message header properties on the request message before putting the request message on the request queue. Then it opens the reply queue and waits for messages with the `correlationId` matching the `messageId` value of the outgoing request message. The responding application servicing the request message gets the request message, prepares the reply message, and sends it to the reply queue under the queue manager specified on the request message. It would also copy the `messageId` from the request message on to the `correlationId` message header field of the response message.

The application, `Requester.java`, is the application that would send the request message and expect a reply from the responding application.

The steps involved are:

- ▶ Import the necessary package.
- ▶ Set the `MQEnvironment` properties for client connection.
- ▶ Connect to the queue manager.
- ▶ Open the request queue for output.
- ▶ Set the put message options.
 - Prepare the request message.
 - Set the reply to queue name.
- ▶ Set the reply to queue manager name.
- ▶ Put the request message on the request queue.
- ▶ Close the request queue.
- ▶ Open the reply queue for input.
- ▶ Set the get message options.
 - Set option to match correlation ID on the response message.
 - Issue the get on the reply queue with wait (for response message with matching correlationId.)

Important: It is recommended that you use a definite wait time on the get call for response messages. The wait interval can be derived from the maximum time the system is allowed to wait for a response.

Example 7-3 Requester.java

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

public class Requester {

    public static void main(String args[]) {

        try
        {

            String hostName = "ITSOG" ;
            String channel = "JAVA.CLIENT.CHNL" ;
            String qManager = "ITSOG.QMGR1" ;
            String requestQueue = "SAMPLE.REQUEST" ;
            String replyToQueue = "SAMPLE.REPLY" ;
```

```

String replyToQueueManager = "ITSOG.QMGR1" ;

//Set up the MQEnvironment properties for Client Connections
MQEnvironment.hostname = hostName ;
MQEnvironment.channel = channel ;
MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                             MQC.TRANSPORT_MQSERIES);

//Connection To the Queue Manager
MQQueueManager qMgr = new MQQueueManager(qManager) ;

/* Set up the open options to open the queue for out put and
   additionally we have set the option to fail if the queue manager is
   quiescing.
*/
int openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;

//Open the queue
MQQueue queue = qMgr.accessQueue(requestQueue,
                                openOptions,
                                null,
                                null,
                                null);

// Set the put message options , we will use the default setting.
MQPutMessageOptions pmo = new MQPutMessageOptions();
pmo.options = pmo.options + MQC.MQPMO_NEW_MSG_ID ;
pmo.options = pmo.options + MQC.MQPMO_SYNCPOINT ;

MQMessage outMsg = new MQMessage(); //Create the message buffer
outMsg.format = MQC.MQFMT_STRING ; // Set the MQMD format field.
outMsg.messageFlags = MQC.MQMT_REQUEST ;
outMsg.replyToQueueName = replyToQueue;
outMsg.replyToQueueManagerName = replyToQueueManager ;

//Prepare message with user data
String msgString = "Test Request Message from Requester program ";
outMsg.writeString(msgString);

// Now we put The message on the Queue
queue.put(outMsg, pmo);

//Commit the transaction.
qMgr.commit();

System.out.println(" The message has been Sussesfully put\n\n#####");
// Close the the Request Queue

```

```

queue.close();

// Set openOption for response queue
openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;
MQQueue respQueue = qMgr.accessQueue(replyToQueue,
                                     openOptions,
                                     null,
                                     null,
                                     null);

MQMessage respMessage = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ; //Get messages under sync
point control
gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait for Response Message
gmo.matchOptions = MQC.MQMO_MATCH_CORREL_ID;
gmo.waitInterval = 10000 ;
respMessage.correlationId = outMsg.messageId ;

// Get the response message.
respQueue.get(respMessage, gmo);
String response = respMessage.readString(respMessage.getMessageLength());
System.out.println("The response message is : " + response);
qMgr.commit();
respQueue.close();
qMgr.disconnect();

}

catch (MQException ex)
{
    System.out.println("An MQ Error Occurred: Completion Code is :\t" +
ex.completionCode + "\n\n The Reason Code is :\t" + ex.reasonCode );
    ex.printStackTrace();
}

catch(Exception e) {
    e.printStackTrace();
}
}
}

```

Responder application

The responder application, Responder.java, processes the request message from the request queue and sends a reply back to the requesting application to the request queue specified.

Example 7-4 Responder.java

```
import com.ibm.mq.* ;
public class Responder {

    public static void main(String args[]) {

        try
        {

            String hostName = "ITSOG" ;
            String channel = "JAVA.CLIENT.CHNL" ;
            String qManager = "ITSOG.QMGR1" ;
            String qName = "SAMPLE.REQUEST" ;

            //Set up the MQEnvironment properties for Client Connections
            MQEnvironment.hostname = hostName ;
            MQEnvironment.channel = channel ;
            MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                         MQC.TRANSPORT_MQSERIES);

            //Connection To the Queue Manager
            MQQueueManager qMgr = new MQQueueManager(qManager) ;

            /* Set up the open options to open the queue for out put and
               additionally we have set the option to fail if the queue manager is
               quiescing.
            */
            int openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;

            //Open the queue
            MQQueue queue = qMgr.accessQueue(qName,
                                             openOptions,
                                             null,
                                             null,
                                             null);

            // Set the put message options.
            MQGetMessageOptions gmo = new MQGetMessageOptions();
            gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ; //Get messages unde sync
            point control
            gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait if no messages on the
            Queue
```

```

        gmo.options = gmo.options + MQC.MQGMO_FAIL_IF QUIESCING ; // Fail if Queue
Manager Quiescing
        gmo.waitInterval = 3000 ; // Sets the time limit for the wait.

        /* Next we Build a message The MQMessage class encapsulates the data buffer
        that contains the actual message data, together with all the MQMD
parameters
        that describe the message.
        To Build a new message, create a new instance of MQMessage class and use
writxxx (we will be using writeString method). The put() method of
MQQueue also
        takes an instance of the MQPutMessageOptions class as a parameter.
        */

        MQMessage inMsg = new MQMessage(); //Create the message buffer

        // Get the message from the queue on to the message buffer.
        queue.get(inMsg, gmo) ;

        // Read the User data from the message.
        String msgString = inMsg.readString(inMsg.getMessageLength());

        System.out.println(" The Message from the Queue is : " + msgString);

        //Check if message if of type request message and reply to the request.
        if (inMsg.messageFlags == MQC.MQMT_REQUEST ) {
            System.out.println("Preparing To Reply To the Request " );
            String replyQueueName = inMsg.replyToQueueName ;
            openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF QUIESCING ;
            MQQueue respQueue = qMgr.accessQueue(replyQueueName,
                                                openOptions,
                                                inMsg.replyToQueueManagerName,
                                                null,
                                                null);

            MQMessage respMessage = new MQMessage() ;
            respMessage.correlationId = inMsg.MessageId;
            MQPutMessageOptions pmo = new MQPutMessageOptions();
            respMessage.format = MQC.MQFMT_STRING ;
            respMessage.messageFlags = MQC.MQMT_REPLY ;
            String response = "Reply from the Responder Program " ;
            respMessage.writeString(response);
            respQueue.put(respMessage, pmo);
            System.out.println("The response Successfully send ");

            qMgr.commit();
            respQueue.close();
        }

        queue.close();

```

```

        qMgr.disconnect();
    }

    catch (MQException ex)
    {
        System.out.println("An MQ Error Occurred: Completion Code is :\t" +
ex.completionCode + "\n\n The Reason Code is :\t" + ex.reasonCode );
        ex.printStackTrace();
    }

    catch(Exception e) {
        e.printStackTrace();
    }
}
}

```

Message grouping

An application program may need to group a set of updates into a *unit of work*. Such updates are usually logically related and must all be successful for data integrity to be preserved. Data integrity would be lost if one update succeeded while another failed. MQSeries supports transactional messaging.

A unit of work *commits* when it completes successfully. At this point all updates made within that unit of work are made permanent and irreversible. Alternatively, all updates are *backed out* if the unit of work fails. *Syncpoint coordination* is the process by which a unit of work is either committed or backed out with integrity.

Logical message

Logical messages within a group are identified by the GroupId and MsgSeqNumber fields. The MsgSeqNumber starts at 1 for the first message within a group, and if a message is not in a group, the value of the field is 1.

Logical messages within a group can be used to:

- ▶ Ensure ordering (if this is not guaranteed under the circumstances in which the message is transmitted).
- ▶ Allow applications to group together similar messages (for example, those that must all be processed by the same server instance).

Each message within a group consists of one physical message, unless it is split into segments. Each message is logically a separate message, and only the GroupId and MsgSeqNumber fields in the MQMD need bear any relationship to other messages in the group. Other fields in the MQMD are independent; some may be identical for all messages in the group whereas others may be different. For example, messages in a group may have different format names, CCSIDs, encodings, and so on.

Simple GroupSender application

Our next example, GroupSender.java, illustrates sending messages in a group. The application would put 10 simple messages on the queue as a group within a unit of work.

The steps involved are:

- ▶ Import the necessary packages.
- ▶ Set MQEnvironment properties for client connection.
- ▶ Connect to the queue manager.
- ▶ Set the queue open options for output.
- ▶ Open the queue for output.
- ▶ Set the put message options.
 - Set option to maintain logical order of messages.
 - Set option to request the queue manager to generate GroupId.
- ▶ Create a message buffer.
- ▶ Set message header properties.
 - Set messageFlags property to indicate that the messages are in a group.
 - Set the format property to String.
- ▶ Create and put the individual messages on the queue.
- ▶ On the last message in the group, set the messageFlags property to indicate that the message is the last one in the group.
- ▶ Commit the transaction.

Example 7-5 GroupSender.java

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

public class GroupSender {

    private MQQueueManager qmgr;
    private MQQueue outQueue;
    private String queueName = "SAMPLE.QUEUE" ;
```

```

private String host = "ITSOG" ;
private String channel = "JAVA.CLIENT.CHNL" ;
private String qmgrName = "ITSOG.QMGR1" ;
private MQMessage outMsg;
private MQPutMessageOptions pmo;

public static void main (String args[]) {

    GroupSender gs = new GroupSender();
    gs.runGoupSender();

}
public void runGoupSender() {
try {
    init();
    sendGroupMessages();
    qmgr.commit();
    System.out.println("\n Messages successfully Send " ) ;
}
catch (MQException mqe) {
    mqe.printStackTrace();
    try{
        System.out.println("\n Backing out Transaction " ) ;
        qmgr.backout();
        System.exit(2);
    }
    catch(Exception e) {
        e.printStackTrace();
        System.exit(2);
    }
}
catch(Exception e) {
    e.printStackTrace();
    System.exit(2);
}

}
private void init() throws Exception {

    // Set The MQEnvironment for Client Connection
    MQEnvironment.hostname = host ;
    MQEnvironment.channel = channel ;
    MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                MQC.TRANSPORT_MQSERIES);

    //Connect to The Queue Manager
    qmgr = new MQQueueManager ( qmgrName);

    // Set queue open option for output

```

```

int opnOptn = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;

// Open the Queue for output
outQueue = qmgr.accessQueue ( queueName , opnOptn,null,null,null ) ;

}

private void sendGroupMessages() throws Exception {

// Set Put Message Options
pmo = new MQPutMessageOptions();
pmo.options = pmo.options + MQC.MQPMO_LOGICAL_ORDER ;
pmo.options = pmo.options + MQC.MQPMRF_GROUP_ID ;

outMsg = new MQMessage();
outMsg.messageFlags = MQC.QMF_MSG_IN_GROUP ; // Set message flag
indicating that                                     // the messages belong to a
group
outMsg.format = MQC.QMFMT_STRING ; // Set the format for the message to
be String

String msgData = null;
// Send 10 simple messages as a group.
int i = 10;
while(i > 0) {

msgData = "This is the " + i + " th message in the group " ;
outMsg.writeString(msgData);
if (i == 1)
outMsg.messageFlags = MQC.QMF_LAST_MSG_IN_GROUP ; //
indicating that the                                     // the 10 th message is
the last in group.

i--;
// Put each message at a time to the queue.
outQueue.put(outMsg, pmo);
outMsg.clearMessage(); // clear the buffer for re-use with
next message in group.
}

}

}

```

Simple GroupReceiver application

Our next example, GroupReceiver.java, illustrates getting messages in a group. The application would get messages in a group (put by GroupSender application) from the queue as a group within a unit of work.

The steps involved are:

- ▶ Import the necessary packages.
- ▶ Set MQEnvironment properties for client connection.
- ▶ Connect to the queue manager.
- ▶ Set the queue open options for input.
- ▶ Open the queue for input.
- ▶ Set the put message options.
 - Set option to get messages under syncpoint control.
 - Set option to process messages only when all messages within the group are available.
 - Set option to process messages in the logical order.
- ▶ Create a message buffer.
- ▶ Set message header properties.
- ▶ Create and put the individual messages on the queue.
- ▶ Get the messages from the queue until the last message is processed.
- ▶ Display the message content on the console.
- ▶ Commit the transaction.

Example 7-6 GroupReceiver.java

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

public class GroupReceiver {

    private MQQueueManager qmgr;
    private MQQueue inQueue;
    private String queueName = "SAMPLE.QUEUE" ;
    private String host = "ITSOG" ;
    private String channel = "JAVA.CLIENT.CHNL" ;
    private String qmgrName = "ITSOG.QMGR1" ;
    private MQMessage inMsg;
    private MQGetMessageOptions gmo;

    public static void main (String args[]) {

        GroupReceiver gs = new GroupReceiver();
```

```

        gs.runGoupReceiver();
    }
    public void runGoupReceiver() {
    try {
        init();
        getGroupMessages();
        qmgr.commit();
        System.out.println("\n Messages successfully Send " );
    }
    catch (MQException mqe) {
        mqe.printStackTrace();
        try{
            System.out.println("\n Backing out Transaction " );
            qmgr.backout();
            System.exit(2);
        }
        catch(Exception e) {
            e.printStackTrace();
            System.exit(2);
        }
    }
    catch(Exception e) {
        e.printStackTrace();
        System.exit(2);
    }
}

private void init() throws Exception {
    //Set the MQEnvironment Properties for client connection
    MQEnvironment.hostname = host ;
    MQEnvironment.channel = channel ;
    MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                MQC.TRANSPORT_MQSERIES);

    //Connect to the Queue Manager
    qmgr = new MQQueueManager ( qmgrName);

    // Set the Queue open option for input
    int opnOptn = MQC.MQOO_INPUT_AS_Q_DEF | MQC.MQOO_FAIL_IF QUIESCING ;

    // Open the Queue for input
    inQueue = qmgr.accessQueue ( queueName , opnOptn,null,null,null ) ;
}

private void getGroupMessages() throws Exception {

    // Set the get message options

```



```

        gmo = new MQGetMessageOptions();
        gmo.options = MQC.MQGMO_FAIL_IF QUIESCING;
        gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ;
        gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait for messages
        gmo.waitInterval = 5000 ; // Set wait time limit in ms
        gmo.options = gmo.options + MQC.MQGMO_ALL_MSGS_AVAILABLE ; //Get
messages only when all                                     // messages of the
group are available
        gmo.options = gmo.options + MQC.MQGMO_LOGICAL_ORDER ; // Get messages in the
logical order.
        //gmo.matchOptions = MQC.MQMO_MATCH_GROUP_ID ;

        // Create the message buffer.
        inMsg = new MQMessage();

