[MS-MQSO]:
Message Queuing System Overview

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation for protocols, file formats, languages, standards as well as overviews of the interaction among each of these technologies.
- **Copyrights.** This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you may make copies of it in order to develop implementations of the technologies described in the Open Specifications and may distribute portions of it in your implementations using these technologies or your documentation as necessary to properly document the implementation. You may also distribute in your implementation, with or without modification, any schema, IDL's, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications.
- **No Trade Secrets.** Microsoft does not claim any trade secret rights in this documentation.
- **Patents.** Microsoft has patents that may cover your implementations of the technologies described in the Open Specifications. Neither this notice nor Microsoft's delivery of the documentation grants any licenses under those or any other Microsoft patents. However, a given Open Specification may be covered by Microsoft Open Specification Promise or the Community Promise. If you would prefer a written license, or if the technologies described in the Open Specifications are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **Trademarks.** The names of companies and products contained in this documentation may be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights.
- **Fictitious Names.** The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications do not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments you are free to take advantage of them. Certain Open Specifications are intended for use in conjunction with publicly available standard specifications and network programming art, and assumes that the reader either is familiar with the aforementioned material or has immediate access to it.

This document provides an overview of the Message Queuing System Overview Protocol Family. It is intended for use in conjunction with the Microsoft Protocol Technical Documents, publicly available

Copyright © 2011 Microsoft Corporation.

standard specifications, network programming art, and Microsoft Windows distributed systems concepts. It assumes that the reader is either familiar with the aforementioned material or has immediate access to it.

A Protocol Family System Document does not require the use of Microsoft programming tools or programming environments in order to implement the Protocols in the System. Developers who have access to Microsoft programming tools and environments are free to take advantage of them.

Abstract

The Message Queuing (MSMQ) System is a communications service that enables reliable and secure asynchronous messaging between applications over a variety of deployment topologies. This document describes the intended functionality of the MSMQ System and how the system of protocols interacts with each other in order to achieve the desired messaging functionality. It provides examples of some of the common user scenarios. It does not restate the processing rules and other details that are specific for each protocol, because these are described in the respective individual protocol documents.

Revision Summary

Date	Revision History	Revision Class	Comments
06/20/2008	0.1	Major	Updated and revised the technical content.
07/25/2008	0.1.1	Editorial	Revised and edited the technical content.
08/29/2008	1.0	Major	Updated and revised the technical content.
10/24/2008	1.0.1	Editorial	Revised and edited the technical content.
12/05/2008	2.0	Major	Updated and revised the technical content.
01/16/2009	2.0.1	Editorial	Revised and edited the technical content.
02/27/2009	2.0.2	Editorial	Revised and edited the technical content.
04/10/2009	2.0.3	Editorial	Revised and edited the technical content.
05/22/2009	3.0	Major	Rewrote content in a new template.
07/02/2009	4.0	Major	Updated and revised the technical content.
08/14/2009	5.0	Major	Updated and revised the technical content.
09/25/2009	6.0	Major	Updated and revised the technical content.
11/06/2009	7.0	Major	Updated and revised the technical content.
12/18/2009	8.0	Major	Updated and revised the technical content.
01/29/2010	9.0	Major	Updated and revised the technical content.
03/12/2010	10.0	Major	Updated and revised the technical content.
04/23/2010	11.0	Major	Updated and revised the technical content.

Copyright © 2011 Microsoft Corporation.

Date	Revision History	Revision Class	Comments
06/04/2010	12.0	Major	Updated and revised the technical content.
07/16/2010	13.0	Major	Significantly changed the technical content.
08/27/2010	14.0	Major	Significantly changed the technical content.
10/08/2010	15.0	Major	Significantly changed the technical content.
11/19/2010	16.0	Major	Significantly changed the technical content.
01/07/2011	17.0	Major	Significantly changed the technical content.
02/11/2011	18.0	Major	Significantly changed the technical content.
03/25/2011	19.0	Major	Significantly changed the technical content.
05/06/2011	20.0	Major	Significantly changed the technical content.
06/17/2011	20.1	Minor	Clarified the meaning of the technical content.

Copyright © 2011 Microsoft Corporation.

Contents

Copyright © 2011 Microsoft Corporation.

Copyright © 2011 Microsoft Corporation.

Copyright © 2011 Microsoft Corporation.

Copyright © 2011 Microsoft Corporation.

1 Introduction

This Protocol Family System Document (PFSD) is primarily intended to cover the Protocol Family as a whole. In conjunction with Member Protocol Technical Documents (TDs), which are intended to cover Member Protocols, it presents the rules for information exchange relevant to those Member Protocols and the Protocol Family that are used to interoperate or communicate with a Windows operating system in its various environments.

In an asynchronous, distributed environment, applications often need to communicate with other applications over heterogeneous networks and systems. The receiving applications, networks, and systems may be temporarily unavailable. The Message Queuing System is a communications service that enables reliable and secure asynchronous message communication between such applications over a variety of deployment topologies.

This document describes the relationships among the protocols that comprise the Message Queuing System. Section 2 of this document gives a brief overview of a Message Queuing System and its basic components, and describes how it provides asynchronous messaging functionality between applications. Section 3 of this document introduces the background and specific concepts of the Message Queuing System, its purpose, and a use case model to elaborate the various actors and functions of the system. Section 4 describes the context, the environment, and the relationship of the various internal and external components of the Message Queuing System. The abstract models of the Message Queuing System are described in section 5. The Message Queuing System details, including examples for this system PFSD, are available in section 6.

This document is primarily intended for readers interested in implementing a new Message Queuing System or interoperating with an existing Message Queuing System. The various people groups that are listed as stakeholders in section 3.3.1 are expected to read this document to the extent that satisfies their respective interests in the Message Queuing System.

1.1 Glossary

The following terms are defined in [MS-GLOS]:

Active Directory
atomic transaction
domain
domain controllers (DCs)
globally unique identifier (GUID)
NetBIOS name
remote procedure call (RPC)
security identifier (SID)
transaction manager

The following terms are defined in [MS-MQMQ]:

application
application protocol
connected network
connector application
connector server
cursor
dead-letter queue
direct format name

Copyright © 2011 Microsoft Corporation.

distribution list
enterprise
external transaction
foreign queue
format name
internal transaction
local queue
management application
message
message queue
Microsoft Message Queuing (MSMQ)
MSMQ
MSMQ management server
MSMQ Directory Service server
MSMQ queue manager
MSMQ queue server
MSMQ routing link
MSMQ routing server
MSMQ site
MSMQ site gate
MSMQ supporting server
order queue
outgoing queue
path name
private queue
public queue
queue
queue manager
remote queue
remote read
routing link
routing link cost
SOAP Reliable Messaging Protocol (SRMP)
system queue
transactional message
transactional queue

The following terms are defined in [MS-DTCO]:

facet
outcome
resource manager (RM)

The following terms are defined in [MC-MQAC]:

negative source journaling
positive source journaling

The following terms are specific to this document:

asynchronous messaging: Communication between two applications or systems, independent of time.

at most once: A message delivery assurance that requires that the Message Queuing System will deliver the message to the destination at most once. Some messages may not be delivered.

Copyright © 2011 Microsoft Corporation.

best effort: Indicates that the Message Queuing System makes a best effort to meet the specified message delivery assurance, but does not raise an error if the delivery assurance is not met.

dequeue: The act of receiving and removing a message from a queue.

Directory Service: A distributed data storage system that allows computers connected to a network to store, edit, and retrieve information.

Directory-Integrated mode: A Message Queuing deployment mode in which the clients and servers use a Directory Service to enable a set of features pertaining to message security, efficient routing, queue discovery, distribution lists, and aliases. See also **Workgroup mode**.

enqueue: The act of placing a message in a queue.

exactly once: A message delivery assurance that requires that the Message Queuing System will deliver the message to the destination once and only once such that each sent message is either delivered once to the destination or an error is raised.

express message: A volatile message that does not persist through queue manager restarts. These messages provide best-effort, at-most-once delivery assurance.

NetBIOS domain name: A **NetBIOS name** that identifies a **domain**.

network address: An address that is used to identify and communicate with a specific computer in a computer network. A computer can have more than one network address. Any of these network addresses can be used to communicate with the computer.

poison message: A **message** in a **queue** that cannot be processed by an application because of errors unrelated to the Message Queuing System. A poison message can cause an application to fail repetitively until the message has been removed from the queue.

receive: An atomic operation that retrieves and removes a message from a message queue.

recoverable message: A message that persists through queue manager restarts. These messages provide best-effort, at-most-once delivery assurance.

routing server: See **MSMQ routing server**.

site: See **MSMQ site**.

supporting server: See **MSMQ supporting server**.

transaction: An atomic transaction context dispensed by a transaction coordinator, such as the Microsoft Distributed Transaction Coordinator (for more information, see [MSDN-DTC]), and used by a queue manager to coordinate its state changes with state changes in other resource managers. For more information, see [MS-DTCO].

Transaction Coordinator: Provides concrete mechanisms for beginning, propagating, and completing atomic transactions. It also provides mechanisms for coordinating agreement on a single atomic outcome for each transaction, and for reliably distributing that outcome to all participants in the transactions. For more information, see [MS-DTCO].

Workgroup mode: A Message Queuing deployment mode in which the clients and servers operate without using a Directory Service. In this mode, features pertaining to message security, efficient routing, queue discovery, distribution lists, and aliases are not available. See also **Directory-Integrated mode**.

Copyright © 2011 Microsoft Corporation.

The following protocol abbreviations are used in this document:

HTTP 1.1: Hypertext Transfer Protocol -- HTTP/1.1 as defined in [RFC2616].

MQAC: Message Queuing (MSMQ): ActiveX Client Protocol as defined in [MC-MQAC].

MQBR: Message Queuing (MSMQ): Binary Reliable Message Routing Algorithm as defined in [MS-MQBR].

MQCN: Message Queuing (MSMQ): Directory Service Change Notification Protocol as defined in [MS-MQCN].

MQDMPR: Message Queuing (MSMQ): Common Data Model and Processing Rules as defined in [MS-MQDMPR].

MQDS: Message Queuing (MSMQ): Directory Service Protocol as defined in [MS-MQDS].

MQDSSM: Message Queuing (MSMQ): Directory Service Schema Mapping as defined in [MS-MQDSSM].

MQMP: Message Queuing (MSMQ): Queue Manager Client Protocol as defined in [MS-MQMP].

MQMR: Message Queuing (MSMQ): Queue Manager Management Protocol as defined in [MS-MQMR].

MQQB: Message Queuing (MSMQ): Binary Reliable Messaging Protocol as defined in [MS-MQQB].

MQQP: Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol as defined in [MS-MQQP].

MQRR: Message Queuing (MSMQ): Queue Manager Remote Read Protocol as defined in [MS-MQRR].

MQSD: Message Queuing (MSMQ): Directory Service Discovery Protocol as defined in [MS-MQSD].

PGM: Pragmatic General Multicast as defined in [RFC3208]

SOAP 1.1: Simple Object Access Protocol (SOAP) 1.1 as defined in [SOAP1.1].

SRMP: Message Queuing (MSMQ): SOAP Reliable Messaging Protocol (SRMP) as defined in [MC-MQSRM].

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as described in [RFC2119]. Note that in [RFC2119] terms, most of these specifications should be imperative, to ensure interoperability. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

Any specification that does not explicitly use one of these terms is mandatory, exactly as if it used MUST.

1.2 References

References to Microsoft Open Specification documents do not include a publishing year because links are to the latest version of the documents, which are updated frequently. References to other documents include a publishing year when one is available.

Copyright © 2011 Microsoft Corporation.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information. Please check the archive site, http://msdn2.microsoft.com/en-us/library/E4BD6494-06AD-4aed-9823-445E921C9624, as an additional source.

[C706] The Open Group, "DCE 1.1: Remote Procedure Call", C706, August 1997, http://www.opengroup.org/public/pubs/catalog/c706.htm

[MC-COMQC] Microsoft Corporation, "Component Object Model Plus (COM+) Queued Components Protocol Specification".

[MC-MQAC] Microsoft Corporation, "Message Queuing (MSMQ): ActiveX Client Protocol Specification".

[MC-MQSRM] Microsoft Corporation, "Message Queuing (MSMQ): SOAP Reliable Messaging Protocol (SRMP) Specification".

[MS-ADA2] Microsoft Corporation, "Active Directory Schema Attributes M".

[MS-ADTS] Microsoft Corporation, "Active Directory Technical Specification".

[MS-DISO] Microsoft Corporation, "Domain Interactions System Overview".

[MS-DTCO] Microsoft Corporation, "MSDTC Connection Manager: OleTx Transaction Protocol Specification".

[MS-MQBR] Microsoft Corporation, "Message Queuing (MSMQ): Binary Reliable Message Routing Algorithm".

[MS-MQCN] Microsoft Corporation, "Message Queuing (MSMQ): Directory Service Change Notification Protocol Specification".

[MS-MQDMPR] Microsoft Corporation, "Message Queuing (MSMQ): Common Data Model and Processing Rules".

[MS-MQDS] Microsoft Corporation, "Message Queuing (MSMQ): Directory Service Protocol Specification".

[MS-MQDSSM] Microsoft Corporation, "Message Queuing (MSMQ): Directory Service Schema Mapping".

[MS-MQMP] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager Client Protocol Specification".

[MS-MQMQ] Microsoft Corporation, "Message Queuing (MSMQ): Data Structures".

[MS-MQMR] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager Management Protocol Specification".

[MS-MQQB] Microsoft Corporation, "Message Queuing (MSMQ): Message Queuing Binary Protocol Specification".

[MS-MQQP] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol Specification".

Copyright © 2011 Microsoft Corporation.

[MS-MQRR] Microsoft Corporation, "Message Queuing (MSMQ): Queue Manager Remote Read Protocol Specification".

[MS-MQSD] Microsoft Corporation, "Message Queuing (MSMQ): Directory Service Discovery Protocol Specification".

[MS-RDPBCGR] Microsoft Corporation, "Remote Desktop Protocol: Basic Connectivity and Graphics Remoting Specification".

[MS-RPCE] Microsoft Corporation, "Remote Procedure Call Protocol Extensions".

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992, http://www.ietf.org/rfc/rfc1321.txt

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[RFC2616] Fielding, R., Gettys, J., Mogul, J., et al., "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999, http://www.ietf.org/rfc/rfc2616.txt

[RFC3208] Speakman, T., Crowcroft, J., Gemmell, J., Farinacci, D., Lin, S., Leshchiner, D., Luby, M., Montgomery, T., Rizzo, L., Tweedly, A., Bhaskar, N., Edmonstone, R., Sumanasekera, R., and Vicisano, L., "PGM Reliable Transport Protocol Specification", RFC 3208, December 2001, http://www.ietf.org/rfc/rfc3208.txt

[RFC3275] Eastlake III, D., Reagle, J., and Solo, D., "(Extensible Markup Language) XML-Signature Syntax and Processing", RFC 3275, March 2002, http://www.ietf.org/rfc/rfc3275.txt

[RFC3377] Hodges, J., and Morgan, R., "Lightweight Directory Access Protocol (v3): Technical Specification", RFC 3377, September 2002, http://www.ietf.org/rfc/rfc3377.txt

[SOAP1.1] Box, D., Ehnebuske, D., Kakivaya, G., et al., "Simple Object Access Protocol (SOAP) 1.1", May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[SP800-32] National Institutes of Standards and Technology, "Introduction to Public Key Technology and the Federal PKI Infrastructure", SP800-32, February 2001, http://csrc.nist.gov/publications/nistpubs/800-32/sp800-32.pdf

[UML] Object Management Group, "Unified Modeling Language", http://www.omg.org/technology/documents/formal/uml.htm

1.2.2 Informative References

[MS-GLOS] Microsoft Corporation, "Windows Protocols Master Glossary".

[MS-WSO] Microsoft Corporation, "Windows System Overview".

[MSDN-DTC] Microsoft Corporation, "Distributed Transaction Coordinator", http://msdn.microsoft.com/en-us/library/ms684146.aspx

[MSDN-WCF] Microsoft Corporation, "Windows Communication Foundation", http://msdn.microsoft.com/en-us/library/ms735119.aspx

[MSFT-PKI] Microsoft Corporation, "Best Practices for Implementing a Microsoft Windows Server 2003 Public Key Infrastructure", July 2004, http://technet2.microsoft.com/WindowsServer/en/library/091cda67-79ec-481d-8a96-03e0be7374ed1033.mspx

Copyright © 2011 Microsoft Corporation.

2 Overview

Section 1, "Introduction", describes this Protocol Family System Document. This section introduces the system that is being documented.

2.1 System Summary

The Message Queuing System is a communications service that temporally decouples the act of sending a **message** from the act of receiving that message, allowing applications to communicate even if their execution lifetimes do not overlap.

The **queue** is the central abstraction in the Message Queuing System. **Applications** send messages to a queue and/or **receive** messages from a queue. The queue provides persistence of the messages, enabling them to survive across application restarts. As such, this abstraction enables an application to send messages even if the receiving application is not executing or is unreachable due to a network outage.

The high-level view of the Message Queuing System is illustrated in the following figure.

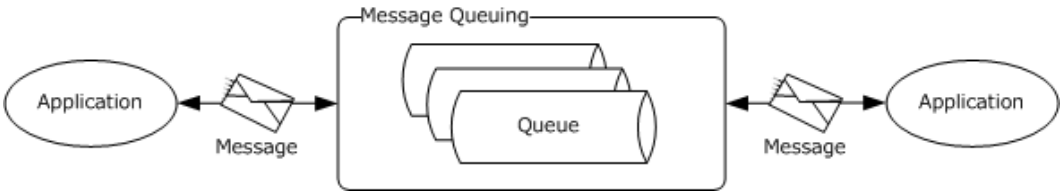


Figure 1: Message Queuing enables asynchronous message exchange

The Message Queuing System enables the following message exchange patterns between applications:

- One-Way Messaging: A source application sends messages to a destination application and does not wait for the outcome of the message processing. A destination application receives messages and processes them.
- Request-Response: A source application sends a message to a destination. The receiving application receives the request message and sends the response message to a queue specified by the sender in the request message. The sending application receives the response message and can correlate it to the original request message.
- Broadcast: A source application sends messages that may be received by zero or more applications. This pattern is useful in implementing publish-and-subscribe types of applications.

Queues are hosted by a communications service called a **queue manager**, which runs in a separate service from the client applications so that the act of sending messages is decoupled from the act of receiving the messages. Applications interact with the queue manager by using the Message Queuing System protocols to accomplish the intended Message Queuing System functionality, as shown in the following figure.

Copyright © 2011 Microsoft Corporation.

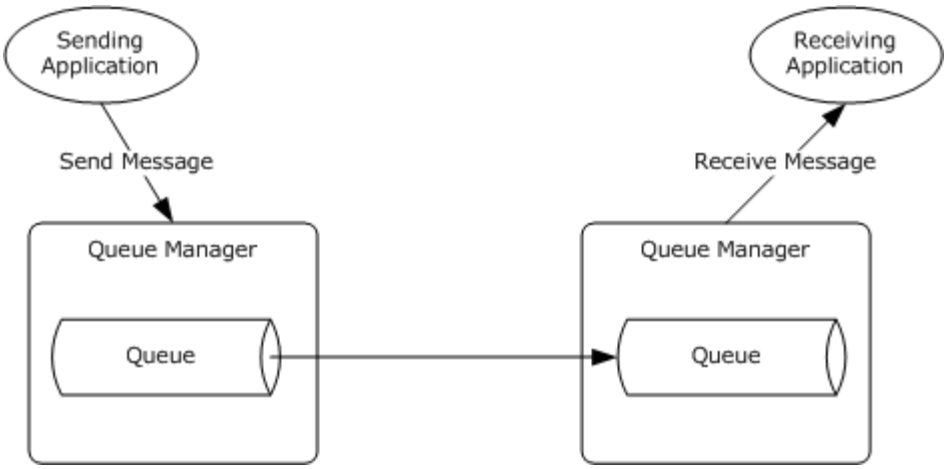


Figure 2: Queues hosted by queue manager service

The Message Queuing System can optionally interact with other components to provide richer functionality to applications. These components include the **Directory Service** and the **Transaction Coordinator**. The participants in a Message Queuing System are shown in the following figure and include:

- Application: An application uses the **application protocols** of the Message Queuing System and the associated components to exchange messages asynchronously with other applications, as well as to perform management and administrative operations on the Message Queuing System.
- Queue Manager: A queue manager is the message communication service that hosts queues and interacts with the applications for sending and receiving messages. The queue manager also interacts with other queue managers to asynchronously transfer messages between queues across a network.
- Directory Service: A Directory Service is an optional subcomponent of the Message Queuing System that stores and provides directory information such as network topology information, security key distribution, queue and system metadata, and queue discovery.
- Transaction Coordinator: A Transaction Coordinator is an optional subcomponent of the Message Queuing System. An application can send or receive messages within the context of a **transaction**, and the Transaction Coordinator interacts with the queue manager to accept or discard these operations, depending on the **outcome** of the transaction, while maintaining atomicity, consistency, isolation, and durability (ACID) behavior throughout the lifetime of the transaction.

Copyright © 2011 Microsoft Corporation.

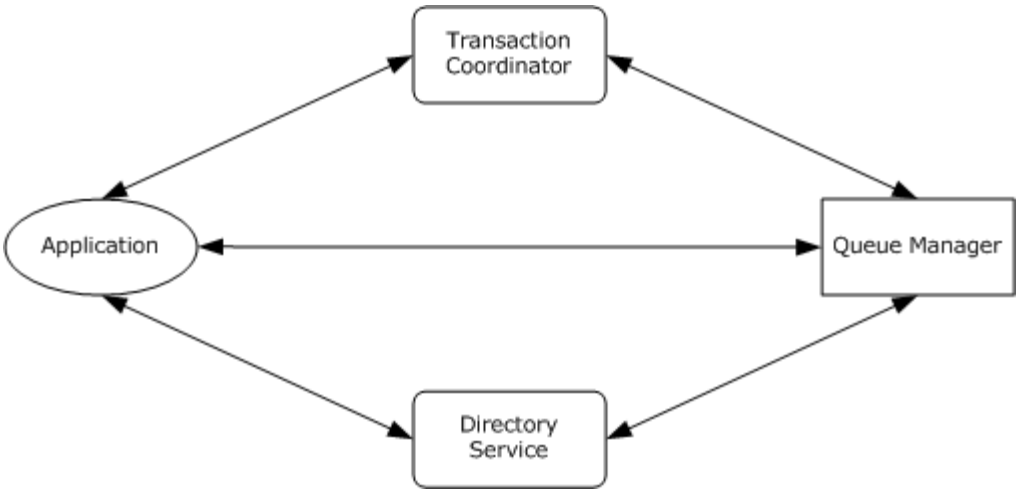


Figure 3: Participating components in a Message Queuing System

2.2 List of Member Protocols

The Message Queuing System implements the protocols defined in [MC-MQAC], [MS-MQMP], [MS-MQRR], [MS-MQQB], [MC-MQSRM], [MS-MQCN], [MS-MQMR], [MS-MQSD], [MS-MQDS], and [MS-MQQP] and uses data structures defined in [MS-MQMQ], common data models and processing rules defined in [MS-MQDMPR], directory schema mappings defined in [MS-MQDSSM], and algorithms defined in [MS-MQBR]. A list of the individual member protocols that make up the MSMQ system is given below:

Message Queuing (MSMQ): Data Structures, as specified in [MS-MQMQ]. The common definition and data structures used by the member protocols of the Message Queuing System.

Message Queuing (MSMQ): Common Data Model and Processing Rules, as specified in [MS-MQDMPR]. The abstract data model, events, and processing rules shared by the member protocols of the Message Queuing System protocol family.

Message Queuing (MSMQ): ActiveX Client Protocol, as specified in [MC-MQAC]. A DCOM protocol that provides a client application programming interface to the Message Queuing System.

Message Queuing (MSMQ): Queue Manager Client Protocol, as defined in [MS-MQMP]. An RPC protocol that provides a client interface to the Message Queuing System through the Supporting Server deployment profile.

Message Queuing (MSMQ): Binary Reliable Messaging Protocol, as specified in [MS-MQQB]. A block protocol over TCP/IP or SPX/IPX for reliable transfer of messages between two queue managers.

Message Queuing (MSMQ): Binary Reliable Messaging Algorithm, as specified in [MS-MQBR]. A routing algorithm used with MQQB by a queue manager to route messages to the appropriate queue manager in a complex network topology.

Message Queuing (MSMQ): SOAP Reliable Messaging Protocol (SRMP), as specified in [MC-MQSRM]. An HTTP-based or Pragmatic General Multicast (PGM)–based protocol that uses SOAP encoding for reliable transfer of messages from one queue manager to others.

Message Queuing (MSMQ): Directory Service Change Notification Protocol, as specified in [MS-MQCN]. A block protocol for notifying a queue manager about changes made to the Directory Service objects owned by that queue manager.

Copyright © 2011 Microsoft Corporation.

Message Queuing (MSMQ): Queue Manager Management Protocol, as specified in [MS-MQMR]. An RPC protocol that provides a client interface for Message Queuing System administration.

Message Queuing (MSMQ): Directory Service Discovery Protocol, as specified in [MS-MQSD]. A block protocol that uses UDP multicast for locating MSMQ Directory Service servers.

Message Queuing (MSMQ): Directory Service Protocol, as specified in [MS-MQDS]. An RPC protocol that implements the MSMQ Directory Service. This protocol is superseded by **Active Directory**.

Message Queuing (MSMQ): Directory Service Schema Mapping, as specified in [MS-MQDSSM]. A mapping between relevant Message Queuing abstract data model elements and a directory service over a Lightweight Directory Access Protocol (LDAP) interface.

Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol, as specified in [MS-MQQP]. An RPC protocol that provides an interface between two queue managers for reading and browsing messages from a remote queue. This protocol has been superseded by MQRR.

Message Queuing (MSMQ): Queue Manager Remote Read Protocol, as specified in [MS-MQRR]. An RPC protocol that provides an interface between two queue managers for reading and browsing messages from a remote queue. This protocol supersedes MQQP.

2.3 Relevant Standards

There are no standards assignments for the Message Queuing System.

Copyright © 2011 Microsoft Corporation.

3 Foundation

This section describes the theoretical and practical information needed to understand this document and this system.

3.1 Background Knowledge and System-Specific Concepts

This section summarizes:

- Background knowledge that is required to understand this document.
- Concepts that are specific to this system.

The reader of this document should be familiar with the following topics and concepts:

- Distributed and Decoupled Applications
- Synchronous versus Asynchronous Message Communication
- Message Oriented Middleware
- Message Delivery Assurance
- Transactions
- Directory Service and Active Directory
- Security Models, Access Control, Authentication, Encryption

Furthermore, the reader should also be familiar with the following concepts that are specific to the Message Queuing System:

- Queue
- Message Transfer and Routing
- Queue Manager

The above system-specific concepts are elaborated in greater detail below.

A queue is a temporary placeholder for messages that are shared between applications. The simplest Message Queuing deployment involves two applications and a single queue that is accessible to both the applications. The queue is hosted and managed by a single Queue Manager. One application sends messages to the queue and the other application receives the messages from the same queue, as shown in the following diagram.

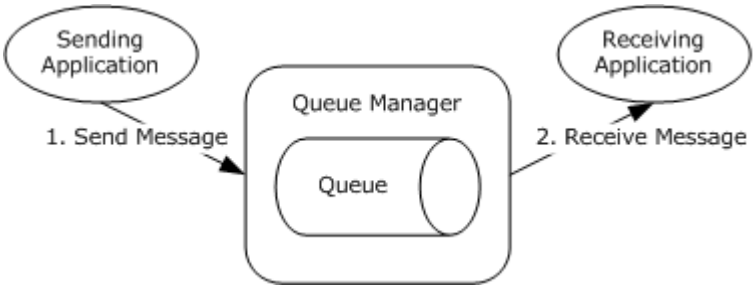


Figure 4: MSMQ deployment with two applications and a single queue

Copyright © 2011 Microsoft Corporation.

The sending application sends a message to the queue [1]. When the send is successful, the application proceeds with other work or terminates. The receiving application subsequently receives the message asynchronously [2]. The message is removed from the queue. This asynchronous message exchange pattern enables the temporal decoupling of the send operation from the receive operation.

The next figure, "MSMQ deployment where applications do not share the same queue", illustrates a topology that is slightly different from the previous one in that the sending application and the receiving application do not share the same queue. Instead, both the sending application and the receiving application interact with separate, directly accessible queues. The sending application interacts with a source queue, and the receiving application interacts with a destination queue. Additionally, the destination queue is directly reachable from the source queue in a network. Each queue is hosted and managed by a separate queue manager.

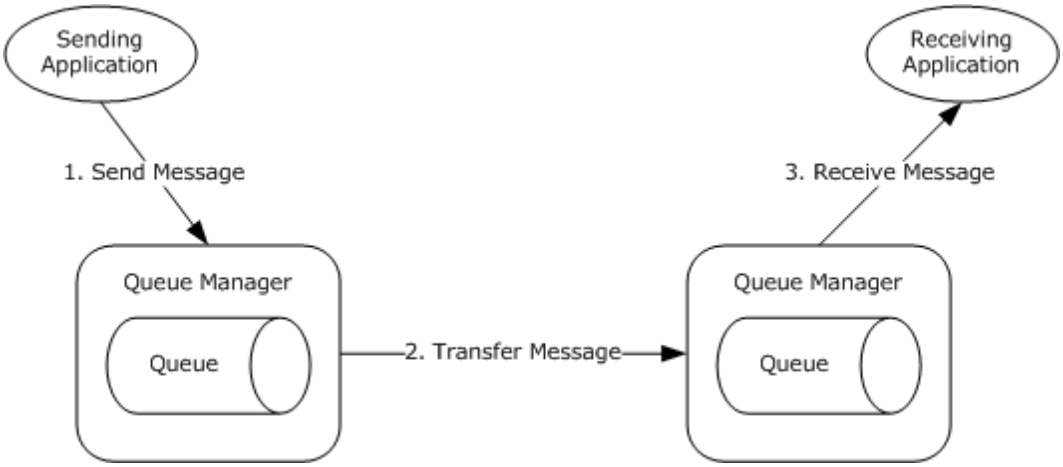


Figure 5: MSMQ deployment where applications do not share the same queue

In the deployment topology shown in the preceding figure, the sending application puts a message in the queue [1]. This source queue works as the temporary placeholder for the message and is called the outgoing queue. Next, the Message Queuing System directly transfers the message to the destination queue and removes the message from the outgoing queue [2]. Finally, the receiving application receives the message from the destination queue [3].

If the destination queue is not directly reachable from the source queue in a network, additional interim queues are required between the source queue and the final destination queue. Each interim queue is hosted and managed by a queue manager. Messages are routed to the final destination queue through one or more interim queues. Although the destination queue is not directly reachable from the source queue, each interim queue is reachable by its preceding queue and its successor queue.

Queues are hosted and managed by a queue manager playing the queue server role. The queue manager hosts and manages a set of local queues, acts as an intermediary placeholder for storing and forwarding messages to their final destinations, and interacts with the applications for sending and receiving messages. The queue manager performs the following tasks:

- On the send side, the queue manager manages its queues, accepts messages from the sending application and optionally transfers messages to other queue managers. If the messages are destined for a queue that the send-side queue manager hosts, they are placed in that queue on the machine (as illustrated in figure 4, "MSMQ deployment with two applications and a single queue"). Alternatively, if the messages belong to a queue not hosted by the queue manager on

Copyright © 2011 Microsoft Corporation.

the send side, they are placed in an outgoing queue and subsequently transferred to the destination queue manager (as illustrated in figure 5, "MSMQ deployment where applications do not share the same queue").

- On the receive side, the queue manager manages its queues, accepts messages transferred from other queue managers, and delivers messages to the receiving application. Figure 4, "MSMQ deployment with two applications and a single queue" illustrates the simple topology where the send-side queue manager is the same as the receive-side queue manager. In this illustration, the single queue manager manages its queues, accepts messages from the sending application and delivers them to the receiving application. Figure 5, "MSMQ deployment where applications do not share the same queue",illustrates a topology that involves two separate queue managers, one at the send side that interacts with the sending application, and the other at the receiving side that interacts with the receiving application. These two queue managers interact with each other for transferring messages between queues. As the queue manager on the send side transfers messages from its outgoing queue, the queue manager on the receive side accepts and stores those messages. Subsequently, the receive-side queue manager delivers the messages to the appropriate receiving applications.
- Optionally, there can be other queue managers between the send and the receive queue managers. This is needed for facilitating efficient message routing between the source and the destination queues. These interim queue managers store incoming messages and route them to the next hop so that they can eventually reach the final destination queue.

The queue manager undertakes one or more of the preceding tasks; for each task, the queue manager can manage more than one queue. In other words, a queue manager manages all its hosted queues and its outgoing queues, interacts with the sending and receiving applications, and interacts with other queue managers to transfer messages between queues.

An example of a simple Message Queuing System deployed on a network is shown in the following figure.

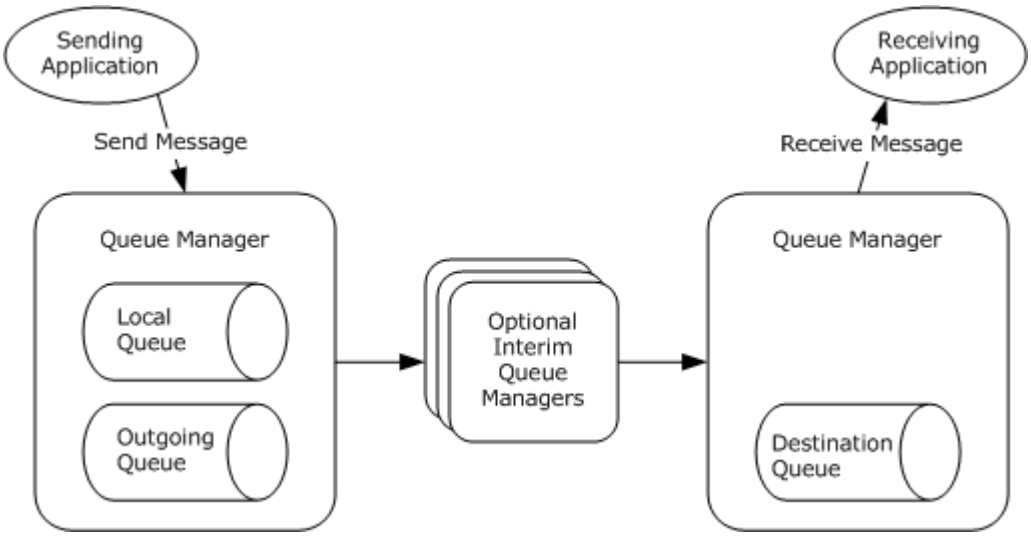


Figure 6: Message Queuing System deployed over a network

As depicted in the preceding figure, a sending application sends a message to a nearby queue manager. If the destination queue is hosted by the queue manager (a local queue), the queue manager stores the message in the local queue. Alternatively, if the destination queue is hosted by another queue manager on a different machine, the queue manager places the message in an

Copyright © 2011 Microsoft Corporation.

outgoing queue. In either case, the sending application can proceed to do other work. The queue manager asynchronously transfers the message from the outgoing queue to the queue manager of the destination queue, optionally through interim queue managers for routing the message. Subsequently, a receiving application reads the message from the destination queue.

3.2 System Purposes

The primary purpose of the Message Queuing System is to enable temporal decoupling of applications by providing an asynchronous messaging service between applications that need to reliably communicate with each other by sending and receiving messages. The Message Queuing System accounts for transient networking failure, system availability, and queue location. Specifically, the Message Queuing System provides the following capabilities relevant for messaging applications.

3.2.1 Message Delivery Assurance

The **Message Queuing** system provides the following levels of message delivery assurance:

- **Best-effort Express Delivery**: The Message Queuing system provides **best-effort**, **at-most-once** express delivery assurance through **express messages** that are volatile and hence can be lost in transit during transient unavailability or failure of the Message Queuing components or of the underlying network. These messages do not live through system restarts.
- **Best-effort Delivery**: The Message Queuing system provides best-effort, at-most-once delivery assurance through **recoverable messages**. These messages are persisted by the Queue Manager and so they live through system restarts. These messages can be lost in transit during transient unavailability or failure of the underlying network.
- **Exactly-once Delivery**: The Message Queuing system provides **exactly-once** delivery assurance for messages that are sent to **transactional queues**. A source application sends one or more messages to a transactional queue as part of a transaction. Depending on the outcome of the transaction, the queue manager accepts and persists these messages. Subsequently the messages are transferred between queue managers for placment in the destination queue. A destination application receives and processes messages as part of a transaction. Whether the messages are considered consumed or not depends on the outcome of the transaction. The Message Queuing system MUST receive and deliver these messages only in the scope of a transaction.

3.2.2 Message Transfer and Routing

The Message Queuing system supports message transfer across complex network topologies and over a variety of network transport protocols, including TCP/IP or IPX/SPX, UDP, HTTP, HTTPS, and PGM. To optimize message throughput, the Message Queuing system defines an optional routing mechanism to find and use the least-cost routing path between two machines, as specified in [MS-MQBR].

3.2.3 Message Security

The Message Queuing system enables secure messaging between applications by supporting a variety of security features, including user authentication, authorization, message integrity, and message encryption and decryption. The Message Queuing security architecture is described in section 7 of this document.

Copyright © 2011 Microsoft Corporation.

3.2.4 Management and Administration

The Message Queuing system defines protocols for **management applications** for the configuration, management, and administration of the Message Queuing system.

3.3 System Use Cases

3.3.1 Stakeholders and Interests Summary

- Architects: Architects can use the Message Queuing System as a building block for architecting and designing a generic distributed application framework that uses asynchronous messaging as a communication backbone between system components. Application architects use the Message Queuing System for designing distributed applications requiring reliable and asynchronous message support. Architects can also use this system to design interoperability with other queuing systems.
- Developers: Developers can use the Message Queuing System to build and implement asynchronous messaging functionality that can be used by a number of user applications. Developers can also use the system to implement seamless interoperability of the existing Message Queuing Systems with newly implemented ones. Application developers can use the various application protocols of the Message Queuing system to build applications that interact with a Message Queuing system and accomplish message queuing functionality within the user applications.
- Testers: Quality assurance teams require the Message Queuing System for testing implementations created by the developers in order to verify the conformance of such implementations with the protocol specifications.
- System Administrators: System administrators are the primary actors for the management and administrative operations of the Message Queuing System. System administrators require the system for:
  - Understanding the various operations and management aspects of a distributed application that uses message queuing functionality.
  - Performing various management and administrative operations of a distributed application that uses message queuing functionality.
  - Enumerating the protocols and all the artifacts of these protocols (for example, message format, error codes, and retry logic) that they expect to see flowing over the networks in their enterprise.
- Application: Applications use the various Message Queuing application protocols to interact with the Message Queuing System. The role of an application is described in detail in sections 4.3.1.1 and 5.2.1.1 of this document. The application is the direct actor for all Message Queuing use cases except Transfer Message (section 3.3.4.5).
- Application Users: Application users are the primary actors of the Message Queuing System. The application users perform business operations by using the functionality of the distributed applications. Such application operations invoke various message queuing activities within the system.

Copyright © 2011 Microsoft Corporation.

3.3.2 Supporting Actors and System Interests Summary

Transaction Coordinator: In order to manage a queue as a transactional resource, the queue manager acts as a resource manager to interact with a Transaction Coordinator. The details of such interaction are specified in section 4.3.1.2.