        String msgData = null;
        // Process the messages of the group
        while(true) {
            inQueue.get(inMsg, gmo);
            int msgLength = inMsg.getMessageLength();

            msgData = inMsg.readString(msgLength);
            System.out.println(" The message is \n " + msgData);
            char x = gmo.groupStatus ;
            // Check for the last message flag
            if(x == MQC.MQGS_LAST_MSG_IN_GROUP) {
                System.out.println("B Last Msg in Group") ;
                break;
            }
            inMsg.clearMessage();
        }
    }
}

```

Although we have covered publish/subscribe patterns in each of the other chapters for Java, it is highly recommended that you follow this up by reading *MQSeries Publish/Subscribe Applications*, SG24-6282, which can be downloaded from:

<http://www.ibm.com/redbooks>



Programming with JMS

In this chapter we discuss the Java Messaging Service (JMS) Interface concepts and MQSeries implementation, and programming with JMS. The JMS concepts are discussed in the context of programming patterns in messaging.

8.1 What is JMS?

Java Message Services (JMS) is the standard API for messaging, such as the JDBC API for databases. The JMS specification (1.0.2) was developed by Sun Microsystems with the active involvement of IBM, other enterprise messaging vendors, transaction processing vendors, and RDBMS vendors. JMS provides a common model for Java programs to interact with messaging systems performing various operations against the messaging systems objects. The common operations that a program uses against a messaging systems's objects are creating messages, sending messages, receiving messages and reading messages from the enterprise messaging system. JMS provides a common way for programs being developed in Java to access these messaging system operations.

JMS has two messaging styles, or in other words two domains:

- ▶ One-to-one, or point-to-point model
- ▶ Publish/subscribe model

JMS is only a specification. Each Enterprise Messaging System vendor must provide classes that implement the specification for their specific messaging system.

In this chapter we describe the MQSeries implementation of the JMS API, discuss the JMS API concepts and the MQSeries JMS implementation capabilities, and illustrate the use of MQSeries JMS with different scenarios in which MQSeries JMS implementation can be used.

Why JMS?

The JMS standard is important because:

- ▶ It is the first enterprise messaging API that has achieved wide cross-industry support.
- ▶ It simplifies the development of enterprise applications by providing standard messaging concepts and conventions that apply across a wide range of enterprise messaging systems.
- ▶ It leverages existing, enterprise-proven messaging systems.
- ▶ It allows you to extend existing message-based applications by adding new JMS clients that are interpreted fully with their existing non-JMS clients.
- ▶ It allows you to write portable, message-based business applications.

8.2 Overview

JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product. Messages, as described here, are asynchronous requests, reports or events that are consumed by enterprise applications. They contain vital information needed to coordinate these systems. They contain precisely formatted data that describes specific business actions. Through the exchange of these messages, each application tracks the progress of the enterprise.

JMS defines a common set of enterprise messaging concepts and facilities. It attempts to minimize the set of concepts a Java language programmer must learn to use enterprise messaging products. It strives to maximize the portability of messaging applications. The JMS standard provides a vendor-independent programming interface, but does not define a communications protocol.

The JMS model

JMS defines a generic view of a message passing service. It is important to understand this view, and how it maps onto the underlying MQSeries transport. The generic JMS model is based on interfaces that are defined in Sun's `javax.jms` package and shown in Figure 8-1.

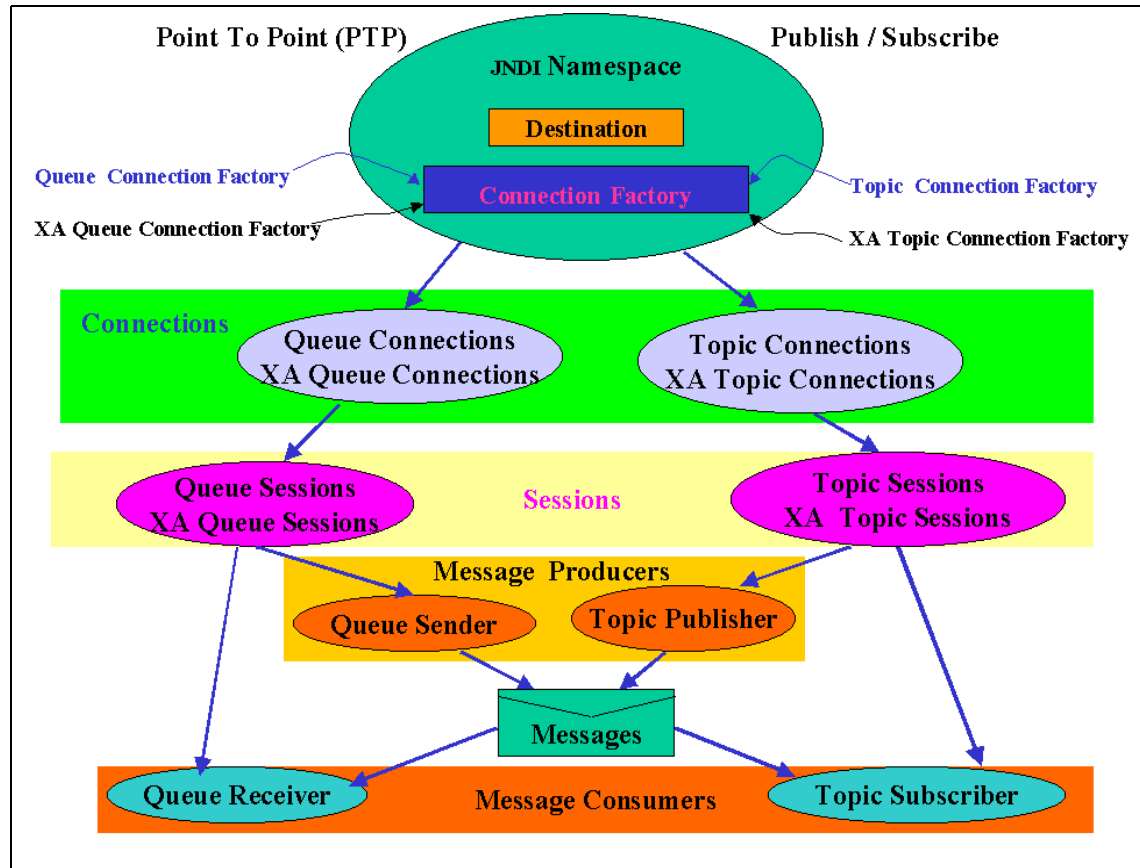


Figure 8-1 JMS model

Connections

Connections provide access to the underlying transport, and are used to create sessions. In MQSeries context, a connection provides a place to hold the parameters, such as queue manager name, remote host name (in Java client connectivity), etc. In other words, an MQSeries JMS Connection typically allocates MQSeries resources outside the Java virtual machine. Connections also support concurrent use.

A connection can provide the following benefits:

- ▶ Encapsulate an open connection with a JMS provider. It typically represents an open TCP/IP socket between a client and a provider service daemon.
- ▶ Its creation is where client authentication takes place.
- ▶ It can specify a unique client identifier.

- ▶ It provides `ConnectionMetaData`.
- ▶ It supports an optional `ExceptionListener`.

Due to the authentication and communication setup done when a connection is created, a connection is a relatively heavyweight JMS object. Most clients will do all their messaging with a single connection. Other more advanced applications may use several connections. JMS does not architect a reason for using multiple connections; however, there may be operational reasons for doing so.

A JMS client typically creates a connection, one or more sessions, and a number of message producers and consumers. When a connection is created, it is in stopped mode. That means that no messages are being delivered.

Important: Connections are created in a *stopped* mode.

It is typical to leave the connection in stopped mode until setup is complete. At that point, the connection's `start()` method is called and messages begin arriving at the connection's consumers. This setup convention minimizes any client confusion that may result from asynchronous message delivery while the client is still in the process of setting itself up.

A connection can immediately be started and the setup can be done afterwards. Clients that do this must be prepared to handle asynchronous message delivery while they are still in the process of setting up.

Note: A message producer can send messages while a connection is stopped.

Connections are not created directly, but are built using a connection factory. Factory objects can be stored in a JNDI namespace, thus insulating the JMS application from provider-specific information. The connection factory objects are created using JMSAdmin, the MQSeries JMS administration tool. This tool enables administrators to define the properties of eight types of MQSeries JMS objects to the JNDI namespace. Please refer to 8.4.9, “Administering JMS JNDI objects with VisualAge for Java using JMSAdmin” on page 260.

Creating a connection

A client uses a connection factory to create connections. The type of connection factory to use depends upon the type of connection you want to make:

- ▶ For a PTP connection, a `QueueConnectionFactory` or `XAQueueConnectionFactory` is used to obtain a `QueueConnection` or `XAQueueConnection`.

- For a publish/subscribe messaging pattern, `TopicConnectionFactory` or `XATopicConnectionFactory` is used to obtain a `TopicConnection` or a `XATopicConnection`.

To create a connection, do the following:

- Retrieve the factory object from JNDI namespace

JNDI API provides naming and directory functionality to applications written in Java. It is designed especially for Java by using Java's object model. Using JNDI, Java applications can store and retrieve named Java objects of any type. In addition, JNDI provides methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes.

Different naming and directory service providers can be plugged in seamlessly behind this common API. This allows Java applications to take advantage of information in a variety of existing naming and directory services, such as LDAP, NDS, DNS, and NIS(YP), and allows Java applications to coexist with legacy applications and systems.

In the JNDI, all naming and directory operations are performed relative to a context. There are no absolute roots. Therefore, the JNDI defines an initial context, which is the starting point for resolution of names for naming and directory operations. Once you have an initial context, you can use it to look up other contexts and objects.

- To retrieve an object from a JNDI namespace, an initial context must be set up, as shown in the following code snippet:

```
import javax.jms.*
import javax.naming.*;
import javax.naming.directory.*;
java.util.Hashtable ;

Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, icf) ;
env.put(Context.PROVIDER_URL, url);
Context ctx = new InitialDirContext(env) ;
```

Where `icf` defines a factory class for the initial context, and `url` defines a context-specific URL.

- Once the initial context is obtained, objects are retrieved from the namespace by using the `lookup()` method. The following code retrieves a `QueueConnectionFactory` name `firstQCF` from an LDAP-based namespace:

```
QueueConnectionFactory qFactory ;
queueFactory = (QueueConnectionFactory)ctx.lookup("cn=firstQCF") ;
```

- Use the factory object to obtain a connection.

The `createQueueConnection()` method on the factory object is used to create a connection, for example:

```
QueueConnection connection ;  
connection = qFactory.createQueueConnection();
```

► Starting the connection

The JMS specification defines that the connections should be created in the “stopped” state. The connection has to be explicitly started before you can send messages using the connection.

The connection is started using the `start()` method, for example:

```
connection.start();
```

Sessions

Provides a context for producing and consuming messages, including the methods used to create message producers and message consumers.

A JMS Session is a single-threaded context for producing and consuming messages. Although it may allocate provider resources outside the Java virtual machine, it is considered a lightweight JMS object.

A session can be created using the `createQueueSession()` or `createTopicSession()` method of the respective connection object.

The create session method takes two parameters:

1. A boolean that determines whether the session is transacted or non-transacted.

A *transacted* session is one in which a group of messages are sent or received as a unit on an all-or-nothing basis.

A *non-transacted* session is one in which the messages are sent or received individually.

2. A parameter that determines the acknowledge mode.

For example:

```
session = connection.createQueueSession(false , Session.AUTO_ACKNOWLEDGE) ;
```

This is the simplest case of creating a non-transacted session with `AUTO_ACKNOWLEDGE`. A connection is thread safe, but sessions and the objects that are created from them are not thread safe. The recommended practice for multi-threaded applications is to use a separate session for each thread.

A session has several purposes:

- It is a factory for its message producers and consumers.

- ▶ It supplies provider-optimized message factories.
- ▶ It supports a single or a series of transactions that combine work spanning its producers and consumers into atomic units.
- ▶ A session defines a serial order for the messages it consumes and the messages it produces.
- ▶ A session retains messages it consumes until they have been acknowledged.
- ▶ A session serializes execution of message listeners registered with its message consumers.

A session can create and service multiple message producers and consumers.

One typical use is to have a thread block on a synchronous message consumer until a message arrives. The thread may then use one or more of the session's message producers.

If a client wants to have one thread producing messages while others consume them, the client should use a separate session for its producing thread.

Once a connection has been started, any session with a registered message listener, or listeners, is dedicated to the thread of control that delivers messages to it. It is erroneous for client code to use the session or any of its constituent objects from another thread of control. The only exception to this is the use of the session or connection close method.

It should be easy for most clients to partition their work naturally into sessions. This model allows clients to start simply and incrementally add message processing complexity as their need for concurrency grows.

The close method is the only session method that can be called while some other session method is being executed in another thread.

A session may be optionally specified as transacted. Each transacted session supports a single series of transactions. Each transaction groups a set of message sends and a set of message receives into an atomic unit of work. In effect, transactions organize a session's input message stream and output message stream into series of atomic units. When a transaction commits, its atomic unit of input is acknowledged and its associated atomic unit of output is sent. If a transaction rollback is done, its sent messages are destroyed and the session's input is automatically recovered.

The content of a transaction's input and output units is simply those messages that have been produced and consumed within the session's current transaction.

A transaction is completed using either its session's commit or rollback method. The completion of a session's current transaction automatically begins the next. The result is that a transacted session always has a current transaction within which its work is done.

Message producers

A JMS client uses message producers to send messages to a specific destination. Message producers are created by passing the destination to a create message producer method supplied by a session object.

In point-to-point messaging, this would be a `QueueSender` that is created using the `createSender` method on a `QueueSession` object. A `QueueSender` is normally created for a specific queue, so that all the messages sent using that sender are sent to the same destination. The destination is specified using a `Queue` object. Queue objects can be either created at runtime or built and stored in a JNDI namespace. For example:

```
Queue ioQueue ;
ioQueue = (Queue)ctx.lookup(qLookup) ;
sender = session.createSender(ioQueue);
```

In publish/subscribe messaging, this would be a `TopicPublisher` that is created using a `createPublisher` method on a `TopicSession` object.

Normally the Topic is specified when a `TopicPublisher` is created. In this case, attempting to use the methods for an unidentified `TopicPublisher` will throw an `UnsupportedOperationException`.

For example:

```
Topic topic ;
topic = (Topic)ctx.lookup( "cn=first.topic" ) ;
TopicPublisher pub = session.createPublisher(topic);
```

A client also has the option of creating a message producer without supplying a destination. In this case, a destination must be input on every send operation. A typical use for this style of message producer is to send replies to requests using the request's `replyTo` destination.

A client can specify a default delivery mode, priority and time-to-live for messages sent by a message producer. It can also specify delivery mode, priority and time-to-live per message.

A client can specify a time-to-live value in milliseconds for each message it sends. This value defines a message expiration time, which is the sum of the message's time-to-live and the GMT time at which it is sent (for transacted sends, this is the time the client sends the message, not the time the transaction is committed).

Sending a message

Messages are sent using message producers. So when sending a message in the point-to-point (PTP) model, you would use `QueueSender` objects, and in the publish/subscribe model, you would use `TopicPublisher` objects.

In the PTP model, use the `send()` method of the `QueueSender` to send messages. For example:

```
outmessage = session.createTextMessage();
outmessage.setText("Sample Message ");
sender.send(outMessage);
```

In the publish/subscribe model, use the `publish` method of the `TopicPublisher` object to publish messages. For example:

```
pub.publish(outMessage);
```

Message consumers

A message consumer is used to receive messages. The `MessageConsumer` interface is the parent interface for all message consumers. In the PTP model, this would be `QueueReceiver`. In the publish/subscribe model, this would be `TopicSubscriber`.

A message consumer can be created with a *message selector*, which allows the clients to select a subset of messages based on the criteria specified in the message selector. Please refer to 8.7, "Message selectors" on page 289 for details.

A message consumer client may either synchronously receive a message or have the messages asynchronously delivered as they arrive. A client can request the next message from a message consumer using one of its receive methods. There are several variations of receive that allow a client to poll or wait for the next message.

A client can register a `MessageListener` object with a message consumer. As messages arrive at the message consumer, it delivers them by calling the `MessageListeners onMessage` method. Refer to 8.6, "Asynchronous processing" on page 285 for further information.

In the PTP model, a `QueueReceiver` is created by using the `createReceiver()` method on the `QueueSession` object. The method takes a `Queue` parameter that defines where the messages are to be received from. For example:

```
QueueReceiver queueReceiver = session.createReceiver(ioQueue) ;
```

In the publish/subscribe model, a `TopicSubscriber` object is created by using `createSubscriber()` method of the `TopicSession` object. For example:

```
TopicSubscriber sub = session.createSubscriber(topic);
```

Receiving messages

Messages are received using message consumers. In PTP messaging, `QueueReceiver` objects would be used to receive messages, and in publish/subscribe messaging, `TopicSubscriber` objects would be used to receive messages.

In the PTP model, you can use the `receive` method of the `QueueReceiver` object to receive messages. For example:

```
Message inMessage = queueReceiver.receive(800) ;
```

The parameter specified is the timeout in milliseconds. The method calls to receive the next message that arrives within the specified timeout interval.

In the publish/subscribe model, use the `receive()` method of the `TopicSubscriber` to receive subscriptions. For example:

```
Message inMsg = sub.receive() ;
```

The fragment of code performs a `get` with a wait.

Important: Note that a connection is thread safe, but sessions, message producers, and message consumers are not. The recommended strategy is to use one session per application thread.

In MQSeries terms, a connection provides a scope for temporary queues. Also, it provides a place to hold the parameters that control how to connect to MQSeries. Examples of these parameters are the name of the queue manager and the name of the remote host if you use the MQSeries Java client connectivity. Session contains an `HCONN` and therefore defines a transactional scope.

The message producer and message consumer contain an `HOBJ` that defines a particular queue for writing to or reading from. Note that normal MQSeries rules apply. Only a single operation can be in progress per `HCONN` at any given time. Therefore, the message producers or message consumers associated with a session cannot be called concurrently.

This is consistent with the JMS restriction of a single thread per session. Puts can use remote queues, but gets can only be applied to queues on the local queue manager. The generic JMS interfaces are subclassed into more specific versions for point-to-point and publish/subscribe behavior. The point-to-point versions are:

- QueueConnection QueueSession QueueSender QueueReceiver

A key idea in JMS is that it is possible to write application programs that only use references to the interfaces in `javax.jms`. This is strongly recommended.

All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory TopicConnectionFactory
 - Queue
 - Topic

These are known as “administered objects”, that is, objects that can be built using a vendor-supplied administration tool and can be stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation. MQSeries classes for Java Message Service consists of a number of Java classes and interfaces that are based on the Sun `javax.jms` package of interfaces and classes.

Clients should be written using the Sun interfaces and classes that are listed below, and that are described in detail in the following sections. The names of the MQSeries objects that implement the Sun interfaces and classes have a prefix of “MQ” (unless stated otherwise in the object description). The descriptions include details about any deviations of the MQSeries objects from the standard JMS definitions.

Transactions

A session is a sequence of messages sent and received on a connection between a client and the messaging system. When a session is created, it is created as either non-transacted (default) or transacted. A transacted session ensures that a group of messages is sent and received on an all-or-none basis. A non-transacted session means that messages are sent and received individually. An example of the use of a transacted session would be online shopping. The customer opens an order (beginning the transaction). Each item selection the customer makes is a message to add an item to the order. The customer closes the order (ending the transaction). If the sender commits the transaction, all messages in the group are sent. If the sender rolls back the transaction, none of the items are ordered.

8.3 JMS messages

JMS provides different message types. Each contains some information about its content.

The message types defined in JMS are:

- ▶ BytesMessage
- ▶ MapMessage
- ▶ ObjectMessage
- ▶ StreamMessage
- ▶ TextMessage

JMS messages are comprised of three parts:

- ▶ Header

Header fields contain information used by the provider for routing and identifying the messages, as well as information that can be used by the client.

- ▶ Properties

In addition to the fields in the header, JMS messages provide a facility for adding optional header fields to a message. These properties can be categorized as:

- Application-specific properties, which are used by clients to add processing-specific information for clients processing the messages
- Standard properties, which are JMS-specific properties
- Provider-specific properties, which are required by the native client

- ▶ Body

This is where the data portion of the message is. JMS provides different types of message bodies, which cover the majority of messaging styles currently being used.

8.3.1 Mapping JMS messages onto MQSeries messages

MQSeries messages have three components:

- ▶ The MQSeries message descriptor (MQMD)
- ▶ An MQSeries MQRFH2 header
- ▶ The message body

The MQRFH2 header is optional. The MQRFH2 header is an extensible header with a fixed portion and a variable header portion. The fixed portion is modeled on the standard MQSeries header pattern. The MQRFH2 header can carry JMS-specific data that is associated with the message content, and can also carry additional information not directly associated with JMS.

The JMS messages are translated or transformed into MQSeries messages in two ways:

- Mapping

Mapping involves translating the JMS message header and message properties to the equivalent MQMD values for which there are equivalent MQMD fields.

- Copying

In cases where the JMS message header and message property values do not have an equivalent MQMD field, the JMS header field or property is passed, possibly transformed as a field in the MQRFH2 header.

The inclusion of the MQRFH2 header is optional. In an outgoing message, its inclusion is determined by a flag in the JMS Destination class. The flag can be set when defining the JMD administered objects using the JMSAdmin tool. The administrator can indicate that the JMS client is communicating with a non-JMS client by setting the MQSeries Destination's TargetClient value to JMSC.MQJMS_CLIENT_NONJMS_MQ. Since the MQRFH2 header carries JMS-specific information, you should include it when the sender knows that the receiving client is a JMS client. If the receiver is not a JMS client, omit the MQRFH2 header.

Working with JMS message types

In this section, we illustrate how the different JMS message types can be used, using an example of a shipping tracking ID being sent as a message from a JMS client.

Using the TextMessage class

Information in a message can be sent as a human-readable text string, which can be read and processed, or displayed by a client.

We can send the tracking ID as a string in a TextMessage object as follows:

```
String trackingId ;
TextMessage message;
message = session.createTextMessage();
message.SetText(trackingId);
```


Using the BytesMessage class

In BytesMessage, information is sent in a binary format. The information in the message can be constructed as a bytes array.

Such a message can be prepared as:

```
byte[] trackingId ;
BytesMessage message ;
message = session.createBytesMessage();
message.writeBytes(trackingId);
```

Using a map message

In map messages, information can be sent as name value pairs. This way, the message itself can include the metadata about the data contained in the message.

In our example of sending a message requesting information on a trackingId, the name (metadata) can be trackingId and the actual value (say AMX100000) would be the value of the name value pair.

The message can be constructed as:

```
String attributeName = "trackingId";
String attributeValue = "AMX100000"
MapMessage message;
message = session.createMapMessage();
message.setString(attributeName, attributeValue) ;
```

Note: If the value part was of data type say, long, in that case you can use the message.setLong method. Use the appropriate method for the data type you are working with.

Using a stream message

In a stream message, similar to the map message, a message can consist of various fields written in sequence, each field with its own primitive types. In a map message, a client can set any number of fields in the map and the processing client can read specific fields without having to process the entire map. However, in a stream message, even if the processing client is interested only in a subset of the fields in the stream message, the client would have to read each field in the message (and discard the fields it is not interested in). Another important difference is that in a map message the order of the fields are not important, but in the stream message the fields are read in the same order they were written.

A stream message can be constructed as:

```
String attributeName = "trackingId" ;
String attributeValue = "AMX100000" ;
```

```
StreamMessage message;
session.createStreamMessage();
message.writeString(attributeName) ;
message.writeString(attributeValue) ;
```

Using an object message

Object messages can be used to pass a Java object as a message, which the message receiving client can use methods in the object to extract the data.

In the following illustration, we use a tracking object:

```
public class TrackingObject {
    private String attributeName ;
    private String attributeValue ;

    public void setAttributeName ( String name ) {
        attributeName = name ;
    }

    public void setAttributeValue( String value) {
        attributeValue = value ;
    }

    public String getAttributeName() {
        return attributeName;
    }
    public String getAttributeValue() {
        return attributeValue ;
    }
}
```

Using the above tracking object we can sent the tracking information as a tracking object.

We can construct such a message as:

```
String attributeName = "trackingId" ;
String attributeValue = "AMX100000" ;

TrackingObject trackingObject = new TrackingObject();
ObjectMessage = message ;

trackingObject.setAttributeName(attributeName) ;
trackingObject.setAttributeValue(attributeValue) ;

message = session.createObjectMessage();
message.setObject(trackingObject) ;
```

On the receiving side, the client can use the get methods in the tracking object to extract the data.

Message acknowledgment

JMS messages support the acknowledge method for use when a client has specified that a JMS consumer's messages are to be explicitly acknowledged. If a client uses automatic acknowledgment, calls to acknowledge are ignored.

There are three types of message acknowledgment. The type of message acknowledgment is specified when creating a session. These different types are:

► AUTO_ACKNOWLEDGE

With AUTO_ACKNOWLEDGE mode, the session automatically acknowledges a message when it has either successfully returned from a call to the receiver, or the message listener registered by the message consumer to process the message successfully returns.

```
Session session = queueConnection.createQueueSession(false,  
Session.AUTO_ACKNOWLEDGE);
```

► CLIENT_ACKNOWLEDGE

With CLIENT_ACKNOWLEDGE mode, the client explicitly acknowledges a message by calling the acknowledge method on the message.

```
Session session = queueConnection.createQueueSession(false,  
Session.CLIENT_ACKNOWLEDGE);
```

```
Then once the message is processed, the client can issue  
message.acknowledge() ;  
method to acknowledge the message.
```

When using the CLIENT_ACKNOWLEDGE mode, care must be taken to avoid accumulation of large number of unacknowledged messages while processing the messages. This buildup of unacknowledged messages can cause resource exhaustion, leading to failures.

► DUPS_OK_ACKNOWLEDGE

The DUPS_OK_ACKNOWLEDGE mode instructs the session to lazily acknowledge the delivery of the messages. This is likely to result in the delivery of duplicate messages if JMS fails. It should be used by consumers who are tolerant in processing duplicate messages. In cases where the client is tolerant of duplicate messages, some enhancement in performance can be achieved using this mode, since a session has lower overhead in trying to prevent duplicate messages.

8.3.2 JMS additional features

Other features that JMS provides include:

- ▶ Asynchronous message delivery
 - Employs the concept of message listeners
 - Uses event-based model that triggers a specified function on preset events

A JMS client can register a listener object that implements the `MessageListener` interface with a message consumer (message receiver). When the messages arrive for the registered consumer, the message is made available to the message consumer by calling the listeners `onMessage` method.

- ▶ Message selectors
 - Content-based retrieval of specific messages
 - Use of SQL92-based query functions

The JMS message provides a facility to provide user-defined metadata to the JMS message header (outside the actual body of the message). JMS programs can take advantage of this facility to select a subset of messages based on a selection criteria or, in other words, a JMS client can choose only those messages that it is interested in.

8.4 MQSeries JMS implementation

The key points of JMS implementation with MQSeries are:

- ▶ Platform coverage: AIX, HP-UX, Windows NT, Solaris, and Linux
- ▶ Comprised of JAR files encapsulating the key functionality:
 - `com.ibm.mq.jms.jar`
 - `com.ibm.mq.jar`
- ▶ Administration tool for defining administered objects to a JNDI namespace:
 - JMSAdmin
- ▶ Available as a product extension via Web download
- ▶ MQSeries classes for Java and JMS with MQSeries V5.2

8.4.1 MQSeries JMS installation

For enabling JMS support for MQSeries, you must install the Java classes that implement the JMS interfaces. The JMS classes for MQSeries can be downloaded in the MA88 SupportPac from the IBM Web site at:

<http://www-3.ibm.com/software/ts/mqseries/txppacs/>

This SupportPac is free. You can download the appropriate version for the supported platforms and install it following the installation instructions provided with the SupportPac.

Important: When you install the JMS SupportPac, all files in the Java subdirectory of MQSeries installation are backed up during the install and the new files are copied from the SupportPac. If you had AMI support installed in your environment, you may need to reinstall the AMI SupportPac.

Once you have installed the JMS SupportPac, in order to run publish/subscribe applications with JMS you need to run the MQSC script called MQJMS_PSQ.mqsc located in the java\bin subdirectory under the MQSeries installation directory. To run the script on the default queue manager, run the following command from an operating system command window:

```
runmqsc < "C:\Program Files\IBM\MQSeries\java\bin\MQJMS_PSQ.mqsc"
```

where C:\Program Files\IBM\MQSeries\ is assumed to be the MQSeries installation directory.

8.4.2 JMS administered objects - JNDI and JMSAdmin

Directory services incorporate a naming facility to provide abstract level namespaces that encapsulate the arrangement and identification of various entities such as machine name, services, users, provider-specific objects such as QueueManagers, Queues, Topics etc., in a messaging parlance. This enables the use of logical namespaces that allow easier discovery and identification of the objects in the network. The Java Naming and Directory Interface (JNDI) API implementation provides directory and naming functionality to programs developed in Java. This allows programs in Java to discover and retrieve objects of any type from the JNDI namespace.

The JMS administered objects are the objects that a JMS application stores and retrieves from the JNDI namespace. The JMS administered objects are normally created by administrators. The JMS administered objects contains the configuration information of the underlying messaging service provider information. In this section we will be discussing defining, administering and using these objects for developing and deploying JMS applications with MQSeries.

Important: In our sample programs, we use the Persistent Name Server provided by WebSphere Application Server as the JNDI server. We developed the sample programs using VisualAge for Java Enterprise Edition and WebSphere Application Server.

8.4.3 JMSAdmin tool

The administration tool JMSAdmin enables administrators to define MQSeries JMS objects and to store them within a JNDI namespace. JMS clients can retrieve these administered objects from the namespaces by using the JNDI interface. The tool also allows administrators to manipulate directory namespace subcontexts within JNDI.

There are eight administered objects you can administer with JMSAdmin tool:

- ▶ MQQueueConnectionFactory
- ▶ MQTopicConnectionFactory
- ▶ MQQueue
- ▶ MQTopic
- ▶ MQXAQueueConnectionFactory
- ▶ MQXATopicConnectionFactory
- ▶ JMSWrapXAQueueConnectionFactory
- ▶ JMSWrapXATopicConnectionFactory

Note: JMSWrapXAQueueConnectionFactory and JMSWrapTopicConnectionFactory are classes that are specific to WebSphere and are contained in the package `com.ibm.ejs.jms.mq`.

8.4.4 Invoking the administration tool

The administration tool has a command-line interface. You can use this interactively or use it to start a batch process. The interactive mode provides a command prompt where you can enter administration commands. In the batch mode, the command to start the tool includes the name of a file that contains an administration command script.

To start the tool in interactive mode, enter the command:

```
JMSAdmin [-t ] [-v ] [-cfg config_filename ]
```

Where:

- ▶ `-t` enables trace (the default is trace off)
- ▶ `-v` produces verbose output (default is terse output)
- ▶ `-cfg config_filename` is the name of an alternative configuration file

8.4.5 JMSAdmin tool configuration

JMSAdmin tool uses a configuration file with the following parameters in it:

► INITIAL_CONTEXT_FACTORY

This indicates the service provider that the tool uses. There are currently three supported values for this property:

- `com.sun.jndi ldap.LdapCtxFactory`

You use this class name when you are using an LDAP server as the JNDI server. (We used IBM Secureway Server 3.2.1 in our tests.)

- `com.sun.jndi.fscontext.RefFSContextFactory`

Use this class name when you are using a file system to store the JNDI objects.

- `com.ibm.ejs.ns.jndi.CNInitialContextFactory`

Use this class name when you are using the Persistent Name Server of WebSphere.

► PROVIDER_URL

This indicates the URL of the session's initial context, the root of all JNDI operations carried out by the tool. Three forms of this property are currently supported:

- `ldap://hostname/contextname` (for LDAP)

- `file:[drive:]/pathname` (for file system context)

The tool will not create the directory specified in the path. Make sure the directory exists prior to using the JMSAdmin tool.

- `iiop://hostname[:port] /[/?TargetContext=ctx]` (with WebSphere Persistent Name Server)

► SECURITY_AUTHENTICATION

This indicates whether JNDI passes security credentials to your service provider. This parameter is used only when an LDAP service provider is used. This property can currently take one of three values:

- None (anonymous authentication)
- Simple (simple authentication)
- CRAM-MD5 (CRAM-MD5 authentication mechanism)

The above values are specified in a configuration file and you can specify the configuration file when invoking the JMSAdmin tool with the `-cfg` parameter. A sample configuration file named `jmsadmin.config`, located under `MQSeriesIntalldirectory\java\bin`, can be edited or used to create the configuration file with the property values specific to your environment.

8.4.6 Using JMSAdmin with the Persistent Name Server

Edit the JMSAdmin.config file that comes with the installation (you can find it under MQSeriesInstallDirectory\java\bin) to indicate that the JMSAdmin tool should use the Persistent Name Server and edit the PROVIDER_URL property value to point to the Persistent Name Server. Then save the file in another folder, for example C:\temp.

In the JMSAdmin.config file, the values we provide are:

- ▶ INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory

This value is valid for the Persistent Name Server provided by VisualAge for Java Enterprise Edition and by WebSphere Application Server. For another JNDI server, the INITIAL_CONTEXT_FACTORY variable must be provided in the accompanying documentation.

- ▶ PROVIDER_URL=iiop://hostname/

where hostname is the name of the machine where the Persistent Name Server is running. When the protocol IIOP is used, the default port is 900 and doesn't need to be mentioned. If you change the default port of the Persistent Name Server (the default port is 900) in the bootstrap port parameter, you must indicate the port in the PROVIDER_URL variable, for example iiop://hostname:901/ (assuming you changed to port number 901 instead of the default port of 900).

Invoke the JMSAdmin tool using the -cfg parameter and give the path where you saved the JMSAdmin.config file after editing.

8.4.7 Using the Persistent Name Server with VisualAge for Java

The Persistent Name Server's database should be accessible from VisualAge for Java using JDBC. If it is the first time setting up the Persistent Name Server, you will need to create a database (we used UDB 7.1 as our database server). We created a database named jndi. For the Persistent Name Server to access the database using JDBC, the VisualAge for Java workspace class path should also include Db2UdbInstallDirectory\java\db2java.zip.

Starting the Persistent Name Server from VisualAge for Java

- ▶ From the VisualAge for Java menu bar, select **Workspace -> Tools -> WebSphere Test Environment** (see Figure 8-2).

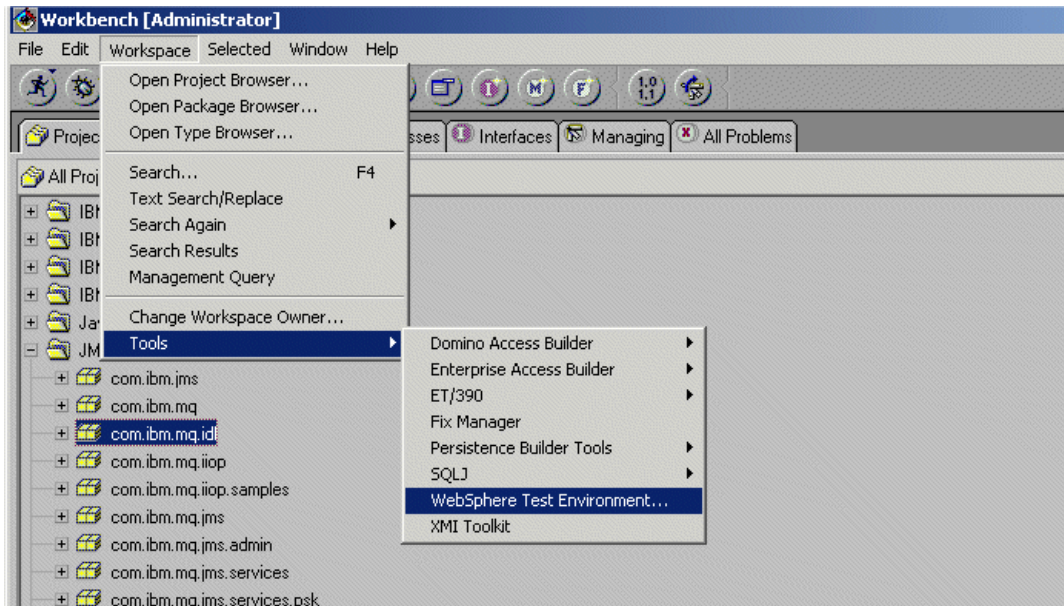


Figure 8-2 Starting the Persistent Name Server from VisualAge for Java step 1

- ▶ Select the Persistent Name Server in the left-hand pane and enter the parameter to connect to the configuration database of the Persistent Name Server, as shown in Figure 8-3.

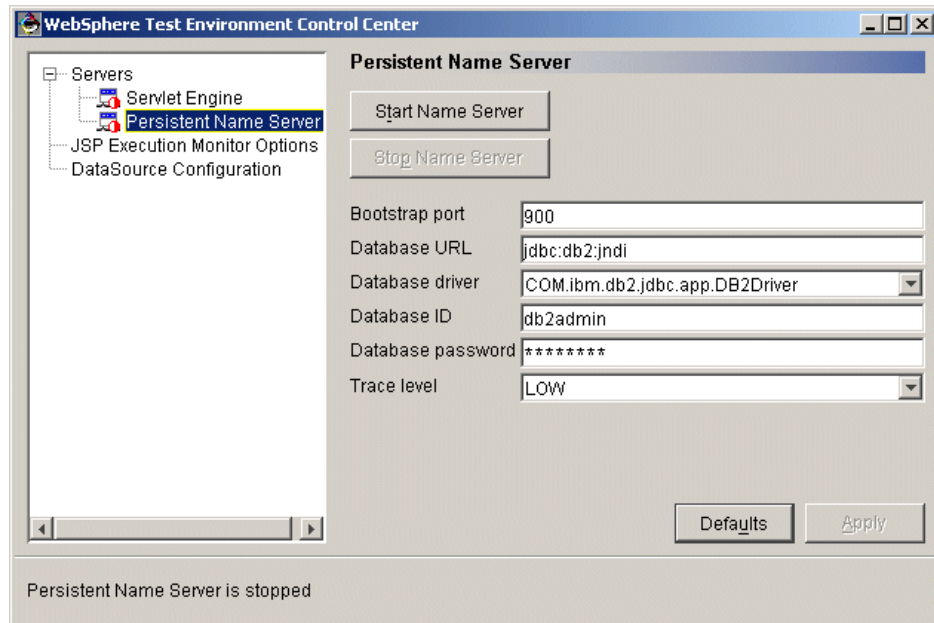


Figure 8-3 Persistent Name Server configuration panel

Important: The Persistent Name Server by default uses port number 900. If you specify a different port, programs using the name server for JNDI lookup should include the port number in the PROVIDER_URL parameter.

- ▶ Start the Persistent Name Server by clicking the **Start Name Server** button.

8.4.8 Configuring VisualAge for Java for use with JMS

The steps involved in setting up VisualAge for Java to work with JMSAdmin or developing JMS Applications are:

- ▶ Verify that IBM Enterprise Extension Libraries and IBM WebSphere Test Environment are loaded in the workspace.
- ▶ Import the following JAR files in to the project. The JAR files can be found under MQInstallationDirectory\Java. We created a new project in VisualAge for Java named JMSTest and this project is used in our illustration.
 - com.ibm.mq.jar
 - com.ibm.mqbind.jar
 - com.ibm.mqjms.jar
 - com.ibm.mq.iiop.jar

- jms.jar
 - jndi.jar
 - ldap.jar
 - fscontext.jar
 - providerutil.jar
- Add the directory MQSeriesInstallDirectory\java\lib to the VisualAge for Java workspace class path. This can be done by selecting **Window -> Options->Resources -->Edit** and adding the directory (refer to Figure 8-4).

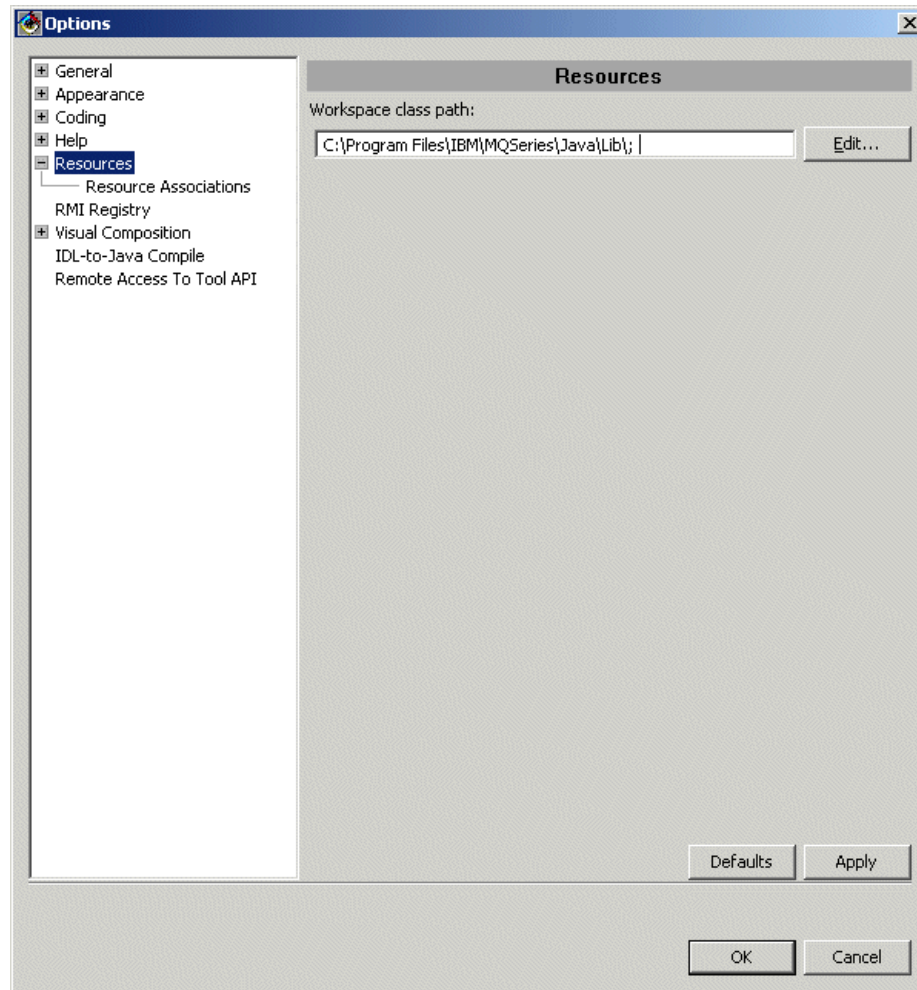


Figure 8-4 Setting VisualAge For Java workspace class path

- If you want to use the binding mode when accessing the queue manager, the path system environment variable should include the MQSeriesInstallDirectory\Java\bin directory. This directory contains the mqjbnd02.dll which is required for using the binding mode.

8.4.9 Administering JMS JNDI objects with VisualAge for Java using JMSAdmin

In this section we describe how to define and administer the JNDI objects required for JMS implementation using the MQSeries JMS administration tool JMSAdmin with VisualAge for Java.

We illustrate the use of the tool with the objects that are used in the subsequent programming samples in this chapter for both the point-to-point and the publish/subscribe messaging patterns.

MQSeries objects used

Table 8-1 lists the MQSeries objects that are used in the sample programs.

Table 8-1 MQSeries objects used in sample programs

Object Name	Description
SAMPLE.QMGR1	Queue manager that will be used in the sample programs
PTP.QUEUE.LOCAL	Queue used by point-to-point samples
PTP.REPLY.QUEUE.LOCAL	Queue used with request/reply to send reply messages
JMS.SVR.CHNL	Server Connection Channel used for client connections
SYSTEM.JMS.ADMIN.QUEUE	The JMS publish/subscribe Administration queue
SYSTEM.JMS.PS.STATUS.QUEUE	The JMS publish/subscribe Status queue
SYSTEM.JMS.REPORT.QUEUE	The JMS publish/subscribe Report queue
SYSTEM.JMS.MODEL.QUEUE	The JMS publish/subscribe Subscribers Model queue. (This model queue is used by subscribers to create a permanent queue for subscriptions)
SYSTEM.JMS.ND.SUBSCRIBER.QUEUE	JMS publish/subscribe Default Non-Durable Shared queue (the default shared queue used by non-durable subscribers)
SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE	JMS publish/subscribe Default Non-Durable Shared Queue for ConnectionConsumer functionality

Object Name	Description
SYSTEM.JMS.D.SUBSCRIBER.QUEUE	JMS publish/subscribe Default Durable Shared Queue (the default shared queue used by durable subscribers)
SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE	JMS publish/subscribe Default Durable Shared Queue for ConnectionConsumer functionality

8.4.10 Defining JMS administered objects

First, we implement the administered objects that are used with the point-to-point sample program. We will work with the Persistent Name Server as the JNDI name server and invoke the JMSAdmin tool from VisualAge for Java. You should have VisualAge set up to work with JMS, and the Persistent Name Server should be configured and started as explained in 8.4.7, “Using the Persistent Name Server with VisualAge for Java” on page 256 and 8.4.8, “Configuring VisualAge for Java for use with JMS” on page 258.

Invoking the JMSAdmin tool from VisualAge for Java

- Update the JMSAdmin.config file to indicate that the JMSAdmin will be using the Persistent Name Server by editing the property value for the initial context factory:

```
INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory
```

Note: In the LDAP server, it would be `com.sun.jndi.ldap.LdapCtxFactory` and in the file system context, the value would be `com.sun.jndi.fscontext.RefFSContextFactory`.

Edit the PROVIDER_URL to point to the JNDI name server. In this case we are running the Persistent Name Server on a server named ITSOG, so it will be:

```
PROVIDER_URL=iiop://itsog/
```

- Copy the above JMSAdmin.config file in the Project Resources of the project you are working with (we are using a project named JMSTest). This can be done from VisualAge for Java by doing the following:
 - On the Resources tab, right-click the project and click **Add --> Resources**, as shown in Figure 8-5.

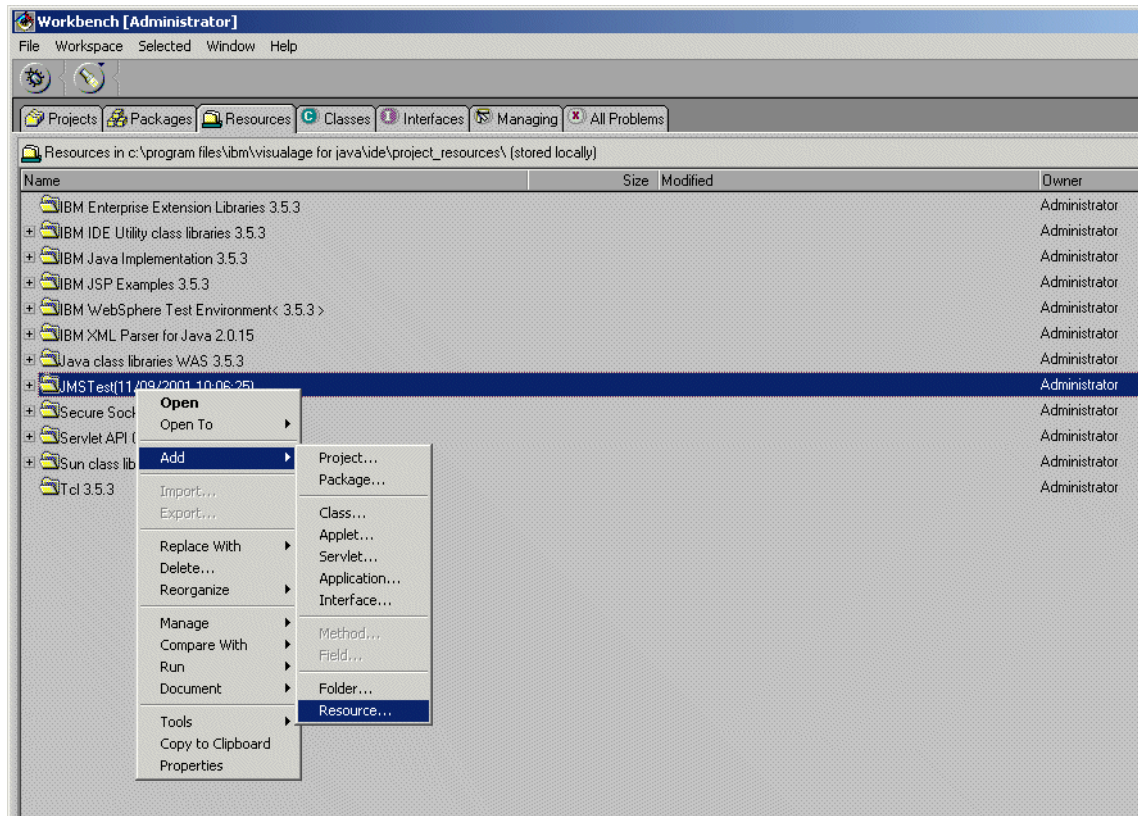


Figure 8-5 Adding resources to VisualAge Project (step 1)

- ▶ Select the directory where the JMSAdmin.config file is located and click the **OK** button (see Figure 8-6).

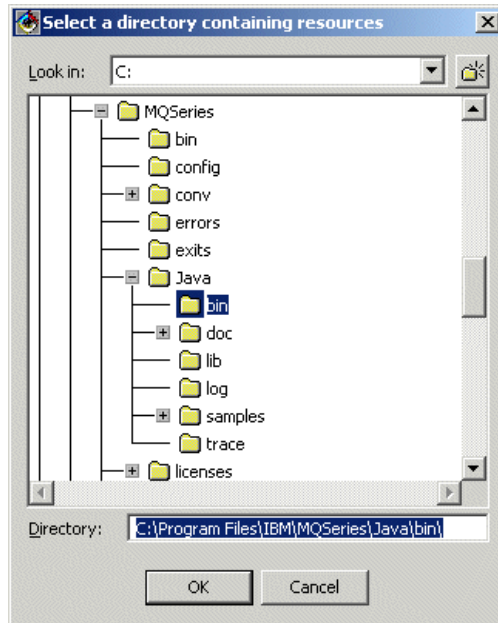


Figure 8-6 Selecting the Resource Directory

- In the Add resources window shown in Figure 8-7, select the JMSAdmin.config file and click **OK**.

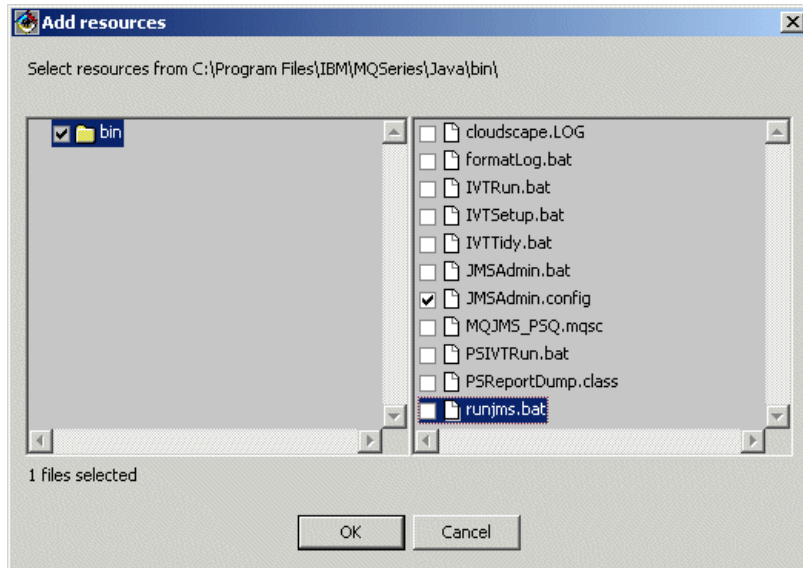


Figure 8-7 Adding the JMSAdmin configuration file resource

- Expand your project and locate the JMSAdmin class (it is in the package `com.ibm.mq.jms.admin`). Right-click the JMSAdmin class and choose **Properties** (see Figure 8-8).

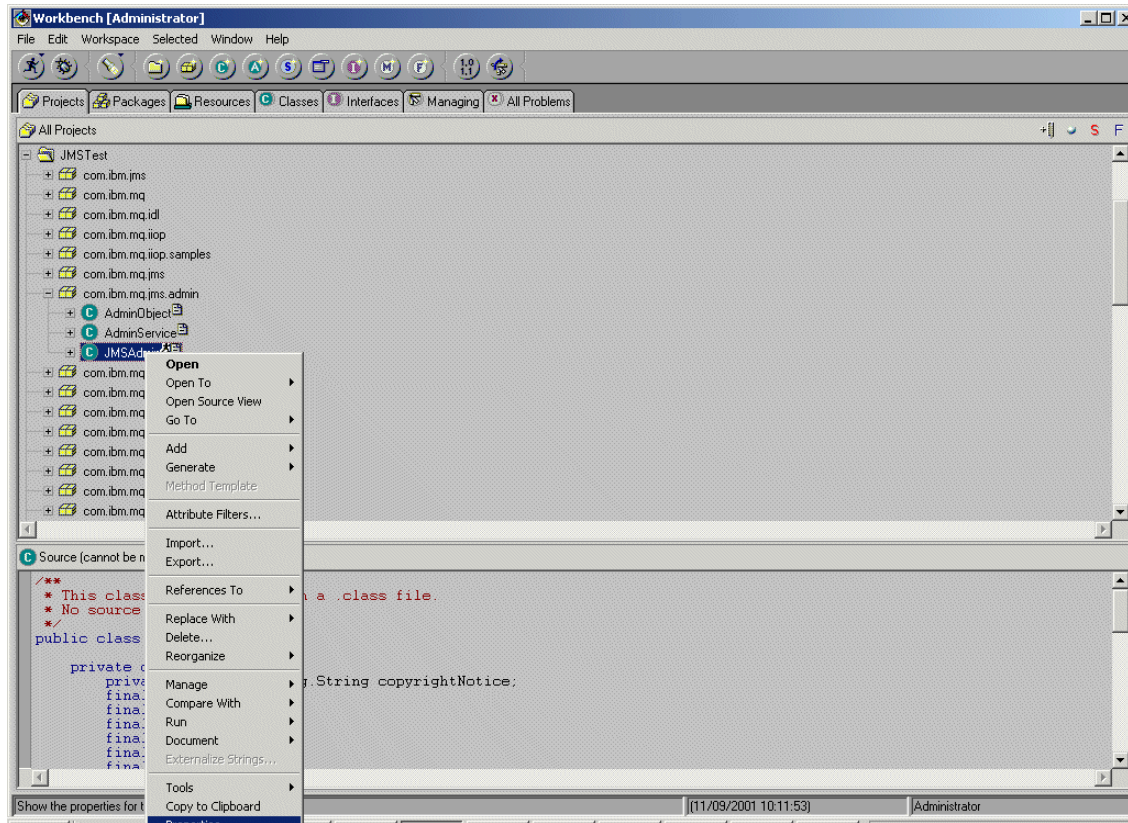


Figure 8-8 Setting up class path for JMSAdmin (step 1)

- In the resulting Properties for JMSAdmin window shown in Figure 8-9, select the **Classpath** tab, then click **Edit** and select the **IBM Enterprise Extension Libraries** and **IBM WebSphere Test Environment**, and then click **OK**.

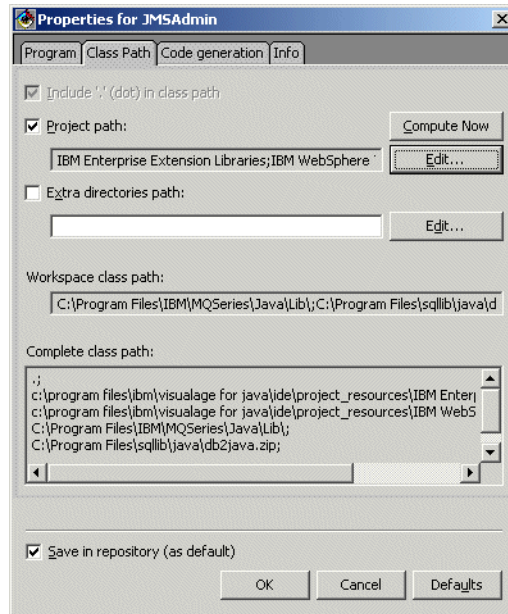


Figure 8-9 Class path for JMSAdmin

The setup for JMSAdmin is complete. Now we can run the JMSAdmin tool. Highlight the JMSAdmin class and click the **Run** button to run the tool. On successful startup, the console should look like that shown in Figure 8-10.

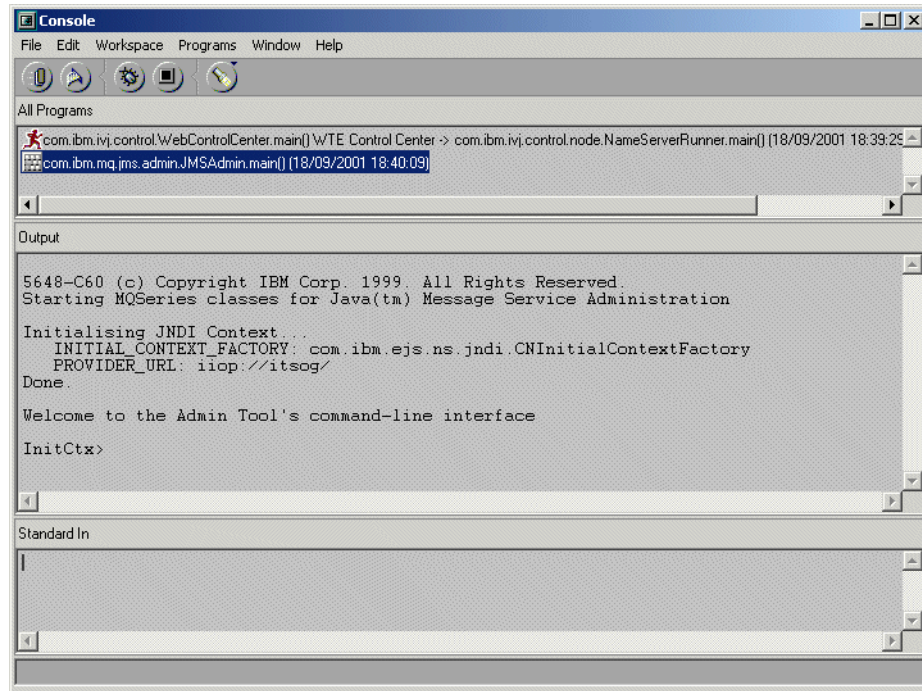


Figure 8-10 JMSAdmin console

Important: If you have problems starting up the JMSAdmin tool, verify that the class path settings are correct and that the Persistent Name Server is started.

Now we are ready to create the JMS administered objects in JNDI.

1. Create a context. We use the JMSAdmin command **DEFINE CONTEXT(context)** name to create the context for the context named ptpCtx that will be used in the point-to-point sample programs.

In the console Standard input area (see Figure 8-11 on page 268), type the command:

```
def ctx(ptpCtx)
```

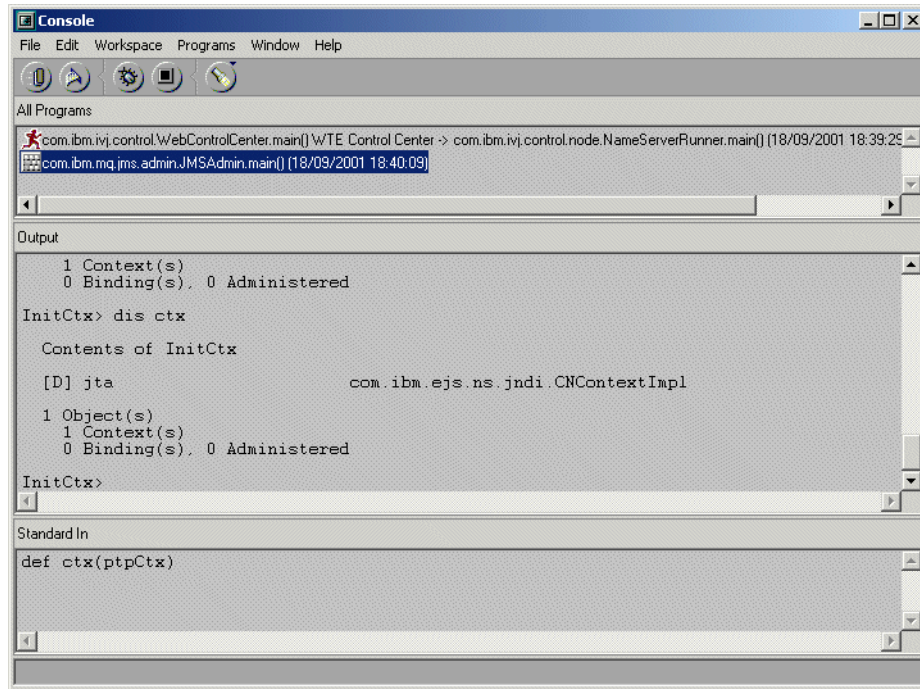


Figure 8-11 Creating a JNDI context using JMSAdmin tool

To display the context you just defined, use the **dis ctx** command. On displaying the context, you should see the context you just created as shown in Figure 8-12.

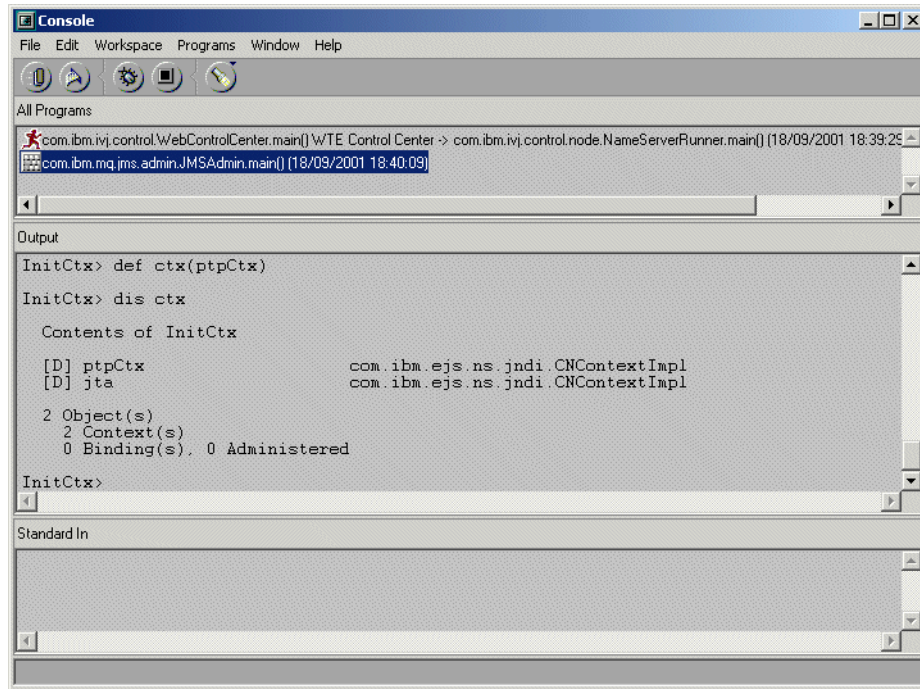


Figure 8-12 Context in the JMS admin tool

2. Change to the context you just created using the following command:

```
chg ctx(ptyCtx)
```

3. Create a QueueConnection Factory named ptpQcf, with the following commands:

```
def qcf(ptyQcf) transport(CLIENT) +  
channel(JMS.SRV.CHNL) qmanager(SAMPLE.QMGR1) host(ITSOG)
```

4. Create a Queue object named ptpQueue, as follows:

```
def q(ptyQueue) queue(PTP.QUEUE.LOCAL) +  
qmanager(SAMPLE.QMGR1)
```

You can verify the objects you created under the current context by using the **dis ctx** command. You should see the QueueConnectionFactory (ptyQcf) and the Queue (ptyQueue) objects as shown in Figure 8-13.

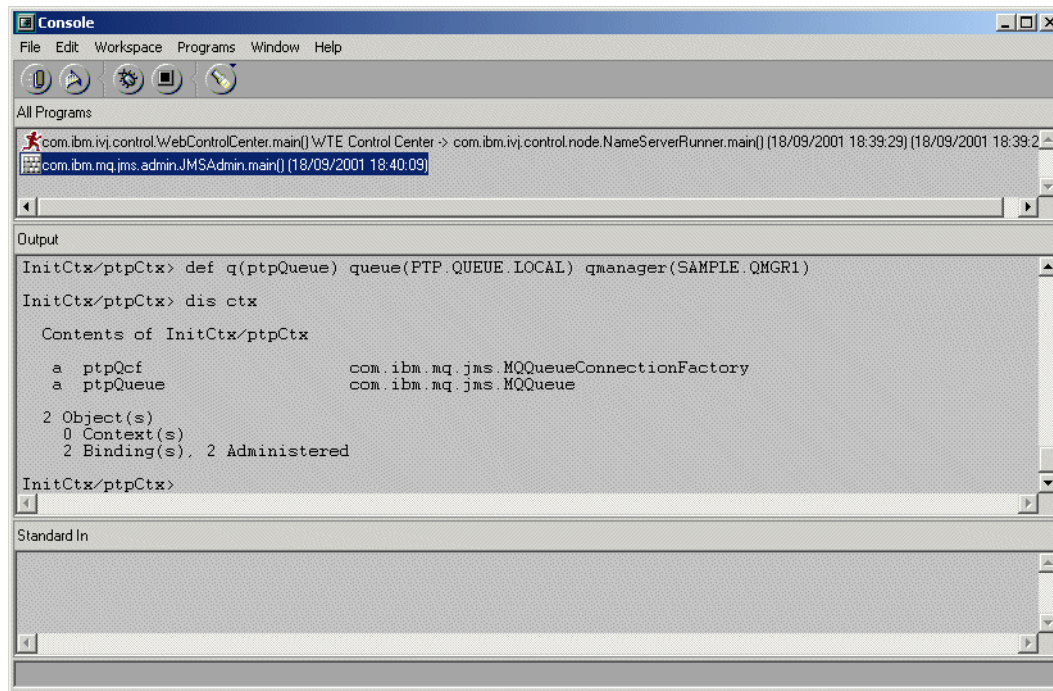


Figure 8-13 Display context

8.5 JMS application development

A JMS application uses either the point-to-point (PTP) or publish/subscribe style of messaging. Nothing prevents these styles from being combined in a single application; however, JMS focuses on applications that use one or the other. JMS defines these two styles because they represent the two dominant approaches to messaging currently in use. Since many messaging systems only support one of these styles, JMS provides a separate domain for each and defines compliance for each domain.

8.5.1 JMS point-to-point (PTP) model

Point-to-point messaging involves working with queues of messages. The sender sends messages to a specific queue to be consumed normally by a single receiver. In point-to-point communication, a message has at most one recipient. A sending client addresses the message to the queue that holds the messages for the intended (receiving) client. You can think of the queue as a mailbox. Many clients might send messages to the queue, but a message is

taken out by only one client. And, like a mailbox, messages remain in the queue until they are removed. Thus the availability of the recipient client does not affect the ability to deliver a message. In a point-to-point system, a client can be a sender (message producer), a receiver (message consumer), or both.

8.5.2 Programming approach in point-to-point messaging

Figure 8-14 shows the high-level steps involved in developing a point-to-point messaging program in JMS.

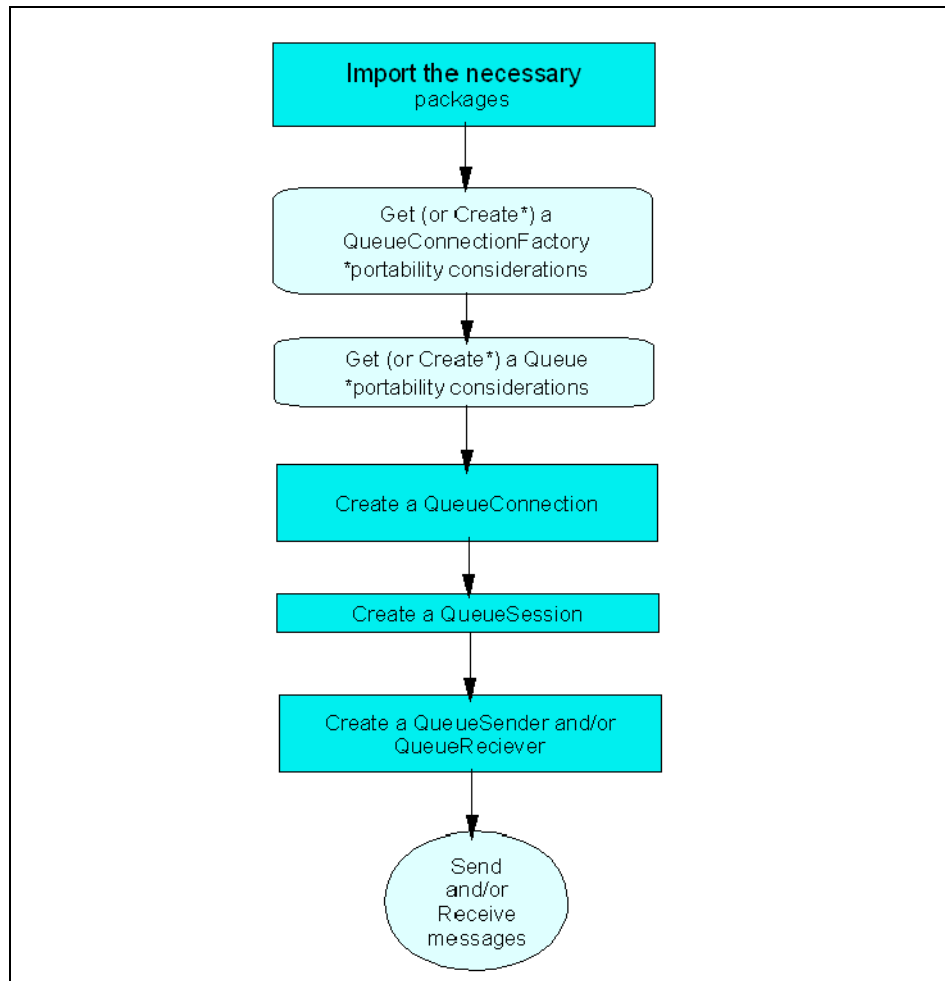


Figure 8-14 JMS PTP programming approach overview

The JMS interfaces in a point-to-point model are:

- ▶ QueueConnection
- ▶ QueueSession
- ▶ QueueSender
- ▶ QueueReceiver

When using JMS, connections are not made directly, but are built using a connection factory. These Factory objects can be stored in a JNDI namespace, thus hiding the vendor-specific implementation.

In point-to-point messaging, there are generally two messaging patterns as shown in Figure 8-15. The first approach is the send-and-forget model, and the second approach is the request/reply pattern.

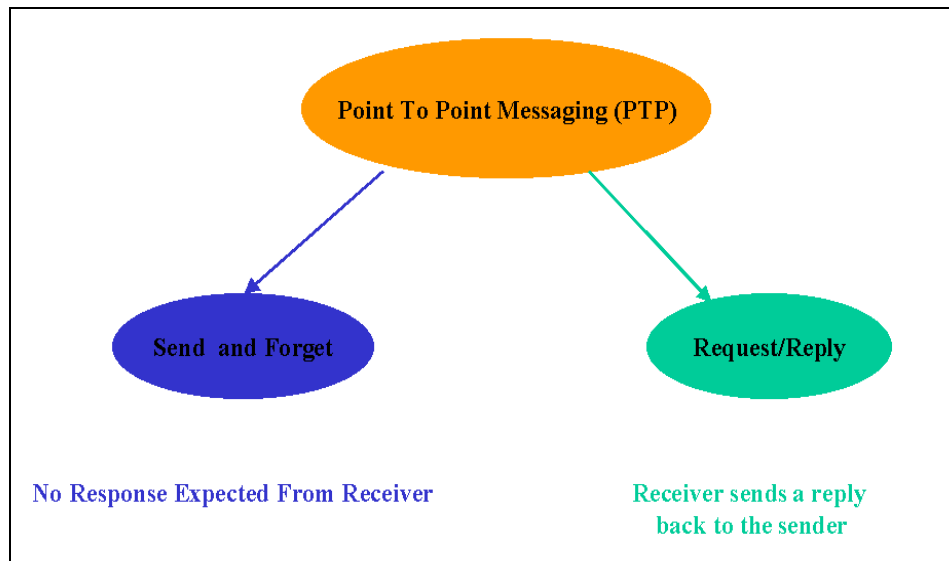


Figure 8-15 Point-to-point messaging patterns

8.5.3 Send-and-forget

With send-and-forget (or fire-and-forget) no response is expected from the receiver of the (datagram) messages.

Simple message producer application

Our first PTP client program creates a text message and sends it to an MQSeries queue. We also illustrate the receiver program that processes the application sent by the sender. The steps involved are:

- ▶ Look up the JNDI namespace for the QueueConnectionFactory and the Queue
- ▶ Get a Queue Connection object
- ▶ Create a QueueSession
- ▶ Create a QueueSender
- ▶ Create a TextMessage
- ▶ Send the message to the Queue
- ▶ Close and disconnect the connection objects

The program PtpSender.java illustrates the steps involved in developing a point-to-point message sending application. The program sends a simple text message to the MQSeries queue.

Example 8-1 PtpSender.java

Step 1 Import Necessary Packages