Directory Service: The Message Queuing System optionally uses a directory by invoking the client directory service interfaces. The details of such interactions are specified in section 4.3.1.3.

3.3.3 Use Case Diagrams

This section breaks down the functionality of the Message Queuing System into small, granular use cases that can then be combined to build and manage complex distributed applications.

The application is the direct actor for all use cases of the Message Queuing System. The Directory Service is the supporting actor for all use cases of the Message Queuing System. The Transaction Coordinator is the supporting actor for the use cases that require transacted work.

The main functionality of the Message Queuing System is divided into two broad categories: "Message Exchange" and "MSMQ Management". The following use case diagram shows how the actors and Message Queuing use cases fit together into a complete Message Queuing System.

Copyright © 2011 Microsoft Corporation.

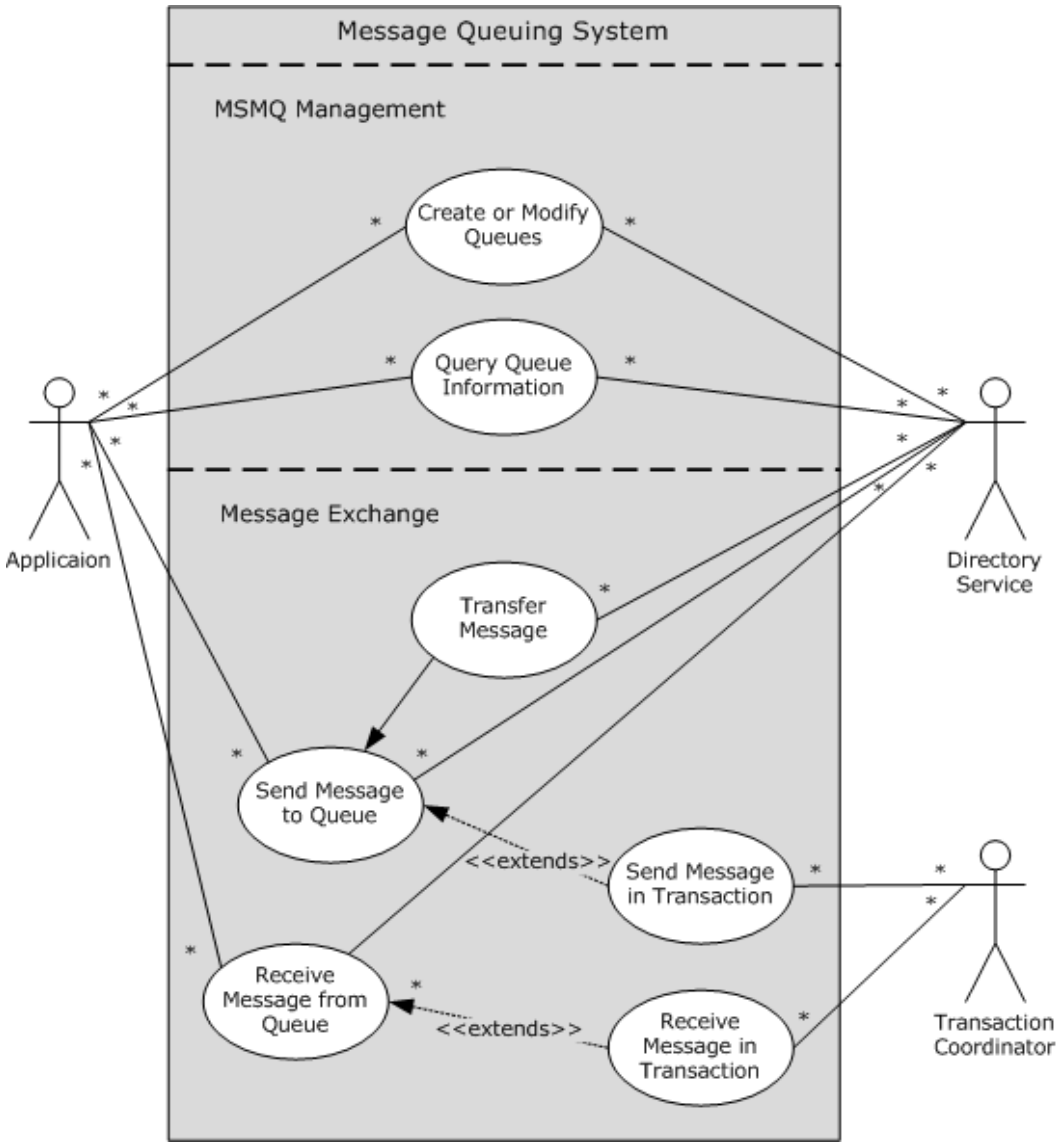


Figure 7: Message Queuing System use case diagram

3.3.4 Use Case Descriptions

3.3.4.1 Create or Modify Queue - Application

Goal: This use case is initiated by an application to create a queue or modify the properties of a queue.

Context of Use: The application makes changes to the queuing environment to facilitate message exchange operations between other participating applications of the overall system.

Direct Actor: Application

Primary Actor: System administrators

Copyright © 2011 Microsoft Corporation.

Supporting Actors: If the use case operation involves a public queue, Directory Service is a supporting actor.

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the following preconditions:

- The Message Queuing System MUST complete initialization as specified in section 6.6.
- The application MUST have access to the machine on which a queue is to be created or modified.
- The application MUST have the necessary administrative rights to execute the operation.

Minimal Guarantee: The queue creation or modification is either not initiated, or has been completely effected.

Success Guarantee: The application gets a response indicating success or failure of the operation. On success, the queue is created or modified. On failure, no changes are made.

Trigger: The direct actor triggers this use case based on the actions of the primary actor.

The steps involved in the use case for creation or modification of a private queue are:

1. The application sends a request to the queue manager to create a private queue on that queue manager, or to modify an existing private queue hosted by the queue manager.
2. The queue manager creates or modifies the queue and sends a response back to the application.

Main Success Scenario:

The steps involved in public queue creation or modification are:

1. The application sends a request to a queue manager to create a public queue on any queue manager in the directory, or to modify an existing public queue hosted by any queue manager in the directory.
2. The queue manager sends a message to the Directory Service to make the necessary changes in the directory. On receipt of this message, the Directory Service creates the public queue in the directory.
3. The queue manager sends a notification message to the queue manager that is hosting the queue to notify it of the changes. On receipt of such a notification message, the hosting queue manager synchronizes the changes identified in the notification message.
4. The queue manager sends a response back to the application.

Upon successful completion, this use case meets the following postcondition:

- The queue is created or modified.

Extensions: None

3.3.4.2 Query Queue Information - Application

Goal: This use case is initiated by an application to query configuration and runtime information about the Message Queuing System.

Copyright © 2011 Microsoft Corporation.

Context of Use: The application collects administrative information about the Message Queuing System and presents the information for management purposes.

Direct Actor: Application.

Primary Actor: System administrators.

Supporting Actors: If the use case operation involves a public queue, Directory Service is a supporting actor.

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the following preconditions:

- The Message Queuing System MUST complete initialization as specified in section 6.6.
- The application administrator MUST complete the application-specific configuration of the Message Queuing System, such as creating the necessary queues.
- The application MUST have access to the machine and queue.
- The application MUST have the necessary administrative rights to execute the query.

Minimal Guarantees: The state of the queue does not change.

Success Guarantee: The application gets a response indicating the success or failure of the operation. On successful completion, the application receives the requested information.

Trigger: The direct actor triggers this use case based on the actions of the primary actor.

Main Success Scenario:

The processing of the use case is as follows:

1. The application sends a request to the queue manager to query the necessary data.
2. The queue manager responds to the query by providing the information requested.

Upon successful completion, this use case meets the following postconditions:

- The state of the queue does not change.
- The application receives the requested information.

Extensions: None

3.3.4.3 Send Message to Queue - Application

Goal: This use case places a message in a queue.

Context of Use: An application creates a message and interacts with the queue manager to send the message. The application optionally uses a Directory Service for looking up the queue name.

Direct Actor: Application, Exchange Message - Application

Primary Actor: Application users

Copyright © 2011 Microsoft Corporation.

Supporting Actors: If the use case operation involves a **public queue**, Directory Service is a supporting actor.

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the following preconditions:

- The Message Queuing System MUST complete initialization as specified in section 6.6.
- The queue MUST exist.
- The application MUST be authorized to send messages to the queue.
- If a Directory Service is not being used, the application MUST be configured with the address of the queue prior to the send operation.

Minimal Guarantee: None

Success Guarantee: The application gets a response indicating success or failure of the operation. On success, the message is accepted by the queue manager.

Trigger: The direct actor triggers this use case based on the actions of the primary actor. It is also triggered by the Exchange Message - Application use case.

Main Success Scenario:

The steps involved in this use case are:

1. The application constructs a message to send.
2. The application optionally obtains the queue name from the Directory Service.
3. The application sends the message to the queue manager.
4. The queue manager performs validation checks and fails the operation in case of an error.
5. If the queue is hosted by this queue manager, the queue manager puts the message in the queue and returns a response back to the application. An extension of this step is described in section 3.3.4.4, Send Message in Transaction.
6. If the queue is hosted by a different queue manager, the queue manager puts the message in an outgoing queue and invokes a separate use case to complete the message transfer operation, as described in section 3.3.4.5, Transfer Message.

Upon successful completion, this use case meets the following postcondition:

- The message is placed in the destination queue.

Extensions: Send Message in Transaction use case, as described in section 3.3.4.4.

3.3.4.4 Send Message in Transaction - Application

Goal: This use case places one or more messages in a queue under the context of an atomic transaction.

Context of Use: This use case extends section 3.3.4.3, Send Message to Queue, by adding transactional semantics to the message send operation. In this use case, the application and the

Copyright © 2011 Microsoft Corporation.

queue manager interact with a Transaction Coordinator to send one or more messages under the context of a transaction. The messages are visible only after a successful outcome of the transaction. A failed outcome of the transaction undoes the entire message send operation.

Direct Actor: Application, Exchange Message – Application

Primary Actor: Application users

Supporting Actors: Transaction Coordinator

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the following precondition, in addition to those described in the extending use case in section 3.3.4.3, Send Message to Queue:

- The transaction coordinator MUST be accessible to the application and the queue manager in order to coordinate the transaction execution.

Minimal Guarantee: If the outcome of the transaction is success, the messages are accepted by the queue manager. If the outcome of the transaction is abort, the messages are not accepted by the queue manager.

Success Guarantee: The application gets a response indicating success or failure of the operation.

Trigger: The direct actor triggers this use case based on the actions of the primary actor. It is also triggered by the Exchange Message – Application use case.

Main Success Scenario:

The steps involved in this use case are as follows:

1. The application creates a transaction and follows steps 1 through 4 of section 3.3.4.3, Send Message to Queue, to send each message to the queue manager under the context of the transaction.
2. The queue manager enlists in the transaction if it has not enlisted in the transaction yet and does not make the messages visible outside the context of the transaction.
3. The queue manager returns a response back to the application.
4. The application commits or aborts the transaction, and the transaction coordinator communicates the outcome of the transaction to the resource manager facet of the queue manager. Upon a successful outcome, the queue manager makes the messages visible, and this use case continues to the next step. Conversely, upon a failed outcome of the transaction, the queue manager undoes the message send operations and this use case completes.
5. Follow step 6 of section 3.3.4.3, Send Message to Queue, for each message that will be sent.

Upon completion, this use case meets the following postconditions:

- If the outcome of the transaction is successful, the messages are placed in the destination queue as with the postcondition in section 3.3.4.3, Send Message to Queue.
- If the outcome of the transaction is failed, the messages do not appear in the destination queue.

Extensions: None

Copyright © 2011 Microsoft Corporation.

3.3.4.5 Transfer Message

Goal: Transfer a message from one queue manager to another.

Context of Use: This use case is optionally used by the use case in section 3.3.4.3, Send Message to Queue, as well as by the use case in section 3.3.4.4, Send Message in Transaction, to enable messages to be sent to queues that are not hosted by the source queue manager. This use case can be invoked only by the Send Message use cases described in sections 3.3.4.3 and 3.3.4.4 and is not intended to be directly invoked by the actors described in the Message Queuing use case diagram (section 3.3.3).

Direct Actor: Send Message to Queue – Application, Send Message in Transaction - Application

Primary Actor: None

Supporting Actors: If the use case operation involves a public queue, Directory Service is a supporting actor.

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the following preconditions, in addition to those of the invoking use cases (section 3.3.4.3, Send Message to Queue, or section 3.3.4.4, Send Message in Transaction):

- If invoked from the use case in section 3.3.4.3, Send Message to Queue, steps 1 through 5 in section 3.3.4.3 are performed and the message is placed in an outgoing queue, OR
- If invoked from the use case in section 3.3.4.4, Send Message in Transaction, steps 1 through 4 in section 3.3.4.4 are performed, the transaction has a successful outcome, and as a result the message is placed in an outgoing queue.

Minimal Guarantee: If the identifier of an incoming message is already known by the destination queue manager, indicating a duplicate message, the message is discarded.

Success Guarantee:

- If the message transfer is successful, the message is placed in the destination queue.
- If the message transfer fails, the message is optionally placed in the designated dead-letter queue.

Trigger: This use case is triggered by the Send Message to Queue use case, as described in section 3.3.4.3, or the Send Message in Transaction use case, as described in section 3.3.4.4.

Main Success Scenario:

The steps involved in this use case are:

1. The source queue manager determines the destination queue manager and transfers the message.
2. The destination queue manager accepts the message and performs duplication and validation checks and places the message in the destination queue. If the identifier of an incoming message is already known by the destination queue manager, it is considered a duplicate and is discarded without any further processing. If the message fails validation checks, a negative acknowledgment is sent to the source queue manager. On successful transfer, the destination queue manager sends the appropriate acknowledgments back as specified in the In Transit portion of section 5.1.1.3.

Copyright © 2011 Microsoft Corporation.

3. Steps 1 and 2 are repeated until the source queue manager gets the appropriate acknowledgments back as specified in the In Transit portion of section 5.1.1.3 or rejects the message for further retransmission.

Extensions: In a variation of this use case, the Message Queuing System uses the MQBR algorithm to perform multi-hop message transfer. In this variation, the queue manager MUST use the MQBR algorithm in step 1 to determine the next hop queue manager, and the steps described in this section are repeated until the message reaches the final destination queue manager.

3.3.4.6 Receive a Message from a Queue - Application

Goal: This use case receives a message from a queue.

Context of Use: This use case covers the counterpart of the Send Message use case. An application receives a message from a queue. The application optionally uses a Directory Service for looking up the queue name.

Direct Actor: Application, Exchange Message – Application

Primary Actor: Application users

Supporting Actors: If the use case operation involves a public queue, Directory Service is a supporting actor.

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the same preconditions as in section 3.3.4.3, Send Message to Queue, with the exception that the receiving application MUST be authorized to receive messages from the queue.

Minimal Guarantee: None

Success Guarantee: The application gets a response indicating success or failure of the operation. On success, if a message is available to receive from the queue, it is returned to the application as a result of the receive operation and the message is removed from the queue.

Trigger: The direct actor triggers this use case based on the actions of the primary actor. It is also triggered by the Exchange Message – Application use case.

Main Success Scenario:

The steps involved in this use case are:

1. The application optionally obtains the queue name from the Directory Service.
2. The application sends a request to the queue manager to receive a message from the queue and waits for a response from the queue manager within a specific timeout.
3. The queue manager performs validation checks and fails the operation in case of an error.
4. If a message is not available, the queue manager blocks the application until a message is available, or until the receive operation in step 2 times out or is canceled. In the latter case, an error response is returned to the application and this use case completes.
5. The queue manager removes the available message from the queue and returns it to the application as part of the response.

Copyright © 2011 Microsoft Corporation.

Upon successful completion, this use case meets the following postcondition:

- If a message is available to receive in the queue and is returned to the application as a result of the receive operation, the message is removed from the queue.

Extensions: None

3.3.4.7 Receive Message in Transaction – Application

Goal: This use case receives one or more messages from a queue under the context of an atomic transaction.

Context of Use: This use case extends the use case described in section 3.3.4.6, Receive a Message from a Queue, by adding transactional semantics to the message receive operation. In this use case, the application and the queue manager interact with a Transaction Coordinator to receive one or more messages in the context of a transaction. Whether the messages are consumed or not depends on the outcome of the transaction. The messages are removed from the queue only upon a successful outcome of the transaction. Conversely, the messages are returned back to the queue upon a failed outcome of the transaction.

Direct Actor: Application, Exchange Message – Application

Primary Actor: Application users

Supporting Actors: Transaction Coordinator

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: This use case has the following preconditions, in addition to those described in the extending use case in section 3.3.4.6, Receive a Message from a Queue:

- The Transaction Coordinator MUST be accessible to the application as well as the queue manager in order to coordinate the transaction execution.

Minimal Guarantee: If the outcome of the transaction is success, the messages are delivered to the application and removed from the queue. If the outcome of the transaction is abort, the messages are not removed from the queue.

Success Guarantee: The application gets a response indicating success or failure of the operation.

Trigger: The direct actor triggers this use case based on the actions of the primary actor. It is also triggered by the Exchange Message – Application use case.

Main Success Scenario:

The steps involved in this use case are similar to those in section 3.3.4.6, Receive a Message from a Queue, with the exception of additional interactions with the Transaction Coordinator system. The combined steps are as follows:

1. The application creates a transaction and follows steps 1–4 of section 3.3.4.6, Receive a Message from a Queue, to send each receive request to the queue manager under the context of the transaction.

2. The queue manager enlists in the transaction if it has not enlisted in the transaction yet. For each receive request, if a message was available from the previous step, the queue manager locks the message, and hence makes it invisible outside the context of the transaction, as if the message

Copyright © 2011 Microsoft Corporation.

has been temporarily removed from the queue. Each message is returned to the application for the corresponding receive request.

3. The application commits or aborts the transaction, and the Transaction Coordinator communicates the outcome of the transaction to the resource manager facet of the queue manager. Upon a successful outcome of the transaction, the queue manager removes the messages from the queue. Conversely, upon a failed outcome, the queue manager undoes the receive operation and returns the messages back to the queue.

Upon successful completion, this use case meets the following postconditions:

- If the outcome of the transaction is successful, the postcondition is the same as in section 3.3.4.6, Receive a Message from a Queue.
- If the outcome of the transaction is not successful, the messages are placed back in the queue.

Extensions: None

3.3.4.8 Exchange Message – Application

Goal: This use case enables two applications to exchange messages asynchronously. Messages are sent to a queue by a sending application and are received from the queue by a receiving application.

Context of Use: The sending application creates the messages and sends them to the queue manager that hosts the queue. The receiving application receives the messages from the queue. This use case invokes the following supporting use cases:

- Send Message to Queue – Application (or Send Message in Transaction – Application if transaction is used for sending messages)
- Receive Message from a Queue – Application (or Receive Message in Transaction – Application if transaction is used for receiving messages)

Direct Actor: Application

Primary Actor: Application users

Supporting Actors: The supporting actors of the supporting use cases

Stakeholders and Interests: Developers, testers, and application users, as described in section 3.3.1, expect the use case to function correctly and meet the application needs.

Preconditions: The preconditions of the supporting use cases

Minimal Guarantee: The minimal guarantee is dependent on the supporting use cases that are used.

Receive a Message	Send Message to Queue	Send Message in Transaction
Receive a Message from a Queue (section 3.3.4.6)	None.	If the Send Message transaction is successful, the messages are accepted by the queue manager. If the transaction is aborted, the messages are not accepted by the queue manager. Receipt of the message is not guaranteed.
Receive a Message in Transaction	None.	If the Send Message transaction is successful, the messages are accepted by the queue manager. If the transaction is aborted, the

Copyright © 2011 Microsoft Corporation.

Receive a Message	Send Message to Queue	Send Message in Transaction
(section 3.3.4.7)		messages are not accepted by the queue manager. If the Send Message transaction was successful and the message was successfully received, the messages are delivered to the application and removed from the queue. If the receive transaction was aborted, the messages are not removed from the queue.

Success Guarantee: On success, the messages generated by the sending application are delivered to the receiving application.

Trigger: The direct actor triggers this use case based on the actions of the primary actor.

Main Success Scenario:

The steps involved in this use case are:

1. On the sending application side, the use case Send Message to Queue – Application (or Send Message in Transaction – Application if transaction is used) is triggered to send messages to a queue. This is repeated when the sending application has more messages to send.
2. On the receiving application side, the use case Receive Message from a Queue – Application (or Receive Message in Transaction – Application if transaction is used) is triggered to receive the messages. This is repeated when the receiving application needs more messages from the sending application.

Upon successful completion, this use case meets the following postcondition:

- The receiving application receives the messages that the sending application intends to send.

Extensions: None

Copyright © 2011 Microsoft Corporation.

4 System Context

This section describes the relationship between this system and its environment.

4.1 System Environment

The Message Queuing System is designed to provide asynchronous messaging capabilities to applications across a heterogeneous networking environment. Various parts and pieces of the Message Queuing System reside on each participating machine in which the system operates. Furthermore, the Message Queuing applications are installed or located on remote machines. This requires the distributed components of the Message Queuing System to be reachable by one another and also by the applications that require Message Queuing functionality.

Applications MUST have the client side implementation of the application protocols in order to participate in the Message Queuing system. To support the queue manager roles defined in section 5.2.1.2, each participating machine of a Message Queuing System MUST have an instance of a queue manager. The queue manager maintains its state and data across system restarts, which requires the use of a persistent storage mechanism. The queue manager MUST have access to a secure and reliable file system that can be used to persist state and data in an implementation-specific manner.

The Message Queuing System shares common information across various queue manager instances and applications located on remote machines. The common information is maintained in a directory that is accessible to other Message Queuing components and applications. The Directory Service and its relation with the Message Queuing System is described in section 4.3.1.3. In order to use the directory, the participating Message Queuing machines MUST be domain-joined. Versions 1 and 2 of the Message Queuing System use MQDS (as specified in [MS-MQDS]) for communicating with the directory, whereas latter versions use Active Directory through the Lightweight Directory Access Protocol (LDAP), as specified in [RFC3377]. If the MQDS protocol is used in a Message Queuing deployment, the MSMQ Directory Service server MUST be running on a domain controller. The MSMQ Directory Service server is described in section 5.2.1.3.

The Message Queuing System provides transactional messaging capabilities by supporting transactional queues. The system MUST have access to a Transaction Coordinator system in order to provide such capabilities. The Transaction Coordinator uses the queue manager as a resource manager, as specified in [MS-DTCO] (section 1.3.3.2). Transactional messaging in the Message Queuing System is described in section 4.3.1.2.

The Message Queuing System uses the Identity, Authentication, and Authorization services of the Windows operating system, as described in [MS-WSO]. For providing message layer security, a public key infrastructure model, as specified in [SP800-32] and in [MSFT-PKI], MUST be available. The security aspects of the Message Queuing System are described in section 7.

4.2 System Assumptions and Preconditions

The following assumptions and preconditions MUST be satisfied in order for the Message Queuing System to start operation successfully:

- The Message Queuing System MUST be installed on the participating machines and the queue manager, if any, on each machine MUST be initialized.
- The storage devices configured for the system MUST be available to the system and have enough space to store the system state and data.

Copyright © 2011 Microsoft Corporation.

- The transport protocols used by the Message Queuing System on the participating machines MUST be available and fully initialized.
- If transactions are used, a Transaction Coordinator MUST be available and fully initialized.
- Security package providers MUST be available to the system.
- The queue manager MUST possess valid security credentials suitable for authentication.

The following additional assumptions and preconditions MUST be satisfied if the Message Queuing System is operating in Directory-Integrated mode as specified in section 4.3.1.3:

- At least one domain controller MUST exist for the domain.
- Each machine MUST be joined to the domain.
- A Directory Service MUST be initialized and available. If versions 1 or 2 of the Message Queuing System are used, the MSMQ Directory Service Server MUST be initialized and available on a domain controller.
- Appropriate Directory Service objects MUST exist and MUST be configured correctly. Details of these directory objects are specified in [MS-MQDS] and [MS-MQDSSM].

4.3 System Relationships

This section describes the relationships between the Message Queuing System and external components, system dependencies, and influences.

4.3.1 Black Box Relationship Diagram

This section describes the externally visible view of the Message Queuing System and the components within the system.

The following figure expands on the figure "Participating components in a Message Queuing System" from section 2.1 and illustrates the various internal and external components of the Message Queuing System.

Copyright © 2011 Microsoft Corporation.

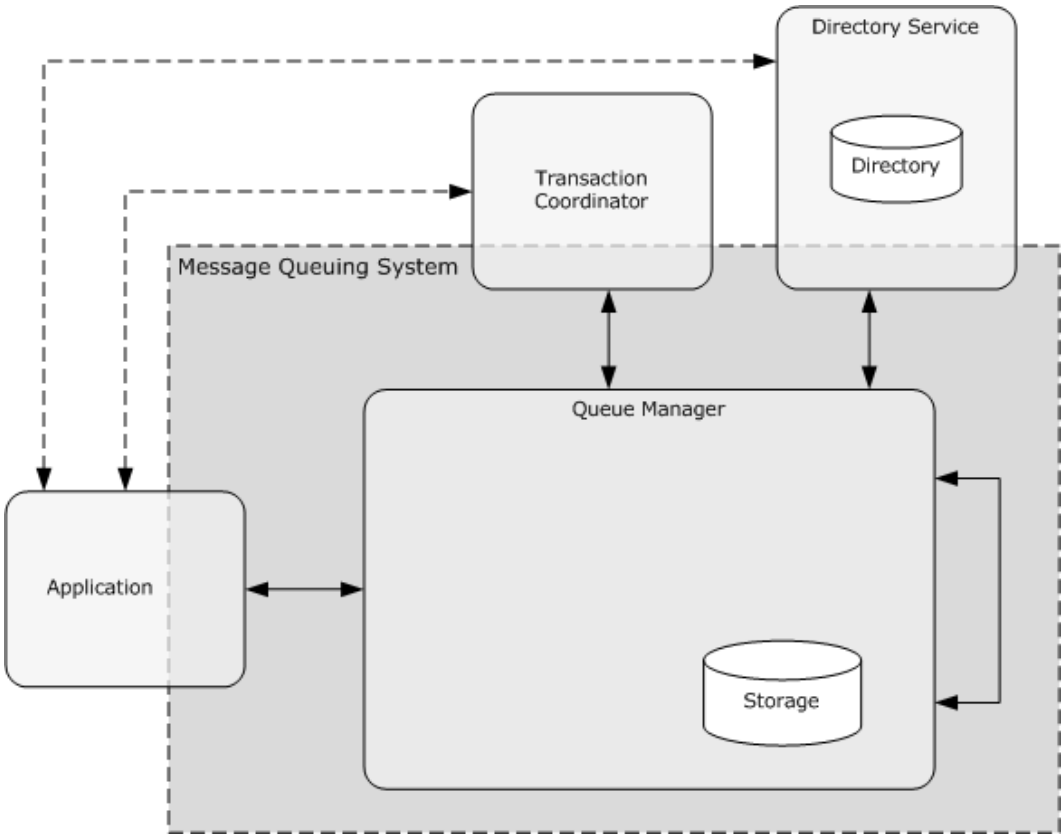


Figure 8: Black box relationships in a Message Queuing System

The queue manager interacts with its internal storage subsystem for persistence of its configuration data, application data (namely, queues and nonvolatile messages), and state. The exact storage mechanism used is an implementation detail of a particular Message Queuing System; however, it MUST retain the persistent data and state across system restarts. The queue manager interacts with other queue managers in the Message Queuing System.

The queue manager optionally interacts with an external Transaction Coordinator to provide transactional capabilities while sending or receiving messages to or from individual queues. Applications may create an external transaction from the Transaction Coordinator and pass it on to Message Queuing to atomically perform certain messaging operations. The external Transaction Coordinator functionality is specified in [MS-DTCO].

The Message Queuing System optionally uses a directory through a Directory Service component. The directory stores and provides information such as network topology, security key distribution, queue and system metadata, and queue discovery. For more information about the Directory Service, see section 4.3.1.3.

The following subsections describe the relationship of the queue manager with the preceding subcomponents.

4.3.1.1 Message Queuing and Applications

Applications use the application protocols of the Message Queuing System to accomplish asynchronous messaging functionalities. An application typically interacts with one queue manager.

Copyright © 2011 Microsoft Corporation.

A queue manager MUST implement the **supporting server** role or the **queue server** role, or both, in order to communicate with applications seeking message exchange functionality.

A queue Manager acts as a **supporting Server** by implementing the server side of **MQMP** and thereby providing a subset of the Message Queuing functionality, as described in [MS-MQMP] section 1. An application interacts with the supporting server through the client side of MQMP. This deployment mode is illustrated in the following figure.

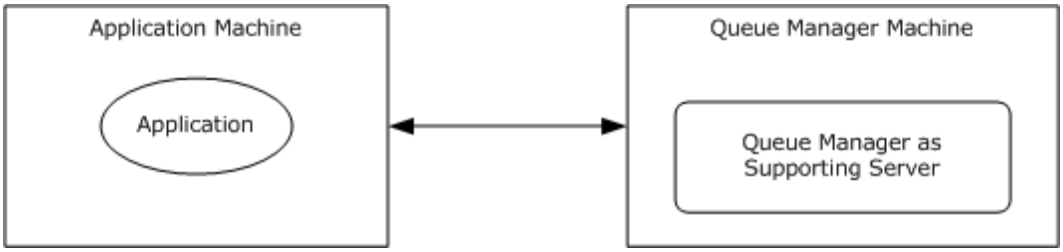


Figure 9: Queue manager as Supporting Server

Alternatively, a queue manager acts as a queue server by implementing the server side of **MQAC** and both the client and server sides of **MQQP** or **MQRR** protocols, thereby providing the full set of message exchange functionality of the Message Queuing system. An application interacts with the queue server through the client sides of the MQAC protocol. This deployment mode is illustrated in the following diagram.

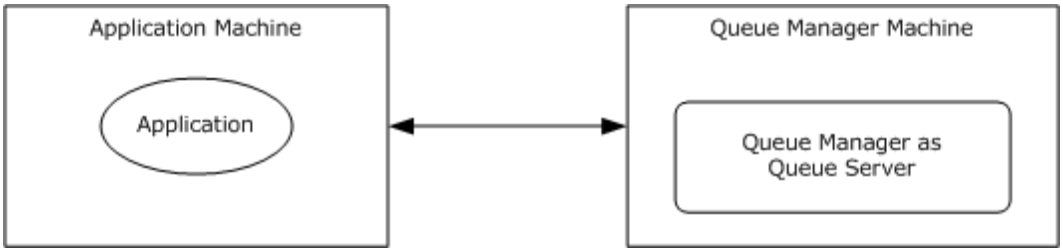


Figure 10: Queue manager as Queue Server

A Queue Manager acts as a **Management Server** by implementing the server side of **MQMR** and the management interfaces of the MQAC protocol, as well as the client and server side of the **MQCN** protocol, thereby providing the management and administrative capabilities of the Message Queuing system. An application interacts with the Management Server through the client side of the MQMR protocol and the client side of the management interfaces of the MQAC protocol. This deployment mode is illustrated in the following diagram.

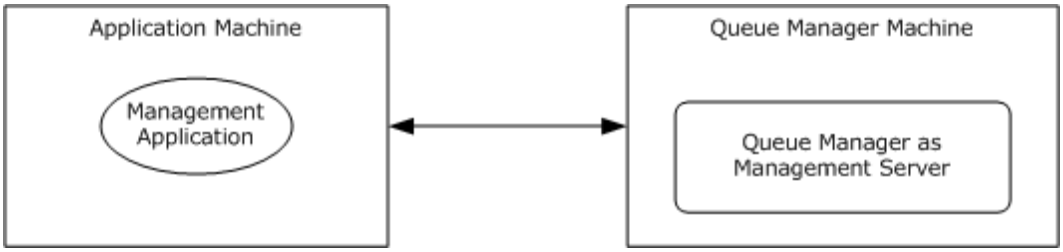


Figure 11: Queue manager as Management Server

The Message Queuing roles for application interaction are described in details in section 5.2.1.1.

Copyright © 2011 Microsoft Corporation.

4.3.1.2 Reliable Message Processing Using Transactions

As specified in section 3.2, Message Queuing supports end-to-end exactly-once delivery assurance through transactional messages, as opposed to best-effort delivery assurance through express or recoverable messages. The Message Queuing System interacts with a Transaction Coordinator to support transactional messaging. This section elaborates on the mechanism by which transactions are used together with exactly-once message delivery assurance to facilitate end-to-end reliable message processing.

The Message Queuing system provides transactional semantics by defining explicit transaction boundaries in end-to-end messaging and also by defining the error-handling/compensation semantics. This functionality is achieved through the following steps:

1. An application sends a message to a transactional queue as part of a transaction.
2. If the destination queue is hosted on a remote machine with respect to the sending queue, the message is reliably transferred to the remote queue by using the message transfer and routing protocols as described in section 5.3.2.2.
3. The receiving application receives and processes the message as part of another transaction.

In order to manage the queue as a transactional resource, the Queue Managers for both the sending side and the receiving side MUST implement the **resource manager** role as specified in [MS-DTCO] section 1.3.3.2. As part of the resource manager role, each Queue Manager MUST participate with the resource manager **facet** of a **Transaction Manager** coordinated by an external Transaction Coordinator, as specified in [MS-DTCO] section 1.3.3.3.

The participating applications MUST fulfill the Application role as specified in [MS-DTCO] section 1.3.3.1. The sending application initiates and completes the transactions within the message send boundary. The receiving application initiates and completes the transaction within the message receipt boundary.

The transactional semantics, illustrated in the figure which follows, includes the following steps:

1. Transacted Send:
    1. The sending application creates a transaction and performs certain application-specific tasks on the transaction.
    2. Subsequently, the application sends the message in the transaction.
    3. The Resource Manager facet of the Queue Manager enlists in the transaction.
    4. The Queue Manager enqueues the message and makes this change invisible outside the scope of the transaction.
    5. Subsequently, the application commits the transaction.
    6. The Transaction Coordinator coordinates the transaction with all the resource managers enlisted in the transaction, including the queue manager. For simplicity, the Prepare and Commit steps of a two-phase commit have been illustrated as a single TxCommit message in figure 12.
    7. The queue manager participates in the two-phase commit with the Transaction Coordinator, and upon a commit outcome of the transaction makes the message visible. In case of an abort outcome of the transaction, the message is discarded.

Copyright © 2011 Microsoft Corporation.

The transaction boundary on the send side ends here.

2. **Reliable Transfer:**

   If the destination queue is hosted by a different Queue Manager than the one accepting the original message, the sending Queue Manager reliably transfers the message to the receiving Queue Manager. If the destination queue is hosted by the same Queue Manager, this step is not required.

   1. The Queue Manager uses the **MQQB** or MQSRM transfer protocols to transfer the message to the destination queue. This operation is retried until an acknowledgment is received from the destination Queue Manager.

   2. The destination queue manager receives and stores the message, and sends a transfer acknowledgement.

   3. Upon receipt of a transfer acknowledgment from the destination Queue Manager, the source queue manager removes the message from the source queue.

3. **Transacted Receive:**

   The receiving application creates a transaction and performs certain application-specific tasks on the transaction.

   1. Once the message is available on the destination queue, the receiving application can proceed with consuming the message within the scope of the transaction. The receiving application creates a transaction.

   2. The application receives the message within the scope of the transaction.

   3. The resource manager facet of the receiving queue manager enlists in the transaction.

   4. The receiving queue manager makes the message invisible outside the context of the transaction.

   5. The receiving queue manager returns the message to the application.

   6. Subsequently, the application processes the message and commits the transaction.

   7. The Transaction Coordinator coordinates the transaction with all the resource managers enlisted in the transaction, including the queue manager. For simplicity, the Prepare and Commit steps of a two-phase commit have been illustrated as a single TxCommit message in the figure below.

   8. The receiving queue manager participates in two-phase commit with the Transaction Coordinator, and on a commit outcome of the transaction, deletes the message from the queue. In case of an abort outcome of the transaction, the message is returned back to the queue.

      The transactional boundary of the receiving application ends here.

Copyright © 2011 Microsoft Corporation.

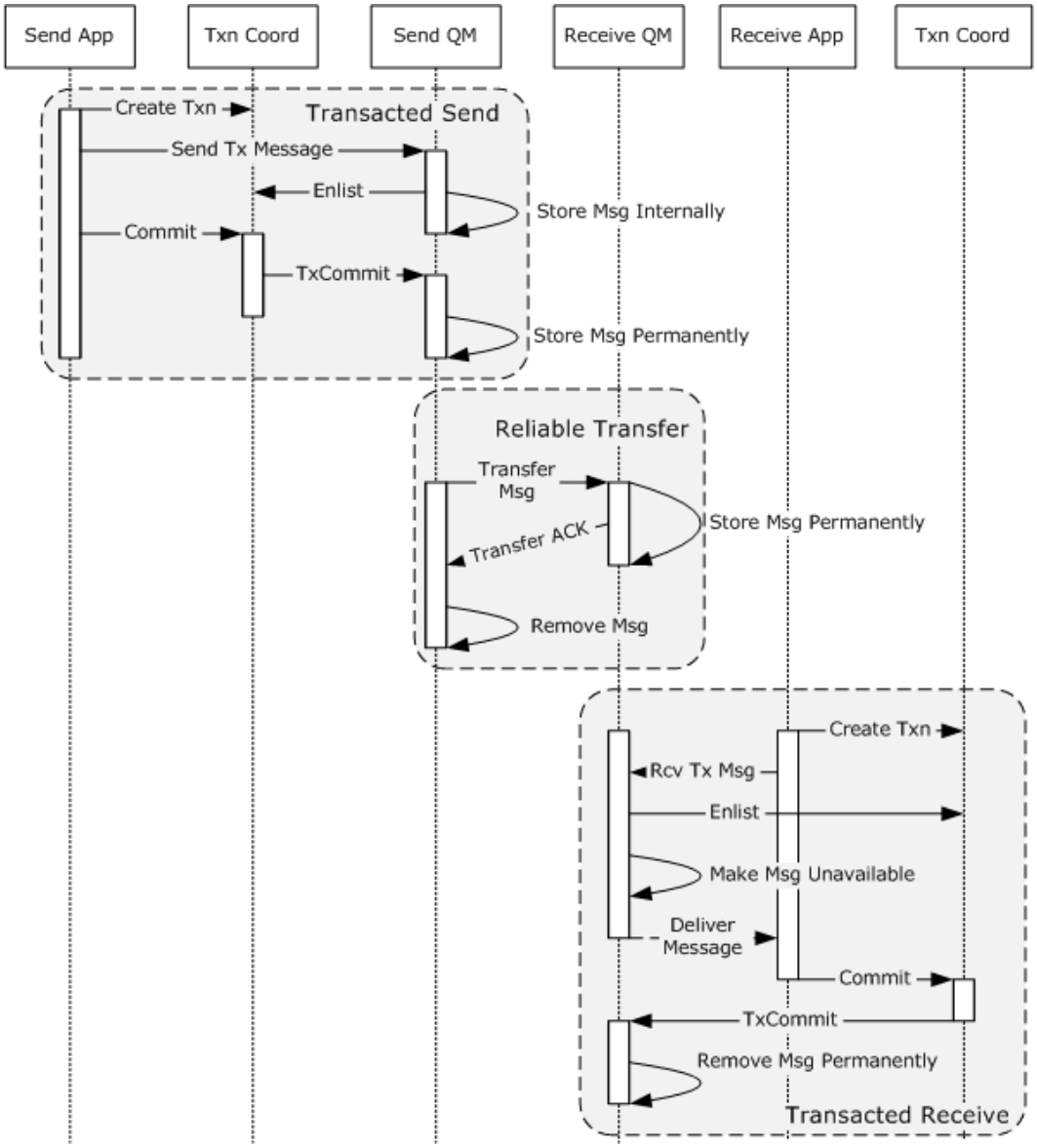


Figure 12: Transaction boundaries in end-to-end message exchange

The Message Queuing system also implements a lightweight internal Transaction Coordinator to avoid the overhead of **external transactions**. The Message Queuing system can dispense an **internal transaction** to an application, and this transaction can be used to coordinate actions only with the dispensing Queue Manager. Internal transactions can be used for sending messages to transactional queues, as long as no other external transacted resource, including other transactional queues of a different Queue Manager, is participating in the same internal transaction. The Queue Manager acts as a standalone Transaction Manager, and no external Transaction Coordinator or other Transaction Managers are involved when an internal Transaction is used. In an end-to-end message exchange, internal and external transactions can be mixed. Thus, a message can be sent using an internal transaction and received using an external transaction and vice versa.

Copyright © 2011 Microsoft Corporation.

As part of reliable message processing, the Message Queuing system also provides the error handling semantics through **negative source journaling** and Message Timers as specified in [MC-MQAC] section 3.1.2. If the message times out or a failure is encountered in delivering the message to the destination queue, the Queue Manager MUST place the message in the designated **dead-letter queue** for the sending application to perform application-specific error handling.

4.3.1.3 Message Queuing and Directory Service

The Message Queuing system optionally uses a Directory Service to enable a set of features pertaining to message security, efficient routing, and the publishing of queues, **distribution lists**, and queue aliases. The Directory provides Message Queuing with global storage and an access mechanism for Message Queuing system metadata that is shared by the entire Message Queuing system.

The Message Queuing system MUST operate in one of the predefined modes in terms of Directory Service integration: **Workgroup mode** or **Directory-Integrated mode**.

- Workgroup mode

  In this mode, Message Queuing does not use the Directory Service and is limited to **private queues** using **direct format name** addressing.

- Directory-Integrated mode

  In this mode, the Message Queuing system interacts with the Directory Service for publishing and accessing public queue metadata, network topology information, and certificate and encryption metadata. In this mode all Queue Managers participating in the system MUST be located on machines that are **domain**-joined.

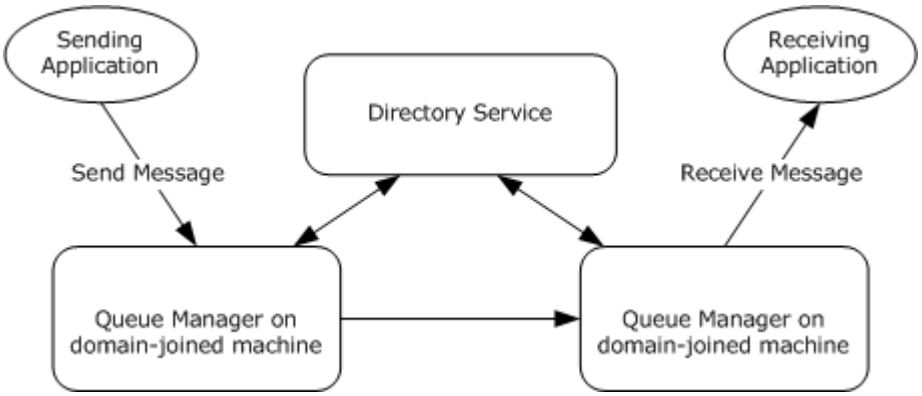


Figure 13: Message Queuing in Directory-Integrated mode

The Directory Service and its function within the Message Queuing system are specified in more detail in [MS-MQDS] section 1.3 and section 3 and [MC-MQAC] section 1.4. Versions 1 and 2 of the Message Queuing system implement their own Directory Service through the **MQDS** protocol. Subsequent versions of the Message Queuing system use Active Directory as the Directory Service through the Lightweight Directory Access Protocol (LDAP).

- Directory-Integrated mode with routing

  The Message Queuing system supports least-cost routing of messages within a computer network. This provision is particularly useful in complex network topologies—for example, where Message Queuing machines belong to different administrative regions. This is accomplished by

Copyright © 2011 Microsoft Corporation.

deploying the Message Queuing system in Directory-Integrated mode with routing enabled. In this mode, Message Queuing MUST retrieve the network topology information from the Directory Service and use this information for appropriate routing of messages from one Queue Manager to another. If a direct connection to the final destination Queue Manager is not possible, the source Queue Manager makes use of interim special queue managers known as **routing servers**, which can efficiently route messages to the next hop.

An overview of this deployment is specified in [MS-MQBR] section 1.3. The following diagram illustrates a typical deployment of Message Queuing with routing servers across a complex network topology.

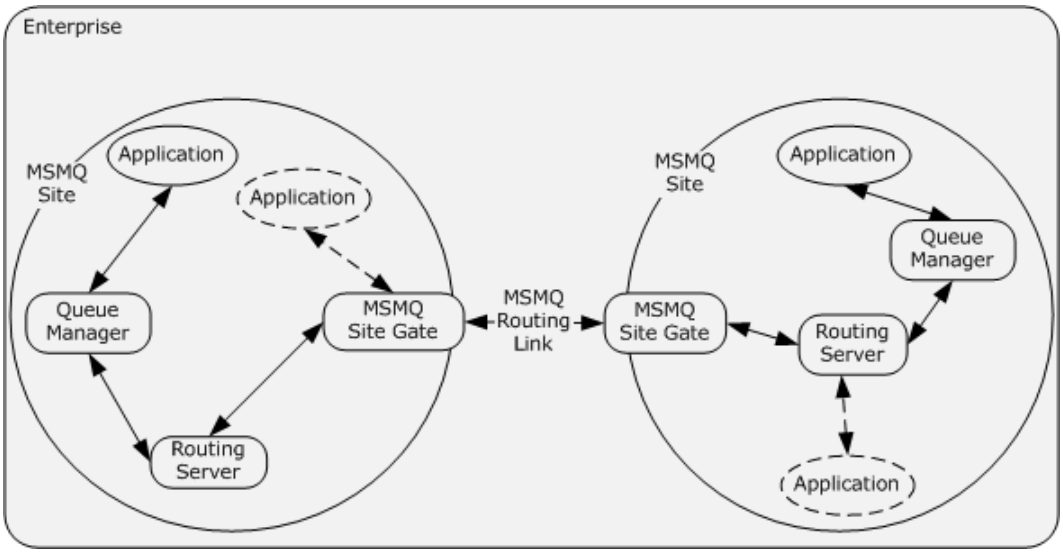


Figure 14: Enterprise deployment of Message Queuing with routing

The preceding figure illustrates Message Queuing deployment in an **enterprise** environment over a complex deployment topology. Message Queuing routing functionality allows applications deployed on one part of the enterprise network to communicate with remote applications deployed on another part of the enterprise in such a way that, although there is no direct network connectivity between these two applications, Message Queuing can use interim routing servers to transfer messages between the applications in a reliable way.

An **MSMQ site** is a network of computers, typically physically collocated, that have high connectivity as measured in terms of latency (low) and throughput (high).

An **MSMQ routing link** is a communication link between two **sites**.

A routing server is a Queue Manager role that enables Store-and-Forward messaging between computers within an MSMQ site.

An **MSMQ site gate** is a routing server that bridges two or more MSMQ sites such that all intersite messaging traffic flows through MSMQ site gates.

In an enterprise multisite deployment as shown in the previous illustration, a message flows in the following manner:

- The sender application sends the message to a source Queue Manager.
- The Queue Manager transfers this message to a routing server in the same MSMQ site.

Copyright © 2011 Microsoft Corporation.

- The routing server transfers the message to an MSMQ site gate.
- The source MSMQ site gate transfers the message to the destination MSMQ site gate through the MSMQ routing link.
- The destination site gate transfers the message to the destination Queue Manager, optionally through one or more routing servers.
- The destination Queue Manager places the message in the destination queue.

Because both site gates and routing servers are Queue Managers themselves, applications can also use them as Queue Managers. This is illustrated in the preceding figure through dotted association lines.

The routing server algorithm as specified in [MS-MQBR] section 3.1 is used to determine the least-cost message route between the source and destination Queue Managers.

4.3.2 System Dependencies

The following sections list the systems that use the interfaces provided by this system of protocols and other systems that depend on this system.

4.3.2.1 External Interfaces from This System

The following systems use the interfaces provided by the Message Queuing System:

- **Component Object Model Plus (COM+)** uses the Message Queuing System for implementing the COM+ Queued Components protocol as specified in [MC-COMQC].
- **RPC** Message Queuing MAY use the Message Queuing System as a transport for supporting asynchronous RPC calls.<1>
- **Windows Communication Foundation (WCF)** uses the Message Queuing System as a transport for supporting queued communication. WCF uses the Message Queuing primitives to provide additional message processing functionalities such as **poison message** handling. This functionality is supported by keeping track of the number of times an application attempted to process a particular message in a queue and when the number of such attempts exceeds a certain application-defined limit, moving the message from the queue to a separate designated queue such that the application does not receive the poison message anymore and can process other messages. See [MSDN-WCF] for more information.

4.3.2.2 Dependencies on Other Systems

The Message Queuing System does not have mandatory dependency on any external subsystem other than those called out in the individual protocol specifications, as well as the following:

- A durable storage system to persist state and data
- Various transports as specified in section 6.3
- Security infrastructure as specified in section 7

4.3.3 System Influences

The following components influence the overall functionality and capability of the Message Queuing System.

Copyright © 2011 Microsoft Corporation.

- Transaction Coordinator: The Message Queuing System interacts with a Transaction Coordinator via the protocol specified in [MS-DTCO] to provide support for distributed transactions. In the absence of an external Transaction Coordinator, the transactional operations on queues cannot be performed in the context of an external distributed transaction. However, as specified in section 5.2.1.3, the Message Queuing System implements an internal Transaction Coordinator, and these internal transactions can still be used for isolated send and receive operations on a transactional queue.
- Directory Service: The Message Queuing Systems communicate with the Directory Service via the MQDS or ADTS [MS-ADTS] protocols.<2> In the absence of a Directory Service, the basic message exchange functionalities are available, although a set of features as specified in section 4.3.1.3 are not available.

4.4 System Applicability

The usefulness of the Message Queuing System is limited to asynchronous messaging; therefore, it SHOULD NOT be used in the following messaging applications:

- Synchronous Messaging: The asynchronous nature of the Message Queuing message exchange patterns introduces latency in comparison with a direct, synchronous communication pattern. Therefore, the Message Queuing System is not suitable for applications that require message delivery within a short and predefined amount of time.
- Use a Queue as a Database: Unlike a database system, a queue is neither intended to be used as a long-term storage medium, nor intended to support a general query function. The queue SHOULD provide enough storage to enable temporal decoupling of applications and to deal with time intervals when the applications are offline or the network is out of service, regardless of the volume of traffic. The queue SHOULD NOT be used as a database.
- Traditional End-to-End Transaction Flow: The Message Queuing System does not provide true end-to-end transactional capability between the participating applications. Instead, Message Queuing supports local transactional boundaries, reliable message transfer, and well-defined failure handling semantics. Therefore, the Message Queuing System MUST NOT be used for atomic update between applications.

4.5 System Versioning and Capability Negotiation

There are multiple versions of the Message Queuing System.<3> A summary of different system versions and the protocols or protocol subsets implemented by these versions is given in the following table.

Message Queuing System versioning and capability negotiation

System version	Protocols implemented	Protocol subsets implemented
MSMQ 1.0	MQQB, MQBR, MQCN, MQQP, MQSD, MQMP, MQMR, MQDS	MQAC: Version 1.0 of the COM interfaces listed in [MC-MQAC] MQDS: Sections 3.1 and 3.2 listed in [MS-MQDS]
MSMQ 2.0	MQQB, MQBR, MQCN, MQQP, MQSD, MQMP, MQMR, MQDS	MQAC: Version 2.0 of the COM interfaces listed in [MC-MQAC]
MSMQ 3.0	MQQB, MQBR, MQCN, MQQP, MQSD, MQMP,	MQAC: Version 3.0 of the COM interfaces listed in [MC-MQAC]

Copyright © 2011 Microsoft Corporation.

System version	Protocols implemented	Protocol subsets implemented
	MQMR, SRMP	MQRR: Sections 3.1.4.1 through 3.1.4.9 inclusive from [MS-MQRR] MQMP: The client version<4> of MSMQ 3.0 provides only the client-side implementation. The server version<5> of MSMQ 3.0 provides client-side and server-side implementations. MQDS: Only the server side of MQDS is implemented to provide directory service to MSMQ 1.0 and 2.0 clients.
MSMQ 4.0	MQQB, MQBR, MQCN, MQQP, MQSD, MQMP, MQMR, SRMP, MQRR	MQAC: Version 4.0 of the COM interfaces listed in [MC-MQAC] MQDS: Only the server side of MQDS is implemented to provide directory service to MSMQ 1.0 and 2.0 clients.
MSMQ 5.0	MQQB, MQBR, MQCN, MQQP, MQSD, MQMP, MQMR, SRMP, MQRR	MQAC: Version 4.0 of the COM interfaces listed in [MC-MQAC]

Any deviations from a specific version's implementation of these protocol specifications are documented in the respective protocol documents. Abstracts for these protocols are given in section 2.2.

Capability negotiations between client and server implementations of these protocols are specified in the "Versioning and Capability Negotiation" sections in the respective protocol specifications.

4.6 System Vendor-Extensible Fields

This system has no vendor-extensible fields other than those specified in the protocol documents.

Copyright © 2011 Microsoft Corporation.

5 System Architecture

This section describes the basic structure of the system and the interrelationships among its parts, consumers, and dependencies.

5.1 Abstract Data Model

The abstract data model of Message Queuing includes the data model of the queue manager as well as the data model of the optional directory. The shared data model and processing rules common to all Message Queuing protocols are described in [MS-MQDMPR] section 3.1.1.

The following section provides a high-level description of the Message Queuing System abstract data model and describes the interaction between the components through this abstract data model.

5.1.1 Queue Manager Abstract Data Model

The following UML [UML] static class diagram illustrates the high-level abstract data model of the queue manager as viewed by the external entities.

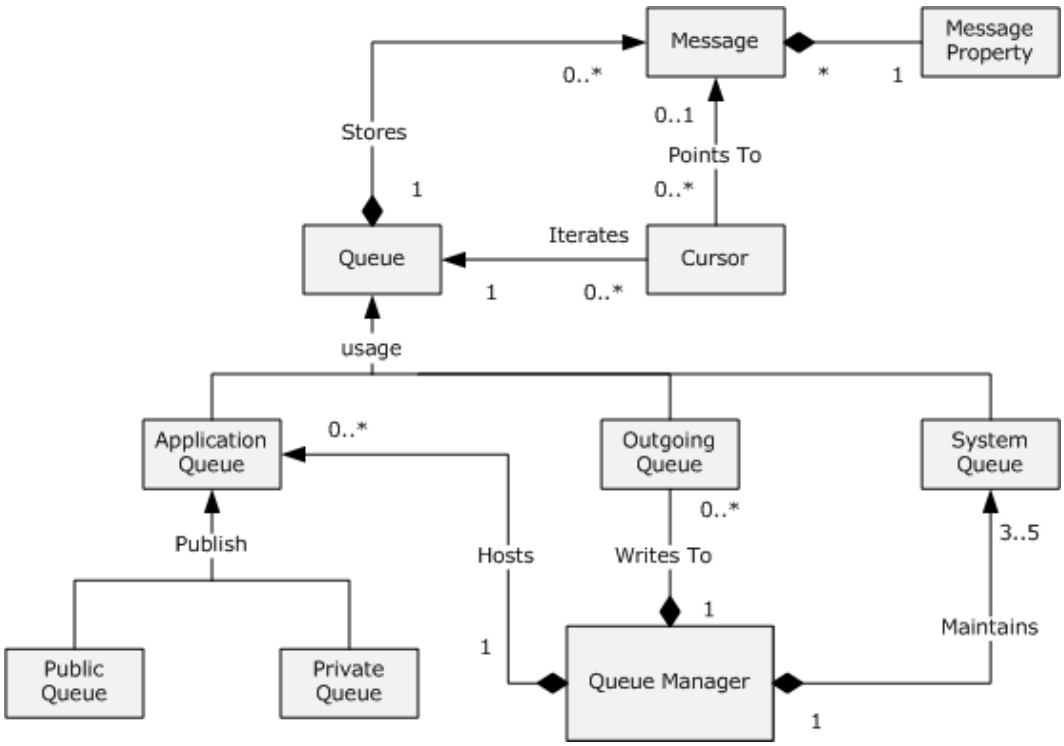


Figure 15: UML static class diagram for the Queue Manager

The queue manager represents a single instance of the Message Queuing System that is available on a machine with unique network addresses. The queue manager interacts with other queue manager instances. The queue manager manages creation, modification, deletion, and message exchange operations on a queue. Additionally, the queue manager MUST listen to and handle all incoming Message Queuing network traffic for the network addresses of the machine. The abstract data model of a queue manager is specified in [MS-MQDMPR] section 3.1.1.1.

Copyright © 2011 Microsoft Corporation.

A queue holds messages. The abstract data model of a queue is specified in [MS-MQDMPR] section 3.1.1.2.

A message is a unit of data transferred between applications through Message Queuing. A message has a body and other message properties that travel along with the message. The abstract data model of a message is specified in [MS-MQDMPR] section 3.1.1.12.

Applications may use a **cursor** to iterate over the messages in a queue. A cursor represents the position within the queue. Message Queuing does not preserve cursor state across restarts. The abstract data model for cursors can be found in [MS-MQDMPR] section 3.2.

5.1.1.1 Queue Manager States

The Queue Manager exhibits state transitions as shown in the following figure.

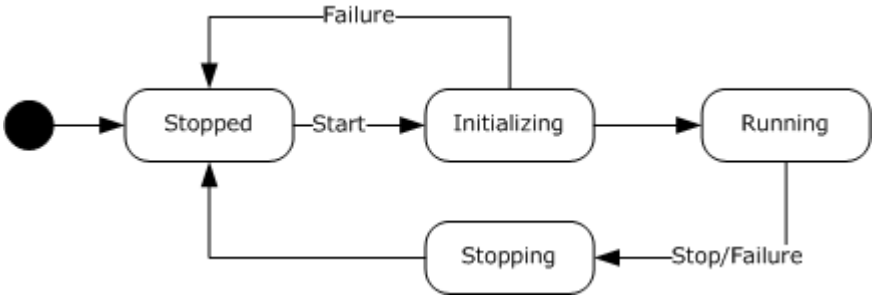


Figure 16: Queue manager state transition

The state of the queue manager corresponds to the QueueManager.QueueManagerState data element attribute as described in [MS-MQDMPR] section 3.1.1.1.

- **Stopped**: This is the initial state of the queue manager. The queue manager MUST enter this state if a failure event is raised in the subsequent initializing state. In the Stopped state, the queue manager does not perform any of its roles; however, it has all the necessary state information preserved in such a way that a start event can trigger the queue manager to transition to the Initializing state. The start event is an implementation-specific external event and is generated externally to start up the queue manager.

- **Initializing**: This is a transient state of the queue manager. The queue manager enters this state from the Stopped state upon receipt of a start event. In this state, the queue manager performs its own initialization. The queue manager also establishes necessary handshaking with other dependent components, such as the Directory Service and the Transaction Coordinator. The queue manager fetches its internal state and related data specific to its implementation. The queue manager initiates all protocol initializations for its supported protocols. On successful initialization, the queue manager transitions to the Running state. The queue manager MUST transition to the Stopped state if a failure event is generated. A failure event can be generated if any incoherency is detected during the initialization process, or due to the malfunctioning of any external subcomponent, or if a protocol initialization fails. On failure, the start event SHOULD be periodically generated as per the following retry mechanism, to retry the queue manager initialization. Although the Message Queuing System does not mandate any specific periodic reinitialization, and any such retry mechanism is entirely an implementation choice, this retry mechanism ensures that a Queue Manager can automatically start and participate in a Message Queuing system when no transient failure conditions preventing a prior initialization exist:

  1. If a failure occurs during initialization, wait for 120 seconds and retry.

Copyright © 2011 Microsoft Corporation.

2. If the retry operation in step 1 fails, wait for 900 seconds and retry. Continue to retry every 900 seconds until the initialization sequence completes successfully.

- **Running**: Also known as the Ready state, this is the fully functional state of the queue manager, and the queue manager performs all its roles and operations, including sending and receiving messages to and from applications; accepting, transferring, and routing messages to and from other queue managers; and performing management roles. On receipt of a stop or failure event, the queue manager MUST transition to the Stopping state. The stop event is a protocol-independent, implementation-specific external event and is generated externally to shut down the queue manager. The failure event is a protocol-independent, implementation-specific event and is generated if any unexpected and fatal failure condition is encountered, such as hardware failure or exhaustion of implementation-specific resources. Within the Running state, the queue manager maintains the following abstract data model (ADM) variables, each of which represents a constrained mode of operation of the queue manager and constitute a specific substate of the queue manager that is independent of the other substates:

  - **Queue Manager.Throttled**: This variable is manipulated during processing of the Begin Flow Control event described in [MS-MQDMPR] section 3.1.7.2.5 and the End Flow Control event described in [MS-MQDMPR] section 3.1.7.2.6. When this variable is set to TRUE, the queue manager MUST stop accepting messages from any source and only begin accepting new messages again after this value is set to FALSE. This variable SHOULD<6> be used for throttling incoming messages during low resource situations such as low memory on the machine or limited storage capacity. When the low resource condition is true, the Message Queuing System raises the Begin Flow Control event described in [MS-MQDMPR] section 3.1.7.2.5 When the condition becomes normal, the Message Queuing System raises the End Flow Control event described in [MS-MQDMPR] section 3.1.7.2.6.

  - **QueueManager.ConnectionActive**: This variable is manipulated during processing of the Take Offline event described in [MS-MQDMPR] section 3.1.4.12 and the Bring Online event described in [MS-MQDMPR] section 3.1.4.13. When this variable is set to FALSE, the queue manager restricts its network activities and does not receive messages from remote queue managers until this value is set to TRUE.

  - **QueueManager.DirectoryOffline**: This ADM variable is maintained when the queue manager is operating in Directory-Integrated mode; that is, the QueueManager.DirectoryIntegrated ADM variable is set to TRUE. This variable is manipulated during processing of the Create Directory Object, Delete Directory Object, Read Directory, Write Directory and the Check Directory Online events described in [MS-MQDMPR] sections 3.1.7.1.18, 3.1.7.1.19, 3.1.7.1.20, 3.1.7.1.24 and 3.1.7.1.25, respectively. When the queue manager is unable to communicate with the Directory Service, it MUST set this variable to TRUE. When this value is TRUE, the queue manager MUST continue to accept messages from applications and place them in the outgoing queues but MUST NOT attempt to transfer the messages to other queue managers until Directory Service connectivity is restored and this variable is set to FALSE.

- **Stopping**: This is a transient state in which the queue manager stops its normal operations and starts a graceful shutdown. The queue manager MUST transition to the Stopping state when it receives a stop or failure event. In this state, the queue manager stops all of its protocol operations, shuts down its subsystems, and disconnects from any dependent resources. It then transitions to the Stopped state.

5.1.1.2 Queues and Queue States

Queues hold messages and maintain their ordering. Messages are stored in a queue in a first-in-first-out order.

Copyright © 2011 Microsoft Corporation.

Queues can be subclassed into three broad categories based on their uses. These are application queue, **outgoing queue**, and system queue.

- **Application Queue**

  The most common type of queue is an application queue. They are created by applications, and their life cycles are managed solely by the applications. Applications can send messages to or receive messages from application queues.

  Application queues can be further subcategorized based on whether they are published to the Directory. Queues that are published to the Directory through the Directory Service are called public queues. Metadata for the public queue is maintained by the directory. Other applications can discover public queues by browsing the directory.

  Alternatively, a queue can be defined locally without being published to the directory. This category is known as a private queue. An application uses the actual name and location of a private queue to send and receive messages. Other than publishing the queue information in the directory, there is no difference between a public queue and a private queue.

  The state transition of an application queue is illustrated in the following diagram.

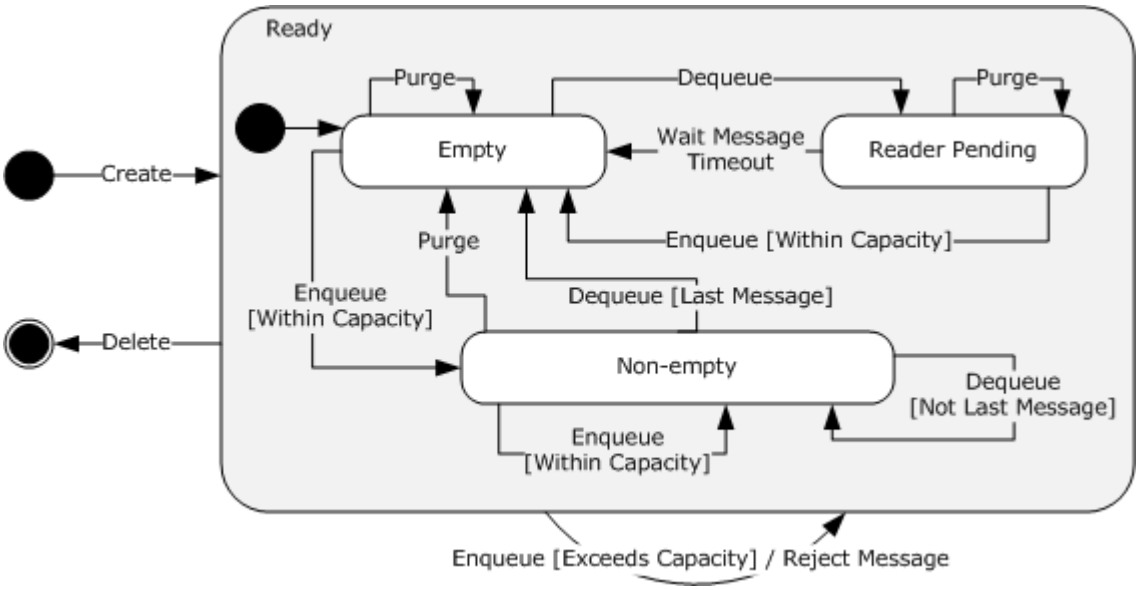


Figure 17: Application queue state transition

An application queue is created by an application through the create event. The create event corresponds to the Create Queue event described in [MS-MQDMPR] section 3.1.7.1.3. The application queue has a single composite state named Ready, which means that the queue can be used for normal send or receive operations. The Ready state has three substates: Empty, Reader Pending, and Non-empty. The Empty state is the state of the queue when it contains no messages and no pending readers. The Reader Pending state is the state of the queue when it contains no messages but has pending readers. The Non-empty state indicates that the queue has at least one message ready for consumption.

An enqueue event is generated when an incoming message arrives for a queue. The enqueue event corresponds to the Enqueue Message event described in [MS-MQDMPR] section 3.1.7.1.9. If the queue does not have the capacity to enqueue a new message, the message is rejected. If

Copyright © 2011 Microsoft Corporation.

the queue is in the Empty substate, the enqueue operation transitions the queue to the Non-empty state. If the queue is in the Reader Pending substate, the enqueue operation transitions the queue to the Empty state. Similarly, a dequeue event removes a message from the queue if one is available. The dequeue event corresponds to the Dequeue Message event described in [MS-MQDMPR] section 3.1.7.1.10. If the dequeue event occurs on the last message, the queue transitions to the Empty substate. If the dequeue event occurs on an empty queue, the queue transitions to the Reader Pending substate. The queue transitions back to the Empty substate when an enqueue event or a wait message timeout event occurs. The wait message timeout event corresponds to the Waiting Message Read Timer Expired event, as described in [MS-MQDMPR] section 3.1.7.3.5, which occurs when the dequeue operation times out or gets canceled.

A purge event removes all messages from the application queue. The purge event corresponds to the Purge Queue event described in [MS-MQDMPR] section 3.1.7.1.7. If the purge event is generated when the queue is in the Non-empty substate, the queue transitions to the Empty substate. Otherwise, no action is taken and the queue state remains the same.

On a delete event, the application queue and all of its associated messages are permanently removed from the queue manager. The delete event corresponds to the Delete Queue event described in [MS-MQDMPR] section 3.1.7.1.4.

Applications can receive or peek messages from a queue using a message iterator called a cursor. An application can create a cursor for a queue. The cursor points to a message in the queue, and the pointer can be moved through cursor operations. The queue manager maintains the pointer of each cursor with respect to its associated queue. Cursors are volatile and are therefore discarded on queue manager restart. The ADM attributes and the state diagram of a cursor are described in [MS-MQDMPR] section 3.2.1.

- **Outgoing Queue**

  The queue manager creates a temporary queue for outbound messages that are destined for a queue on another queue manager. Such a queue is called an outgoing queue. For each unique destination queue, the queue manager creates a separate outgoing queue and attempts to transfer the messages from the outgoing queue to the destination queue.

  The state transitions for the outgoing queue are shown in the following figure.

Copyright © 2011 Microsoft Corporation.

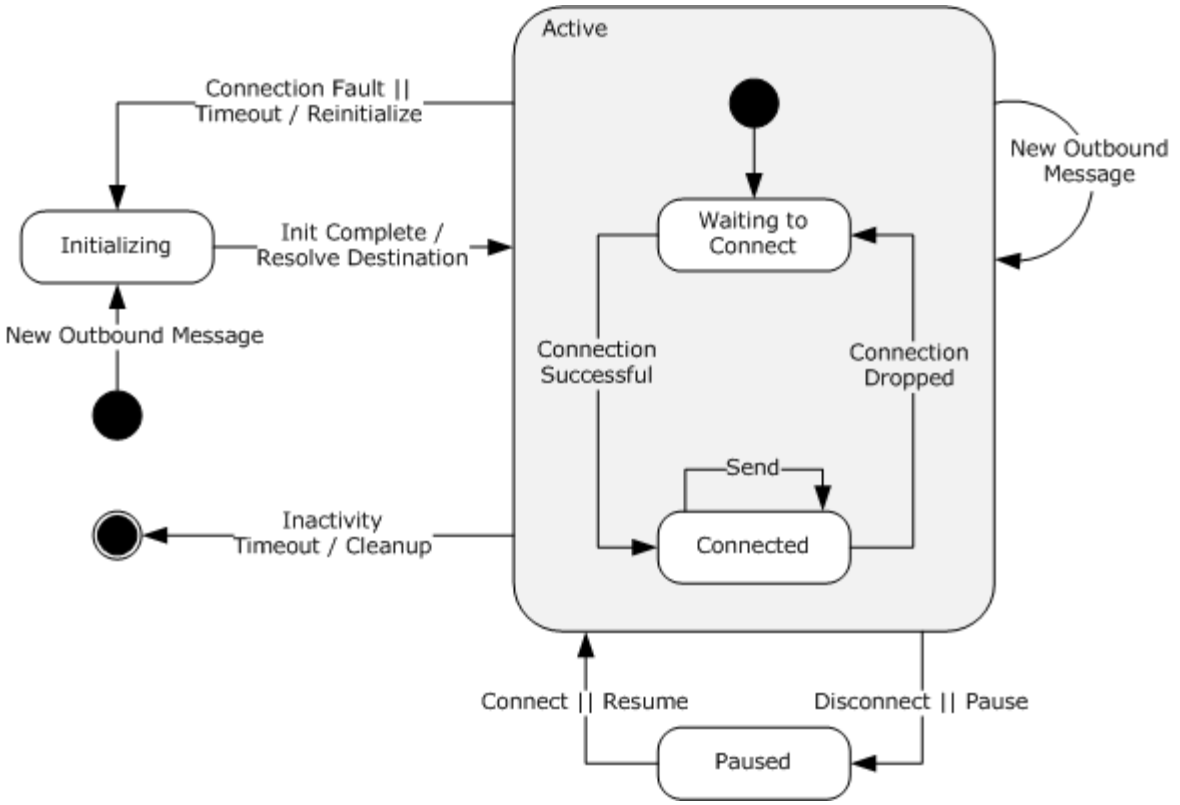


Figure 18: Outgoing Queue state transition

The outgoing queue starts in the Initializing state. The session initialization happens in this state; the corresponding protocol state is specified in [MS-MQQB] section 3.1.1.1.1 and [MC-MQSRM] section 3.1.1.1. In this state, the queue manager resolves the address of the destination machine. On receipt of the init complete event indicating successful completion of the initialization, the queue enters the Active state.

The Active state is a composite state, which represents the fact that the outgoing queue is either establishing a connection or sending messages to the destination queue manager. Arrival of new outbound messages while in the Active state does not cause a state transition. On receipt of an internal implementation-specific inactivity timeout event, the outgoing queue cleans up and destroys itself. If a connect fault event is raised, or if there is a timeout, the outgoing queue transitions to the Initializing state.

- The Waiting to Connect substate represents the conditions that exist when the outgoing queue attempts to connect to the destination queue manager. After the connection successful event is generated, the outgoing queue transitions to the Connected substate.
- The Connected substate represents the conditions that exist when the outgoing queue can transfer outstanding messages to the remote destination queue. If the connection drop event is generated while a queue is in the Connected substate, it transitions to the Waiting to Connect state.

The Paused state represents the condition that exists when the outgoing queue in an Active state responds to external disconnect or pause events. In this state, the outgoing queue remains idle.

Copyright © 2011 Microsoft Corporation.

On receipt of a subsequent connect or resume event, the outgoing queue transitions to the Active state and reestablishes the connection in order to resume the message transfer operation.

- **System Queue**

  The queue manager has some built-in queues called **system queues**. These system queues are protected for use only by the queue manager. Each system queue serves a specific purpose for the Message Queuing System protocols, described as follows:

  - **Source Journal Queue**: A placeholder queue used by the source queue manager to hold all messages for which an application has requested **positive source journaling**.
  - **Dead-Letter Queue**: A queue that contains messages that could not be delivered, and negative source journaling is requested. There is a dead-letter queue for transactional messages and a dead-letter queue for non-transactional messages.
  - **OrderAck Queue**: An internal queue that is used by the message transfer protocol MQQB and **MQSRM** to implement exactly-once delivery assurance.
  - **Notification Queue**: An internal queue that is used by the MQCN protocol for communicating change notifications to queue managers.
  - **Connector Queue**: A temporary queue to store messages that are forwarded to foreign queues.

The states and state transitions of the system queues are the same as those of the application queues.

5.1.1.3 Messages and Message States

A message is a unit of data transfer between applications and the Message Queuing System, or between Message Queuing subcomponents. For a detailed composition of the message, refer to the **Message** ([MS-MQDMPR] section 3.1.1.12) ADM element.

An application sends a message to a queue. After the message reaches the destination queue, another application can receive the message and process it. Similarly, the queue manager or the Directory Service can send internal messages to other queue managers. The message undergoes several state changes in between. The following figure describes the various states and the inter-state transitions of a message, as visible from an application's perspective.

Copyright © 2011 Microsoft Corporation.

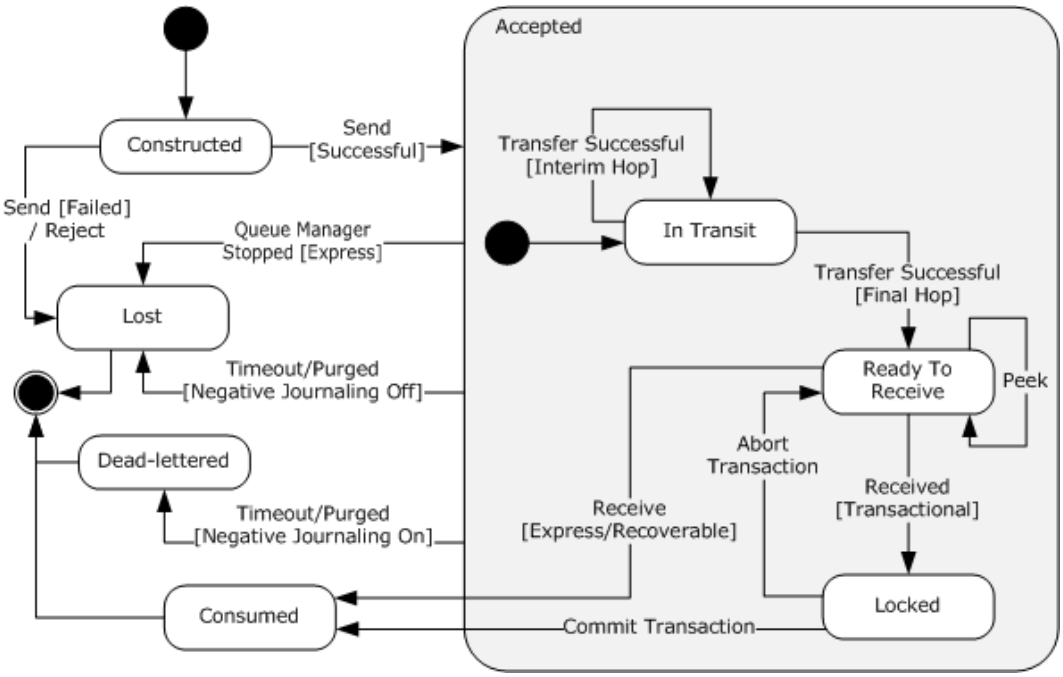


Figure 19: Message state transition

- Constructed: This is the initial state of the message; it represents that the message has been constructed and is ready to be handed off to the queue manager using one of the available protocols. The sender MUST generate a send event in order for the message state to transition to the Accepted state (if the send operation succeeds) or transition to the Lost state (if the send operation fails). The send event corresponds to the Enqueue Message event described in [MS-MQDMPR] section 3.1.7.1.9.

- Accepted: This is a composite state representing the fact that the queue manager has accepted ownership of the message through a successful send event. For express messages, if the Queue Manager is stopped, as described in the Queue Manager Stopped event in [MS-MQDMPR] section 3.1.4.2, the messages enter the Lost state, denoting that the messages are lost from the Message Queuing system. Each message has two timers associated with it, as described in section 6.4.2. When either of these timers expire, a timeout event is generated to discard the message. For messages with negative journaling disabled, i.e. the Message.NegativeJournalingRequested element is FALSE as described in [MS-MQDMPR] section 3.1.1.12, a timeout event causes the message to enter the Lost state. For messages with negative journaling enabled, a timeout event causes the message to enter the Dead-lettered state. The Accepted state has the following substates:

  - In Transit: During this state, the message is placed in an outgoing queue of the queue manager and is transferred to the destination queue. If the destination queue is not directly accessible from the source machine, the message is sent to the queue manager of the next hop, until it reaches the final destination queue. A transfer successful event on the final hop transitions the message to the Ready to Receive state. The transfer successful event is generated when all the acknlowledgements as specified in [MS-MQQB] section 3.1.1.6 are received by the sending machine. For express and recoverable messages, the receipt of the Session Acknowledgement, as described in [MS-MQQB] section 3.1.1.6.2 and 3.1.5.5, marks a successful transfer. For transactional messages, the receipt of the Transactional

Copyright © 2011 Microsoft Corporation.

Acknowledgement, as described in [MS-MQQB] section 3.1.1.6.1 and 3.1.5.6, marks a successful transfer.

- Ready to Receive: This state represents that the message is in the destination queue. The message is now available to the receiving application. A peek event allows the receiving application to perform a nondestructive read on the message without changing its state or removing it from the queue. The peek event corresponds to the Peek Message event as described in [MS-MQDMPR] section 3.1.7.1.15. Messages can be destructively received through the receive event that corresponds to the Dequeue Message event described in [MS-MQDMPR] section 3.1.7.1.10. For transactional messages, a transaction is used and the receive event transitions the message to a Locked state so that no other application can receive or peek the message. For express and recoverable messages, the receive event removes the message from the Message Queuing system and transitions the message to the Consumed state.

- Locked: This state indicates that an application is preparing to exclusively receive the message and the message is not visible to any other application. As described in [MS-MQDMPR] sections 3.1.1.11 and 3.1.1.12, this state is denoted by setting the MessagePosition.State element of the Message.MessagePositionReference element to Locked. If the transaction is aborted, the message transitions back to the Ready to Receive state. If the transaction is committed, the message is deemed consumed and it message-transitions out of the Message Queuing system into the Consumed state.

- Consumed: This state represents that the message has been received by the receiving application and that the message has exited the Message Queuing System.

- Lost: This state represents that the Message Queuing System could not deliver the message to the receiving application and that the message has been lost. Express messages reach this state if a Queue Manager Stopped event is triggered. All messages reach this state if the initial send to the queue manager fails and the queue manager is therefore unable to accept the message. If Negative Journaling is disabled and the messages time out before they are successfully delivered to the destination, messages enter the lost state.

- Dead-lettered: This state represents that the Message Queuing System did not deliver the message before the message timers, as described in section 6.4.2, expired. Messages are dead-lettered only if Message.NegativeJournalingRequested is set to TRUE.

5.1.2 Directory Abstract Data Model

As specified in section 4.3.1.3, the Message Queuing System optionally operates in the Directory-Integrated mode by using a directory for storing metadata and configuration details. The Message Queuing System ADM elements that are stored in the directory are explicitly categorized as "directory attributes" in [MS-MQDMPR] section 3.1.1.

The following UML [UML] static class diagram illustrates the high-level abstract data model of the Directory as viewed by the external entities.

Copyright © 2011 Microsoft Corporation.

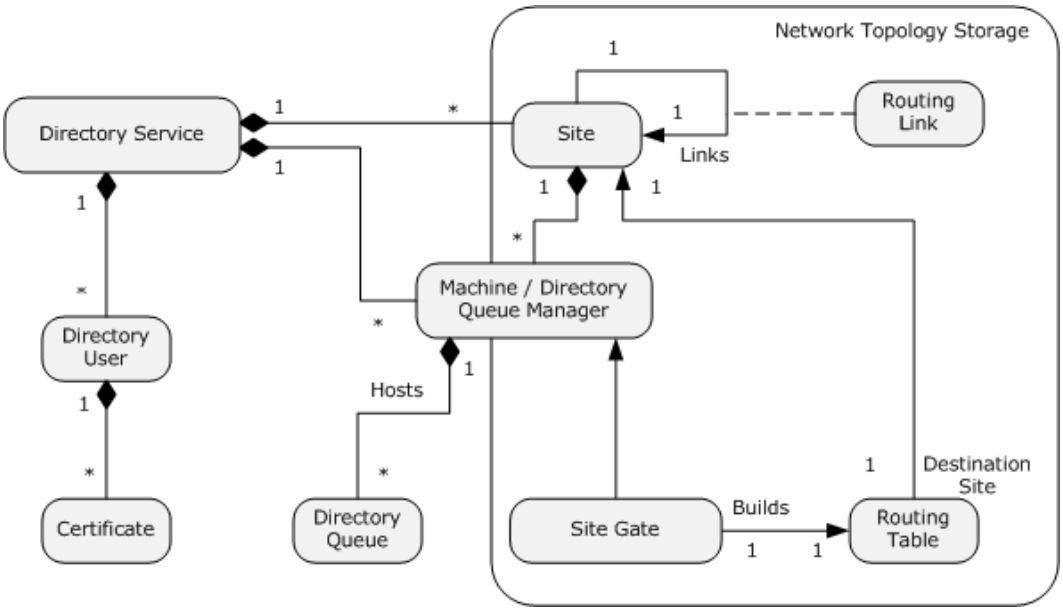


Figure 20: UML static class diagram for the Directory

The abstract data model of the directory objects is specified in detail in [MS-MQDS] section 3.1.1.

The Directory Service is a common object that is shared across the entire Message Queuing System and all the queue managers in different roles. This object is also represents the Enterprise data element of the directory as described in [MS-MQDMPR] section 3.1.1.6. The directory maintains the following categories of information:

- The directory maintains configuration data for each queue manager in the Message Queuing System. When a queue manager initializes in the Directory-Integrated mode, the configuration data of the queue manager gets stored in the directory as a Directory Queue Manager object under the Machine object. The elements of the Directory Queue Manager object are described in [MS-MQDMPR] section 3.1.1.1 as the "directory attributes" of the queue manager ADM.
- The queue metadata corresponding to each public queue of a specific queue manager is stored as a Directory Queue object under the Directory Queue Manager object that owns the public queue. Whenever public queue information in the directory gets updated, the owning queue manager is notified of the changes through MQCN.
- A message-queuing user is represented in the directory as a Directory User object. The Directory User object associates the cryptographic certificate information for a specific user that is used for message authentication purposes by the queue managers. For more information about these ADM elements, see [MS-MQDMPR] section 3.1.1.16.
- The Directory Service also maintains the network topology information for the enterprise, and the relevant pieces of the abstract data model are shown as the Network Topology Storage section of the figure. The queue managers MUST use this information to operate in the routing server and site gate roles to transfer messages across and within sites.
    - A site is a collection of machines with high connectivity within the network. The site is represented by a site object in the directory. The ADM elements of a site are described in [MS-MQDMPR] section 3.1.1.7.

Copyright © 2011 Microsoft Corporation.

- A **routing link** is a communication link between two sites. A routing link is represented by a **RoutingLink** object in the Directory Service. Routing links can have associated **link costs**. Routing links and their costs are used by the routing servers to compute the least-cost routing paths for store-and-forward machines. The ADM elements of a routing link are described in [MS-MQDMPR] section 3.1.1.8.
- A machine represents a computer in the enterprise network, within a site. A machine that resides on the boundary of a site is called a site gate. Site gate machines allow message transfer to neighboring sites.
- A site gate maintains a routing table. The routing table is dynamically constructed based on the sites, the routing links, and the link costs. Each entry of the routing table specifies the least-cost next-hop site for a given final destination site. The ADM elements of a routing table are described in [MS-MQBR] section 3.1.1.2.

5.2 White Box Relationships

This section describes the white box relationships between the system and the external entities. Section 5.2.1 describes the various roles exhibited by the Message Queuing System. Section 5.2.2 expands the black box relationships of the Message Queuing System as described in section 4.3.1, and describes the interactions between various roles of the Message Queuing System with the external entities.

5.2.1 Message Queuing System Roles

The Message Queuing System and its components exhibit the following roles to the interacting components of the system.

5.2.1.1 Application Roles

The following roles represent the external actors of the Message Queuing System:

- Application: The Application role represents the messaging actions performed by applications of the Message Queuing System. The Application role typically includes message sending and receiving operations to or from application queues. The Application role uses the application protocols for MQMP or MQAC or both to interact with the Message Queuing System.
- Management Application: This is a special Application role that performs the management and administrative aspects of the Message Queuing System. This role uses the client side of MQMR and MQAC protocols to interact with the Management Server role of the queue manager, as described in the following section.

5.2.1.2 Queue Manager Roles

This section describes the roles exhibited by the queue manager in the Message Queuing System.

5.2.1.2.1 Queue Manager Roles for Application Interaction

The Message Queuing System exhibits the following server roles that allow interaction with the applications, as described in section 4.3.1.1:

- **Queue Server**: This role provides the message exchange functionality of the Message Queuing System. This role implements the server side of MQAC and both the server and client sides of MQQP and MQRR.

Copyright © 2011 Microsoft Corporation.

- **Management Server**: In this role, the queue manager provides management and administrative operations on the Message Queuing server. This role allows a management application to retrieve administrative information specific to queues and messages. The Management Server role also performs management operations on a queue. This role implements the server side of the MQMR protocol, as specified in [MS-MQMR] section 3.1, as well as of the MQCN protocol, as specified in [MS-MQCN] section 3.1. This role also implements the server side of the management interfaces of the MQAC protocol, as specified in [MC-MQAC] sections 3.2, 3.3, and 3.4.
- **Supporting Server**: This is a queue manager role that implements the server side of the supporting server protocol as specified in [MS-MQMP].

5.2.1.2.2 Queue Manager Roles for Message Transfer and Routing

The following roles of the queue manager involves transferring messages from the source to the destination.

- **Message Transfer**: This role performs the actual transfer of messages between two queue managers that are directly reachable from one another. This role implements both the client and server sides of the message transfer protocols, namely MQQB and SRMP.
- **Routing Server**: The queue manager performs the routing server role for facilitating the transfer of messages within a site or across sites. The routing server implements Store and Forward messaging in an enterprise Message Queuing System deployment. This role implements the Message Queuing: Binary Reliable Message Routing Algorithm, as specified in [MS-MQBR]. A more specialized version of the routing server role is the Site Gate role where the queue manager provides routing between sites.

5.2.1.2.3 Queue Manager Role for Remote Read and Management

This queue manager role involves message read and management operations from a remote queue manager. Such operations are triggered by user applications as described in section 5.2.1.1. In response to the application requests involving remote message read operations, the queue manager role for application integration, as described in section 5.2.1.2, uses this role to accomplish the functionality. This role implements both the client and server roles of the remote read protocols, namely MQQP and MQRR.

5.2.1.3 Subcomponent Roles

As specified in section 4.3.1, the Transaction Coordinator and Directory Service subsystems interact with the Message Queuing System. In addition to external implementations for these services, the Message Queuing System also has its own implementations of these subsystems, with roles described as follows:

- **Internal Transaction Coordinator**: The queue manager implements an internal Transaction Coordinator and the associated resource manager to dispense and support internal transactions. The dispensing queue manager is the only resource manager participant supported by the transaction context, and therefore no other resource managers, including other queue managers, are able to participate with internal transactions
- **MSMQ Directory Service Server**: A queue manager collocated with a directory can perform the role of a Directory Service by implementing the server side of MQDS. The queue manager playing this role is known as the MSMQ Directory Service server. Versions 1.0 and 2.0 of the Message Queuing System use the client side of MQDS and therefore can only interact with the **MSMQ Directory Service server** as its Directory Service. The MSMQ Directory Service server predates

Copyright © 2011 Microsoft Corporation.

and is superseded by Active Directory. Subsequent versions of the Message Queuing System use LDAP to interact with the Active Directory implementation of the Directory Service.

5.2.2 Message Queuing System Component Interactions

This section expands the black box relationships of the Message Queuing System as described in section 4.3.1 by showing the various interactions between the components and external entities of the Message Queuing System.

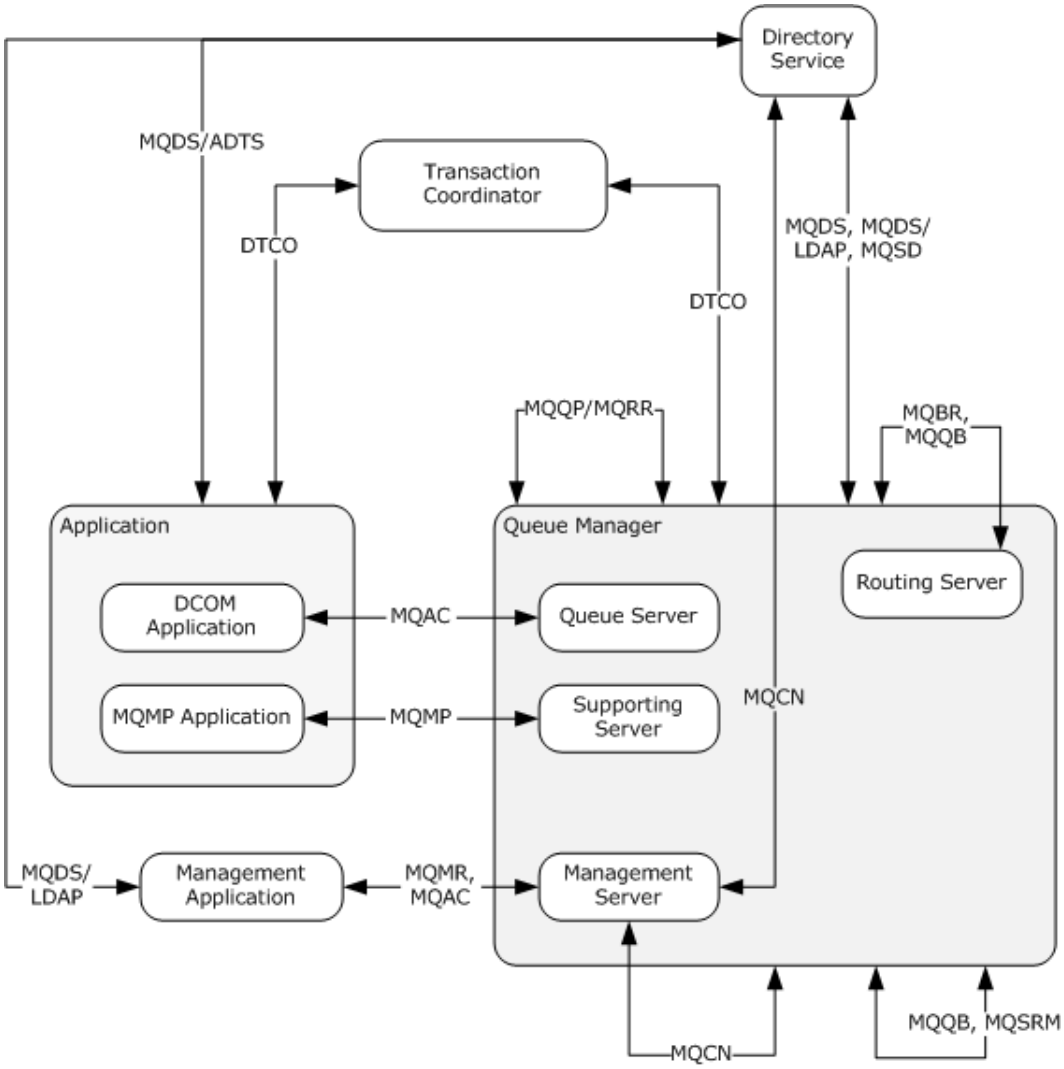


Figure 21: Message Queuing System white box relationships

The preceding figure illustrates the protocols used in the interaction between the various components of the Message Queuing System and the various roles of the queue manager.

Each link in the figure represents a protocol associated between two components or roles, represented by single or composite boxes.

Copyright © 2011 Microsoft Corporation.

The queue manager box on the right side in the figure is the composite component that represents multiple roles of the queue manager. A link associated with this box represents an association with all roles of the queue manager. A link associated with a specific inner box applies to that particular role only. The Queue Server, Supporting Server and Management Server roles of the queue manager interact with different types of application programs, as introduced in section 4.3.1.1. The Routing Server role of the queue manager is responsible for message routing in a distributed network, as introduced in section 4.3.1.3.

The Application box on the left side in the figure represents applications in different deployment modes. A link associated with this box represents protocols that apply to all three of these deployment modes. A link associated with a specific inner application mode applies to that particular application only. A DCOM application interacts with the Queue Server role of the queue manager using the MQAC protocol, whereas an MQMP application interacts with the Supporting Server role of the queue manager using the MQMP protocol. A management application interacts with the Management Server role of the queue manager using the MQMR and MQAC protocols to accomplish administrative and management related activities of the Message Queuing System.

As specified in section 4.3.1.2, a Transaction Coordinator comes into play within a Message Queuing System for transactional send and receive of messages. Applications interact with the Transaction Coordinator through the application role, and the queue manager interacts with the Transaction Coordinator through the resource manager role of the Transaction Coordinator component.

As specified in section 4.3.1.3, the Directory Service is an optional component of the Message Queuing System and enables certain functionalities of the queue manager that require interaction with a Directory. In Message Queuing System versions 1 and 2, the queue manager that is collocated with a directory also assumes the role of a Directory Service through MSMQ Directory Service server. In such a configuration, the Directory Service becomes another queue manager role in the preceding figure. In subsequent versions of the Message Queuing System, Active Directory is used as the Directory Service. Applications and the queue manager interact with the Directory Service.

5.3 Member Protocol Functional Relationships

5.3.1 Member Protocol Roles

The following table summarizes the roles of each individual protocol of the Message Queuing System protocol family.

Protocol name	Protocol role
[MS-MQMQ] Message Queuing (MSMQ): Data Structures	The common definition and data structures used by the member protocols of the Message Queuing System.
[MS-MQDMPR] Message Queuing (MSMQ): Common Data and Processing Rules	The common abstract data model, events, and processing rules shared by the member protocols of the Message Queuing System protocol family.
[MC-MQAC] Message Queuing (MSMQ): ActiveX Client Protocol	A DCOM protocol that provides a client application programming interface to the Message Queuing System. This protocol exposes the message queuing functionalities, such as queue management and discovery and sending and receiving of messages, to the client applications through a collection of DCOM objects. The queue manager implements the server side of this protocol, and

Copyright © 2011 Microsoft Corporation.

Protocol name	Protocol role
	applications use the client interfaces to invoke the needed messaging functionality.
[MS-MQMP] Message Queuing (MSMQ): Queue Manager Client Protocol	An RPC protocol that provides a client interface to the Message Queuing System through the Supporting Server deployment profile. The queue manager implements the server side of this protocol to fulfill the Supporting Server role. Applications use the client side of this protocol to perform various message queuing operations such as managing local private queues, managing cursors, sending and receiving of messages, and so on. This protocol ties closely with MQQP for implementing various processing rules, as described in the respective protocol documents, and therefore if one protocol is implemented, the other one MUST be implemented as well.
[MS-MQQB] Message Queuing (MSMQ): Binary Reliable Messaging Protocol	A block protocol designed for reliable transfer of messages between two queues, hosted by two different queue managers located on different machines. This protocol uses TCP/IP or SPX/IPX to transform the data between queue managers, but augments it with additional levels of acknowledgments that ensure that the messages are reliably transferred across regardless of TCP or SPX connection failures, application failures, or node failures. The queue manager implements both the client and the server sides of this protocol. This protocol relies on LDAP or MQDS to look up persistent entries in the directory. This protocol also relies on MQBR to implement the Routing Server role of the queue manager.
[MS-MQBR] Message Queuing (MSMQ): Binary Reliable Messaging Algorithm	A routing algorithm used with MQQB by a queue manager to route messages to the appropriate queue manager in a complex network topology.
[MC-MQSRM] Message Queuing (MSMQ): SOAP Reliable Messaging Protocol (SRMP)	An HTTP-based or PGM-based protocol that uses SOAP encoding for reliable transfer of messages from one queue manager to others. The queue manager implements both the client and the server sides of this protocol.
[MS-MQCN] Message Queuing (MSMQ): Directory Service Change Notification Protocol	A block protocol for notifying a queue manager about changes made to the Directory Service objects owned by that queue manager. The types of notifications that can be performed by using this protocol include notifying a queue manager that a queue object has been created, changed, or deleted; and notifying a queue manager that its machine object has been changed. This protocol uses MQQB to transfer notification messages between queue managers. Both the client and the server sides of this protocol are implemented by the queue manager.
[MS-MQMR] Message Queuing (MSMQ): Queue Manager Management Protocol	An RPC protocol that provides a client interface for management operations on the Message Queuing System. The various monitoring and administrative operations provided by this protocol include retrieval of information about queue managers and queues, taking queues and queue managers offline, and so on. The management applications use the client side of this protocol and the queue manager implements the server side of this protocol.
[MS-MQSD] Message Queuing (MSMQ): Directory Service Discovery Protocol	A block protocol that uses UDP multicast for locating MSMQ Directory Service servers. The client side of this protocol is used by a server implementation of MQMP. The queue manager implements both the client and the server sides of this protocol.

Copyright © 2011 Microsoft Corporation.

Protocol name	Protocol role
[MS-MQDS] Message Queuing (MSMQ): Directory Service Protocol	An RPC protocol that implements the MSMQ Directory Service. The queue manager and applications use the client side of this protocol to communicate with the Directory Service. The MSMQ Directory Service Server role of the queue manager implements the server side of MQDS. Versions 1.0 and 2.0 of the Message Queuing System implement the client and server side of this protocol. Versions 3.0 and 4.0 only implement the server side of this protocol. This protocol is not supported in its entirety in version 5.0 of the Message Queuing System.
[MS-MQDSSM] Message Queuing (MSMQ): Directory Service Schema Mapping	A mapping between relevant Message Queuing abstract data model elements, as described in MQDMPR, and a directory service over an LDAP interface.
[MS-MQQP] Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol	An RPC protocol that provides an interface between two queue managers for reading and browsing messages from a remote queue. The queue manager implements both the client and the server sides of this protocol. This protocol is tightly coupled with MQMP in that the processing rules of MQMP invoke methods of this protocol. Therefore if one protocol is implemented, the other one MUST be implemented as well. This protocol has been deprecated, and MQRR is the preferred protocol for implementing the remote read functionality. MQRR, however, does not replace MQQP, and it does not change the co-implementation requirement of MQMP and MQQP.
[MS-MQRR] Message Queuing (MSMQ): Queue Manager Remote Read Protocol	An RPC protocol that provides an interface between two queue managers for reading and browsing messages from a remote queue. This protocol is preferred for implementing the remote read functionality. The queue manager implements the client and server sides of this protocol. This protocol does not replace MQQP and does not change the co-implementation requirement of MQMP and MQQP. When this protocol is implemented, MQMP and MQQP can also be implemented for backward compatibility.

5.3.2 Member Protocol Groups

This section describes the member protocol groups that are used together to accomplish a conceptually separate sub-goal of the system.

5.3.2.1 Common Data Structure and Model

This protocol group defines the common data structures, abstract data models, and processing rules for all protocols in the Message Queuing System family of protocols. This group consists of the following protocols:

- Message Queuing (MSMQ): Data Structures, as described in [MS-MQMQ]
- Message Queuing (MSMQ): Common Data and Processing Rules, as described in [MS-MQDMPR]

5.3.2.2 Message Transfer and Routing Protocols

This protocol group is responsible for transferring messages between queues hosted by different queue managers. This group consists of the following protocols:

- Message Queuing (MSMQ): Binary Reliable Messaging Protocol, as described in [MS-MQQB]

Copyright © 2011 Microsoft Corporation.

- Message Queuing (MSMQ): Binary Reliable Messaging Algorithm, as described in [MS-MQBR]
- Message Queuing (MSMQ): SOAP Reliable Messaging Protocol (SRMP), as described in [MC-MQSRM]

5.3.2.3 Messaging Functionality Protocols

5.3.2.3.1 Core Messaging Functionality Protocols

This protocol group provides the message queuing functionalities to the applications. Member protocols of this group are invoked either directly by client applications, or indirectly by the queue manager as a result of a client application seeking a specific messaging functionality from the Message Queuing System. This group consists of the following protocols:

- Message Queuing (MSMQ): ActiveX Client Protocol, as described in [MC-MQAC]
- Message Queuing (MSMQ): Queue Manager Client Protocol, as described in [MS-MQMP]
- Message Queuing (MSMQ): Queue Manager to Queue Manager Protocol, as described in [MS-MQQP]
- Message Queuing (MSMQ): Queue Manager Remote Read Protocol, as described in [MS-MQRR]

Additionally, the following external protocol is also invoked for providing functionality if this protocol group:

- MSDTC Connection Manager: OleTx Transaction Protocol Specification, as described in [MS-DTCO]

5.3.2.3.2 Management, Administration and Configuration Protocols

This protocol group provides the administrative functionalities to the applications. Member protocols of this group are invoked either directly by client applications, or indirectly by the queue manager as a result of a client application seeking a specific message queuing administration functionality from the Message Queuing System. This group consists of the following protocols:

- Message Queuing (MSMQ): ActiveX Client Protocol, as described in [MC-MQAC]
- Message Queuing (MSMQ): Queue Manager Management Protocol, as described in [MS-MQMR]
- Message Queuing (MSMQ): Directory Service Change Notification Protocol, as specified in [MS-MQCN]

5.3.2.4 Directory Service Protocols

This protocol group provides storage and access to a directory in Message Queuing System versions 1.0 and 2.0. This functionality has been superseded by Active Directory. This group consists of the following protocols:

- Message Queuing (MSMQ): Directory Service Protocol, as described in [MS-MQDS]
- Message Queuing (MSMQ): Directory Service Discovery Protocol, as described in [MS-MQSD]
- Message Queuing (MSMQ): Directory Service Schema Mapping, as described in [MS-MQDSSM]

Additionally, the following external protocols are also invoked for providing functionality of this protocol group:

Copyright © 2011 Microsoft Corporation.

- LDAP: Lightweight Directory Access Protocol [RFC3377]
- Active Directory Technical Specification, as specified in [MS-ADTS]

5.4 System Internal Architecture

The queue manager is the central piece of the Message Queuing communication service. Conceptually, the queue manager deals with every queued messaging aspect of the Message Queuing System. The queue manager represents an instance of the Message Queuing System available on a machine with a unique network address. The queue manager interacts with other queue manager instances by implementing the client and server sides of the relevant Message Queuing protocols, as described in section 5.3.2.2. A queue manager instance interacts with applications to provide creation, modification, deletion, and message exchange operations on a queue. A queue manager listens to and handles all incoming Message Queuing network traffic for the network identifier of the machine. Additionally, the queue manager listens to and handles all the higher-layered triggered events as described in [MS-MQDMPR] section 3.1.4.

The following diagram describes the different facets of the queue manager on one machine (denoted by the central box marked as "Queue Manager" in the diagram) and the communications between these facets and other remote queue managers (denoted by the box named "Remote Queue Manager" on the right hand side of the diagram), as well as the other subcomponents of the Message Queuing System, such as client applications (denoted by the box named "Client" on the left-hand side of the diagram), the Transaction Coordinator (denoted by the box named "Transaction Coordinator" at the top of the diagram), and the Directory Server (denoted by the box named "Directory Server" at the bottom of the diagram).

Copyright © 2011 Microsoft Corporation.

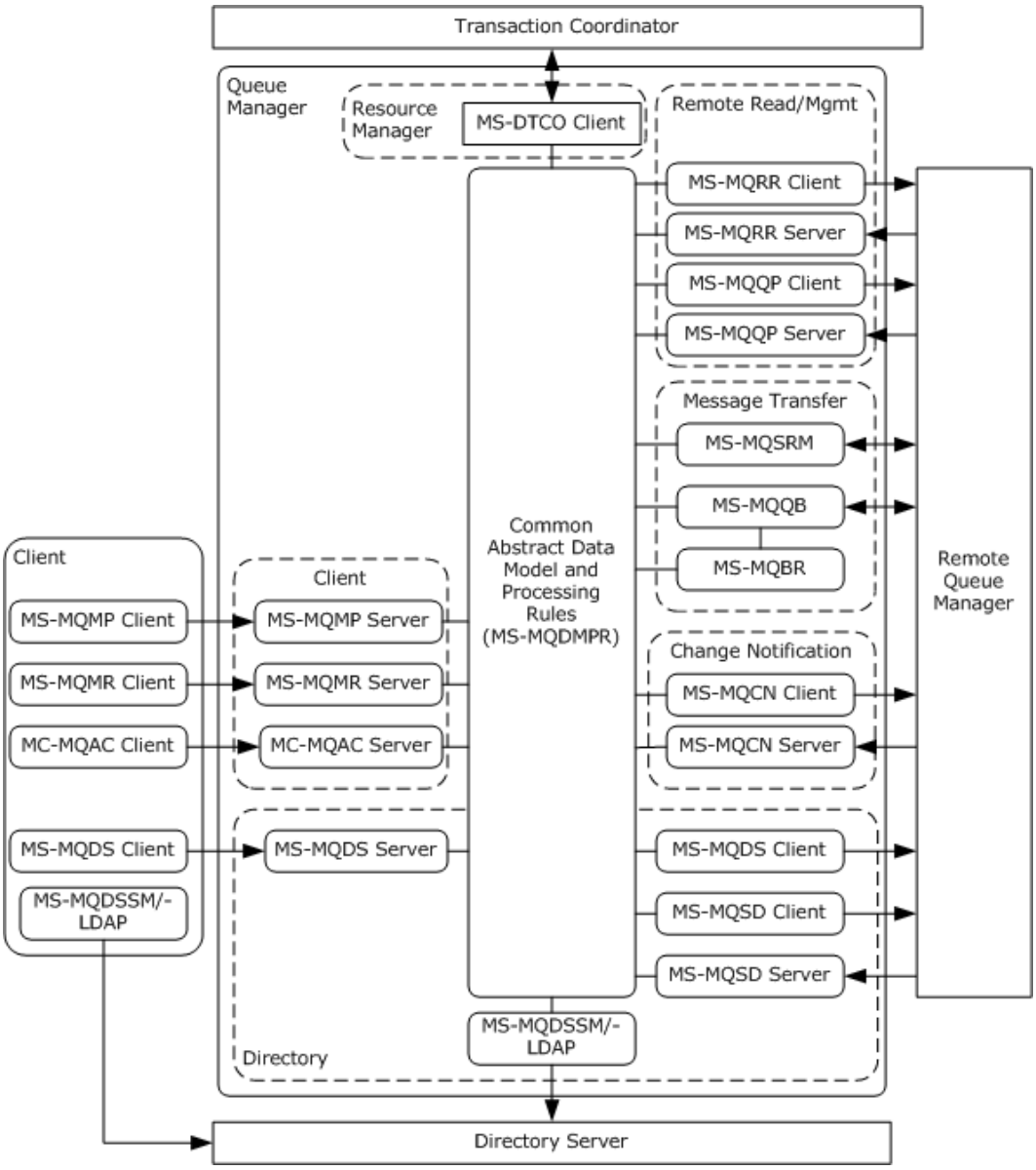


Figure 22: Message Queuing System internal architecture

The abstract data model and processing rules shared by the different facets of the queue manager are described in [MS-MQDMPR] section 3. This common model describes the shared data elements between the various protocols participating in the Message Queuing System.

The Client facet of the queue manager (denoted by the dotted box named "Client" in the above diagram) interacts with the client applications. This facet implements the server sides of MQMP, MQMR, and MQAC. The applications use the client side of these protocols to interact with the queue manager for message exchange and management functionalities.

The Directory facet of the queue manager (denoted by the dotted box named "Directory" in the above diagram) handles the various directory-related operations. On Message Queuing System versions 1.0 and 2.0, the server side of the MQDS protocol is implemented by the MSMQ Directory

Copyright © 2011 Microsoft Corporation.

Service Server, and this is collocated with the queue manager as part of the Directory facet. Client applications on these versions of the Message Queuing System use the client side of MQDS protocol to interact with the Directory Server. Similarly, a queue manager that is not collocated with the MSMQ Directory Service Server uses the client side of MQDS to interact with a remote MSMQ Directory Service server. Furthermore, the queue manager uses the client protocol of MQSD to discover the other MSMQ Directory Service servers. A queue manager running the role of an MSMQ Directory Service server implements the server side of MQSD to broadcast its presence. MQDS and MQSD protocols are superseded by Active Directory in subsequent versions of the Message Queuing System. With Active Directory, both the client applications as well as the Directory facet of the queue manager use LDAP and MQDSSM to communicate directly with the directory. The operations on the Directory facet are invoked only when the Message Queuing System is functioning in Directory-Integrated mode, as described in section 4.3.1.3.

The Resource Manager facet of the queue manager (denoted by the dotted box named "Resource Manager" in the above diagram) handles the interaction with the Transaction Coordinator to enable transactional operations on the transacted queues as specified in section 4.3.1.2. In order to manage the queue as a transactional resource, the queue manager implements the resource manager role of the Transaction Coordinator, as specified in [MS-DTCO] section 1.3.3.2, and participates with the resource manager facet of a transaction manager coordinated by an external Transaction Coordinator, as specified in [MS-DTCO] section 1.3.3.3.

The Remote Read and Management facet of the queue manager (denoted by the dotted box named "Remote Read/Mgmt") handles certain message exchange and management functionalities that involve queues hosted by remote queue managers. The operations on this facet are initiated by the actions performed by the Client facet of the queue manager. The queue manager implements the client side of MQRR and MQQP to invoke the remote queue operations on a remote queue manager. The Remote Read and Management facet of the remote queue manager implements the server sides of MQRR and MQQP to perform the remote operations requested.

The Change Notification facet of the queue manager (denoted by the dotted box named "Change Notification") handles changes made to the resources owner by one queue manager to another. The queue manager implements the client side of MQCN to send the change notifications to a remote queue manager that implements the server side of MQCN. The operations of this facet are initiated by the actions performed by the Client facet and the Directory facet of the queue manager.

The Message Transfer facet of the queue manager (denoted by the dotted box named "Message Transfer") implements the client and server sides of MQSRM, MQQB, and MQBR. The queue manager uses the client side of those protocols to send messages to the remote queue manager. The remote queue manager uses the server protocols to accept such messages. Operations on the Message Transfer facet are initiated by the actions performed by the Client facet as well as the Change Notification facet of the queue manager.

The following section details the various communication flows within the Message Queuing System components.

5.4.1 Communications Within the System

Components of the Message Queuing System communicate with one another using the following protocols and algorithms:

- MQQB: Used by one queue manager to transfer messages to another queue manager.
- MQBR: Used in conjunction with MQQB by a queue manager to route messages to a final destination queue manager, optionally hopping through other interim queue managers.
- SRMP: Used by one queue manager to transfer messages to another queue manager.

Copyright © 2011 Microsoft Corporation.

- MQQP: Used by a queue manager to perform message reading and management operations on a remote queue hosted by a remote queue manager. This protocol is superseded by MQRR.
- MQRR: Used by a queue manager to perform message reading and management operations on a remote queue hosted by a remote queue manager.
- MQCN: Used in conjunction with MQQB by a queue manager to send update notifications to a remote queue manager for resources owned by the remote queue manager.
- MQSD: Used by a queue manager that is operating as an MSMQ Directory Service server in versions 1.0 and 2.0 of the Message Queuing System to send server discovery details to other queue managers that are not operating in the role of an MSMQ Directory Service server.
- MQDS: Used by a queue manager in versions 1.0 and 2.0 of the Message Queuing System to communicate with another queue manager that is operating in the role of an MSMQ Directory Service server.

Abstracts for these protocols and the specific communication within the Message Queuing components are tabulated in sections 2.2 and 5.3.

5.4.2 Communications with External Systems

The Message Queuing System communicates with external systems via different protocols specified in their respective documents:

- MQMP: Used by a client application to communicate with a queue manager operating in the role of a Supporting server, as described in section 4.3.1.1 and 5.2.1.2.1.
- MQAC: Used by a client application to communicate with a queue manager operating in the role of a Queue Server or Management Server, or both, as described in section 4.3.1.1 and 5.2.1.2.1.
- MQMR: Used by a client application to communicate with a queue manager operating in the role of a Management Server, as described in section 4.3.1.1 and 5.2.1.2.1.
- LDAP as specified in [MS-ADTS] section 3.1.1.3: Used by the queue manager operating in Directory-Integrated mode, to communicate with Active Directory.
- DTCO in a Resource Manager role as specified in [MS-DTCO] section 1.3.3.2: Used by the queue manager to enable transactional operations on transacted queues.

Abstracts for these protocols and the specific communication with the external systems are tabulated in sections 2.2 and 5.3.

5.5 Failure Scenarios

This section describes the common failures encountered by the Message Queuing System and the system behavior under such conditions.

5.5.1 Queue Manager Restart

The queue manager can undergo both controlled and uncontrolled shutdown, resulting from either a planned system downtime or an unexpected failure of a component in the underlying operating system that requires a system reboot. The queue manager MUST be resilient to such system stoppage, and when the system restarts, the queue manager MUST reinitialize itself, restore to the state immediately preceding the shutdown, and honor the delivery assurance of the application

Copyright © 2011 Microsoft Corporation.

messages as described in the section 3.2.1. Specifically, the following rules apply for different message types:

- A transactional message sent by an application to deliver to a remote queue MUST live through system restart until it is successfully transferred, or a permanent delivery failure is encountered, or the message times out.
- A transactional message in a local queue hosted by the queue manager MUST live through system restart until it is consumed by the receiving application or the message times out.
- A recoverable message sent by an application to deliver to a remote queue MUST live through system restart if the message was never sent to the destination queue manager prior to the shutdown. Such a message remains in the outgoing queue until the queue manager makes a successful delivery attempt to the remote queue manager, or a permanent delivery failure is encountered, or the message times out.
- A recoverable message in a local queue hosted by the queue manager MUST live through system restart until it is consumed by the receiving application or the message times out.
- An express message MUST be discarded following a system restart.

5.5.2 Transient Network Failure

The Message Queuing System MUST gracefully handle network outages and restore normal operations when the network comes back online according to the following rules:

- The message transfer protocols such as MQQB and MQSRM handle network failures as part of their protocol as described in the respective protocol documents. When the network becomes available, these protocols resume their normal message transfer activities without requiring any additional external intervention. All other dependent protocols, such as MQCN, MUST remain unaffected by such interim network outages.
- The messaging activities that require synchronous communication with another queue manager or the directory MUST be unavailable during network outages. Examples of such activities include the functions invoked by the Remote Read and Management facet and the Directory facet of the queue manager, as described in section 5.4. During a network outage, the Message Queuing System MUST fail the client operations that require synchronous communication across the network. When the network becomes available, the Message Queuing System MUST resume these operations without requiring any additional external intervention.

5.5.3 Transaction Coordinator Unavailable

The external Transaction Coordinator subcomponent of the Message Queuing System provides transactional message send and receive, as described in section 4.3.1.2. The Transaction Coordinator interact with the Resource Manager facet of the queue manager, as specified in section 5.4, to enable transacted send to a local outgoing queue, and transacted receive from a local or remote queue. Being an external component, it is possible that this subcomponent remains temporarily unavailable. The queue manager MUST gracefully handle the unavailability of this subcomponent according to the following rules:

- If the Transaction Coordinator is unavailable during a queue manager startup, the queue manager MUST be able to initialize and start up.
- If the Transaction Coordinator becomes unavailable when the queue manager is in the Running state (as described in section 5.1.1.1), the queue manager MUST gracefully abort and clean up all pending transactions maintained by the queue manager. The queue manager MUST fail any

Copyright © 2011 Microsoft Corporation.

message send or receive operation on a transactional queue. The queue manager MUST perform all other activities that do not involve the use of a Transaction Coordinator (including message transfer to a remote Transactional Queue that does not require the Transaction Coordinator) in an undisrupted manner.

- When the Transaction Coordinator becomes available, the queue manager MUST resume normal transacted receive and send operations, without requiring any additional external intervention.

5.5.4 Directory Unavailable

The Message Queuing System uses a directory if it is operating in the Directory-Integrated mode. The directory can become temporarily unavailable to one of more queue managers in the entire Message Queuing System. The queue manager MUST gracefully handle the unavailability of this subcomponent according to the following rules:

- If the directory is unavailable during the queue manager startup, the queue manager MUST set the QueueManager.DirectoryIntegrated ADM element to FALSE, denoting that it is operating in the Workgroup mode. The queue manager MUST support all functionality available in Workgroup mode, and the queue manager MUST fail all functionality that requires access to a directory. When the directory becomes available, the queue manager MUST be externally restarted to resume operations in Directory-Integrated mode.

- If the directory becomes unavailable when the queue manager is running, the QueueManager.DirectoryOffline ADM element MUST be set to TRUE to denote that the queue manager is running under a constrained mode with no access to the directory. The queue manager MUST fail all operations invoked by the Directory facet of the queue manager. The queue manager MUST support all functionality that does not require directory access. When the directory becomes available, the queue manager MUST reset the QueueManager.DirectoryOffline to FALSE and resume operations in Directory-Integrated mode.

5.5.5 Internal Storage Failure

The queue manager uses a local persistent store to persist its state and data in an implementation-specific protocol-independent manner. The Message Queuing System does not mandate any specific redundancy strategy, inconsistency-detection mechanism, or backup-restore requirement on the implementation of the persistent store. The following rules are generally applied in case of an internal storage failure:

- If the storage system is unable to persist data due to exceeded capacity, the Message Queuing System MUST fail the entire related operation and perform any necessary clean-up operation to restore coherency of the store.

- If the queue manager detects inconsistent configuration data such as queue configuration, and the storage implementation is capable of completely repairing the inconsistency, the queue manager MUST bring the store to a consistent state before proceeding with any other operation.

- If the queue manager detects inconsistency in the storage and the specific storage implementation mechanism is unable to repair the inconsistency, the queue manager MUST stop all operations and shut down until the storage is restored to a consistent state by an external administrator. The Message Queuing System does not mandate any backup or restore mechanism for its state and data. If the state cannot be restored to a consistent state, the queue manager MUST be completely uninstalled and reinstalled on the particular machine, resulting in permanent loss of configuration and data.

Copyright © 2011 Microsoft Corporation.

5.5.6 Directory Inconsistency

As described in [MS-MQDMPR] (section 3.1.1), certain ADM elements of the Message Queuing System are persisted in the directory and shared across all queue managers of the entire deployment. Each queue manager maintains certain attributes in the directory under the Machine Directory Service object of the machine domain, as described in [MS-MQDS] (section 2.2.10.3), and in [MS-MQDSSM] sections 2.2.1, 2.2.2, and 3.1.6.4.1. The queue manager on a particular machine MUST verify that the state maintained under the directory belongs to this queue manager. An inconsistency arises if the machine and the directory go out-of-sync due to a manual reinstallation of the queue manager, or malicious corruption. During startup, the queue manager MUST verify that the QueueManager.Identifier ADM element matches the PROPID_QM_MACHINE_ID property value of the Machine Directory Service object (for MQDS) or the objectGUID attribute of mSMQConfiguration object (for MQDSSM), if one exists. If the Directory object exists and there is a mismatch, the queue manager SHOULD set the QueueManager.DirectoryOffline attribute to TRUE and start the Directory Online Timer.

Copyright © 2011 Microsoft Corporation.

6 System Details

This section contains the details that complete the descriptions in earlier sections of the document. These details are needed to understand and implement this system. Information already in the TDs should be referenced whenever available.

6.1 Architectural Details

This section describes the architecture of the Message Queuing System in terms of user scenario example. Each scenario described in this section demonstrates how a specific user goal is accomplished by the Message Queuing System using one or more use cases described in section 3.3. The following scenarios are described.

#	Scenario	Synopsis	Member protocols used
1	Disconnected Data Entry	An application requiring exactly-once delivery guarantee to transfer business messages asynchronously between machines.	MQQB
2	Web Order Entry	An application requiring independent scaling out of the front-end application servers and the back-end processing servers.	MQQB, MQRR
3	Modify a Public Queue	A management application requiring the ability to administer Message Queuing operations locally and remotely.	MQDSSM, MQCN
4	Creating and Monitoring a Remote Private Queue	A management application requiring the ability to configure and monitor queues locally and remotely.	MQMR, MQAC, MQQB, MQRR
5	Branch Office Order Processing	An application requiring asynchronous and efficient message transfer functionality in a complex network topology.	MQDSSM, MQBR, MQQB
6	Business-to-Business Messaging Across Firewall	An application requiring message communication between two or more business units across organizational network boundaries.	MQSRM, MQRR
7	Server Farm	An application requiring scaled-out processing of messages that are asynchronously and reliably transferred from multiple application machines to a central intermediary server.	MQQB, MQMP, MQDS
8	Stock Ticker	An application requiring broadcasting of messages from a central machine to multiple recipient machines.	MQSRM
9	Business-to-Business Messaging Across Heterogeneous Systems	An application requiring asynchronous message communication between two or more business units when at least one business unit has a different message queuing system other than the Message Queuing (MSMQ) System.	MQDSSM, MQQB, MQBR

Each of the user scenario is described in the following subsections in terms of the following:

Copyright © 2011 Microsoft Corporation.

- Scenario Description: Describes the user scenario in terms of a hypothetical end-to-end application, in which the Message Queuing System plays a role to provide certain functionalities related to asynchronous message communication. This subsection describes the different constraints and goals relevant to the particular scenario that are accomplished by the use of the Message Queuing System.
- Message Queuing Operation Details: Describes how the Message Queuing System accomplishes the specific goals of the user scenario.
- Message Queuing System Overview: Describes how the relevant components and protocols of the Message Queuing System fits into the user scenario.
- Message Queuing Initial State: Describes the initial states and necessary configurations of the Message Queuing System.
- Sequence of Events: Describes the high-level conceptual events that take place between various Message Queuing protocol instances within the system. These events are described at a conceptual action level, and the exact sequence of protocol messages for such actions are described in the specifications of the respective protocols supported by the system, as listed in section 2.2.
- Message Queuing Final State: Describes the final state of the Message Queuing System.

6.1.1 Example 1: Disconnected Data Entry

6.1.1.1 Scenario Description

A company has a large number of field sales agents. In order to reduce paperwork and speed the ordering process, it plans to give laptops to these sales agents, allowing the agents to enter orders electronically via an order entry application and to transmit those orders to the central office for processing.

The desired ordering system operates under a number of constraints:

- The sales agents are frequently traveling and might not have network connectivity at all times. The order entry application must allow sales agents to enter orders even when network connectivity is not available. Therefore, communications between the order entry application and the central office must be asynchronous and persistent, storing orders when connectivity is not available and delivering them when it is.
- Each order must be received by the central office exactly once. If the order cannot be delivered due to prolonged network outage or some other reason, the application on the sales agent laptop must be able to get that information and take application-specific actions.
- In the event of stock shortages, orders are filled in a first come, first served manner. Therefore, orders from any given sales agent must be received by the central office in the same order that the agent entered them.

6.1.1.2 Message Queuing Operational Details

In order to meet these constraints, the company designs an order entry application for use on sales agent laptops and a matching order processing application for the central office, using the Message Queuing (MSMQ) System to deliver orders from the order entry application to the order processing application. The use of **MSMQ** automatically meets the asynchronous requirement of constraint 1. The company chooses to use transactional queues, as specified in section 3.2.1. Transactional

Copyright © 2011 Microsoft Corporation.

queues provide exactly-once and in-order delivery assurances, meeting the second and third constraints described previously.

The use cases associated with this example are covered in Send Message in Transaction – Application (section 3.3.4.4) and Receive Message in Transaction – Application (section 3.3.4.7).

6.1.1.3 Message Queuing System Overview

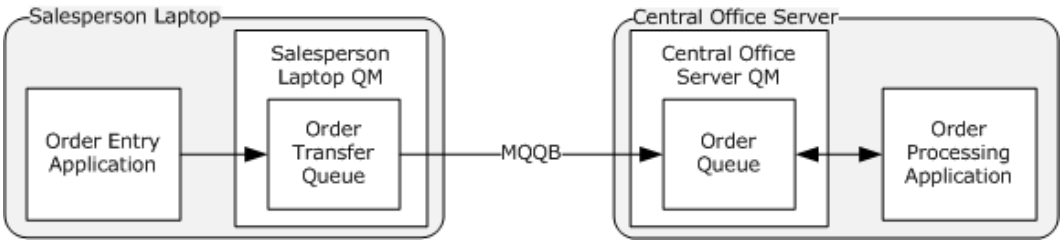


Figure 23: Message Queuing System view for Example 1

The **message queue** protocol usage is as follows:

- The Salesperson Laptop Queue Manager plays the client role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.2.
- The Central Office Server Queue Manager plays the server role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.8.

6.1.1.4 Message Queuing Initial State

In order to execute this example, both of the queue managers are brought to an initial state as described in section 6.6.

One queue, the **order queue**, is configured on the Central Office Server computer, and the network address of the Central Office Server computer is provided to the order entry application.

6.1.1.5 Sequence of Events

The following figure shows the sequence of events for this example.

Copyright © 2011 Microsoft Corporation.

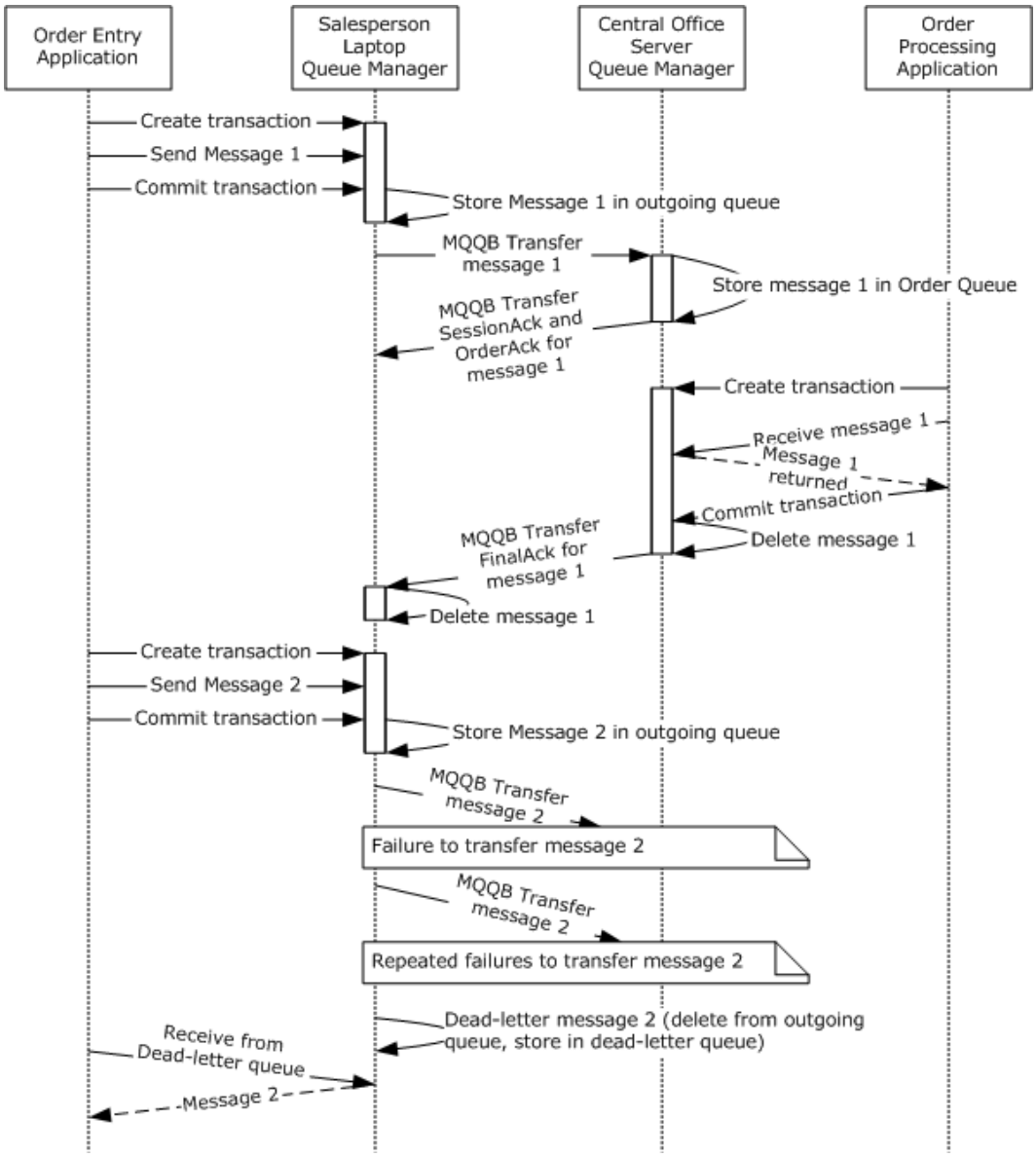


Figure 24: Sequence diagram for Example 1

In this example, a sales agent enters an order in the order entry application that is running on her laptop. The order entry application encapsulates the order into a message (message 1 in the figure above) and assembles other necessary details, such as the delivery assurances required (section 3.2.1), a request for negative source journaling (see [MS-MQQB] section 1.3.4.2), and the format name of the messages' destination, which is the order queue. The application uses a local interface to create an internal transaction, pass the message and associated details to the Salesperson Laptop Queue Manager in the context of that transaction, and then commit the transaction. The Salesperson Laptop Queue Manager stores the message in an outgoing queue when the transaction is committed.

When network connectivity to the Central Office Server computer is available, the Salesperson Laptop Queue Manager transfers the message to the Central Office Server Queue Manager, using

Copyright © 2011 Microsoft Corporation.

MQQB. The states and actions of the Salesperson Laptop Queue Manager in sending this message are specified in [MS-MQQB] section 3.1.1.1.1.7. The Central Office Server Queue Manager receives the message from the network and stores it in the order queue, and then sends a SessionAck and an OrderAck for the first message back to the Salesperson Laptop Queue Manager using MQQB. The states and actions of the Central Office Server Queue Manager in receiving this message are specified in [MS-MQQB] section 3.1.1.1.1.8. The Salesperson Laptop Queue Manager receives the SessionAck and OrderAck from the network. Note that the Salesperson Laptop Queue Manager retains the first message in the outgoing queue.

The order processing application uses a local interface to the Central Office Server Queue Manager to create an internal transaction, receive the first message from the order queue in the context of that transaction, and commit the transaction. When the transaction is committed, the Central Office Server Queue Manager deletes the first message from the order queue and sends a FinalAck for the first message to the Salesperson Laptop Queue Manager, using MQQB. The Salesperson Laptop Queue Manager receives the FinalAck from the network and deletes the first message from the outgoing queue.

The sales agent subsequently enters another order in the order entry application running on her laptop. Similar to the first message, the order entry application encapsulates the order into another message (message 2 in the figure above) and assembles other necessary details, such as the delivery assurances required (section 3.2.1), a request for negative source journaling (see [MS-MQQB] section 1.3.4.2), and the format name of the messages' destination, which is the order queue. The application uses a local interface to create an internal transaction, pass the message and associated details to the Salesperson Laptop Queue Manager in the context of that transaction, and then commit the transaction. The Salesperson Laptop Queue Manager stores the message in an outgoing queue when the transaction is committed.

The Salesperson Laptop Queue Manager attempts to transfer this second message to the Central Office Server Queue Manager, but the transfer fails and no protocol acknowledgment message is received for the second message indicating that the second message is successfully received by the Central Office Server Queue Manager. The Salesperson Laptop Queue Manager retains the second message in the outgoing queue.

If the second message is not expired, the Salesperson Laptop Queue Manager makes further attempts to transfer the second message to the Central Office Server Queue Manager. However, the second message is never successfully received by the Central Office Server Queue Manager. When the second message expires, the Salesperson Laptop Queue Manager honors the request for dead-lettering, storing the second message in the dead-letter queue and deleting it from the outgoing queue.

The order entry application periodically attempts to receive messages from the dead-letter queue using a local interface to the Salesperson Laptop Queue Manager. On its next attempt, it receives the second message from the dead-letter queue and takes an application-defined action, perhaps informing the sales agent that the order was not transferred successfully and requesting permission to attempt to send it again.

6.1.1.6 Message Queuing Final State

The final state of MSMQ in this example is the same as the initial state.

Copyright © 2011 Microsoft Corporation.

6.1.2 Example 2: Web Order Entry

6.1.2.1 Scenario Description

A retail company wants to expand its business to the Internet by creating a Web site that allows a customer to browse the catalog of products, place items in a shopping cart, and then submit the contents of the shopping cart as an order. Customers use a standard Web browser as the client application when shopping. When the customer submits an order to the Web server, the application on the Web server sends the order off to the company's order-processing application on a back-end server for processing, and presents the customer with an order completion page explaining that the order was submitted and that a follow-up e-mail will be sent to the customer when the order has been processed.

In designing the system architecture, the company identifies a need for an option in the future to scale out the Web server application by adding additional Web servers and distributing the customer load. When expanding the front end, the company prefers not to add more order-processing servers, so the need is for a solution that allows loose coupling between the multiple Web servers and the order-processing application. This requirement entails a need to buffer the single order-processing server from the potentially high volume of incoming orders, to prevent stressing that machine.

System constraints are as follows:

- The back-end processing must be buffered from the front end.
- Orders must be processed exactly once.
- The system must allow for scaling out the front end without requiring the back end to scale out, and vice versa.

6.1.2.2 Message Queuing Operational Details

To meet these requirements, the company uses an intermediary server, with a message queue that functions as a temporary repository of orders that the order-processing application receives from the queue and processes when it is ready. The ability of the queue to buffer the input to the order-processing application meets the first constraint listed above.

To fulfill the second constraint, the message queue is transactional, and messages are sent to and received from the queue inside of transactions; see section 3.2.1, Message Queuing Capabilities, for more information on transactional messaging. A transaction coordinator is required to perform the transactions.

The Web servers, the Intermediary server, and the Order-Processing server are all members of a single domain, so the company is able to limit access control on the queue such that only the domain account used for the Web server application can add messages to the queue, and only the domain account used for the order processing application can see or remove messages from the queue. For more information on these permissions, see section 7.2. Since the Web application will be running on each Web server under the same domain account, the company can add as many Web servers as needed, which is required for the third constraint. Similarly, the back-end order-processing application can scale out as needed.

The use cases associated with this example are covered in Send Message in Transaction – Application (section 3.3.4.4) and Receive Message in Transaction – Application (section 3.3.4.7).

Copyright © 2011 Microsoft Corporation.

6.1.2.3 Message Queuing System View

The following figure shows the system view for Web order entry.

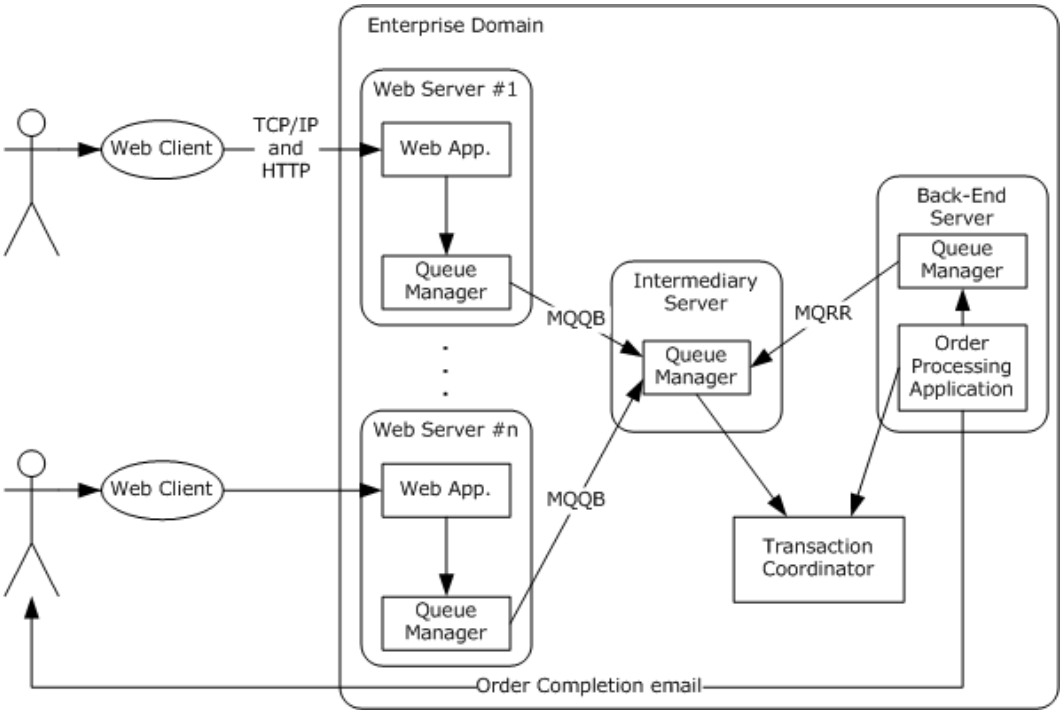


Figure 25: Web order system view for Example 2

The message queue protocol usage is as follows:

- The Web Server Queue Managers play the client role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.2.
- The Intermediary Server Queue Manager plays the server role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.8, and the server role for the MQRR protocol, as specified in [MS-MQRR] section 3.1.
- The Order Processing Application interacts with the local queue manager that plays the client role for the MQRR protocol, as specified in [MS-MQRR] section 3.2.

6.1.2.4 Message Queuing Initial State

To execute this example, the queue managers on the Web Servers, Intermediary Server, and Back-End Server are brought to their initial states, as specified in section 6.6, and are operating in the Queue Server roles.

The queue manager on the Intermediary Server is initialized to contain a transactional queue. The queue manager on each web server dynamically creates an outgoing queue if there are recoverable messages pending transfer to the transactional queue. There is no requirement to create queues on the other queue managers.

Copyright © 2011 Microsoft Corporation.

6.1.2.5 Sequence of Events

The sequence of events for Web order entry is shown in the following figure.

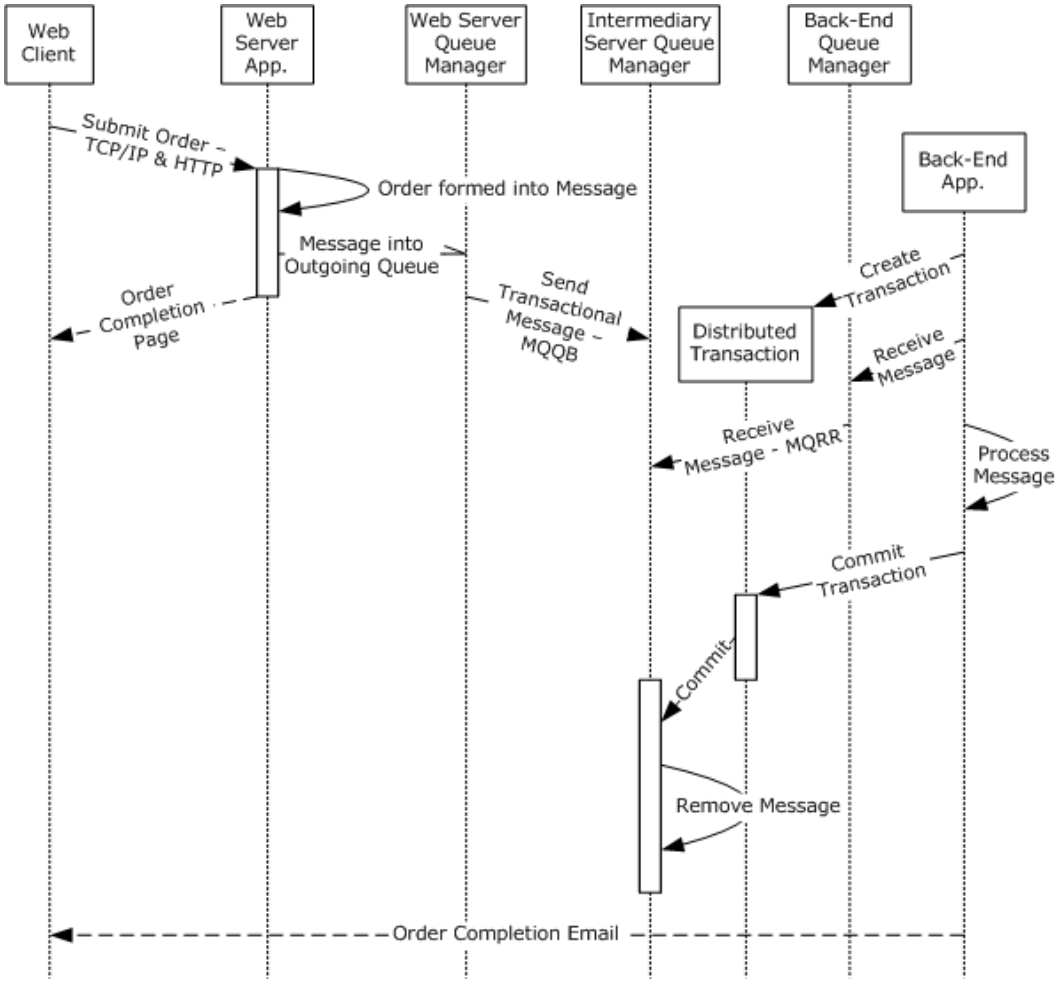


Figure 26: Sequence diagram for Example 2

When the customer submits an order on the Web client, the order is transmitted to the Web server application, which transforms the order into an MSMQ message and gives it transactionally to the local queue manager through a private interface, telling the queue manager where the message is to be sent. An outgoing queue is dynamically created (if not already present) to hold this message pending transfer to the **remote queue** (see [MS-MQDMPR] section 3.1.7.1.5). The message is then sent in to the remote queue over the MQQB protocol (see section 6.1.1, Example 1: Disconnected Data Entry), where it waits until the order processing application receives the message in a transaction using the local queue manager over the MQRR protocol. In this case, the order-processing application creates an atomic transaction, receives the message under the scope of the atomic transaction, communicates with a credit card company to verify the credit transaction, writes information about the order to a database under the scope of the same atomic transaction, and subsequently commits the transaction. The order-processing application then sends an e-mail message to the customer informing them that the processing has been completed. There is a business requirement that dictates that these steps (starting with the receipt of the message) be part of a transaction and coordinated across processes.

Copyright © 2011 Microsoft Corporation.

6.1.2.6 Message Queuing Final State

The final state for the queue managers is equal to their initial state. The Intermediary Server queue manager contains a transactional queue, and the queue is in a ready state.

6.1.3 Example 3: Modify a Public Queue

6.1.3.1 Scenario Description

In many enterprises, message queuing (MSMQ) is deployed widely, with multiple machines hosting message queuing services. The system administrator for such an enterprise needs to have the ability to manage and monitor the state of the message queuing service both locally and remotely. The administrator is required to create and delete multiple public queues on multiple machines and to modify and monitor the properties of the public queues. The application group requires test queues to be created and monitored prior to deployment of the message queuing application. Existing queues need to be modified to support additional functionality, to allow access to new users, and to increase queue quotas.

This example describes one such scenario for a large enterprise where the administrator is responsible for deploying and modifying public queues hosted by all MSMQ services. In this example, the administrator uses an application to modify the properties of an existing public queue hosted by a remote machine. The enterprise also has a Directory Service that is accessible by the administrator application.

This example describes how MSMQ ensures consistency of the queue object state by updating the Directory Service and the remote queue manager that owns the public queue.

System constraints are as follows:

- The modification of the public queue properties must be done remotely.
- Administrative operations must be authenticated.
- Any modification of the public queue properties must be reflected on the Directory Service server and on the message queuing server that hosts the public queue.

6.1.3.2 Message Queuing Operational Details

In this example, MSMQ needs to modify the properties of the public queue on the Directory Service and also to notify the owner of the public queue of the modified properties. The use case associated with this example is covered in section 3.3.4.1, Create or Modify Queue.

The Directory Service holds a database of objects representing entities related to message queuing. The message queuing system provides methods to create, update, retrieve, and delete these objects from the Directory Service through the LDAP protocol, as specified in [MS-ADTS]. This meets the first constraint above, since these methods can be invoked either by the administrator application or by a remote queue manager. See section 5.3.2.3.2 for a summary of the Management, Administration, and Configuration protocols. The Directory Service enforces secure access to the message queuing objects by using the object's access control lists. This meets the second constraint. The administrator application provides the properties of the public queue to be modified and the public queue's unique identifier or distinguished name on the Directory Service, as specified in [MS-MQDS], to the client queue manager. The client queue manager uses this information to update the Directory Service and notify the owner of the public queue object of the update via the protocol, as specified in [MS-MQCN] sections 3.3.5.3 and 3.2.5. This meets the third constraint.

Copyright © 2011 Microsoft Corporation.

6.1.3.3 Message Queuing System View

The following figure shows the key players involved in executing this example. The purpose is to highlight the MSMQ components as well as the protocols required to enable this example.

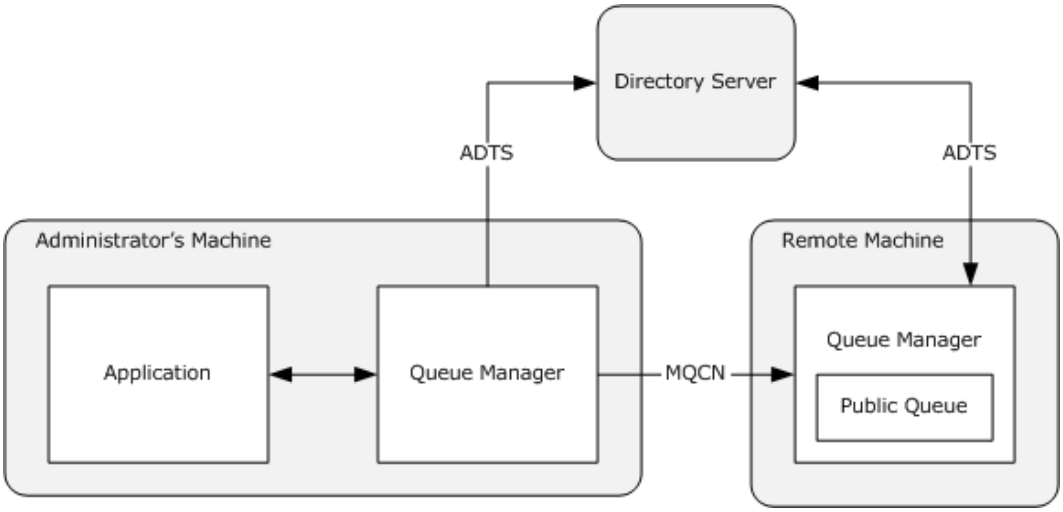


Figure 27: Message Queuing System view for Example 3

The message queue protocol usage is as follows:

- The Administrator's Queue Manager plays the client role for MQCN, as specified in [MS-MQCN] section 3.3, and the client role for ADTS.
- The Remote Machine Queue Manager plays the server role for MQCN, as specified in [MS-MQCN] section 3.2, and the client role for ADTS.

6.1.3.4 Message Queuing Initial State

In order to execute this example, the remote machine must have already created a public queue with a corresponding Active Directory object representing this public queue.

The client application is initialized with the unique identifier of the public queue object on the Directory Service.

6.1.3.5 Sequence of Events

The following figure shows the sequence of events for modifying a public queue.

Copyright © 2011 Microsoft Corporation.

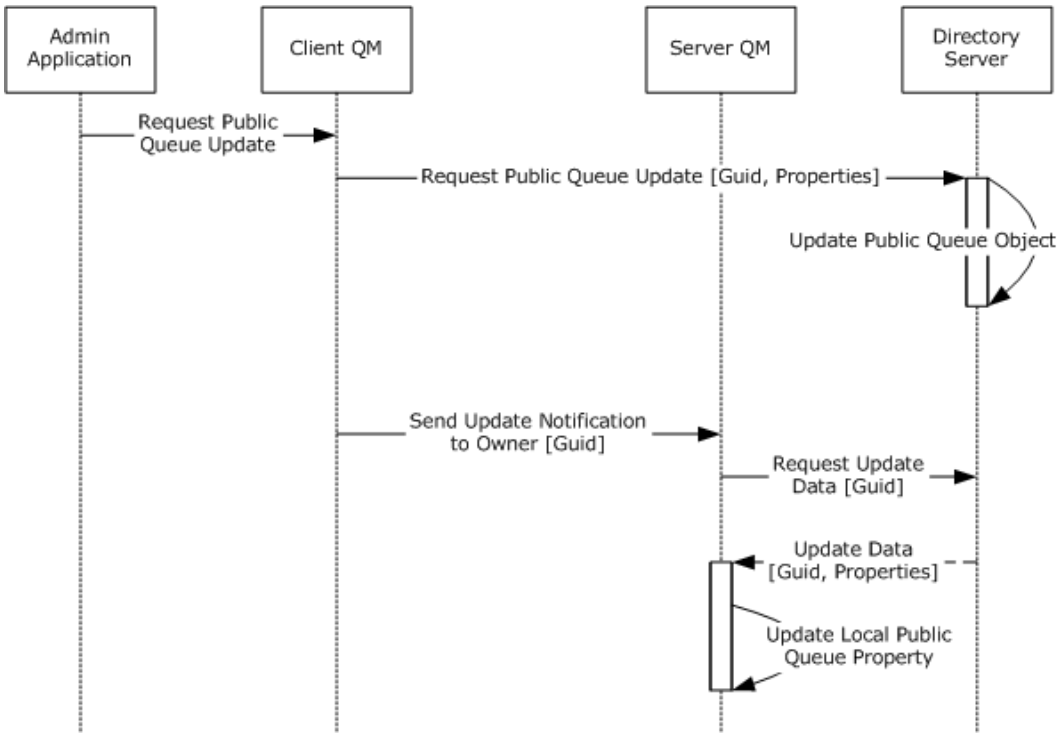


Figure 28: Sequence diagram for Example 3

In this example, the Admin Application makes a private interface call to the Client Queue Manager to update the public queue object in the Directory Service by supplying the public queue's distinguished name on the directory server and the properties of the public queue to be modified. The Client Queue Manager then requests the directory server of the corresponding domain to update the public queue object with the modified properties. This request occurs over the LDAP protocol, as specified in [MS-ADTS].

After the public queue object on the Active Directory has been successfully modified, the Client Queue Manager sends a notification message containing the unique identifier of the public queue that has been modified to the notification queue on the Server Queue Manager. This notification is sent over the MQCN protocol. The Server Queue Manager is constantly listening for incoming change notification messages on its local private notification queue.

Upon receiving the notification message, the Server Queue Manager requests information from the Directory Service for the object represented by the unique identifier that was received in the notification message. This request occurs over the LDAP protocol, as specified in [MS-ADTS]. After the Directory Service responds with the requested data, the Server Queue Manager updates its local public queue state.

6.1.3.6 Message Queuing Final State

In the final state of the message queuing system, both the Active Directory object representing the public queue on the Server Queue Manager and the public queue state on the Server Queue Manager have been updated with the new properties. The public queue in the Server Queue Manager is in a ready state.

Copyright © 2011 Microsoft Corporation.

6.1.4 Example 4: Creating and Monitoring a Remote Private Queue

6.1.4.1 Scenario Description

In many enterprises, MSMQ is deployed widely, with multiple machines hosting message queuing services. The system administrator for such an enterprise needs to have the ability to manage and monitor the state of the message queuing service, both locally and remotely. The administrator is required to create and delete multiple private queues on multiple machines, and to modify and monitor the properties of the private queues. The application group requires test queues to be created and monitored prior to deployment of the message queuing application. Existing queues need to be modified to support additional functionality, to allow access to new users, and to increase queue quota.

This example describes a scenario where the administrator is required to create a new private queue that receives multiple messages that are eventually processed by an internal application. The administrator is required to constantly monitor the number of messages in the private queue to ensure that the queue size does not grow beyond its allocated quota, causing MSMQ to drop messages destined to the private queue.

This example adheres to the following constraints:

- The administrator is able to create new private queues remotely.
- Administrative operations are authenticated.
- The monitoring of the queue state is done in real time.

6.1.4.2 Message Queuing Operational Details

This example is a simple application of MSMQ's remote management functionality. The administrator has built a custom application that uses the MQAC and MQMR protocols to instruct the server to perform message queuing operations, as if the administrator application is executing locally on the remote computer. See section 5.3.2.3.2 for a summary of the Management, Administration, and Configuration protocols. The use cases associated with this example are covered in section 3.3.4.1, Create or Modify Queue and section 3.3.4.2, Query Queue Information.

In this example, the Admin Application and the Server Queue Manager to be managed are running on two distinct machines as shown in the figure called Sequence diagram in the Sequence of Events section for this example (section 6.1.4.5). The administrator's machine does not have a queue manager. The administrator application uses the client sides of the MQAC and MQMR protocols to communicate to the remote server queue manager over RPC and DCOM transports. The administrator application establishes a connection to an RPC server, as well as a DCOM server on the remote machine. The Server Queue Manager acts as the RPC server and the DCOM server. This meets the first constraint listed above. The server enforces security by using the underlying **RPC** protocol to retrieve the identity of the invoking client application and by reading the access control list of the private queue to verify access. The client application must have administrator privileges on the server machine. This meets the second constraint. The client side of this system is simply a pass-through. That is, no additional state is required on the client side of this protocol. Calls made by the application are passed directly to the transports, and the results returned by the transports are passed directly back to the application.

Operations such as creating a private queue are actually invoked from the administrator application by using the client side of MQAC, and the operations are executed on the Server Queue Manager that implements the server side of MQAC. After the private queue is created, other business applications send and receive messages to the said private queue. The administrator application's query to retrieve the number of messages in the private queue causes the Server Queue Manager to

Copyright © 2011 Microsoft Corporation.

capture the state of the private queue and to return the data back to the administrator application. This meets the third constraint.

6.1.4.3 Message Queuing System View

The following figure shows the key players involved in this application architecture.

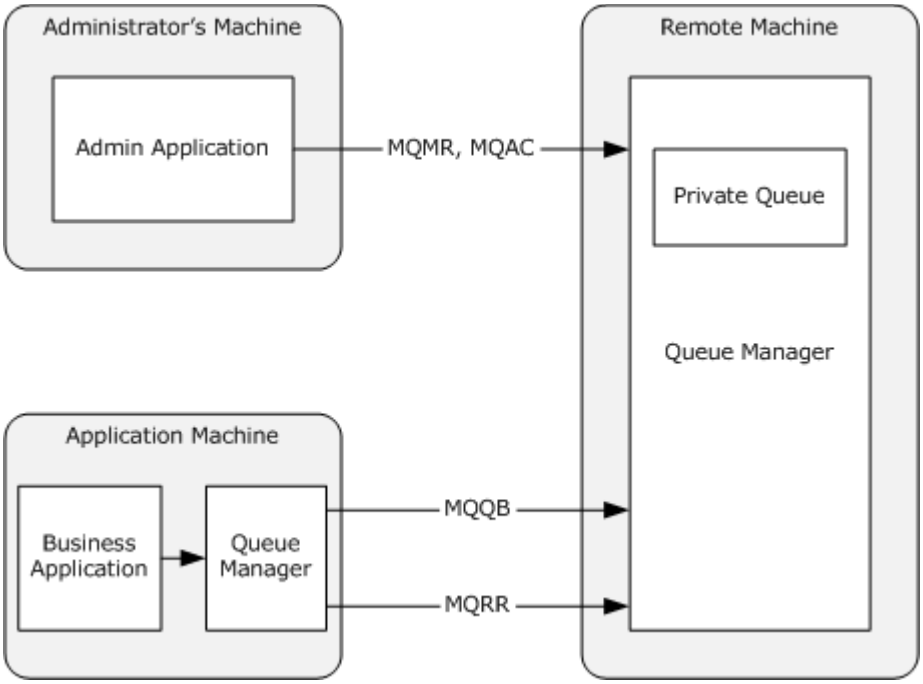


Figure 29: Message Queuing System view for Example 4

The message queue protocol usage is as follows:

- The administrator's application plays the client role of MQAC, as specified in [MC-MQAC] section 3.10, and the client role of MQMR, as specified in [MS-MQMR] section 3.1.4.1.
- The remote machine queue manager plays the server role of MQAC, as specified in [MC-MQAC] section 3.10, and the server role of MQMR, as specified in [MS-MQMR] section 3.1.4.1.

Additionally, the business applications running on other Application Machines uses the local queue managers to send messages to the private queue using the MQQB protocol, and receive messages from the private queue using the MQRR protocol. However, for the purpose of this example, the usage of these protocols are not relevant to demonstrate the management operations aspects of the Message Queuing System.

6.1.4.4 Message Queuing Initial State

In order to execute this example, the queue manager on the remote machine is in the initial state as described in section 6.6, Initialization, and is operating in the Queue Server role.

The administrator application and the business applications are configured with the remote machine's address and the private queue's address on the remote queue manager.

Copyright © 2011 Microsoft Corporation.

6.1.4.5 Sequence of Events

The following figure shows the sequence of events for creating and monitoring a remote private queue.

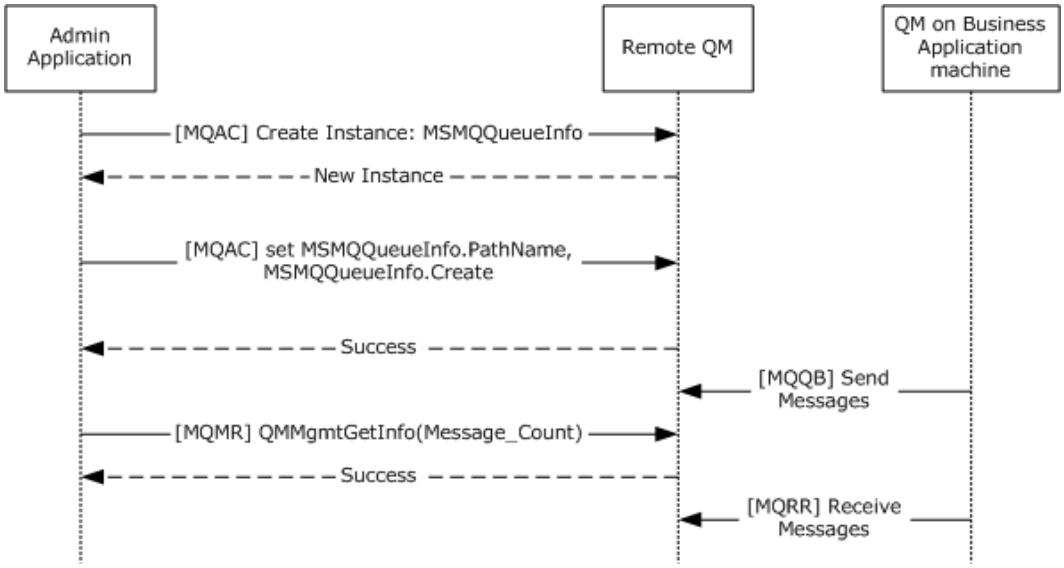


Figure 30: Sequence diagram for Example 4

In this example, the Admin application uses the MQAC and MQMR protocols to instruct the server to perform message queuing management operations. Directory Services are not required to perform these operations.

The client application requests the RPC server on the remote queue manager to create an instance of the MSMQQueueInfo (see [MC-MQAC] section 3.10) object and calls the Create method on this instance by specifying the path name and properties of the private queue. The Create method creates a private queue with the specified properties on the remote machine.

Once the private queue is created on the remote queue manager machine, other business applications running on various application machines send messages to and receive messages from the private queue, using the local queue managers on the respective machines, depending on the functionality of the business applications. Although these activities involve the Message Queuing protocols, the specifics of these activities and the related protocols are not relevant for the purpose of this example.

The client application then proceeds to monitor the state of the newly created private queue by retrieving the number of messages with a user-specified periodicity. The application calls the R_QMMgmtGetInfo method (see [MS-MQMR] section 3.1.4.1) and specifies the remote machine name and private queue name for which to retrieve state information. The queue manager on the remote machine returns information about the number of messages in the local private queue to the client application. The administrator uses this information to deduce the health of the business applications.

6.1.4.6 Message Queuing Final State

The final state of the remote queue manager has a new private queue, and the queue is in a Ready state. The query to retrieve the number of messages from the remote queue manager's private queue does not alter the state in any way.

Copyright © 2011 Microsoft Corporation.

6.1.5 Example 5: Branch Office Order Processing

6.1.5.1 Scenario Description

A national company has business operations in many different geographic regions of the country. There is one headquarters for the entire company, each region has a regional headquarters, and sales are conducted at branch offices distributed throughout each region. These offices and headquarters are connected by WAN links of varying speeds and reliabilities. Each branch office has a connection to its regional headquarters, and the regional headquarters is connected to neighboring regional headquarters and directly to the company headquarters. All computers are in the same domain and the same Message Queuing (MSMQ) enterprise, but each office and headquarters is a separate site.

As sales occur, sales data is entered at each branch office. The data is accumulated locally, then periodically is transmitted to both the regional headquarters and the company headquarters for aggregation and analysis by region and nationally. Since this is highly confidential corporate data, the application uses MSMQ security features to protect the data from snooping and tampering and to ensure the identity of the sender. The application also assures that the data is sent and uses the asynchronous nature of MSMQ to allow data to be accumulated when a branch office loses connectivity, and to be sent when connectivity is restored.

Since each branch office has only one WAN link, there is only one way for data to flow from a branch office server to a regional office server. However, due to interconnectivity among regional headquarters, there are multiple routes from any regional headquarters to the company headquarters. The application uses the intersite routing features of MSMQ to allow data delivery to company headquarters for a region even when that regional headquarters' direct WAN link is down, and to prioritize the data away from congested links or toward underutilized links.

System requirements and constraints are as follows:

- Branch offices must be able to send sales data to company headquarters.
- The system must be able to use optimal routes where available.
- The system must be resilient to unreliable WAN links between regional and company headquarters.
- Sales data must be encrypted in transit.
- The solution must allow asynchronous data transmission.

6.1.5.2 Message Queuing Operational Details

MSMQ on the branch office's server needs to reliably send confidential sales data to the regional headquarters' (HQ) server and the distant company headquarters server. Since the branch offices and company headquarters are not directly linked and might be on different networks, any messages sent to the company headquarters server by the branch office server need to be routed via the regional headquarters' server. See the figure called Sequence diagram in the Sequence of Events section for this example (section 6.1.5.5). The use case associated with this example is presented in section 3.3.4.3, Send Message to Queue.

To enable this requirement, the administrator for the enterprise updates the directory server with information about all the available machines in the enterprise, the connected networks for each of the machines, sites in the enterprise, and all available routing links. The administrator also specifies the routing link cost between each of the sites. Each of the site gates obtains this information via the MQDS protocol or over the LDAP protocol using **MQDSSM**, and computes the routing table, as

Copyright © 2011 Microsoft Corporation.

specified in [MS-MQBR] section 3.1.3.1, to identify the next hop for messages sent to a given destination machine. The routing link cost information for the WAN links in the Directory Service is used by the site gates to compute a least-cost path to a destination machine. The routing table may contain entries for more than one route for a given destination machine arranged in ascending order of the route cost. The first entry in the routing table represents the optimal route for a given destination. The queue manager on the site gate attempts to transfer messages to the first entry in the routing table, as specified in [MS-MQBR] section 3.1.5.1. This meets the second constraint.

If the queue manager on the site gate is unable to establish a connection to the first entry of the routing table, it attempts to establish a connection to the next entry in the routing table. This feature of MSMQ allows messages to be routed around dead network links. This meets the third constraint.

Confidential data transmitted by the branch office is encrypted by using the security features of MSMQ (see section 7.4). This ensures complete confidentiality between the sender and the receiver and meets the fourth constraint. The use of MSMQ automatically meets the asynchronous requirement of the fifth constraint.

Copyright © 2011 Microsoft Corporation.

6.1.5.3 Message Queuing System View

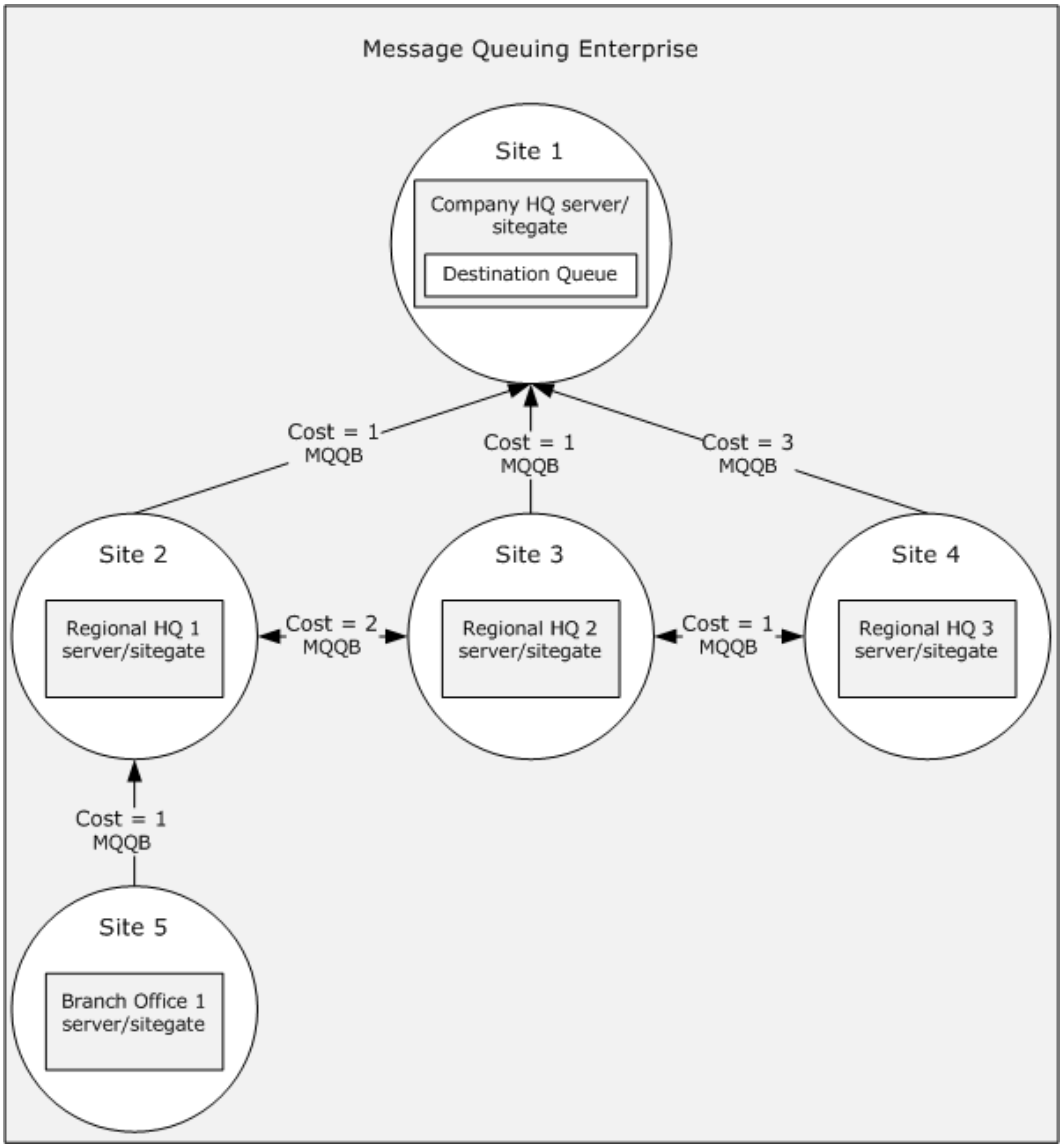


Figure 31: Message Queuing System view for Example 5

The message queue protocol usage is as follows:

- The Branch Office 1 Queue Manager plays the client role of MQQB and the server role of **MQBR**.
- The Regional HQ 1, 2, and 3 Queue Managers play both the client and the server roles of MQQB and the server role of MQBR.
- The Company HQ Queue Manager plays the server role of MQQB and the server role of MQBR.

Copyright © 2011 Microsoft Corporation.

6.1.5.4 Message Queuing Initial State

In order to execute this example, all queue managers are in the initial state as described in section 6.6, operating in the Queue Server role.

At least one Message Queuing System in each site must be designated as an MSMQ site gate, as specified in [MS-MQBR]. The In-routing and Out-routing servers must be configured on all the MSMQ site gates. The Directory Service must contain sufficient information to build a routing table, including:

- The available machines in the enterprise.
- The connected networks for each of the machines.
- The sites in the enterprise.
- All available routing links and the associated costs for each link.

The directory server has the public keys of all the machines in the enterprise. The queue manager servers on the site gates compute the routing tables that contain the most optimal routes on startup.

One queue is configured in the company headquarters (HQ) server to receive sales data.

6.1.5.5 Sequence of Events

The following figure shows the sequence of events for branch office order processing.

Copyright © 2011 Microsoft Corporation.

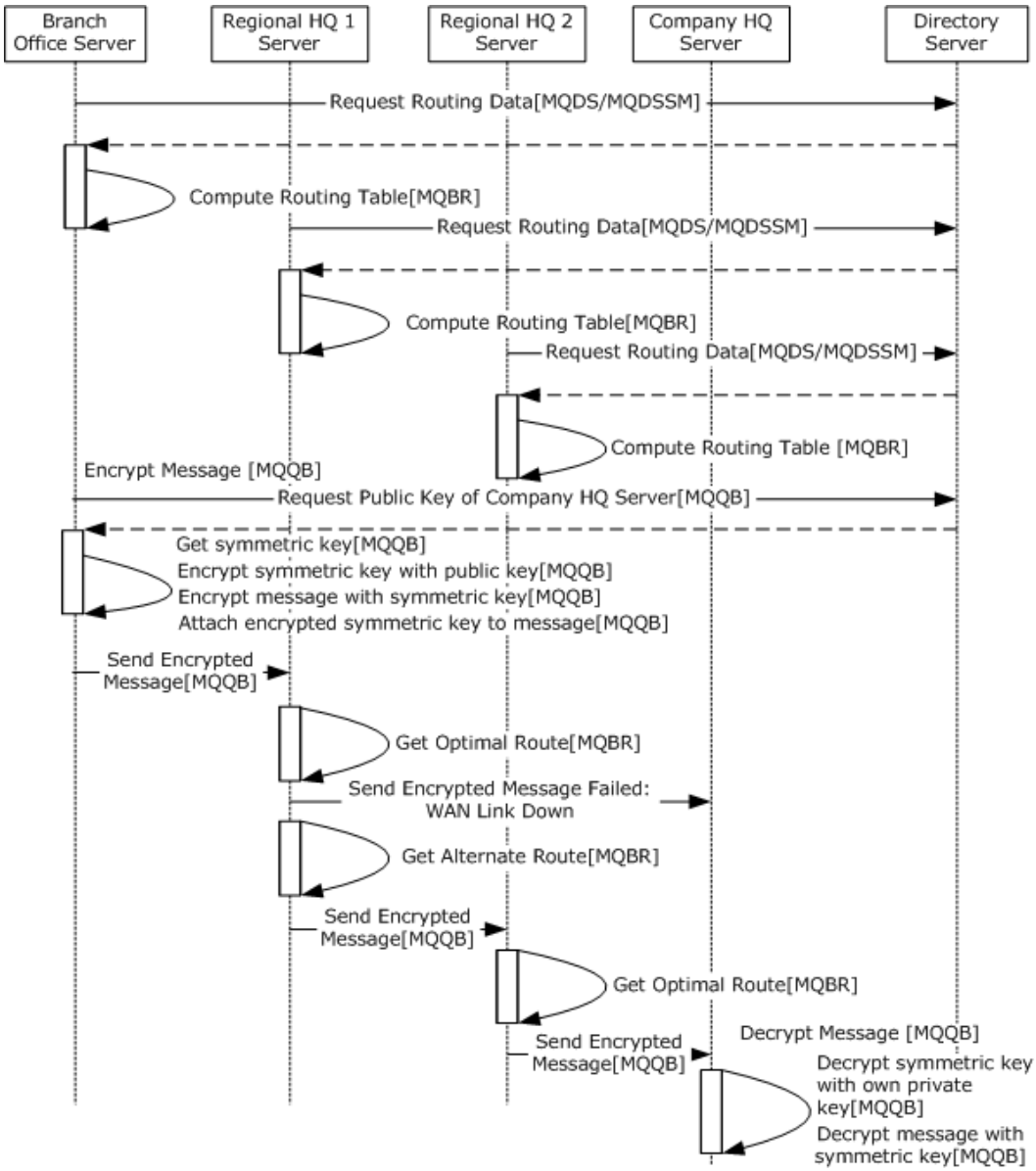


Figure 32: Sequence diagram for Example 5

In this example, the queue manager on the branch office server attempts to send an encrypted message to the company HQ server. As there is no direct connection between the two servers, the branch office server routes the message through MSMQ routing servers in the regional headquarters. The MSMQ services designated as site gates retrieve the data mentioned in section 6.1.5.3 to construct the routing table containing the next-hop information for each destination machine in the enterprise.

Encrypting the message (see [MS-MQQB] section 3.1.7.1.5 for details)

The queue manager of the branch office server uses the following steps to encrypt the message:

Copyright © 2011 Microsoft Corporation.

1. Retrieves the public key information of the company HQ server from the Directory Service (via the [MS-MQDSSM] algorithm).

2. Dynamically generates a symmetric key. The symmetric key generation SHOULD use the key length and provider information in the company HQ server's public key information to ensure that the company HQ server can decrypt the message.

3. Encrypts the symmetric key with the company HQ server's public key.

4. Encrypts the message with the symmetric key.

5. Attaches the encrypted symmetric key to the message. The message format is specified in [MS-MQMQ] section 2.2.20.6.

Calculating Optimal Routes:

Once the routing table is constructed and initialized ([MS-MQBR] section 3.1.3.1) with the various sites and the costs between them, Dijkstra's algorithm is used to discover the least-cost paths to each destination site.

In this example, the regional headquarters server for the branch office has more than one route to the company headquarters in its routing table. Here, the least cost path for a message from site 5 to site 1 is as follows: Site5-Site2-Site1 with a cost of 2. The next least cost path is: Site5-Site2-Site3-Site1 with a cost of 4. See the sequence diagram for Example 5. In this example, the WAN link between site 2 and site 1 is lost after the routing tables on the site gates have been initialized.

Branch Office Routing Table

DestinationSite	NextHopSite
Company HQ Site (Site 1)	Regional HQ Server 1 Site (Site 2)

The branch office server encrypts the sales data message and routes the encrypted message to the regional HQ 1 server after consulting its routing table.

Calculating Alternate Routes:

In case a site is not reachable, the GetNextHopsForSiteGate algorithm ([MS-MQBR] section 3.1.5.3) is used to get a list of alternate next hops from which the least cost alternate route to the ultimate destination is constructed using Dijkstra's algorithm.

In this example, the regional HQ 1 server consults its routing table to get the next site gate to which to route the message. The regional HQ 1 server attempts to establish a connection to the company HQ 1 server but fails to do so as the WAN link between site 1 and site 2 is dead. The queue manager on the regional HQ 1 server consults its routing table yet again to retrieve the next available least-cost path. In this example, the regional HQ 1 server establishes a connection to the regional HQ 2 server in site 3 and sends the encrypted message.

The regional HQ 2 server consults its routing table and establishes a connection to the company HQ server, which happens to be the next hop, and transfers the encrypted message. The company HQ server decrypts the message with its private key and processes the sales data.

Decrypting the message (see [MS-MQQB] section 3.1.5.8.3 for details)

The queue manager of the company HQ server uses the following steps to decrypt the message:

1. Extracts the encrypted symmetric key from the message.

Copyright © 2011 Microsoft Corporation.

2. Decrypts it with its own private key.

3. Decrypts the message with the decrypted symmetric key.

Note that overviews of Message layer security features such as authentication, encryption and decryption are also described in sections 7.4.1.4 and 7.4.1.5 of this document.

6.1.5.6 Message Queuing Final State

The final state of the Message Queuing System in the company HQ contains the sales data sent from the branch office server. The final state of all other Message Queuing Systems in this example is the same as the initial state.

6.1.6 Example 6: Business-to-Business Messaging Across Firewall

6.1.6.1 Scenario Description

A manufacturer wishes to allow designated employees to place orders electronically with a supplier and receive responses and status updates concerning those orders, with the overall effect of making the ordering process more efficient, cheaper, and faster. These companies are geographically and organizationally separate, each having its own network and enterprise. The only connection between them is the Internet and, for security, each company has a firewall protecting its internal network from the Internet.

The desired ordering system operates under a number of constraints:

- Internet connectivity between the two companies is not guaranteed to be available or error-free. Any communications between the two companies must be asynchronous and persistent, storing orders and responses when connectivity is not available and delivering them when connectivity returns, allowing the system to tolerate both transient errors and extended outages.
- For security reasons, the IT departments in both companies strictly limit what traffic is allowed through the firewall and what machines may receive that traffic directly.
- Orders and responses are confidential data. They must be protected against:
  - Information disclosure
  - Tampering
- Orders and responses represent financial transactions between the two companies.
  - Only certain employees of the manufacturer are authorized to place orders. Every order must be verified as originating from an authorized employee and be able to be tracked to the employee who placed it.
  - All responses from the supplier must be verified as originating from an authorized responder, such as an order processing application that is deployed for this purpose.
- One order entered by a manufacturer employee must result in exactly one order filled by the supplier; if the same order is received multiple times, this must not result in multiple orders filled by the supplier. Furthermore, orders must not be lost.

Copyright © 2011 Microsoft Corporation.

6.1.6.2 Message Queuing Operational Details

In order to meet these constraints, the two companies design two applications, an order entry application and an order processing application, using the Message Queuing System to deliver orders and responses across the Internet. The use of MSMQ automatically meets the asynchronous requirement of the first constraint, and the companies choose to use recoverable messages, as specified in section 3.2.1, meeting the persistence requirement of the first constraint.

The companies choose HTTPS as the transport, as specified in section 6.3.1 of this document and in [MC-MQSRM] section 2.1. This meets the second constraint, since HTTPS traffic is already allowed through the firewalls to certain servers. This traffic is not allowed to employee desktops, which limits the choices available to meet other constraints.

The encryption included in HTTPS protects against information disclosure, meeting the first part of the third constraint, protection against information disclosure.

MSMQ can protect messages against tampering, as specified in section 7.3. Since the two applications are in separate organizations, the companies deploy certificates provided by a trusted third-party certificate authority to enable this feature. These certificates are used to sign the messages, allowing the recipient to detect tampering if it occurs, meeting the second part of the third constraint, protection against tampering.

The same certificates that are used for tampering detection are also used to verify identity, meeting the fourth constraint. Every manufacturer employee who is authorized to order from the supplier receives a unique certificate, and those certificates are also provided to the supplier to allow orders to be verified for authorization and are tracked to the ordering employee. Similarly, the supplier obtains a certificate for the order processing application and provides it to the manufacturer to allow responses to be verified as genuine. Because these user certificates, as specified in section 7.4.1.3, are not registered in the directory service, MSMQ itself does not verify the sender's identity; instead, the application extracts the sender's certificate from the message and compares it against the locally stored set of valid certificates. A match provides a verified identity, which is used for tracking and authorization; lack of a match indicates an unauthorized sender.

MSMQ can provide an exactly-once delivery assurance to meet the fifth constraint, but doing so would require HTTPS traffic directly to the desktops that are running the order entry application, as specified in [MC-MQSRM] sections 3.1.1.2, 3.1.1.3.2, and 3.1.1.4.2, which the manufacturer's IT department will not allow. Instead, the companies design the applications to detect and handle transmission failures and duplicates.

The mechanism for detecting transmission failures is partly based on MSMQ's ability to generate acknowledgment messages, both positive (confirming that an operation has occurred) and negative (indicating that an operation failed), which allow a sending application to determine when a message has been successfully received by the destination application, or when there was a failure. These acknowledgments are specified in [MC-MQSRM] sections 3.1.1.3.1 and 3.1.1.3.4 and are also discussed in [MS-MQQB] section 1.3.5.2. This feature allows the order entry application to detect most failures early and to resend the order immediately. Any other failures are detected by the lack of a response from the order processing application within a timeout period specified in the applications' design, again causing the order entry application to resend the order.

The mechanism for detecting duplicates is based on a globally unique order identifier, which is generated and assigned to an order by the order entry application at the time that the order is initially sent. When the order processing application receives an order, it checks the order database for an existing order with the same identifier and retransmits the original response if the order is already present in the database. This mechanism does not use any MSMQ features, except that the order identifier is transmitted via MSMQ as part of the order.

Copyright © 2011 Microsoft Corporation.

Since HTTPS traffic is not allowed directly to the desktops running the order entry application, acknowledgments and response messages are sent to a server in the manufacturer's enterprise that is allowed to receive HTTPS traffic, and the order entry application receives the acknowledgments and responses from a queue on that server.

The use cases associated with this example are presented in sections 3.3.4.3, Send Message to Queue, and 3.3.4.6, Receive Message from Queue.

6.1.6.3 Message Queuing System View

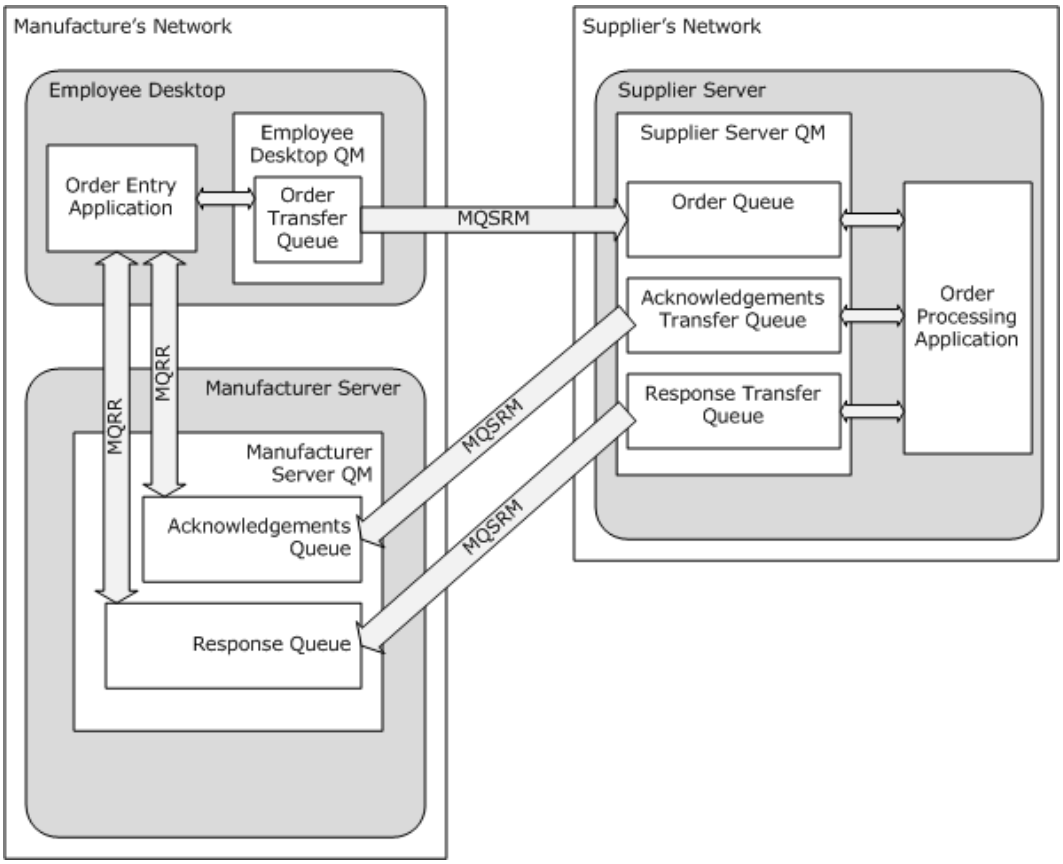


Figure 33: Message Queuing System view for Example 6

The message queue protocol usage is as follows:

- The order entry application plays the client role for the MQRR protocol, as specified in [MS-MQRR] section 3.2.
- The Employee Desktop Queue Manager plays the client role for the SRMP protocol, as specified in [MC-MQSRM].
- The Supplier Server Queue Manager plays both the client and server roles for the SRMP protocol, as specified in [MC-MQSRM] section 3.1.5.1.6.
- The Manufacturer Server Queue Manager plays the server role for the SRMP protocol and the server role for the MQRR protocol, as specified in [MS-MQRR] section 3.1.

Copyright © 2011 Microsoft Corporation.

6.1.6.4 Message Queuing Initial State

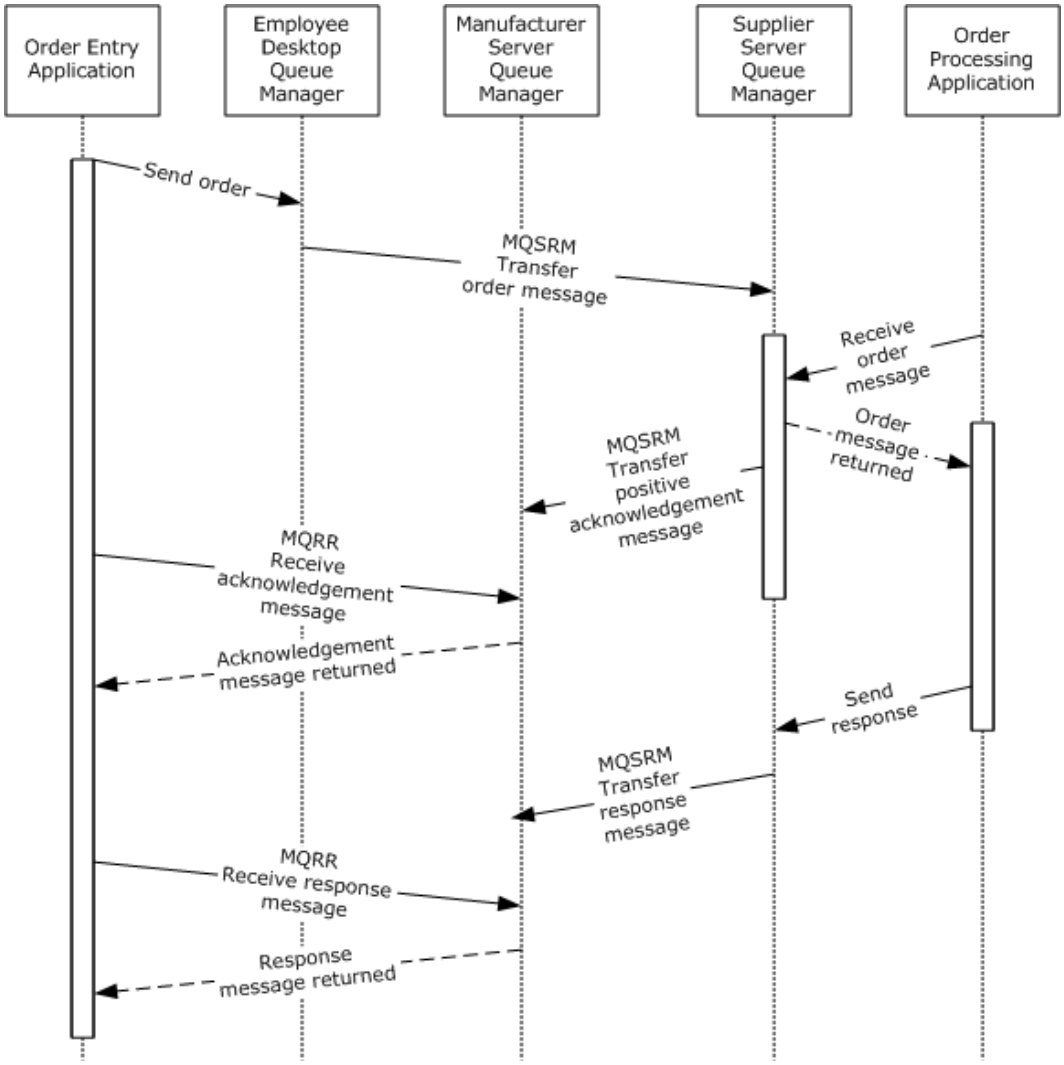
In order to execute this example, all three queue managers are brought to an initial state as described in section 6.6.

One queue, the order queue, is configured on the Supplier Server computer, with a name specified in the ordering system design, and the network address of the Supplier Server computer is provided to the order entry application.

Two queues, the Acknowledgments queue and the Response queue, are configured on the Manufacturer Server computer, with names specified in the ordering system design, and the network address of the Manufacturer Server computer is provided to the order processing application and the order entry application.

6.1.6.5 Sequence of Events

The following figure shows the sequence of events for business-to-business messaging across a firewall.



Copyright © 2011 Microsoft Corporation.

Figure 34: Sequence diagram for Example 6

In this example, an authorized manufacturer employee enters an order into the order entry application running on the Employee Desktop computer. The order entry application assigns the order a globally unique identifier and encapsulates the order into a message and assembles other necessary details such as the delivery assurances required (recoverable messages, section 3.2.1), the request to generate acknowledgments (signified in the message, as specified in [MC-MQSRM] section 2.2.5.2.3), the format names of the Acknowledgments queue and the Response queue, the Time To Be Received for the message, and the format name of the message's destination, which is the order queue. The application uses a local interface to pass the message and associated details to the Employee Desktop Queue Manager, as shown in the preceding figure. The Employee Desktop Queue Manager stores the message in an outgoing queue.

When network connectivity to the Supplier Server computer is available, the Employee Desktop Queue Manager transfers the order message to the Supplier Server Queue Manager, using SRMP over the HTTPS transport, as specified in [MC-MQSRM] section 3.1.1.1.2.1. The Supplier Server Queue Manager receives the message from the network, as specified in [MC-MQSRM] section 3.1.5.1.6, and stores it in the order queue.

In this example, it is up to the application to detect when a message was not transferred successfully. The order entry application uses MSMQ's Time To Be Received and acknowledgment features to determine the message's status, the server side of which are specified in [MC-MQSRM] sections 3.1.5.1.4 and 3.1.5.1.10. Because acknowledgments were requested, when the order processing application receives the message from the order queue, the Supplier Server Queue Manager sends a positive acknowledgment to the Acknowledgments queue, including the message ID ([MC-MQSRM] section 2.2.4.3) of the original message. The order entry application attempts to receive a message from the Acknowledgments queue. When the order entry application receives a positive acknowledgment, it switches to attempting to receive a message from the Response queue.

Not shown in the sequence diagram is what happens if the order processing application does not receive the message before the Time To Be Received expires. In that case, the queue manager holding the message at the time of expiry (which may be the Employee Desktop Queue Manager or the Supplier Server Queue Manager) sends a negative acknowledgment, using SRMP over HTTPS, to the Acknowledgments queue, including the message identification of the original message and the reason for the negative acknowledgment ([MC-MQSRM] section 2.2.6.1). The order entry application receives the negative acknowledgment and retransmits the original order message.

The order processing application receives the order message from the order queue via a local interface. The order processing application extracts the order and checks whether the order's identifier is already present in the order database. In this example, the identifier is not present, so the order is not a duplicate. The order processing application processes the order and creates a response. The order processing application encapsulates the response into a message and uses a local interface to pass the message to the Supplier Server Queue Manager, along with other necessary details such as the delivery assurances required (recoverable messages, section 3.2.1) and the format name of the message's destination, which is the Response queue. The Supplier Server Queue Manager stores the message in an outgoing queue.

When network connectivity to the Manufacturer Server computer is available, the Supplier Server Queue Manager transfers the response message to the Manufacturer Server Queue Manager, using SRMP over HTTPS. The Manufacturer Server Queue Manager receives the message from the network and stores it in the Response queue.

When the order entry application receives the response message, it displays the response to the employee. Not shown in the sequence diagram is what happens if the order entry application does not receive a response message within a timeout period specified in the application's design. In that

Copyright © 2011 Microsoft Corporation.

case, the order entry application assumes a failure in the order processing application and retransmits the original order message.

6.1.6.6 Message Queuing Final State

The final state of MSMQ in this example is the same as the initial state.

6.1.7 Example 7: Server Farm

6.1.7.1 Scenario Description

A national insurance company employs agents located all over the country who sell insurance policies. Before each policy is sold, the agent collects information about the applicant and the item(s) to be insured and sends the information electronically to the company's central office. There, the applicant data is analyzed and a price quote is generated for the agent to give to the applicant.

The system constraints are as follows:

- Because the data being sent is personally identifiable and confidential, the system needs to guard against information disclosure and to ensure that the information received at the central office originated from one of their agents.
- Because applicant data loss is unacceptable, the company's system needs a guarantee that customer data will be processed.
- The agent enters the information into the sales office application during the applicant's appointment, so the applicant data must be processed quickly to give the applicant a quote during the appointment.

6.1.7.2 Message Queuing Operational Details

To meet the first system constraint, the company uses message queuing encryption and authentication (as described in section 7.4.1.4) to secure the applicant data as it travels to the central office.

The company uses a single message queue on a single intermediary server as the location to which all applicant data is sent, because the traffic does not require more messaging capacity than a single server can provide. However, each set of applicant data requires substantial analysis, and a single analysis server cannot keep up with the flow of incoming data. Therefore, in order to maintain a reasonable response time, the analysis system runs on multiple servers, separate from the intermediary server. This fulfills the second constraint.

When the company built the system, there was no option to receive remote messages in a transaction, so in order for the analysis servers to receive applicant data messages transactionally from the intermediary server, the intermediary server was configured as a supporting server, and the analysis servers were configured to use the MQMP protocol to communicate with the intermediary server. This strategy allows the system to take advantage of the benefits of transactions, while distributing the intensive data analysis across multiple computers, which satisfies the third constraint, and contributes to the second constraint.

The use cases associated with this example are covered in sections 3.3.4.3, Send Message to Queue, and 3.3.4.6, Receive a Message from a Queue.

Copyright © 2011 Microsoft Corporation.

6.1.7.3 Message Queuing System View

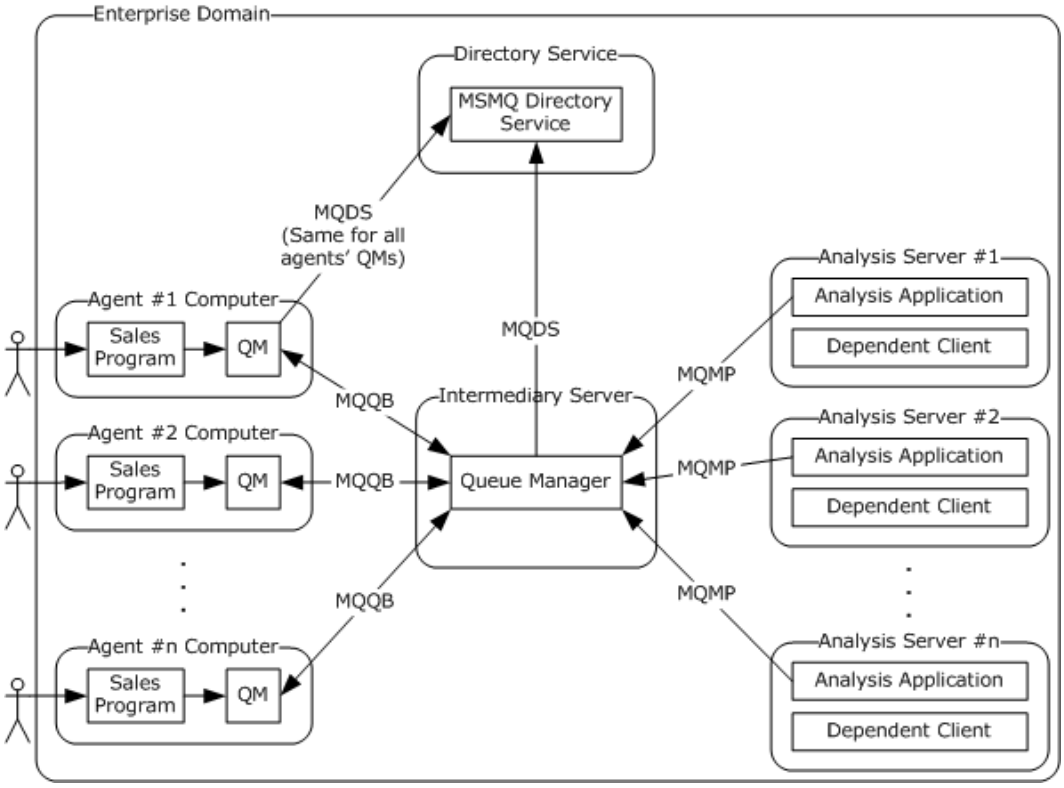


Figure 35: System view for Example 7

The message queue protocol usage is as follows:

- The Agent Computer Queue Managers play the client role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.2, as well as the server role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.8.
- The Intermediary Server plays the server role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.8, as well as the client role for the MQQB protocol, as specified in [MS-MQQB] section 3.1.5.2. It also plays the server role for the MQMP protocol, as specified in [MS-MQMP] section 3.2.4.12.
- The Analysis Application plays the client role for the MQMP protocol, as specified in [MS-MQMP] section 1.3.

6.1.7.4 Message Queuing Initial State

To execute this example, the queue managers on the agents' computers and the Intermediary Server are brought to an initial state as described in section 6.6, Initialization. The agents' computers are operating in the Queue Server role, and the Intermediary Server is operating in the Supporting Server role. The analysis servers are operating in the application role.

The queue manager on the Intermediary Server is initialized to contain a transactional queue and configured to accept only authenticated messages. The queue managers on the agents' computers are initialized to contain queues to which response messages are sent by the analysis applications.

Copyright © 2011 Microsoft Corporation.

6.1.7.5 Sequence of Events

The following diagram shows the sequence of events for the server farm example.

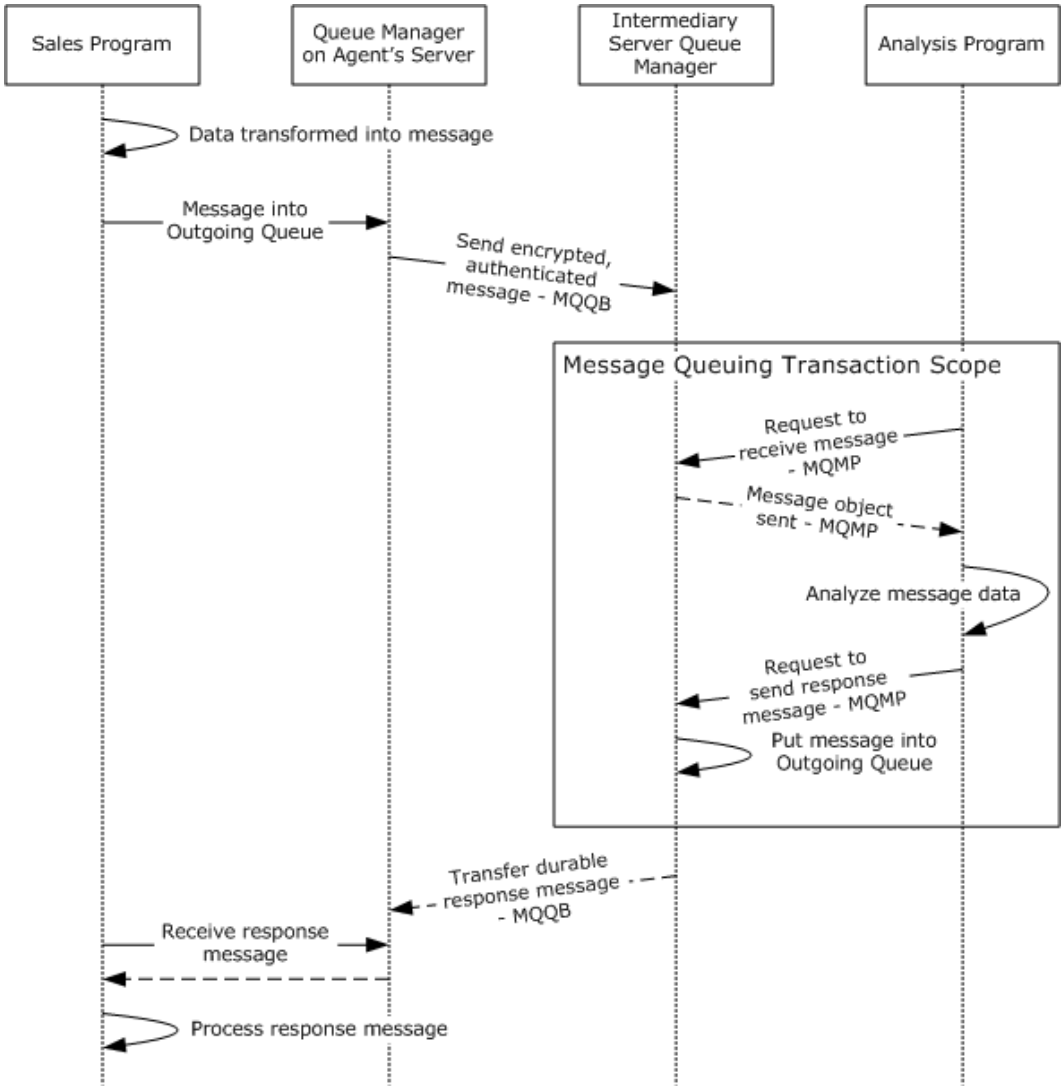


Figure 36: Server farm message sequence

The process starts when the agent submits the applicant information to the sales program. The program transforms the information into an MSMQ message, specifies the use of authentication and encryption on the message, and gives it to the local queue manager through a private interface, along with information about both the destination queue and the queue on the agent's computer to which the response message should be sent after processing. The queue manager puts the message into an outgoing queue and sends it over the MQQB protocol to the message queue on the Intermediary Server.

When an analysis program is ready to analyze a message, it creates an MSMQ transaction scope and sends a request to its supporting server, the Intermediary Server, over the MQMP protocol to receive the next message from the MSMQ queue. The Intermediary Server queue manager receives

Copyright © 2011 Microsoft Corporation.

the next message off the queue and sends the message object over the MQMP protocol to the analysis program. Still inside the transaction, the message is analyzed and the analysis program sends a request to the Intermediary Server to send a recoverable message to the queue manager on the agent's computer. The Intermediary Server puts the requested message in an outgoing queue and the message is sent to the queue on the agent's computer that was specified in the message (see [MS-MQMP] section 4.5 for more information). The transaction scope ends here.

The response message waits in the queue until the sales program receives it from the queue through a private interface and informs the insurance agent of the analysis results.

6.1.7.6 Message Queuing Final State

The final state for the queue managers is equal to their initial state. The Intermediary Server Queue Manager contains a transactional queue, and the queue is in a ready state. The agent's queue manager contains a queue, and the queue is in the ready state.

6.1.8 Example 8: Stock Ticker

6.1.8.1 Scenario Description

An investment company has a feed of stock ticker information from a third party. In addition to using this information for their trading, they wish to display the current prices of selected stocks in various common areas around their buildings, such as lobbies and lunch rooms, especially those that may be visited by non-employees. These displays will allow traders to follow the market while not at their desks and will reinforce the "high-tech" image that the company wishes to portray to the public.

The display system must operate under a number of constraints:

- Each display point consists of a computer and monitor, running an application that obtains the ticker information from the network and displays it in the desired format. The ticker information is placed on the network by a separate computer located in the data center.
- For ease of use, it should be possible to set up new display points and remove existing ones without requiring configuration of any part of the system except the display point itself.
- By the nature of a stock ticker, the price for any particular stock is updated periodically and frequently. Prices other than the most recent available are not of interest, so there is no requirement for persistence of pricing information.
- The transmission of prices to the displays should impose as low a load on the network and the display point computers as possible.

6.1.8.2 Message Queuing Operational Details

To meet these constraints, the company designs a pair of applications, one to obtain the stock ticker information from the third party and send it to the display points (the "back-end application"), and one to run on the display points and display the information (the "display application"). For communications between the back end and the display application, the company decides to use the message queuing system, which allows them to be separated, meeting the first constraint.

The back-end application takes the incoming stock ticker information and converts it to message queuing (MSMQ) messages, then sends those messages using the multicast feature of the MSMQ, described in [MC-MQSRM] sections 1.3.11 and 2.1. The multicast feature allows a sender to send to all queues configured with a particular IP multicast address. This decouples the sender from the number and location of receivers, so display points can be deployed and removed as desired without

Copyright © 2011 Microsoft Corporation.

impacting the back-end application or other display points in any way, meeting the second constraint.

Volatile messages, described in section 3.2.1, are used because persistence is explicitly not desired, as stated in the third constraint, and because they impose a lower load on both the sending and receiving computers, contributing to meeting the fourth constraint.

The multicast feature also contributes to meeting the fourth constraint, since a single multicast message serves all the display points. Otherwise, the back-end application would have to send multiple messages containing the same information, one for each display point.

The use cases associated with this example are covered in section 3.3.4.3, Send Message to Queue, and section 3.3.4.6, Receive a Message from a Queue.

6.1.8.3 Message Queuing System View

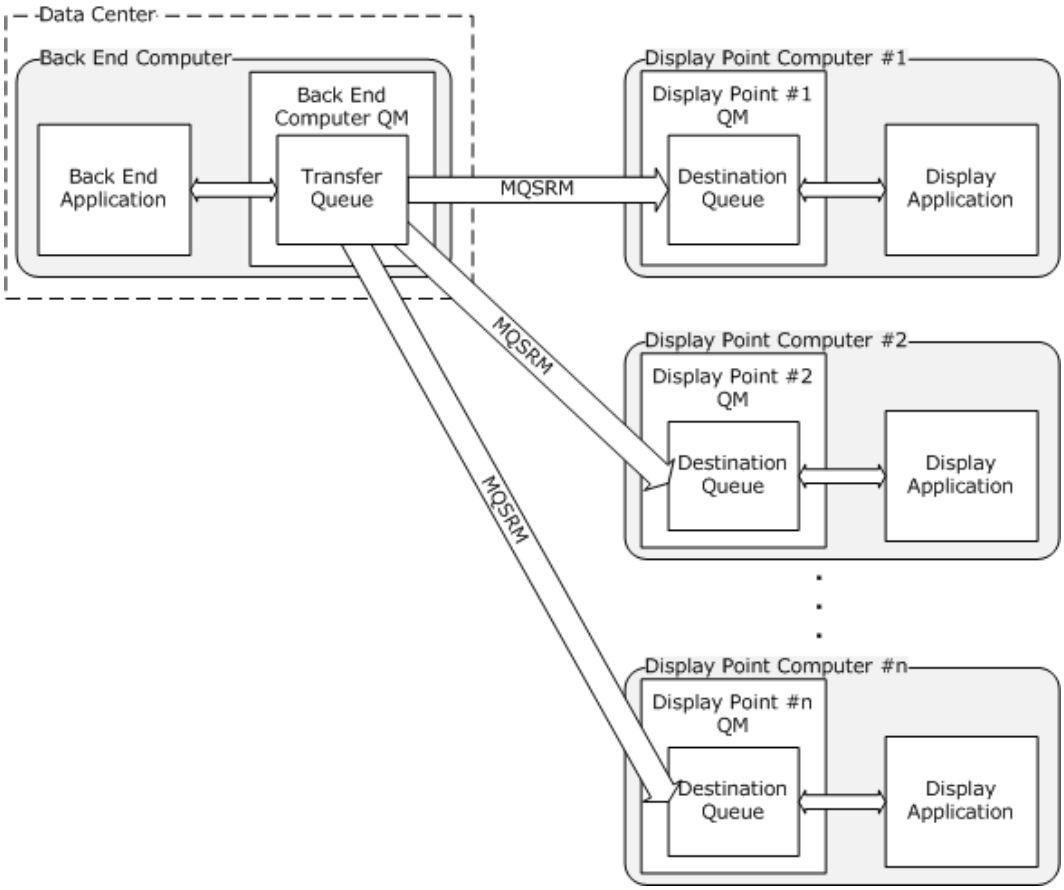


Figure 37: System view for Example 8

The message queue protocol usage is as follows:

- The Back End Computer Queue Manager plays the client role for the MQSRM protocol, as specified in [MC-MQSRM].

Copyright © 2011 Microsoft Corporation.

- The queue managers on the Display Point Computers play the server roles for the MQSRM protocol, as specified in [MC-MQSRM] section 3.1.5.1.6.

6.1.8.4 Message Queuing Initial State

In order to execute this example, all queue managers are brought to an initial state as described in section 6.6. There is one queue manager on the Back End Computer and one on each Display Point Computer.

One queue is configured on each Display Point Computer and each queue is configured with an IP multicast address. The queue name and IP multicast address are specified in the design of the applications.

6.1.8.5 Sequence of Events

The sequence of events for the stock sticker example is shown in the following figure.

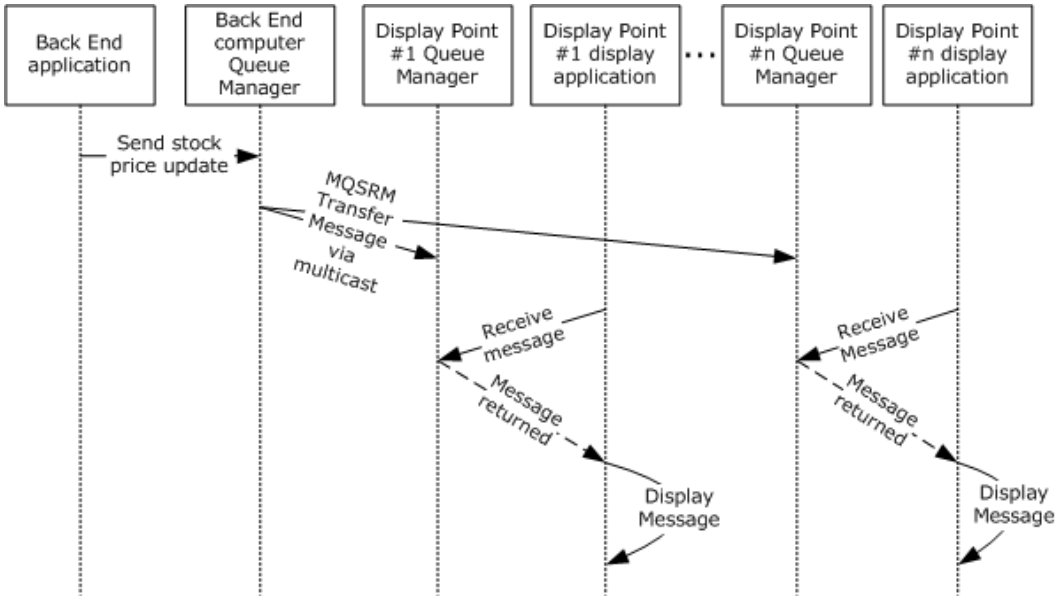


Figure 38: Sequence diagram for Example 8

In this example, the back-end application is running on the Back End Computer in the data center. The back-end application processes a stream of stock ticker information, consisting of stock exchange company symbols and the corresponding stock prices, extracting the symbols and prices for companies of interest. It encapsulates each pair of ticker symbol and price into a message and assembles other necessary details such as the IP multicast address to which the message should be sent, which is specified in the application design. The application uses a local interface to pass the message and associated details to the Back End Queue Manager, as shown in the preceding diagram. The Back End Queue Manager stores the message in an outgoing queue.

When network connectivity is available, the Back End Queue Manager transfers the message to the Display Point Queue Managers using MQSRM over the PGM transport, as described in [RFC3208] and in [MC-MQSRM] sections 1.3.11 and 2.1.3.

On each Display Point Computer, the queue manager located on that computer has configured the PGM transport to receive messages on the IP multicast address specified in the application design.

Copyright © 2011 Microsoft Corporation.

The message is distributed across the network to the Display Point Queue Managers as specified in [RFC3208]. Each Display Point Queue Manager stores the message in the destination queue hosted on that Display Point Computer. The display applications on the Display Point Computers then receive the message, extract the symbol and price, and display them.

6.1.8.6 Message Queuing Final State

The final state of MSMQ in this example is the same as the initial state.

6.1.9 Example 9: Business-to-Business Messaging Across Heterogeneous Systems

6.1.9.1 Scenario Description

A small manufacturing company has a major customer that is moving to electronic communications for their supply-chain management. The customer has Message Queuing (MSMQ) deployed in their enterprise and requires all their suppliers to use it for communications with them. However, the small manufacturing company already has a different message queuing system deployed; the cost and disruption of replacing it potentially outweighs the value of moving to a new system. A way to pass messages between the two message queuing systems, operating transparently to applications using the systems, solves this conflict.

6.1.9.2 Message Queuing Operational Details

MSMQ supports the ability to route messages to queues hosted on other message queuing systems and vice versa. These foreign queues are represented in MSMQ's Directory Service using the same objects that represent public queues, but they are marked as being foreign and the objects are created manually by administrators. The actual translation of messages from one system to the other is handled by a **connector application** on a **connector server**. The connector application itself is not part of MSMQ. It is an application that uses a specific set of features provided by MSMQ and provides a specific set of services.

The manufacturing company finds that a connector application is commercially available that supports the message queuing system deployed in their enterprise, so they configure and deploy a connector server on a messaging gateway computer in the customer's enterprise. This strategy establishes communications between the two systems and allows the manufacturer to integrate with their customer's supply-chain management without having to replace their message queuing system.

The use cases associated with this example are covered in sections 3.3.4.3, Send Message to Queue, 3.3.4.5, Transfer Message, and 3.3.4.6, Receive a Message from a Queue.

Copyright © 2011 Microsoft Corporation.

6.1.9.3 Message Queuing System View

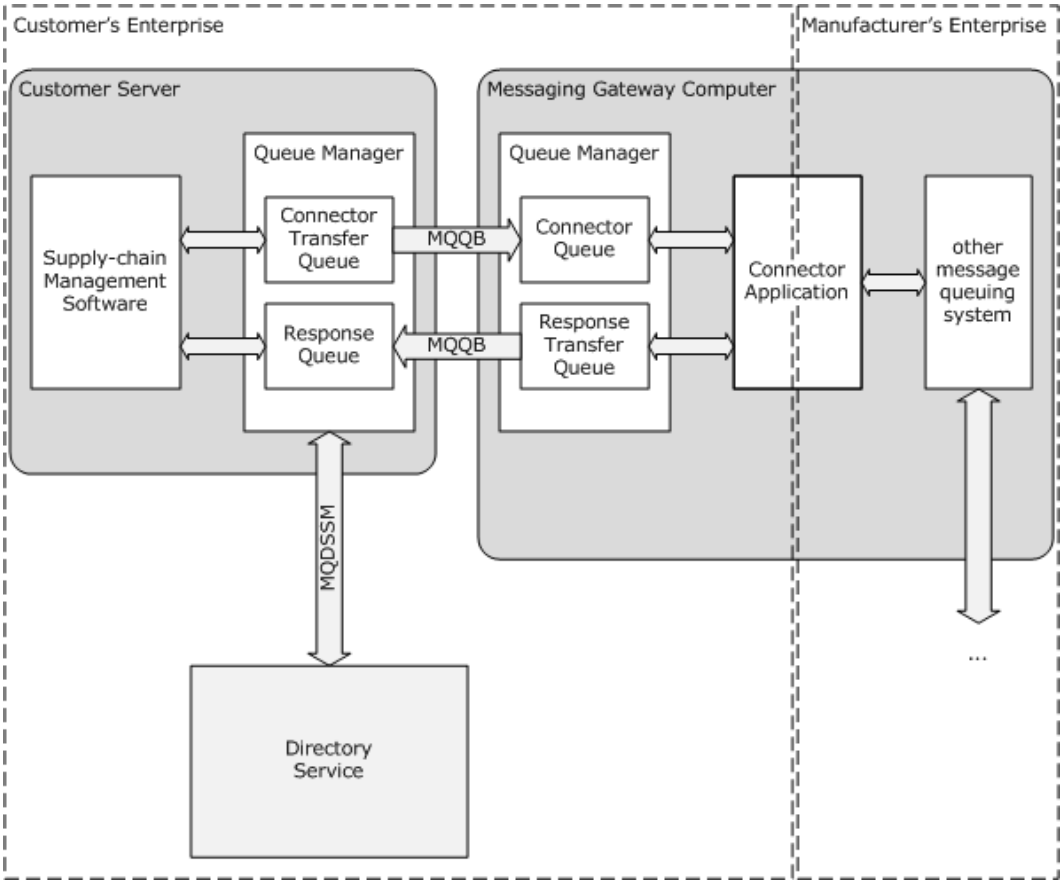


Figure 39: System view for Example 9

The Customer Server Queue Manager plays both client and server roles for the MQQB protocol, as specified in [MS-MQQB] sections 3.1.5.2 and 3.1.5.8; plays the client role for LDAP using MQDSSM to access the Directory Service; and uses the routing algorithms described in [MS-MQBR].

6.1.9.4 Message Queuing Initial State

In order to execute this example, all queue managers are brought to an initial state as described in section 6.6.

The client software for the manufacturer's message queuing system is installed on the Messaging Gateway computer, and then the connector application is installed, which makes the Messaging Gateway computer a connector server. The Connector queue on the Messaging Gateway computer is created as part of the connector application installation.

One queue, the Response queue, is configured on the Customer Server, to which responses from the manufacturer are directed.

An administrator creates a **foreign queue** object in the Directory Service, representing the destination queue for messages traveling from the customer's server to the manufacturer.

Copyright © 2011 Microsoft Corporation.

6.1.9.5 Sequence of Events

The sequence of events for the business-to-business messaging across heterogeneous systems example is shown in the following diagram.

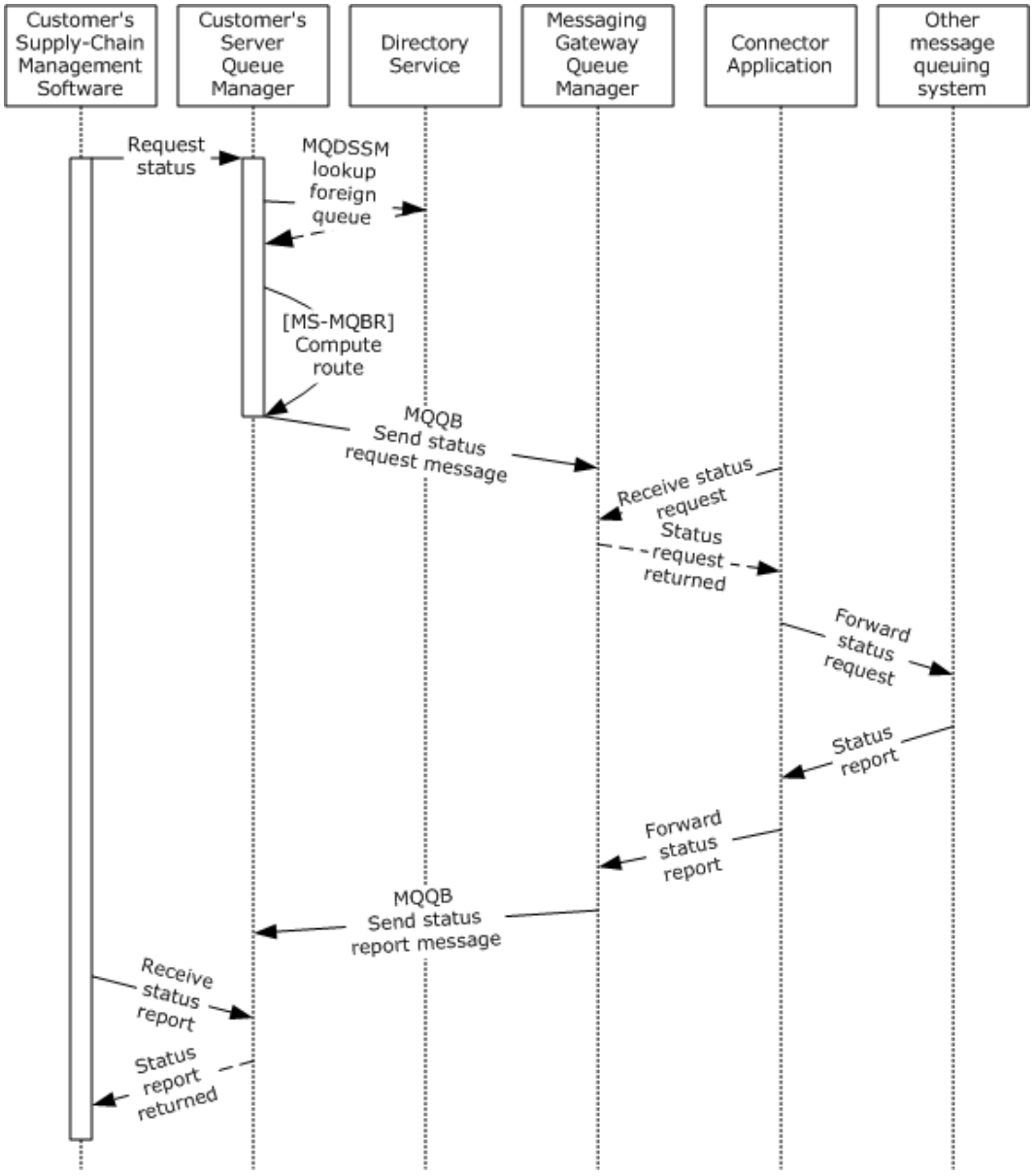


Figure 40: Sequence diagram for Example 9

In this example, the customer's supply-chain management software sends a status report request to the manufacturer's supply-chain management software. The status report request is encapsulated in an MSMQ message along with necessary details such as the destination queue for the message. The customer's supply-chain management software uses a local interface to pass the message and associated details to the Customer Server Queue Manager. The Customer Server Queue Manager stores the message in an outgoing queue and looks up the destination queue in the Directory

Copyright © 2011 Microsoft Corporation.

Service. Discovering that the destination is a foreign queue, the Customer Server Queue Manager computes the route to the foreign queue as specified in [MS-MQBR]. The next hop to get to the foreign queue is the Connector Queue on the Messaging Gateway computer, so the Customer Server Queue Manager transfers the message to the Messaging Gateway computer using MQQB when network connectivity is available.

The Messaging Gateway Queue Manager receives the message from the network and stores it in the Connector Queue. The connector application receives the message from the Connector Queue and forwards the message to its final destination in the manufacturer's enterprise. The details of forwarding are specific to the connector application and the manufacturer's message queuing system and are not included in this example.

When the connector application receives a response, it determines where to send it, which in this case is the Response Queue on the Customer Server. Again, the details of how the connector application translates a destination expressed in the manufacturer's message queuing system syntax to a destination in MSMQ are specific to the connector application and the manufacturer's message queuing system and are not included in this example. The connector application uses a local interface to pass the message and associated details, such as the message's destination, to the Messaging Gateway queue manager. The Messaging Gateway queue manager stores the message in an outgoing queue and transfers it via MQQB to the Customer Server when network connectivity is available.

The Customer Server Queue Manager receives the message from the network and stores it in the destination queue. The customer's supply-chain management software receives the message, extracts the status report, and the process completes.

6.1.9.6 Message Queuing Final State

The final state of MSMQ in this example is the same as the initial state.

6.2 Communication Details

The communications within the Message Queuing System and between the system and external entities are described in sections 5.4.1 and 5.4.2 respectively. The Message Queuing System does not define any communication constraints or additional message types beyond those described in the specifications of the protocols supported by the system, as listed in section 2.2.

6.3 Transport Requirements

This section describes the transports that are used within the system as well as with external entities.

6.3.1 Transports Used Within the System

A queue manager uses a transfer protocol to transfer messages to another queue manager. The Message Queuing System provides two message transfer protocols. The queue manager MUST use the transfer protocol specified by the application.

The MQQB protocol is used to transfer a message from an outgoing queue to a destination queue with a range of delivery assurances. This block protocol uses either TCP/IP or SPX/IPX as the underlying data transport. Details about this protocol are specified in [MS-MQQB].

The SRMP protocol is also used to transfer messages from an outgoing queue to a destination queue with a range of delivery assurances. The SRMP protocol uses SOAP 1.1 [SOAP1.1] as its message format. For its transport, the protocol uses HTTP 1.1 [RFC2616] or PGM [RFC3208]. Details about

Copyright © 2011 Microsoft Corporation.

this protocol are specified in [MC-MQSRM]. The use of PGM with SRMP enables best-effort multicast of messages to multiple destination queues.

The Message Queuing System uses the transports shown in the following table to communicate within the system.

Transports used within the system

Transport	Protocols
TCP/IP	MQQB, MQCN
IPX/SPX	MQQB, MQCN
HTTP	SRMP
HTTPS	SRMP
UDP	MQQB, MQSD
PGM	SRMP
RPC as specified in [C706] and [MS-RPCE]	MQQP, MQRR, MQDS

6.3.2 Transports Used Between the System and External Entities

Applications communicate with the queue manager over several RPC-based and DCOM-based protocols. Applications and queue managers interact with the Directory Service through LDAP.

The following table lists the transports used by the Message Queuing System to communicate with external entities.

Transports used between the system and external entities

Transport	Protocols
RPC	MQMP, MQMR
LDAP	As specified in [MS-ADTS]
DCOM	MQAC

6.4 Timers

The timers of the Message Queuing System are described in the specifications of the protocols supported by the system, as listed in section 2.2. The timers described in this section are those that are important to the state of the overall system, and the Message Queuing System SHOULD maintain these.

6.4.1 Directory Service Synchronization Timers

This section describes the various synchronization timers that the queue manager uses to synchronize the directory. These timers are enabled only if the Message Queuing System is operating in Directory-Integrated with Routing mode as described in section 4.3.1.3.

Copyright © 2011 Microsoft Corporation.

6.4.1.1 Directory Sites Update Timer

This timer is described in [MS-MQDMPR] section 3.1.2.1.

6.4.1.2 Directory Server List Update Timer

This timer controls the periodic update of the **DirectoryServerList** ([MS-MQDMPR] section 3.1.1.1) ADM attribute of the local **QueueManager** ADM element instance. The duration of this timer is implementation-specific.<7> The timer is started when the **DirectoryServerList** ADM attribute of the local **QueueManager** ADM element instance is updated. Upon expiration of this timer the Queue Manager performs the following actions:

- The Queue Manager SHOULD <8> use LDAP over Active Directory, as specified in [MS-ADTS] sections 7.1.1.3.1 and 7.1.1.3.2, to retrieve the **NetBIOS computer names** of the Active Directory **domain controllers (DCs)** for the domain to which the queue manager computer belongs and SHOULD add these names to the **DirectoryServerList** ADM attribute of the local **QueueManager** ADM element instance.

6.4.1.3 Site Gate List Update Timer

This timer is described in MS-MQDMPR section 3.1.2.1.

6.4.2 Message Timers

The message timers are described in [MS-MQDMPR] section 3.1.2.4, and the associated timer events are described in [MC-MQAC] section 3.1.6.1.1.

6.4.3 Security Timers

This section discusses the following security timers:

- Certificate Data Cache Validity Timer
- Queue Manager Public Key Cache Validity Timer

6.4.3.1 Certificate Data Cache Validity Timer

As described in section 7.4.1.3, user certificates are used by applications to sign application messages. The queue manager ensures the message integrity and authentication using the User.Certificates data element as described in [MS-MQDMPR] section 3.1.1.15. For performance reasons, the queue manager SHOULD cache the certificate data in an internal data cache. This timer regulates the amount of time that a queue manager operating in Directory-Integrated mode waits before invalidating the internal certificate data cache. The default, minimum, and maximum values of this timer are implementation-specific.<9> This timer is initialized every time the queue manager populates the certificate data cache from the Directory Service.

The queue manager MUST perform the following action when this timer expires:

- Clear the internal certificate data cache.

6.4.3.2 Queue Manager Public Key Cache Validity Timer

As described in [MS-MQDMPR] section 3.1.1.1, a queue manager operating in Directory-Integrated mode maintains the directory-attributes QueueManager.PublicEncryptionKeyList and QueueManager.PublicSigningKeyList. For performance reasons, these directory attributes are cached

Copyright © 2011 Microsoft Corporation.

locally within each queue manager. This timer regulates the amount of time that the queue manager waits before invalidating the cache. The default, minimum, and maximum values of this timer are implementation-specific.<10> This timer is initialized every time the queue manager populates the public key cache from the Directory Service.

The queue manager MUST perform the following action when this timer expires:

- Clear the internal Queue Manager Public Key Cache entry.

This event has no argument or return value.

6.4.4 Initialization Timers

This section discusses the following initialization timer:

- Initialization Retry Timer

6.4.4.1 Initialization Retry Timer

This timer triggers a new attempt to initialize the Queue Manager. The timer is set when the initialization fails. The duration of the timer is the number of seconds specified in **QueueManager.InitializationRetryTimerDuration**. Upon expiration of this timer the Queue Manager MUST perform the initialization steps described in section 6.6.2.

6.5 Non-Timer Events

The system does not define any additional non-timer events beyond those described in the specifications of the protocols supported by the system, as listed in section 2.2.

6.6 Provisioning and Initialization Procedures

Provisioning of the Message Queuing system MUST start with the following arguments:

- *iQueueManager*: A Boolean value that indicates whether the Message Queuing system is provisioned as a **queue manager**. A default value True SHOULD be used regardless of the supplied value.<11>
- *iSupportingServer* (optional): The address of a Message Queuing server that acts as a supporting server. This argument MUST be specified if *iQueueManager* is False.

If *iQeueManager* is False, the argument *iSupportingServer* MUST be persisted. The software components that implement the **client** side of [MS-MQMP] SHOULD be deployed to facilitate communications between applications and a supporting server through the MQMP protocol.

If *iQueueManager* is True, a queue manager MUST be provisioned and initialized as specified in the following sections to support the queue manager roles specified in section 5.2.1.2.

6.6.1 Provisioning of a Queue Manager

As described in section 5, the queue manager is the central piece of the Message Queuing system. A queue manager is deployed on an individual machine in order to participate in the Message Queuing system. This section describes a conceptual model for the provisioning of a queue manager. The model described in this section is provided to facilitate the explanation of deploying and activating a queue manager in a Message Queuing system. This document does not mandate that implementations adhere to this model as long as their external behavior is consistent with the behavior that is described in this section.

Copyright © 2011 Microsoft Corporation.

Provisioning of a queue manager MUST be triggered with the following arguments:

- *iDirectoryIntegrated*: A Boolean value that indicates whether the queue manager is integrated with a directory service.
- *iClustered*: A Boolean that specifies whether the queue manager is part of a cluster.
- *iDirectoryServer (optional)*: A Boolean value that indicates whether the queue manager will provide directory service. The default value is FALSE.
- *iDirectoryServerType (optional)*: An enumeration that specifies the type of directory service provided by the queue manager. This argument MUST be ignored if iDirectoryServer is FALSE.
- *iRoutingServer (optional)*: A Boolean value that indicates whether the queue manager is configured as an MSMQ Routing Server. The default value is FALSE.
- *iSrmpHttpSupport (optional)*: A Boolean value that indicates whether the queue manager supports HTTP-based **SRMP** message transfer. The default value is FALSE.
- *iSrmpPgmSupport (optional)*: A Boolean value that indicates whether the queue manager supports PGM-based SRMP message transfer. The default value is FALSE.

Provisioning of a queue manager starts with loading the software components that implement the queue manager. When the queue manager starts, it MUST perform the following actions:

- Create an instance of the local **QueueManager** object.
- Initialize QueueManager.Identifier to a new GUID.
- Initialize QueueManager.LastDomain to NULL.
- Initialize QueueManager.Security to NULL.
- Set QueueManager.DirectoryIntegrated to *iDirectoryIntegrated*.
- Set QueueManager.Clustered to *iClustered*.
- Set QueueManager.SrmpHttpSupport to *iSrmpHttpSupport*.
- Set QueueManager.SrmpPgmSupport to *iSrmpPgmSupport*.
- If the QueueManager.DirectoryIntegrated element is TRUE, initialize the following attributes; otherwise, these elements remain uninitialized.
  - Set QueueManager.DirectoryServer to *iDirectoryServer*.
  - Set QueueManager.DirectoryServerType to *iDirectoryServerType*.
  - Set QueueManager.RoutingServer to *iRoutingServer*.
- Initialize the remaining persistent attributes and directory attributes of the **QueueManager** object and set them to their respective default values as specified in [MS-MQDMPR] section 3.1.1.
- Initialize the queue manager by executing the steps as specified in [MS-MQDMPR] section 3.1.3 with the argument *iProvisioning* set to TRUE.
- Initialize the queue manager by executing step 2 to step 4 as specified in section 6.6.2.

Copyright © 2011 Microsoft Corporation.

- End the provisioning process. The queue manager enters the Running state.

The queue manager MUST handle errors raised by the provisioning steps. On error, the queue manager MUST abort the provisioning procedure and return a failure in an implementation-specific manner so that another attempt to provision the queue manager can be made at a later time.

6.6.2 Queue Manager Initialization

The queue manager and all of its subroles SHOULD initialize as described in the following steps:

1. Initialize the queue manager by executing the steps as specified in [MS-MQDMPR] section 3.1.3 with the argument *iProvisioning* set to FALSE.

2. If the queue manager is operating in Directory-Integrated mode (that is, the **QueueManager.DirectoryIntegrated** ADM attribute is TRUE), it SHOULD <12> perform the following sequence of steps:

   - Let *domainChangeStatus* be a string initialized to empty.

   - Compare the **QueueManager.MachineDomainId** ADM attribute with the **QueueManager.LastDomain** ADM attribute and handle any differences as per the following table. Subsequently set the **QueueManager.LastDomain** ADM attribute and *domainChangeStatus* as follows:

     Domain join status and action

Condition	domainChangeStatus	Action
QueueManager.LastDomain is the same as **QueueManager.MachineDomainId**.	"No change"	No action.
QueueManager.LastDomain is empty, and **QueueManager.MachineDomainId** is not empty.	"Joined a new domain"	Create necessary Directory Service objects related to the system as specified in section 6.6.2.1. Set **QueueManager.LastDomain** to **QueueManager.MachineDomainId**.
QueueManager.LastDomain is not empty, and **QueueManager.MachineDomainId** is empty.	"Left a domain"	Move the system to Workgroup mode as specified in section 6.6.2.2. Set **QueueManager.LastDomain** to empty.
QueueManager.LastDomain and **QueueManager.MachineDomainId** are not empty, and they do not match.	"Changed domains"	Create necessary Directory Service objects related to the system as specified in section 6.6.2.2. Set **QueueManager.LastDomain** to **QueueManager.MachineDomainId**.

3. If the Message Queuing System is operating in Directory-Integrated mode with routing as described in section 4.3.1.3 (the **QueueManager.RoutingServer** ADM attribute is set to TRUE), the queue manager initializes the necessary routing information as specified in [MS-MQBR] section 3.1.3.

Copyright © 2011 Microsoft Corporation.

4. Initialize and start listening on the configured transports specified in section 6.3.

5. End the Queue Manager Initialization process. The queue manager enters the Running state.

The queue manager MUST handle errors raised by the initialization steps specified above. On error, the rest of the initialization MUST be aborted, and the following steps SHOULD<13> be executed to perform the initialization retry process, as described in section 5.1.1.1.

- Start the Initialization Retry Timer (section 6.4.4.1) with a timeout of the number of seconds specified in **QueueManager.InitializationRetryTimerDuration**.
- If **QueueManager.InitializationRetryTimerDuration** ([MS-MQDMPR] section 3.1.1.1) equals 120, set **QueueManager.InitializationRetryTimerDuration** to 900.

6.6.2.1 Interactions with Directory Service on Domain Join or Changes

The Message Queuing system SHOULD<14> initialize directory service objects when the machine on which the queue manager is deployed either moves to a new domain or joins a domain for the first time. The control flow for handling the machine's domain join status change is shown in the following diagram.

Copyright © 2011 Microsoft Corporation.

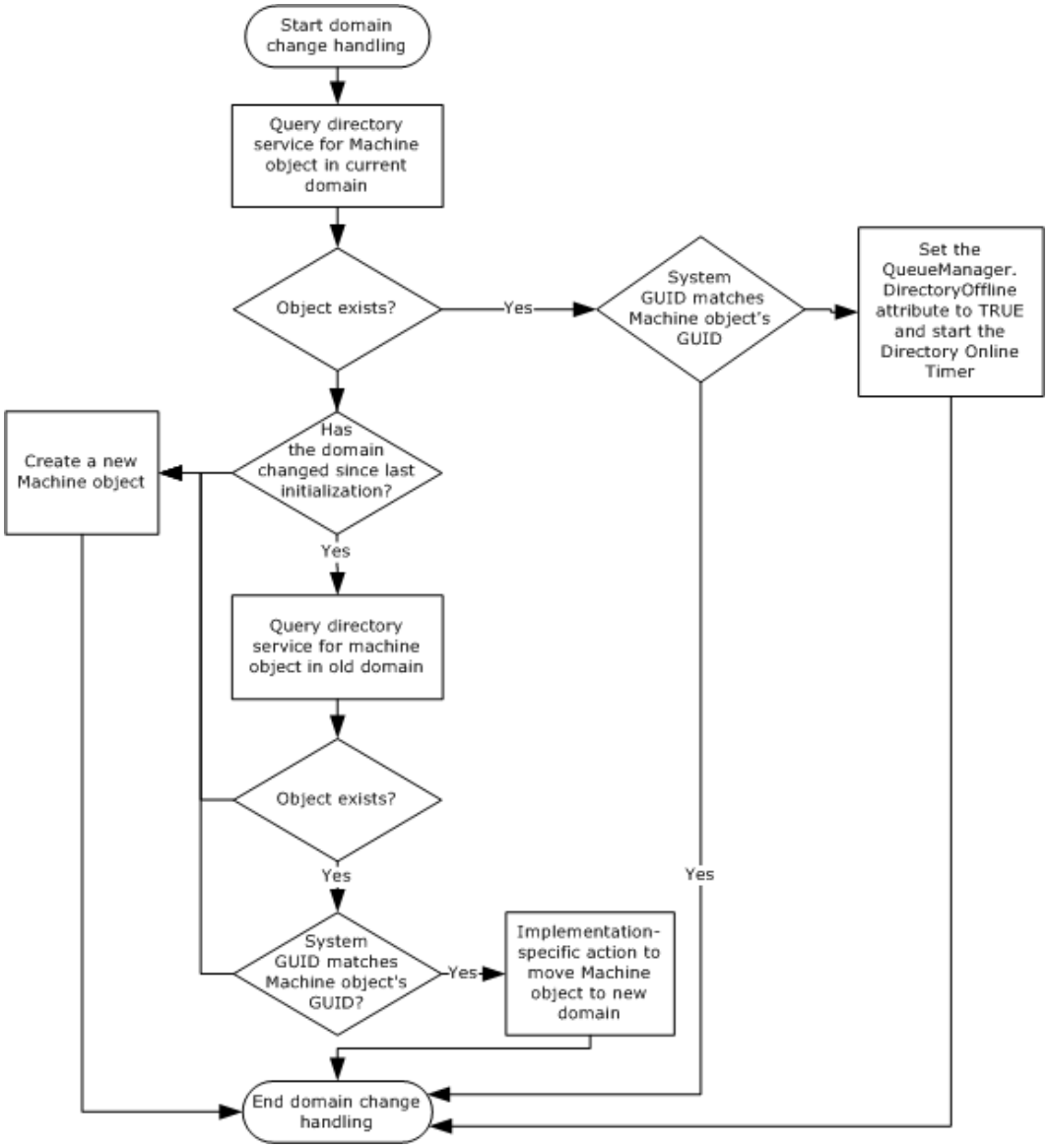


Figure 41: Handling change of domain join status

The queue manager MUST follow these steps for handling a change of domain join status:

1. Query the Directory Service for the Machine Directory Service object in the new domain by generating a Read Directory ([MS-MQDMPR] section 3.1.7.1.20) event with the following arguments:

```
iDirectoryObjectType := "QueueManager"
iFilter := {"QualifiedComputerName" EQUALS QueueManager.QualifiedComputerName}
```

   If *rStatus* of the preceding event is neither **DirectoryOperationResult.Success** nor **DirectoryOperationResult.ObjectNotFound**, raise an error, and take no further action.

Copyright © 2011 Microsoft Corporation.

2. If *rStatus* is **DirectoryOperationResult.Success**:

   - Treat the returned *rDirectoryObject* as a **QueueManager** ADM element and compare **QueueManager.Identifier** with *rDirectoryObject.Identifier*.
   - If the identifiers do not match, set **QueueManager.DirectoryOffline** to TRUE, and start the Directory Online Timer.
   - Complete the domain change handling.

3. Else if *rStatus* is DirectoryOperationResult.ObjectNotFound and *domainChangeStatus* is "Changed domains":

   - The queue manager SHOULD<15> find the identifier of the Machine Directory Service object in the old domain, referred to as *oldQueueManagerId*, as follows:
     - Let DirectoryServerConnection be a variable of type **ADCONNECTION_HANDLE** ([MS-DTYP] section 2.2.2).
     - Perform the "Initialize an ADConnection" task ([MS-ADSO] section 6.2.6.1.1), specifying the following parameters:
       - **TaskInputTargetName** = the <computer name> part of **QueueManager.FullPath** as specified in [MS-MQDSSM] sections 2.2.1 for object type **mSMQConfiguration**.
       - **TaskInputPortNumber** = 389
     - DirectoryServerConnection is set to the **TaskReturnADConnection** result returned by the task.
     - Perform the "Set an LDAP Option on an ADConnection" task ([MS-ADSO] section 6.2.6.1.2), specifying the following parameters:
       - **TaskInputADConnection** = DirectoryServerConnection
       - **TaskInputOptionName** = "LDAP_OPT_PROTOCOL_VERSION"
       - **TaskInputOptionValue** = 3
     - Perform the "Establish an ADConnection" task ([MS-ADSO] section 6.2.6.1.3), specifying the following parameters:
       - **TaskInputADConnection** = DirectoryServerConnection
     - If the **TaskReturnStatus** result is FALSE, raise an error and take no further action.
     - Perform the "Perform an LDAP Bind on an ADConnection" task ([MS-ADSO] section 6.2.6.1.4), specifying the following parameters:
       - **TaskInputADConnection** = DirectoryServerConnection
     - If the **TaskReturnStatus** result is FALSE, raise an error and take no further action.
     - Construct an **LDAPMessage** ([RFC2251] section 4.1):
       - messageID = set as described in [RFC2251] section 4.1.1.1
       - protocolOp = searchRequest

Copyright © 2011 Microsoft Corporation.

- controls = none
- baseObject = **QueueManager.FullPath**
- scope = baseObject
- typesOnly = FALSE
- filter = present "objectClass"
- attributes = list consisting of one element "objectGuid"

- Perform the "Perform an LDAP Operation on an ADConnection" task ([MS-ADSO] section 6.2.6.1.6) with the following parameters:
  - **TaskInputADConnection** = DirectoryServerConnection
  - **TaskInputRequestMessage** = the **LDAPMessage** constructed in the preceding step
- If the value of **TaskReturnStatus** is not success, as defined in [RFC2251] section 4.1.10, unbind the connection as specified in the next step, raise an error and take no further action. Otherwise, extract the value for the **objectGuid** attribute from the result message returned in **TaskOutputResultMessages** and assign it to *oldQueueManagerId*
- Perform the "Perform an LDAP Unbind on an ADConnection" task ([MS-ADSO] section 6.2.6.1.5) with the following parameters:
  - **TaskInputADConnection** = DirectoryServerConnection
- If **QueueManager.Identifier** matches *oldQueueManagerId*, perform implementation-specific action to move the object from the old domain to the new domain<16> and complete the domain change handling.

- Otherwise, continue to step 4.

4. Create a Machine Directory Service object in the new domain by raising a Create Directory Object event ([MS-MQDMPR] section 3.1.7.1.18) with the following arguments:

```
iDirectoryObject := QueueManager
iAttributeList := {"DirectoryServer", "RoutingServer", "OperatingSystemType",
"SupportingServer", "PublicEncryptionKeyList", "PublicSigningKeyList",
"SiteIdentifierList", "Security"}
```

The system MUST set the QueueManager.Identifier to the value of *rObjectGUID* returned from the Create Directory Object event.

6.6.2.2 Interactions with Directory Service on Leaving a Domain

If the queue manager is operating in Directory-Integrated mode and the queue manager machine leaves the Directory Service domain, the system MUST set the QueueManager.DirectoryIntegrated ADM element to FALSE and operate in Workgroup mode as specified in section 4.3.1.3. The system MUST NOT remove any directory objects.

6.7 Status and Error Returns

This section describes the common queue manager and Directory Service Integration errors that are visible to administrators of the Message Queuing System. The system does not define any additional

Copyright © 2011 Microsoft Corporation.

error handling requirements beyond those described in the specifications of the protocols supported by the system, as listed in section 2.2.

6.7.1 Queue Manager Errors

MQ_ERROR_SERVICE_NOT_AVAILABLE (0xC00E000B): The queue manager service is not running. This is related to the failure scenario described in section 5.5.1, "Queue Manager Restart". The administrator needs to examine the event log to get more details and restart the queue manager service as appropriate.

MQ_ERROR_DTC_CONNECT (0xC00E004C): A connection to the Distributed Transaction Coordinator cannot be established. This is related to the failure scenario described in section 5.5.3, "Transaction Coordinator Unavailable". The administrator needs to verify that the Distributed Transaction Coordinator service to which the system is attempting to connect is running.

6.7.2 Directory Service Integration Errors

The following errors are related to the failure scenario described in section 5.5.4, "Directory Unavailable".

MQ_ERROR_NO_DS (0xC00E0013): A connection to the Directory Service could not be established. The administrator needs to verify connectivity of the Message Queuing System with the Directory Service.

MQ_ERROR_CANT_RESOLVE_SITES (0xC00E0089): The sites for the machine on which the Message Queuing System is deployed cannot be retrieved from the Directory Service. The administrator needs to verify connectivity of the Message Queuing System with the Directory Service.

MQ_ERROR_DS_LOCAL_USER (0xC00E0090): A local user is authenticated as an anonymous user and cannot access the Directory Service. The administrator needs to verify that the user account is set up correctly for using the desired functionalities of the Message Queuing System.

The following errors are related to the failure scenarios described in section 5.5.6, "Directory Inconsistency".

MQ_ERROR_MACHINE_EXISTS (0xC00E0040): A Machine Directory Service object for this Message Queuing System already exists in the Directory Service. The administrator needs to investigate the cause of this failure condition and rectify it before making the Message Queuing System available to the users.

MQ_ERROR_CANNOT_JOIN_DOMAIN (0xC00E0076): The machine joined the domain, but the Message Queuing System failed to register itself with the Directory Service. It will continue to run in Workgroup mode. The administrator needs to investigate the cause of this failure condition and rectify it in order to enable the Message Queuing System to provide all the desired functionality to users.

Copyright © 2011 Microsoft Corporation.

7 Security

This section documents system-wide security issues that are not otherwise described in the Technical Documents (TDs) for the Member Protocols. It does not duplicate what is already in the Member Protocol TDs unless there is some unique aspect that applies to the system as a whole.

7.1 Security Elements

The system is composed of components. Components store data and communicate with other components. Both the storage of components and the communications between components SHOULD be secured.

The following figure combines the figure in section 4.3.1, "Components in a Message Queuing System" with the figure in section 4.3.1.3, "Message Queuing in Directory-Integrated mode", and presents an overview of the storage in the system and communications among internal and external components.

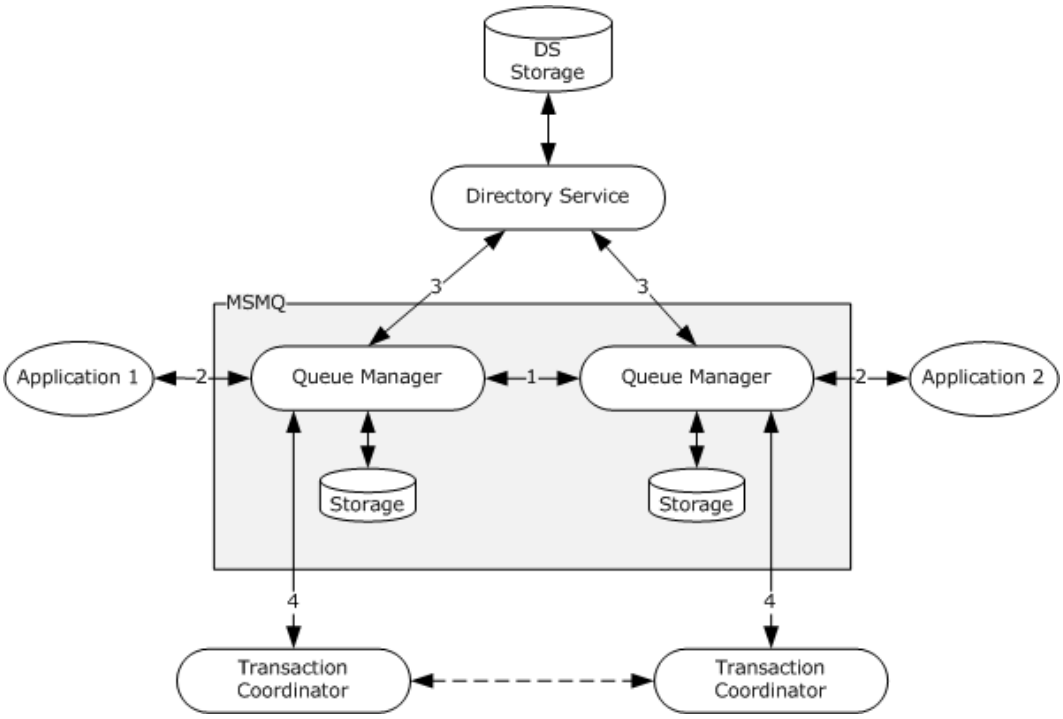


Figure 42: Component storage and communications

The data objects in the storage of a queue manager are listed in the queue manager abstract data model (section 5.1.1), and the data objects in the storage of a Directory Service that the system uses are listed in the Directory abstract data model in section 5.1.2.

The communications between two queue managers (1: internal communication) and between a queue manager and an external entity (2, 3, 4: external communication) are shown in the preceding figure. The communications between external entities are handled by those entities and are not included in the figure.

Copyright © 2011 Microsoft Corporation.

7.2 Security Strategy and Mechanisms

To secure both the data objects and the communications, the system uses a set of security mechanisms that are summarized as follows:

- Uses security identifiers (SIDs) to identify security principals, as described in [MS-WSO] section 3.1.2.1.3.
- Uses authentication mechanisms, as described in [MS-WSO] section 3.1.2.2.
- Uses Access Control Lists (ACLs), Discretionary Access Control Lists (DACLs), and Access Control Entries (ACEs) to specify authorization policy on data objects as described in [MS-WSO] section 3.1.2.3.3.
- Uses communication security mechanisms of authentication, message integrity, and message privacy to protect communications during component interactions.

7.3 Storage Security

Data objects are stored and the storage is protected by the owning component. The system SHOULD define DACLs on each data object so that unauthorized access is not allowed. For queue manager data objects, the owning queue manager SHOULD authorize the user who requests access to these objects. For Directory Service data objects, the application and the queue manager are responsible for defining ACLs, and the Directory Service is responsible for authenticating and authorizing the requester according to the defined ACLs.

Adding a message object to a queue object is controlled by the queue manager according to the ACLs specified on the queue object. The message carries the sender identity, which is used by the queue manager to perform access checks. The queue manager SHOULD authenticate the sender as specified in section 7.4.1.4. Without the implementation of sender authentication, a malicious user can provide a fake user identity in a message and bypass the access control defined for the queue object.

7.4 Communication Security

Communications occur over transports, which are specified in section 6.3. The system relies on the use of transport security features to secure communication. When needed, it augments the security features to provide required communication security support.

7.4.1 Security Layer

Data transmitted between two components can be protected at two layers: the transport layer and the message layer.

7.4.1.1 Transport Layer Security

The transport layer security refers to the security features provided by a transport that the system uses. For example, the RPC and LDAP transports provide security support for authentication, message integrity, and message privacy. The HTTPS transport provides support for server authentication and message privacy.

A complete list of the transports used by the system for internal and external communications is specified in section 6.3. The following table summarizes the security features supported by each transport.

Transport security support

Copyright © 2011 Microsoft Corporation.

Transport	Security features
TCP/IP	None
IPX/SPX	None
HTTP	None
HTTPS	Authentication, message integrity, message privacy
UDP	None
PGM	None
RPC	Authentication, message integrity, message privacy
LDAP	Authentication, message integrity, message privacy
DCOM	Authentication, message integrity, message privacy

When the authentication support of the RPC transport is used, the client and the server MUST agree on the authentication service provider in order to communicate.<17>

As shown in the preceding table, not all transports provide security support for communications, in particular the TCP/IP or IPX/SPX transport that is used by the MQQB protocol to transfer messages. In this case, the system provides a set of security features at the message layer to meet security needs.

7.4.1.2 Message Layer Security

Message layer security, which is independent of the underlying transport, refers to security features that are provided by the system on a per-message basis. It includes message integrity, sender authentication, and message privacy. The former two security features provide end-to-end protection of the message, ensuring that a message is sent by the original sender without being altered during transfer. Message privacy is provided for hop-to-hop transfer of a message. It ensures that the message content is not disclosed to unauthorized users during network transfer.

The message layer security features are built on the public key infrastructure (PKI) model as specified in [SP800-32] and in [MSFT-PKI].

7.4.1.3 Security Model: PKI

PKI is an arrangement that binds a public key certificate with a respective user identity through a trusted third party. The main elements in the PKI are:

- Certification authority (CA): A trusted entity that issues certificates for use by other entities.
- Certificate: An electronic document that includes a digital signature to bind a public key with an identity.
- Public/private key: A pair of keys that are used in the asymmetric cryptographic algorithms. The public key is distributed in the certificate, which can be validated with the CA by other entities. The private key is typically stored on the certificate holder's local computer.

In the system, applications hold user certificates, and each queue manager holds two pairs of cryptography keys (signing keys and encryption keys). The system uses a Directory Service, rather than a CA, as the trusted third party to distribute the public certificates and keys.

Copyright © 2011 Microsoft Corporation.

User Certificates

A user certificate is used to sign an application message to provide the message integrity feature of the message layer security as described in section 7.4.1.4.

User certificates SHOULD be registered in the Directory Service to enable the sender authentication feature of the message layer security as described in section 7.4.1.4. The user certificates are stored in the corresponding user object in the directory and are maintained in the User.Certificates data element as described in [MS-MQDMPR] section 3.1.1.15. The registration associates the certificate with the corresponding user identity. A hash of the certificate (digest) is computed and saved together with the certificate as the User.CertificateDigestList, as described in [MS-MQDMPR] section 3.1.1.15, to facilitate certificate lookup. A user object can have multiple certificates. For more details about the user object attributes, see [MS-RDPBCGR] section 2.2.1.4.3.1.1 and [MS-ADA2] section 2.351. For more details about the user certificate and its digest, see [MS-MQDMPR] section 3.1.1.15 and [MS-MQDSSM] sections 3.1.1.4 and 3.1.6.20.6.

If an application is sending messages to a destination where the Directory Service is unavailable, the application MAY instead provide a user certificate from a **CA** trusted by both the sender and receiver to be used for signing the application message. If such a certificate is used for signing, the queue manager hosting the destination queue will still verify message integrity as described in section 7.4.1.4.1 before storing the message, but cannot authenticate the sender as described in section 7.4.1.4.2. The user certificate remains attached to the message so that the receiving application can verify the owner of the certificate if desired.

The private key associated with a certificate MUST be stored securely and MUST be available to the sender.

Service Cryptography Keys

Each queue manager has two pairs of cryptography keys: one pair for signing system internal messages and one pair for encrypting messages. These keys are represented by QueueManager.PublicSignKeyList and QueueManager.PublicEncryptionKeyList respectively, as described in [MS-MQDMPR] section 3.1.1.1. The private keys MUST be stored securely and MUST be available to the queue manager. The public keys MUST be published in the Directory Service under the Machine object for this queue manager. For more details about the queue manager cryptography keys, see [MS-MQDMPR] section 3.1.1.1 and [MS-MQDSSM] sections 2.2.1 and 3.1.6.20.1.

7.4.1.4 Message Layer Security Features

Using the user certificates, service cryptography keys, and the Directory Service for public key distribution, the system provides three message layer security features: message integrity, sender authentication, and message privacy.

7.4.1.4.1 Message Integrity

Message integrity is achieved through the following sequence:

1. The sending application signs the message in the following steps:
    1. Computes a hash from a set of message properties.
    2. Encrypts the hash with the private key associated with the sender's certificate to generate a signature.
    3. Attaches the signature and the certificate to the message.

Copyright © 2011 Microsoft Corporation.

2. The queue manager hosting the destination queue verifies the message integrity as follows:

    1. Extracts the signature and certificate from the message.

    2. Decrypts the signature with the public key in the certificate to get the sender-generated hash.

    3. Computes the hash from the same set of message properties.

    4. Verifies the signature by comparing the sender-generated hash with the service-computed hash.

The signature format is protocol-specific. See [MS-MQMQ] section 2.2.20.6 for the binary protocol and [RFC3275] for the SRMP protocol.

For more details about the hash algorithms, the message properties used for hashing, and the algorithm to encrypt and decrypt the hash, see [MS-MQMQ] section 2.2.20.6.

7.4.1.4.2 Sender Authentication

When a message contains the signature, the user certificate, and user identity of the sender, the queue manager of the destination queue MUST verify the sender identity that is carried in the message as follows:

1. Verifies message integrity as specified previously under Message Integrity. This ensures that the message was not altered during transmission.

2. Computes the digest as the hash of the attached certificate using MD5 algorithm as defined in [RFC1321].

3. Finds the User object in the Directory Service that satisfies the following conditions:

    - The User.CertificateDigestList data element contains the computed digest.

    - The User.Certificates data element contains the certificate.

4. If a matching user object is found:

    1. Extracts the sender identity from the message.

    2. Compares the extracted identity with the User.SecurityIdentifier element of the matching User object.

    3. Authenticates the sender if the identities match; otherwise, rejects the message.

5. Otherwise, if no matching user object is found, marks the sender as unauthenticated.

When the message does not contain the signature, the user certificate, or the user identity of the sender, the queue manager does not verify the sender identity.

7.4.1.4.3 Message Privacy

Message privacy is achieved through the following sequence.

1. The sending queue manager does the following:

    1. Retrieves the public key information (see details in [MS-MQDS] section 2.2.10.3) of the receiving queue manager from the queue manager's machine object in the Directory Service.

Copyright © 2011 Microsoft Corporation.

2. Dynamically generates a symmetric key. The symmetric key generation SHOULD use the key length and provider information in the receiving queue manager's public key information (see details in [MS-MQDS] section 2.2.19) to ensure that the receiving queue manager can decrypt the message.
3. Encrypts the symmetric key with the receiving queue manager's public key.
4. Encrypts the message with the symmetric key.
5. Attaches the encrypted symmetric key to the message. The message format is specified in [MS-MQMQ] section 2.2.20.6.

2. The receiving queue manager does the following:
   1. Extracts the encrypted symmetric key from the message.
   2. Decrypts it with its own private key.
   3. Decrypts the message with the decrypted symmetric key.

For more details about the encryption algorithm and related message properties, see [MS-MQQB] section 3.1.7.1.5.

7.4.1.5 Message Layer Security Sequences

Taken together, the sequences of the three message layer security features are shown in the following figure.

Copyright © 2011 Microsoft Corporation.

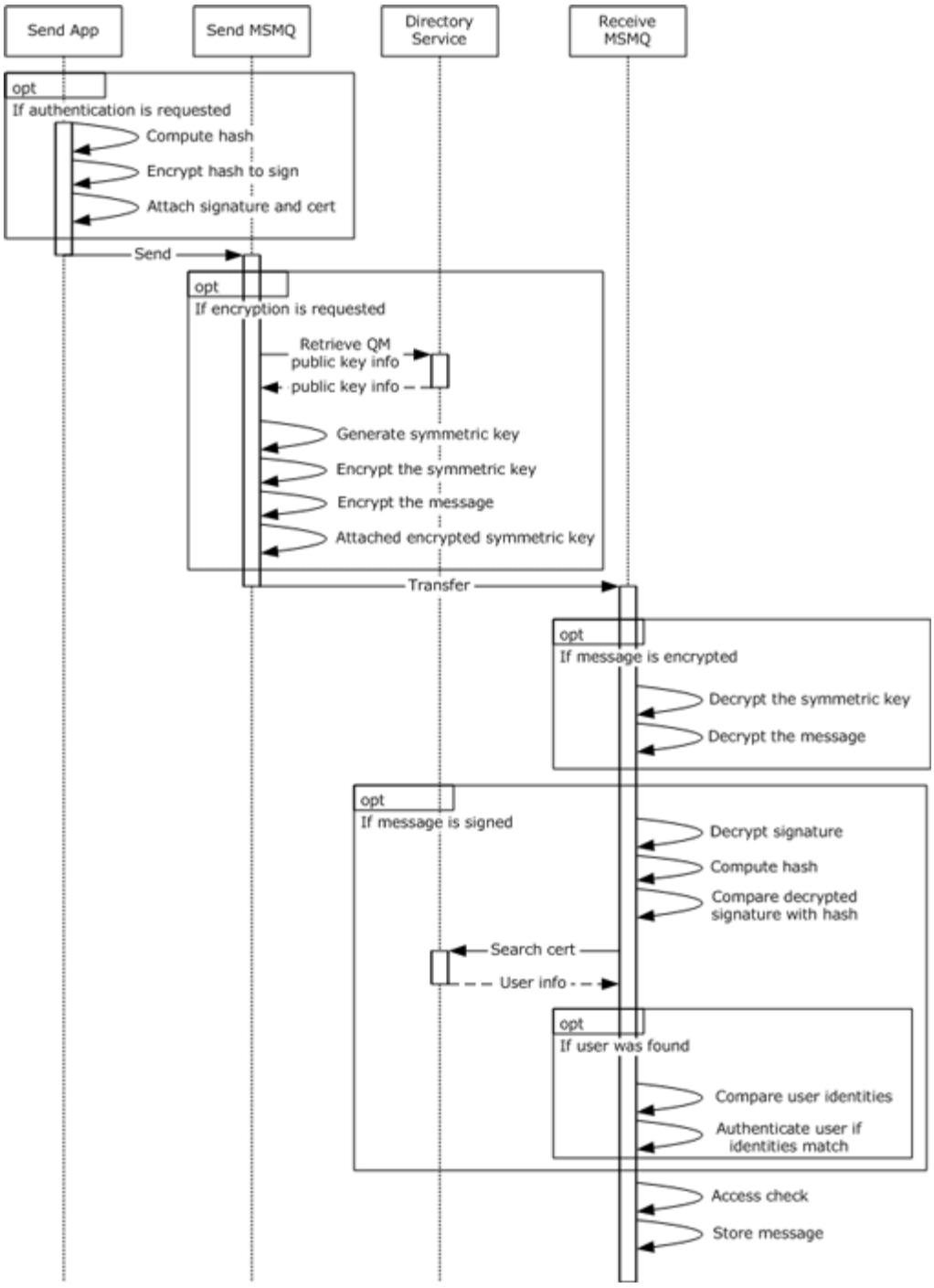


Figure 43: Message layer security sequences

7.5 Internal Security and External Security

Internal and external security is another view of protecting the data and the communications in the system.

Copyright © 2011 Microsoft Corporation.

Internal security is the means by which the system protects its own data and internal communications, and external security is the means by which the system protects external communications. For the system, application messages are external communications. Objects specified in section 7.1 are internal data. For communications labeled in the figure "Component storage and communications" in section 7.1, 1 is internal communication, and 2, 3, and 4 are external communications.

The security mechanisms previously described are used to ensure both internal and external security.

Copyright © 2011 Microsoft Corporation.

8 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include released service packs:

- Microsoft Windows NT® operating system
- Microsoft Windows® 2000 operating system
- Windows® XP operating system
- Windows Server® 2003 operating system
- Windows Server® 2003 R2 operating system
- Windows Vista® operating system
- Windows Server® 2008 operating system
- Windows® 7 operating system
- Windows Server® 2008 R2 operating system

Exceptions, if any, are noted below. If a service pack or Quick Fix Engineering (QFE) number appears with the product version, behavior changed in that service pack or QFE. The new behavior also applies to subsequent service packs of the product unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms SHOULD or SHOULD NOT implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies that the product does not follow the prescription.

<1> Section 4.3.2.1: The RPC system in Windows 2000 Server supports the Message Queuing System as a transport.

<2> Section 4.3.3: The Message Queuing System in Windows NT 4.0 and Windows 2000 Server uses MQDS to communicate with the directory service. All other versions of Windows use LDAP, as specified in [MS-ADTS], to communicate with the directory service.

<3> Section 4.5: The multiple versions of the Message Queuing System that exist are enumerated in the following table.

MSMQ Versions

MSMQ version	Operating system version
MSMQ 1.0	Windows NT 4.0, Windows 95, and Windows 98
MSMQ 2.0	Windows 2000 Server
MSMQ 3.0	Windows XP and Windows Server 2003
MSMQ 4.0	Windows Vista and Windows Server 2008
MSMQ 5.0	Windows 7 and Windows Server 2008 R2

Copyright © 2011 Microsoft Corporation.

<4> Section 4.5: The client version of MSMQ 3.0 is shipped with Windows XP.

<5> Section 4.5: The server version of MSMQ 3.0 is shipped with Windows Server 2003.

<6> Section 5.1.1.1: The Message Queuing System in Windows NT, Windows 2000 Server, Windows XP, and Windows Server 2003 does not monitor resource conditions. In other versions of Windows, as listed in the beginning of section 8, the Message Queuing System monitors the resource condition by listening to the LowPagedPoolCondition system wide kernel event.

<7> Section 6.4.1.2: The default value of the Directory Server List Update timer is 86,400,000 milliseconds in all other versions of Windows as listed in the beginning of section 8. This value is not configurable.

<8> Section 6.4.1.2: For Windows NT and Windows 2000, the queue manager uses the MSMQ: Directory Service Discovery Protocol [MS-MQSD] by calling the Get Directory Server List Higher-Layer Triggered Event ([MS-MQSD] section 3.1.4.1). The **DirectoryServerList** ADM attribute of the local **QueueManager** ADM element instance is populated with the names of **MSMQ Directory Service servers**.

<9> Section 6.4.3.1: The default value of the Certificate Data Cache Validity timer is 1,200 milliseconds in all other versions of Windows as listed in the beginning of section 8.

<10> Section 6.4.3.2: The default value of the Queue Manager Public Key Cache Validity timer is 1,200 milliseconds in all other versions of Windows as listed in the beginning of section 8.

<11> Section 6.6: Only Windows NT, Windows 2000, Windows XP 32-bit, Windows Server 2003 32-bit and Windows Server 2003 R2 32-bit on domain-joined machines use the supplied value to determine whether a queue manager should be deployed.

<12> Section 6.6.2: The Message Queuing System in Windows NT skips this step.

<13> Section 6.6.2: The Message Queuing System does not mandate any specific periodic reinitialization. Any such retry mechanism is an implementation choice.

<14> Section 6.6.2.1: The message queuing system in Windows NT does not perform this sequence of steps.

<15> Section 6.6.2.1: On Windows 2000, the queue manager performs the following actions to get the Machine Service object identifier from the old domain:

- Establish an RPC connection to the [MS-MQDS] protocol server on the old domain controller machine whose name is the <computer name> portion in **QueueManager.FullPath** ([MS-MQDSSM] section 2.2.1 for object type **mSMQConfiguration**).
- If the connection cannot be established, raise an error, and take no further action.
- Generate a Read Directory ([MS-MQDS] section 3.2.6.3) event over the established connection with the following arguments:
  - *iDirectoryObjectType* := "QueueManager"
  - *iFilter* := {"FullPath" EQUALS <full path>}, where <full path> is the **FullPath** attribute of the local queue manager.
  - *iAttributeList* := {"Identifier"}
- If *rStatus* of the event is not **DirectoryOperationResult.Success**:

Copyright © 2011 Microsoft Corporation.

- Close the RPC connection, raise an error, and take no further action.

- Else:

  - Set *oldQueueManagerId* to *rDirectoryObject.Identifier*. Close the RPC connection.

<16> Section 6.6.2.1: The "implementation-specific action to move machine object to new domain" process shown in the diagram captioned "Handling change of domain join status" is implemented in MSMQ by writing an event to the event log that notifies users that they need to manually move the Machine object to the new domain.

<17> Section 7.4.1.1: In the Microsoft implementation, the queue manager RPC servers register authentication services based on the machine's configuration. For example, Kerberos mutual authentication is registered only on a domain-joined machine. When the client has no knowledge of the RPC server's registered authentication services, it determines the authentication service using the logic described in the following figure, in order to establish a connection with the server.

Copyright © 2011 Microsoft Corporation.

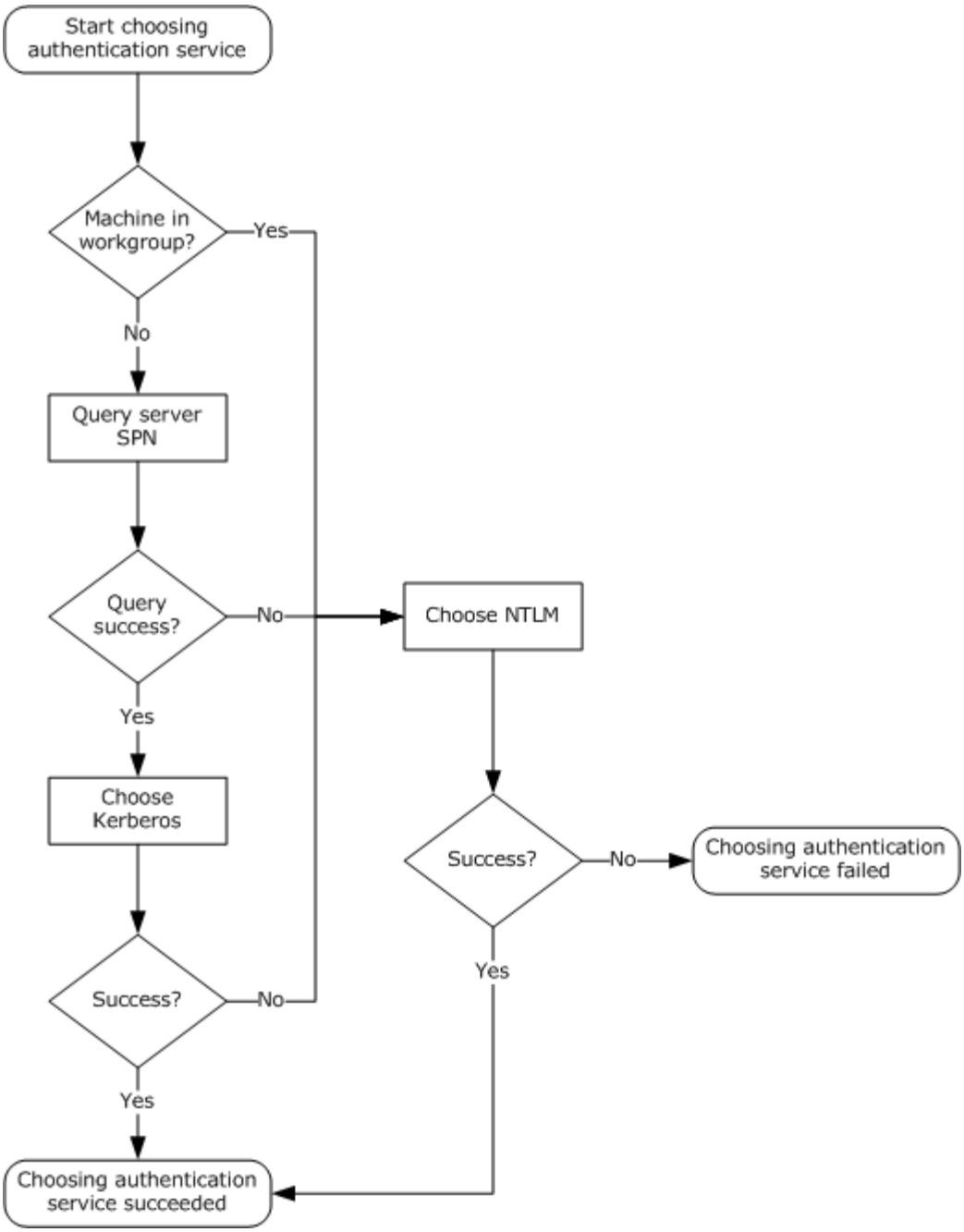


Figure 44: Determine RPC authentication service on the client side

Copyright © 2011 Microsoft Corporation.

9 Change Tracking

This section identifies changes that were made to the [MS-MQSO] protocol document between the May 2011 and June 2011 releases. Changes are classified as New, Major, Minor, Editorial, or No change.

The revision class **New** means that a new document is being released.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements or functionality.
- An extensive rewrite, addition, or deletion of major portions of content.
- The removal of a document from the documentation set.
- Changes made for template compliance.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **Editorial** means that the language and formatting in the technical content was changed. Editorial changes apply to grammatical, formatting, and style issues.

The revision class **No change** means that no new technical or language changes were introduced. The technical content of the document is identical to the last released version, but minor editorial and formatting changes, as well as updates to the header and footer information, and to the revision summary, may have been made.

Major and minor changes can be described further using the following change types:

- New content added.
- Content updated.
- Content removed.
- New product behavior note added.
- Product behavior note updated.
- Product behavior note removed.
- New protocol syntax added.
- Protocol syntax updated.
- Protocol syntax removed.
- New content added due to protocol revision.
- Content updated due to protocol revision.
- Content removed due to protocol revision.
- New protocol syntax added due to protocol revision.

Copyright © 2011 Microsoft Corporation.

- Protocol syntax updated due to protocol revision.
- Protocol syntax removed due to protocol revision.
- New content added for template compliance.
- Content updated for template compliance.
- Content removed for template compliance.
- Obsolete document removed.

Editorial changes are always classified with the change type **Editorially updated.**

Some important terms used in the change type descriptions are defined as follows:

- **Protocol syntax** refers to data elements (such as packets, structures, enumerations, and methods) as well as interfaces.
- **Protocol revision** refers to changes made to a protocol that affect the bits that are sent over the wire.

The changes made to this document are listed in the following table. For more information, please contact protocol@microsoft.com.

Section	Tracking number (if applicable) and description	Major change (Y or N)	Change type
1.2 References	Added explanatory statement regarding the removal of the publishing year from Microsoft Open Specification document references.	N	Content updated.

Copyright © 2011 Microsoft Corporation.

10 Index

Copyright © 2011 Microsoft Corporation.

Copyright © 2011 Microsoft Corporation.

T

U

V

W

Copyright © 2011 Microsoft Corporation.