```
import java.util.*;
import javax.jms.*;
import javax.naming.directory.*;
import javax.naming.*;
public class PtpSender {
public static void main(String[] args) {
String          queueName = "ptpQueue";
String          qcfName = "ptpQcf" ;
Context         jndiContext = null;
QueueConnectionFactory queueConnectionFactory = null;
QueueConnection queueConnection = null;
QueueSession    queueSession = null;
Queue           queue = null;
QueueSender      queueSender = null;
TextMessage      message = null;
String providerUrl = "iiop://itsog:900/ptpCtx" ;
String initialContextFactory = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
```

/**

Step 2 set up an Initial Context for JNDI lookup

*/

```
try{
Hashtable env = new Hashtable() ;
env.put(Context.INITIAL_CONTEXT_FACTORY, initialContextFactory) ;
env.put(Context.PROVIDER_URL , providerUrl );
//env.put(Context.REFERRAL, "throw") ;
jndiContext = new InitialDirContext(env);
}
/**
```

Step 3 get a QueueConnectionFactory. We will retrieve the QueueConnectionFactory object named ptpQcf created in Persistent Name Server using JMSAdmin tool.

*/

```
queueConnectionFactory = (QueueConnectionFactory)jndiContext.lookup(qcfName);
```

```

/**
Step 4 the Queue object from the JNDI namespace.
*/
queue = (Queue)jndiContext.lookup(queueName);
/**
Step 5 Create a QueueConnection from the QueueConnectionFactory
*/
queueConnection = queueConnectionFactory.createQueueConnection();
/**
Step 6 Start the QueueConnection.
*/
queueConnection.start();
/**
Step 7 Create a QueueSession object from the QueueConnection
*/
queueSession = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
/**
Step 8 Create a QueueSender object for sending messages from the queue session.
*/
queueSender = queueSession.createSender(queue);
/**
Step 9 prepare a message object from the queuesession. we will create a
textMessage message object.
*/
message = queueSession.createTextMessage();
/**
Step 10 Set the message you want, to the message object.
*/
message.setText("This is a Test Message from PtpSender Class " );
/**
Step 11 Now we are ready to send the message.
*/
queueSender.send(message);
System.out.println("\n The Message has been sent");
/**
Step 12 Close the Queue Connection Before exiting from the program.
*/
queueConnection.close();

} catch (Exception e) {
e.printStackTrace();
}
}
}

```

Simple message consumer application

Our next client application is a message receiver application, which gets the message that was sent by the PtpSender application and prints out the message on the console. The steps involved are:

- ▶ Look up the JNDI namespace for the QueueConnectionFactory and the Queue.
- ▶ Get a Queue Connection object.
- ▶ Create a QueueSession.
- ▶ Create a QueueReceiver.
- ▶ Receive the message and display it.
- ▶ Close and disconnect the connection objects.

The program PtpReceiver.java illustrates the steps involved in creating a point-to-point message consumer application. The application gets the message from MQSeries queue and display the message.

Example 8-2 PtpReceiver.java

```
// Step 1 Import the Necessary Packages
import java.util.*;
import javax.jms.*;
import javax.naming.directory.*;
import javax.naming.*;
public class PtpReceiver {
/**
 *The Main Method.
 * @param no args
 */
public static void main(String[] args) {
String          queueName = "ptpQueue";
String          qcfName = "ptpQcf" ;
Context         jndiContext = null;
QueueConnectionFactory queueConnectionFactory = null;
QueueConnection queueConnection = null;
QueueSession    queueSession = null;
Queue           queue = null;
QueueReceiver    queueReceiver = null;
TextMessage      message = null;
/* Provider url
/For Persistent Name Server- iiop://iiopservername/contextname
/For LDAP Server use ldap://cn=ContextName,o=OrganizationalSuffix,c=coutrysuffix
eg. ldap://machineName/cn=ptpCtx,o=itso,c=uk
*/
String providerUrl = "iiop://itsog/ptpCtx" ;
String initialContextFactory = "com.ibm.ejs.ns.jndi.CNInitialContextFactory";
```

```

/**
 * Step 2 set up Initial Context for JNDI lookup
 */
try{
    Hashtable env = new Hashtable() ;
    env.put(Context.INITIAL_CONTEXT_FACTORY, initialContextFactory) ;
    env.put(Context.PROVIDER_URL , providerUrl );
    //env.put(Context.REFERRAL, "throw") ;
    jndiContext = new InitialDirContext(env);
}
/**
 * Step 3 get the QueueConnectionFactory from the JNDI Namespace
 */
queueConnectionFactory = (QueueConnectionFactory)jndiContext.lookup(qcfName);
/**
 * Step 4 get the Queue Object from the JNDI Name space
 */
queue = (Queue)jndiContext.lookup(queueName);
/**
 * Step 5 Create a QueueConnection using the QueueConnectionFactory
 */
queueConnection = queueConnectionFactory.createQueueConnection();
/*
 *Step 6 Connections are always created in stopped mode. You have to Explicitly
start them. Start the queueConnection
 */
queueConnection.start();
/**
 * Step 7 Create a queueSession object from the QueueConnection
 */
queueSession = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
/**
 * Step 8 Create a QueueReceiver from the queueSession
 */
queueReceiver = queueSession.createReceiver(queue);
/**
 * Step 9 Receive Messages. Here we are implementing a synchronous message
receiver. The receive call is in a loop so that it would process all the
available messages in the queue
 */
boolean eom = true;
    while (eom) {
        Message m = queueReceiver.receive(1);
        if (m != null) {
            if (m instanceof TextMessage) {
                message = (TextMessage) m;
                System.out.println("Reading message: " +
                    message.getText());
            }
        }
    }
else {

```

```

                                break;
                                }
                                }
                                else eom = false;
                                }

/**
 * Step 10 Close the connections
 */
queueConnection.close();
}
catch(Exception e){
e.printStackTrace();
}
}
}

```

8.5.4 Request/reply

With request/reply messaging, after the receiver receives a request message, it sends a reply back to the sender. On the sender's end, the messages are prepared just like that in PTP, but in addition the `JMSReplyToQueue` values are set to the Queue Name where the sender expects the reply from the receiver. When it gets the message, the receiver can use this reply-to-queue name to send the reply message back to the sender.

JMS provides the `JMSReplyTo` message header field for specifying the destination where a reply to a message should be sent. The `JMSCorrelationId` header field can be used in the reply message to provide reference to the request message.

Application design considerations in request/reply messaging

MQSeries is an asynchronous communications mechanism. In request/reply messaging, when a client puts a message to a queue and then expects a reply, care must be taken at the design level of the application. You do not want to wait indefinitely on the queue for the reply message to arrive, since you do not know how long it will take to get the reply for your request. The application design should cover delayed replies and no reply at all. You may also want to consider setting appropriate timeout values in waiting for the reply message.

If using timeouts in waiting for the reply message, you may want to consider using a proper expiration time value on the remote client side, which generates the reply message. In many request/reply situations, you may consider use of non-persistent messages for performance enhancement. Using non-persistent messages leads to significant performance improvements.

Non-persistent messages can be sent in three ways. The persistence property can be set in one of the following ways:

- ▶ Directly on the queue object within the queue manager while creating the MQSeries queue, or you can alter this on the existing MQSeries Queue.
- ▶ On the queue JMS Administered object in JNDI using JMSAdmin tool. The parameter PERSISTENCE can be used to specify the value of the persistence you want to set. The different values are:
 - APP -- Persistence defined by application (This is the default)
 - QDEF -- Persistence as defined on the MQSeries queue
 - PERS -- Messages are persistent
 - NON -- Messages are non-persistent

If you want to define a queue object name ptpQueue with persistence NON, using JMSAdmin you can specify the command:

```
DEFINE Q(ptpQueue) PERSISTENCE(NON)
```

- ▶ On a per-message basis within the JMS application using the `DeliveryMode.NON_PERSISTENT` field. Message persistence can be modified on the `QueueSender`, or specified on the call to the send method, but not directly on the message, as for example:

```
QueueSender sender = queueSession.createSender(queue) ;  
sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT) ;
```

Or you can set the delivery mode while sending the message by specifying delivery mode, priority, and time to live with the method call.

8.5.5 JMS publish/subscribe model

In contrast to the point-to-point model of communication, the publish/subscribe model enables the delivery of a message to multiple recipients. A sending client addresses, or publishes, the message to a topic to which multiple clients can be subscribed. There can be multiple publishers, as well as subscribers, to a topic. A durable subscription, or interest, exists across client shutdowns and restarts. While a client is down, all objects that will have been delivered to the topic are stored and then sent to the client when it renews the subscription. In a publish/subscribe system, a client can be a publisher (message producer), a subscriber (message consumer), or both.

The JMS publish/subscribe model defines how JMS clients publish messages to, and subscribe to messages from, a well-known node in a content-based hierarchy. JMS refers to these nodes as “topics”.

In this section, the terms *publish* and *subscribe* are used in place of the more generic terms *produce* and *consume* used previously. A topic can be thought of as a mini message broker that gathers and distributes messages addressed to it. By relying on the topic as an intermediary, message publishers are kept independent of subscribers and vice versa. The topic automatically adapts as both publishers and subscribers come and go. Publishers and subscribers are active when the Java objects that represent them exist.

JMS also supports the optional durability of subscribers and “remembers” the existence of them while they are inactive. MQSeries publish/subscribe removes the need for your application to know anything about the target application. All it has to do is send information it wants to share to a standard destination managed by MQSeries publish/subscribe, and let MQSeries publish/subscribe deal with the distribution. Similarly, the target application does not have to know anything about the source of the information it receives.

Figure 8-16 illustrates the steps involved in developing a publish/subscribe application with JMS.

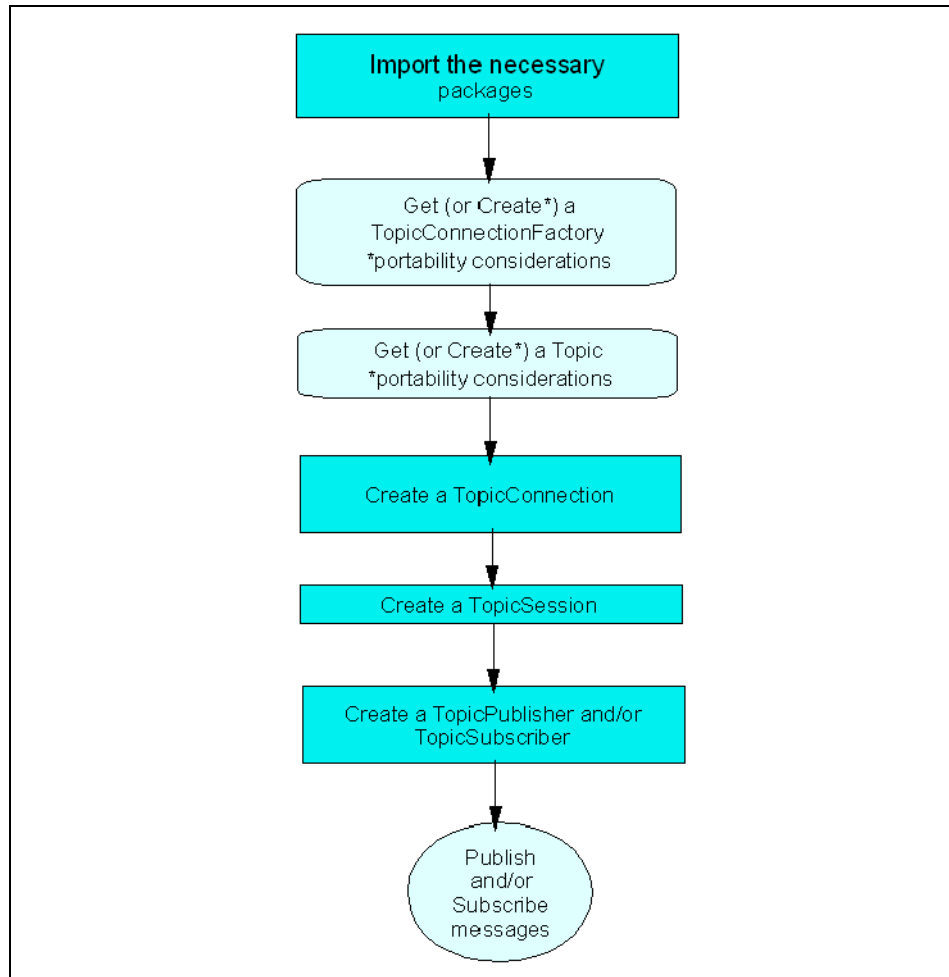


Figure 8-16 JMS publish/subscribe programming overview

In a JMS implementation of the publish/subscribe messaging model, all the vendor-specific implementations can be referenced through the following interfaces in `javax.jms`. All of these are encapsulated in the implementation of the following interfaces:

- ▶ `QueueConnectionFactory`
- ▶ `TopicConnectionFactory`
- ▶ `Queue`
- ▶ `Topic`

In the JMS publish/subscribe model, an asynchronous subscription of topics is made possible when subscribing to a topic.

Simple JMS publish/subscribe application

In this section, we illustrate the steps involved in developing a publishing application with the sample program `JMSPublisher.java`. The steps involved in writing a JMS publish application are:

1. Define the JMS administered objects:

- a. Create a JNDI context.

The following command was used to create the JNDI context named `psCtx` used in our sample programs:

```
DEF CTX(psCtx)
```

- b. Change to the context you just created, using the following command:

```
CHG CTX(psCtx)
```

- c. Create a `TopicConnectionFactory`.

The following command creates a `TopicConnectionFactory` named `psTcf` referring to the `QueueManager` `ITSOG.QMGR1` on host named `ITSOG` listening on the default port 1414. The server connection channel used for client connection is `JMS.SRV.CHNL`:

```
DEF TCF(psTcf) TRANSPORT(CLIENT) QMANAGER(ITSOG.QMGR1) HOST(ITSOG)  
PORT(1414) CHANNEL(JMS.SRV.CHNL) BROKERQMGR(ITSOG.QMGR1)  
BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)  
BROKERPUBQ(SYSTEM.BROKER.DEFAULT.STREAM)  
BROKERSUBQ(SYSTEM.JMS.ND.SUBSCRIBER.QUEUE)  
BROKERCCSUBQ(SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE)
```

- d. Define the JNDI topic.

The following command creates a `Topic` named `psTopic` for the topic named `SampleTopic` under the root topic of `JmsTest`:

```
DEF T(psTopic) TOPIC(JMSTest/SampleTopic)
```

2. Look up the JNDI namespace for the `TopicConnectionFactory` and the `Topic`.
3. Create a `Topic Connection`.
4. Create a `TopicSession`.
5. Create a `TopicPublisher`.
6. Create a `TextMessage`.
7. Publish the message to the `Topic`.
8. Close and disconnect the connection objects.

Our first JMS publish/subscribe client program `JMSPublisher.java` creates a text message and publishes it to a topic “SampleTopic”, which is under the root topic of `JMSTest`. We also illustrate the subscriber program that subscribes to the topic “TestTopic”.

Example 8-3 JMSPublisher.java

```
//Step 1 Import the necessary packages
import java.util.*;
import javax.jms.*;
import javax.naming.directory.*;
import javax.naming.*;

public class JMSPublisher {
    /**
     *The main method
     *@param no args
     */
    public static void main(String[] args) {

        String          topicName = "cn=psTopic";
        String          tcfName = "cn=psTcf" ;
        Context          jndiContext = null;
        TopicConnectionFactory topicConnectionFactory = null;
        TopicConnection   topicConnection = null;
        TopicSession      topicSession = null;
        Topic             topic = null;
        TopicPublisher     publisher = null;
        TextMessage        message = null;

        String providerUrl = "ldap://itsog/cn=psCtx,o=itsog,c=uk" ;
        String initialContextFactory = "com.sun.jndi.ldap.LdapCtxFactory";
        //Step 2 Set up an Initial context for JNDI lookUp.
        try {
            Hashtable env = new Hashtable() ;
            env.put(Context.INITIAL_CONTEXT_FACTORY, initialContextFactory) ;
            env.put(Context.PROVIDER_URL , providerUrl );
            jndiContext = new InitialDirContext(env);
        }
        //Step 3 Obtain a TopicConnection factory
        topicConnectionFactory =
            (TopicConnectionFactory)jndiContext.lookup(tcfName);
        // Step 4 Create a Topic Connection using the connection factory object
        topicConnection = topicConnectionFactory.createTopicConnection();
        //Step 5 Start the topic connection.
        topicConnection.start();
        //Step 6 Obtain a Topic from the JNDI
        topic = (Topic)jndiContext.lookup(topicName);
        // Step 7 Create a Topic Session from the topic connection
```

```

        topicSession = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
//Step 8 Create a topic publisher for the topic from the session.
        publisher = topicSession.createPublisher(topic);
//Step 9 Create a message object
        message = topicSession.createTextMessage();
//Step 10 prepare the body of the message
        message.setText("This is a Test Message from JMSPublisher Class " );
//Step 11 Publish the message.
        publisher.publish(message);
//Step 12 Close the connections.
        publisher.close();
        topicSession.close();
        topicConnection.close();
    }
    catch(Exception e ) {
        e.printStackTrace();
    }
}
}
}

```

Simple JMS subscriber application

In this section, we illustrate the steps involved in developing a JMS subscriber application by creating a sample application.

The steps involved in writing a JMS subscriber application are:

1. Look up the JNDI namespace for the TopicConnectionFactory and the Topic
2. Create a Topic Connection
3. Create a TopicSession
4. Create a TopicSubscriber
5. Receive subscription from the Topic
6. Close and disconnect the connection objects.

The program JMSSubscriber.java is a simple subscriber application that subscribes to messages from the topic “SampleTopic” which is under the root topic of JMSTest. We used a non-durable subscription in the sample.

Example 8-4 JMSSubscriber.java

```

//Step 1 Import the necessary packages.
import java.util.*;
import javax.jms.*;
import javax.naming.directory.*;
import javax.naming.*;
public class JMSSubscriber {
/**
 * The main method

```

```

* @param no args
*/
public static void main(String[] args) {
    String          topicName = "cn=psTopic";
    String          tcfName = "cn=psTcf" ;
    Context          jndiContext = null;
    TopicConnectionFactory topicConnectionFactory = null;
    TopicConnection  topicConnection = null;
    TopicSession     topicSession = null;
    Topic            topic = null;
    TopicSubscriber   subscriber = null;
    TextMessage       message = null;
    String providerUrl = "ldap://itsog/cn=psCtx,o=itsog,c=uk" ;
    String initialContextFactory = "com.sun.jndi.ldap.LdapCtxFactory";

    //Step 2 Set up Initial Context for JNDI lookup
    try{
        Hashtable env = new Hashtable() ;
        env.put(Context.INITIAL_CONTEXT_FACTORY, initialContextFactory) ;
        env.put(Context.PROVIDER_URL , providerUrl );
        env.put(Context.REFERRAL, "throw") ;
        jndiContext = new InitialDirContext(env);
        //Step 3 Get the TopicConnectionFactory from the JNDI Namespace
        topicConnectionFactory = (TopicConnectionFactory)jndiContext.lookup(tcfName);
        //Step 4 Create a TopicConnection
        topicConnection = topicConnectionFactory.createTopicConnection();
        //Step 5 Start The topic connection
        topicConnection.start();
        //Step 6 Create a topic session from the topic connection
        topicSession = topicConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
        //Step 7 Obtain a Topic from the JNDI namespace
        topic = (Topic)jndiContext.lookup(topicName);
        //Step 8 Create a topic subscriber for the topic.
        subscriber = topicSession.createSubscriber(topic);//Non durable subscriber
        // Step 9 Receive Subscription
        message = (TextMessage)subscriber.receive();
        System.out.println("\n *** The Message is " + message.getText());
        //Step 10Close the connection and other open resources
        subscriber.close();
        topicSession.close();
        topicConnection.close();
    }
    catch(Exception e ) {
        e.printStackTrace();
    }
}

```

8.6 Asynchronous processing

MQSeries JMS provides the much-awaited `MessageListener` interface. Using the message listener facility, a client can register a listener object with a message consumer. When a message arrives for the consumer, it is delivered by to the client by calling its `onMessage` method.

This is an alternative to using the other approaches such as triggers, making receive or subscribe calls with the wait or polling mechanism in the application code to check for new messages or subscriptions. When using asynchronous delivery mode with the listener, the entire session associated with the message consumer is marked asynchronous. The same session cannot be used for making any explicit receive calls.

In asynchronous message delivery, application code may not be able to catch exceptions raised by failures to receive messages, since the application does not make explicit receive calls. MQSeries JMS provides a facility to get those exceptions through `ExceptionListener` interface. The `onException` method is called when exceptions occur and the `JMSEException` is passed to the method as its parameter.

8.6.1 Message listeners

A message listener is created by implementing the `MessageListener` interface and providing application-specific processing in the `onMessage` method of the listener. We discuss how we can implement a simple message listener in a message consumer application.

We first create a listener class (`PtpListener.java`), which extends the `JMSMessageListener` class and implements the `onMessage` method. When a message arrives, the message is delivered to the `onMessage` method as its argument. We just display the message whenever a message arrives.

Example 8-5 PtpListener.java

```
import java.io.*;

import javax.jms.*;

public class PtpListener implements MessageListener {
    /**
     * onMessage.
     */
    public void onMessage( Message message) {
        try {

            if (message instanceof TextMessage) {
                System.out.println( ((TextMessage)message).getText());
            }
        }
    }
}
```

```

    }

    } catch (JMSEException e) {
        e.printStackTrace();
    }
}
}

```

In `PtpAsyncConsumer.java`, we illustrate how a message consumer would register a listener, thus using asynchronous processing of the message. The class `PtpAsyncConsumer.java` will use the listener implementation in `PtpListener.java` to process messages asynchronously. You can run the `PtpAsyncConsumer` along with the `PtpSender` application we illustrated in the simple sender application.

Example 8-6 PtpAsyncConsumer.java

```

import java.util.*;
import javax.jms.*;
import javax.naming.directory.*;
import javax.naming.*;

public class PtpAsyncConsumer{
    String          queueName = "cn=ptpQueue";
    String          qcfName = "cn=ptpQcf" ;
    Context          jndiContext = null;
    QueueConnectionFactory queueConnectionFactory = null;
    QueueConnection    queueConnection = null;
    QueueSession       queueSession = null;
    Queue             queue = null;
    QueueReceiver       queueReceiver = null;

    String providerUrl = "ldap://itsog/cn=ptpCtx,o=itsog,c=uk" ;
    String initialContextFactory = "com.sun.jndi.ldap.LdapCtxFactory";

    public static void main(java.lang.String[] args) {
        try {

            PtpAsyncConsumer asyncConsumer = new PtpAsyncConsumer() ;
            asyncConsumer.performTask();
        } catch (Exception e ){
            e.printStackTrace();
        }
    }

    /**
     * Method performTask Control the flow of control of the logical operations

```

```

*/
public synchronized void performTask() throws Exception {
    System.out.println("\n Setting Up Initial JNDI Context ");
    Hashtable env = new Hashtable() ;

    env.put(Context.INITIAL_CONTEXT_FACTORY, initialContextFactory) ;
    env.put(Context.PROVIDER_URL , providerUrl );
    env.put(Context.REFERRAL, "throw") ;
    jndiContext = new InitialDirContext(env);

    System.out.println("\n  Get QueueConnectionFactory  ");

    queueConnectionFactory =
    (QueueConnectionFactory)jndiContext.lookup(qcfName);

    System.out.println("\n  Get Queue  ");
    queue = (Queue)jndiContext.lookup(queueName);

    System.out.println("\n Create Queue Connections  ");
    queueConnection = queueConnectionFactory.createQueueConnection();

    System.out.println("\n  Start Queue Connection  ");
    queueConnection.start();

    System.out.println("\n  Create Queue Session  ");
    queueSession = queueConnection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);

    System.out.println("\n Create Queue Receiver  ");
    queueReceiver = queueSession.createReceiver(queue);

    System.out.println("\n Register the Listener");
    PtpListener yahoo = new PtpListener();
    queueReceiver.setMessageListener(yahoo);
    // Wait for new messages.
    wait();
}
}

```

8.6.2 Exception listeners

When using the asynchronous message delivery model using message listeners, the application code cannot catch exceptions raised by failures to receive messages. This is because the application code does not make explicit calls to `receive()` or `subscribe()` methods. To catch errors in asynchronous processing,

you can register an `ExceptionListener`, which is an instance of a class that implements the `onException()` method. An exception listener allows a client to be notified of a problem asynchronously. If an exception listener is registered, when errors occur the `onMessage()` method is called, with the `JMSEException` object passed to the method as its argument. The exception listeners are set on the connection.

We illustrate how to implement an exception listener with a very simple application:

Example 8-7 JMSEExceptionListener.java

```
import java.io.*;

import javax.jms.*;
public class JMSEExceptionListener implements ExceptionListener {
    /**
     * onException. We will just print the exception and exit from the program
     * execution. Add suitable error handling and recovery logic depending on you
     * use..
     */
    public void onException( JMSEException e) {

        e.printStackTrace();
        System.exit(1);

    }
}
```

Now that we have the exception listener, we can use the listener in an asynchronous consumer by instantiating the `JMSEExceptionListener` and registering the exception listener with the connection object using the `setExceptionListener` method:

```
JMSEExceptionListener exListener = new JMSEExceptionListener( );
queueConnection.setExceptionListener(exListener);
```

The `setExceptionListener` method sets the exception listener for the connection object `queueConnection`.

8.7 Message selectors

JMS provides facilities to query the messages on a queue so that a subset of the messages can be selected based on a given criteria. This can be thought of as a SQL query facility in databases. In fact the syntax for such a search is based on SQL92 standards for conditional expressions. A message selector is a string containing a conditional expression. The message selector can refer to fields in the JMS message header as well as fields in the message properties that are application-defined fields.

The order of evaluation of a message selector is from left to right within precedence level. You can use parentheses to change the order of evaluation. The selector literals and operator names are case sensitive.

A selector may contain:

- ▶ Literals:
 - A string literal is enclosed in single quotes with an included single quote represented by a doubled single quote such as 'literal' and 'literal's. Like Java String literals, these use the unicode character encoding.
 - An exact numeric literal is a numeric value without a decimal point, such as 57, -957, +62. Numbers in the range of Java *long* are supported. Exact numeric literals use the Java integer literal syntax.
 - An approximate numeric literal is a numeric value in scientific notation such as 7E3, -57.9E2 or a numeric value with a decimal such as 7., -95.7, +6.2. Numbers in the range of Java *double* are supported. Approximate literals use the Java floating point literal syntax.
 - The boolean literals TRUE and FALSE.
- ▶ Identifiers:
 - An identifier is an unlimited-length character sequence that must begin with a Java identifier start character and all following characters must be Java identifier part characters. An identifier start character is any character for which the method `Character.isJavaIdentifierStart` returns true. This includes '_' and '\$'. An identifier part character is any character for which the method `Character.isJavaIdentifierPart` returns true.
 - Identifiers cannot be the names NULL, TRUE, or FALSE.
 - Identifiers cannot be NOT, AND, OR, BETWEEN, LIKE, IN, and IS.
 - Identifiers are either header field references or property references.
 - Identifiers are case sensitive.
 - Message header field references are restricted to `JMSDeliveryMode`, `JMSPriority`, `JMSMessageID`, `JMSTimestamp`, `JMSCorrelationID`, and

JMSType. JMSMessageID, JMSCorrelationID, and JMSType values may be null and if so are treated as a NULL value.

- Any name beginning with 'JMSX' is a JMS defined property name.
- Any name beginning with 'JMS_' is a provider-specific property name.
- Any name that does not begin with 'JMS' is an application-specific property name. If a property is referenced that does not exist in a message, its value is NULL. If it does exist, its value is the corresponding property value.
- ▶ Whitespace is the same as that defined for Java: space, horizontal tab, form feed and line terminator.
- ▶ Expressions:
 - A selector is a conditional expression. A selector that evaluates to true matches. A selector that evaluates to false or unknown does not match.
 - Arithmetic expressions are composed of themselves, arithmetic operations, identifiers with numeric values, and numeric literals.
 - Conditional expressions are composed of themselves, comparison operations, logical operations, identifiers with boolean values, and boolean literals.
 - Standard bracketing () for ordering expression evaluation is supported.
- ▶ Logical operators in precedence order: NOT, AND, OR
- ▶ Comparison operators: =, >, >=, <, <=, <> (not equal)
 - Only like type values can be compared. One exception is that it is valid to compare exact numeric values and approximate numeric values (the type conversion required is defined by the rules of Java numeric promotion). If the comparison of non-like type values is attempted, the selector is always false.
 - String and Boolean comparison is restricted to = and <>. Two strings are equal if and only if they contain the same sequence of characters.
- ▶ Arithmetic operators in precedence order:
 - +, - unary
 - *, / multiplication and division
 - +, - addition and subtraction
 - Arithmetic operations must use Java numeric promotion.
- ▶ Arithmetic-expr1 [NOT] BETWEEN arithmetic-expr2 and arithmetic-expr3 comparison operator
 - Age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19

- Age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19
- ▶ identifier [NOT] IN (string-literal1, string-literal2,...) comparison operator where identifier has a string or NULL value.
 - Country IN ('UK', 'US', 'France') is true for 'UK' and false for 'Peru'. It is equivalent to the expression (Country = 'UK') OR (Country = 'US') OR (Country = 'France').
 - Country NOT IN ('UK', 'US', 'France') is false for 'UK' and true for 'Peru'. It is equivalent to the expression NOT ((Country = 'UK') OR (Country = 'US') OR (Country = 'France')).
 - If the identifier of an IN or NOT IN operation is NULL, the value of the operation is unknown.
 - Identifier [NOT] LIKE pattern-value [ESCAPE escape-character] comparison operator, where the identifier has a string value. Pattern-value is a string literal where '_' stands for any single character. '%' stands for any sequence of characters (including the empty sequence). All other characters stand for themselves. The optional escape character is a single character string literal whose character is used to escape the special meaning of the '_' and '%' in pattern-value.
 - Phone LIKE '12%3' is true for '123' '12993' and false for '1234'
 - Word LIKE 'l_se' is true for 'lose' and false for 'loose'
 - Underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'
 - Phone NOT LIKE '12%3' is false for '123' and '12993' and true for '1234'
 - If identifier of a LIKE or NOT LIKE operation is NULL the value of the operation is unknown.
- ▶ Identifier IS NULL comparison operator tests for a null header field value, or a missing property value.
 - prop_name IS NULL
- ▶ Identifier IS NOT NULL comparison operator tests for the existence of a non-null header field value or property value.
 - prop_name IS NOT NULL

JMS providers are required to verify the syntactic correctness of a message selector at the time it is presented. A method providing a syntactically incorrect selector must result in a JMS InvalidSelectorException.

The following message selector selects messages with a message type of *car* and color of *blue* and weight greater than 2500 lbs:

"JMSType = 'car' AND color = 'blue' AND weight > 2500"

Null values: Header fields and Property values may be NULL. The rules of evaluation of the selector expression when NULL values are SQL treats NULL values as Unknown. Hence any comparisons or arithmetic operations involving NULL value would result in Unknown value.

8.7.1 Working with message selectors

Message selectors can be set as a user-defined property on the message. On the sending or publishing side, the set property method that takes a name value pair can be used to set the property name and its value.

To set a property named 'testProperty' with a value of 100 and data type being integer, the property can be set on the object message by:

```
message.setIntProperty("testProperty", 100) ;
```

The set property method takes two arguments. The first argument is type String and is the name of the property value. The second argument is the value of the property. Use the appropriate set property method (setIntProperty for integer values, setStringProperty for values of String type, etc.) depending on the data type of the value you want to set.

On the message consumer side, a message selector string with the appropriate selection criteria is used. The selection criteria is specified at the time of creating the message consumer.

We use a selection criteria of selecting messaging with property name 'testProperty' and value being 100. It can be done by:

```
String selector = "testProperty = 100 " ;  
queueReceiver = session.createReceiver(queueName, selector).
```

The message consumer created with the selector gets a message with a property named "testProperty" and with the value being 100.

JMS specification does not permit the selector associated with a message consumer to be changed once it is created. If you need message receivers with different criteria, you may need to create separate message receivers.

Once a message consumer is created with a message selector, you can check the selector value by using the method getMessageSelector() on the message consumer.

In the queueReceiver we used in the illustration above, the getMessageSelector method returns a string with the selector value queueReceiver.getMessageSelector(). This will return testProperty = 100.



A

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246506>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246506.

Using the Web material

The additional Web material that accompanies this redbook includes the following files:

<i>File name</i>	<i>Description</i>
SG246506.zip	Zipped Code Samples

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	20 MB minimum
Operating System:	Windows NT or Windows 2000
Processor:	300 MHz or higher
Memory:	256 MB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 296.

- ▶ *MQSeries Publish/Subscribe Applications*, SG24-6282
- ▶ *MQSeries Primer*, a redpaper found at <http://www.ibm.com/redbooks>

Other resources

These publications are also relevant as further information sources:

- ▶ *An Introduction to Messaging and Queuing*, GC33-0805
- ▶ *MQSeries Application Programming Reference*, SC33-1673
- ▶ *MQSeries Application Programming Guide*, SC33-0807
- ▶ *MQSeries Application Programming Reference Summary*, SX33-6095
- ▶ *MQSeries Clients*, GC33-1632
- ▶ *MQSeries Application Message Interface Reference*, SC34-5604
- ▶ *MQSeries Using C++*, SC33-1877
- ▶ *MQSeries Using the Component Object Model Interface*, SC34-5387
- ▶ *MQSeries Using Java*, SC34-5456

Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ MQSeries manuals
<http://www-3.ibm.com/software/ts/mqseries/library/manuals/>
- ▶ XA Specification
<http://www.opengroup.org>

- MQSeries SupportPacs

<http://www.ibm.com/software/ts/mqseries/txppacs/>

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also 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.

IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.

Abbreviations and acronyms

AMI	Application Message Interface
API	Application Programming Interface
COM	Component Object Model
DCOM	Distributed Component Object Model
DLQ	Dead Letter Queue
ESQL	Extended Structured Query Language
FIFO	First In First Out
IBM	International Business Machines Corporation
ITSO	International Technical Support Organization
JDK	Java Development Kit
JMS	Java Message Service
JNDI	Java Naming and Directory Interface Interface
JRE	Java Runtime Environment
JVM	Java virtual machine
MQI	Message Queue Interface
MTS	Microsoft Transaction Server
PTP	Point-to-point
SQL	Structured Query Language
XML	Extensible Markup Language

Index

A

ACID 13
acknowledge 251
ActiveX 19, 181
ActiveX/COM 4
addTopic() 111, 112, 113
administered objects 246
administration role 93
administration tool 92
AIX 25, 26, 94, 96, 97, 98, 134, 135, 204, 205, 206, 252
alias queue 36, 187
AMI 18
AMQZSTUB 26
amtc.h 96
amtcpp.hpp 96
anonymous registration 113
API 4, 18
applets 20, 203
Application Message Interface 90
applications 4, 13
array 38
AS/400 25, 26, 41, 94, 97, 134
asynchronous 239, 287
AT&T GIS UNIX 25
Atomicity 13
attributes 100

B

backout 24
BackoutCount 16
BackoutThreshold 16
Balance 12
begin() 116
bidirectional 15
binding mode 24, 207, 211, 260
bindings 90
boolean 241
bootstrap port 256
browse cursor 148
browsing messages 23
buffer 45, 142, 150
business logic 90

BytesMessage 247

C

C 4, 38
C ANSI 59
C++ 4, 19
CCSID 228
character attributes 51
CICS 16, 41, 134
CICS Bridge 19
client channel definition 134
client connection 208
client connection mode 207
close() 114, 123
closing queue objects 23
COBOL 4, 25, 38
COM 182, 197
COM+ 182, 197
commit 14, 16, 24, 115, 144, 151
commit phase 15
commit() 116
compilers 98
completion code 32
Component Object Model 182
connecting to queue manager 23
connection 239
connection factory 239
connection handle 38
connection mode 207
Consistency 14
constants 24
constructor call 210
consumer 252
consuming messages 4
content 4
context 40, 268
control 24
coordinating 14
copybook 97, 98
correlation ID 8
CorrelId 48
CRAM-MD5 255
createDistributionList() 106

createMessage() 107
createPolicy() 102, 108
createPublisher 243
createPublisher() 104
createQueueConnection() 241
createReceiver() 104, 245
createSender 243
createSender() 103
createSession() 101, 119
createSubscriber() 105
createTopicSession() 241

D

data integrity 14
datagram 9, 12
DataLength 51
DB2 14
DC/OSx 25
DCOM 182
dead-letter header 143
dead-letter queue 143
development role 93
Digital OpenVMS 25
disconnecting from queue manager 23
Distributed COM 182
distributed transaction 151
distribution list 37, 45, 90, 92, 99, 106
DNS 240
Durability 14
dynamic queue 36, 37, 140

E

Encina 16, 27
environment variables 25
environments 24
errors 4
exception listener 288
exceptions 4
expire 4
extension 133

F

FIFO 32
FIFO within Priority 32
free-threading 183

G

generic 38
getBytes() 109, 120
getMessageSelector() 292
getting messages 23
global 16
global units of work 55
group 231
Groupid 48
grouping 56

H

handle 32, 38, 40
HCONN 245
hierarchy 7
HOBj 245
HP-UX 25, 27, 94, 96, 97, 98, 134, 135, 204, 205, 206, 252

I

IIOP 256
ILE C/400 26
import 124
ImqBoolean 145
ImqCache 143
ImqDeadLetterHeader 143
ImqGetMessageOptions 147
ImqIMSBridgeHeader 146
ImqItem 146
ImqMessage 143, 146
ImqMessageTracker 147
ImqObject 149
ImqPutMessageOptions 144
ImqPutMessages 151
ImqQueue 145
ImqQueueManager 144, 151
ImqReferenceHeader 146
IMS 16, 134
IMS Bridge 19
indexing 48
in-doubt 15
infinite loop 16
inquire 51
inquiring about object attributes 23
integer attributes 51
interpret 4
iSeries 204, 205, 206
Isolation 14

J

Java 4, 19, 204
Java Development Kit 205
Java Messaging Service 235
Java native method 123
Java virtual machine 118
JavaScript 183, 197
javax.jms 246
JDBC 20, 236
JDK 205
JMS xi, 20, 235
JMS client 237
JMSAdmin 254, 255, 264
JMSWrapXAQueueConnectionFactory 254
JMSWrapXATopicConnectionFactory 254
JNDI 240, 246, 254, 278
Just-in-time 19
JVM 121

L

LDAP 240, 255
libraries 96
Linux 204, 205, 206, 252
Local queues 35
local units of work 55
Logical message 227
logical order 17, 32, 46, 49
lookup() 240

M

MAOC 94
MAOF 92, 94, 101
MA1G 205
MA88 204, 205, 252
many-to-one 3
MapMessage 247
mapping 92, 248
memory 122
message broker 8, 92
message grouping 16, 23, 58
message object 109, 127, 189
Message Queue Interface xi
message selector 244
messaging 4
method 140, 186
Micro Focus 26
Microsoft Internet Explorer 183
Microsoft Transaction Server 196

model queue 37
MQAX200 200
MQBACK 28, 55
MQBEGIN 28, 55
MQBO 28
MQBYTE 30
MQBYTEn 30
MQC 214
MQCD 138
MQChannelDefinition 208
MQCHAR 30
MQCHARn 30
MQCLOSE 28, 30
MQCMIT 28, 55
MQCNO 28
MQCONN 28, 30
MQCONNx 28
MQDH 28
MQDISC 28
MQDistributionList 188, 208
MQDistributionListItem 188, 208
MQDistributionListObject 191
MQDLH 29
MQEnvironment 208, 210
MQException 209
MQGET 28
MQGetMessageOptions 192, 209
MQGMO 29, 55
MQHCONN 30
MQHOBJ 30
MQI 18
MQINQ 28, 51
mqjbnd02.dll 260
MQLONG 30
MQManagedObject 209
MQMD 29, 45, 141, 192, 213, 214, 247, 248
MQMDE 29
MQMessage 209
MQMessage class 197
MQMessageTracker 209
MQOD 29, 38
MQOO 214
MQOPEN 28, 30, 43
MQOR 29, 38
MQPMO 29, 41, 55, 214
MQPMR 29
MQPoolServices 209
MQPoolServicesEvent 209
MQPoolToken 209

- MQProcess 209
- MQPUT 28, 43
- MQPUT1 28, 43
- MQPutMessageOptions 190, 194, 209
- MQueue 188, 210, 254
- MQueueConnectionFactory 254
- MQueueManager 186, 193, 195, 210
- MReceiveExit 210
- MRFH 104, 172
- MRFH2 247, 248
- MRRM 29
- MRR 38
- MQSecurityExit 210
- MSEndExit 210
- MSeries xi, 4
- MSeries classes 208
- MSession 185, 189
- MSET 28
- MSimpleConnectionFactory 210
- MTM 29
- MTMC 29
- MTMC2 29
- MTopic 254
- MTopicConnectionFactory 254
- MQXAQueueConnectionFactory 254
- MQXATopicConnectionFactory 254
- MQXP 30
- MQXQH 30
- MsgId 48
- MsgSeqNumber 48
- MTS 184
- multiple segments 17
- multi-threaded 241
- MVS/ESA 25, 30

N

- namespace 246
- NDS 240
- NIS 240

O

- ObjectMessage 247
- object-oriented 19, 134
- objects 18, 19, 24, 90
- Offset 48
- OLE 182
- one-to-many 3
- one-to-one 3, 4

- onException 285
- onMessage 252, 285
- oolicy 99
- Open Applications Group 115
- Open Applications Middleware 115
- Open Group 15
- open() 108
- open/close calls 31
- opening queue objects 23
- operations 13, 23
- ordering 227
- OS/2 18, 135
- OS/2 Warp 25, 30, 134
- OS/390 98, 204, 205, 206
- OS/400 98, 204, 205, 206

P

- patterns xi, 4, 23, 129, 310
- permanent dynamic queue 40
- persistence 74, 278
- physical order 32, 46, 49
- PL/I 25
- PMQLONG 30
- pointer 33
- point-to-point 4
- policies 92
- policy 91, 100, 103
- policy object 109, 127
- port 208
- predefined queue 13
- prepare phase 15
- Priority 33
- procedural 19
- producer 239
- programming tools 9
- program-to-program 13
- pseudo-code 56
- publication data 74
- publish 8, 112, 126
- publish() 112
- publish/subscribe 3, 4, 7, 9, 71, 90, 233
- publisher 99, 104
- publisher classes 127
- PubSubCommand 83

Q

- QChannelExit 208
- queue 4

QueueConnection 239
QueueConnectionFactory 239
QueueReceiver 245
queuing 12

R

RDBMS 20, 236
readBytes() 111
reason code 32
receive 123
receive() 110, 113, 287
receiver classes 127
receivers 4
Redbooks Web site 296
 Contact us xiv
Reference header 146
register 76
relational database 14, 56
Remote queues 36
reply 5, 12
reply-to 37
report 12, 109
repository 92
request 12
request/reply 3, 4, 8, 221
resource manager 14
responder 225
response message 221
return code 32
rollback 15, 115, 151
rollback() 116
RPG/400 26

S

scope 34, 100
segmentation 17
selectors 51
send() 109
send-and-forget 3, 59
sender services 106
senders 4
sending messages 23
server connection 207
service 92
service point 92
servlets 20
session 239
session factory 101, 119

session object 100, 101, 106, 114, 127
set 53
setExceptionListener 288
setGroupStatus() 117
setting object attributes 23
setWaitTime() 123
single-phase 16
single-phase commit 14
SINIX 25
size 50
specification 32
SQL92 252
start() 239
stream 74, 249
StreamMessage 247
structure 24, 37, 38
style 4
subscribe() 112, 287
subscriber 99, 129
subscriber object 127
subscription 129
Sun Solaris 25, 94, 96, 97, 98, 134, 135, 204, 205, 206, 252
SunOS 25
SupportPacs xi
synchronize 14
syncpoint 144, 227

T

TAL 25
Tandem NonStop Kernel 25
TCP/IP 208, 238
temporary queue 9, 245
temporary topics 9
TextMessage 247
topic 7, 126
TopicConnection 240
TopicConnectionFactory 240
TopicSession 243
transaction 116
transaction management 23
transaction manager 14
transactional 115
transactions 13
transmission queue 36
trigger 252
Trigger message 19
Tuxedo 16, 117

two-phase 16
two-phase commit 14
types 24

U

UML 135
unit of work 14, 15, 16, 24, 115
UNIX 15, 18, 25, 30, 97, 205
unsubscribe() 113

V

VBScript 183, 197
VisiBroker 204, 205
Visual Basic 4, 25, 184
VisualAge 260

W

wait time 222
Windows 25, 97, 98, 204
Windows 2000 93, 94, 183, 184, 197, 205, 206
Windows 3.1 135
Windows 95 135, 183, 205, 206
Windows 98 183, 205, 206
Windows NT 4, 25, 26, 30, 93, 94, 134, 135, 183,
205, 206, 252
writeBytes() 109, 120

X

X/Open 15
XA 116, 195
XA Distributed Transaction Processing 15
XA specification 14
XAQueueConnection 239
XAQueueConnectionFactory 239
XATopicConnection 240
XATopicConnectionFactory 240
XML 92

Y

YP 240

Z

z/OS 204, 205, 206



MQSeries Programming Patterns

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



MQSeries Programming Patterns



Redbooks

**Install, tailor and
configure specialist
tools such as JMS
admin**

**Popular MQSeries
programming
choices discussed**

**Common
programming
pattern examples**

Today MQSeries offers the programmer more choices than ever in which to write new MQSeries applications, from the most traditional Message Queue Interface API all the way through to the popular and highly portable JMS interface.

Because of the many options available, it can sometimes be difficult for an application programmer new to MQSeries to easily understand the differences and benefits of each, or appreciate the implications of using one programming approach versus another.

This redbook will help you install, tailor and configure specialist tools such as JMS admin, and will help you to design/create MQSeries applications. It gives a broad and general understanding of the currently available MQSeries APIs.

We do this first by describing some of the more common examples and coding patterns, and then explaining each one in turn using the different APIs MQSeries supports to show the merits of each particular programming choice.

This redbook positions the different MQSeries programming choices against each other in such a way as to help the application writer to make a clearer and more informed judgment as to which is the most suitable programming method for a particular situation.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks