CHAPTER 19

# Using Message Queues

In this chapter, I want to talk about Microsoft Message Queues, including what they are, how they work, and when they might be useful for your applications.

**In This Chapter**

Introducing the Queuing Object Model

Accessing message queues

Building client and server programs

Viewing message queue information

## How Message Queuing Works

*Message queuing* is a tool that helps two programs communicate with each other in an asynchronous fashion. This means that a client program can send a request to a server and then continue processing without waiting for the server process to complete. When the server process has finished processing the request, it will return the message to the client program, which will be notified with the response it available to be processed.


**Well, kinda, sorta:** While not strictly a part of the COM+ programming environment, Microsoft Message Queuing works closely with COM and COM+ components to implement high-performance applications.

### Synchronous processing

Normally, when you are using a COM component and you issue a method, or request the information in a property, your program must wait until the method or property returns control to your program. This is called *synchronous processing*.

In synchronous processing, the client program begins by sending a request to the server to perform a specific task. The server program receives the request, performs the task, and returns the result back to the client. During the time the server is processing the client's request, the client program is in a wait

state, where it can't perform any processing. It will only resume processing after the server has returned its response (see Figure 19-1).


**Synchronous processing made simple:** When thinking about synchronous processing, visualize how a Web browser works. The URL is used to generate a request to the Web server and then the browser waits for the Web page to be returned.

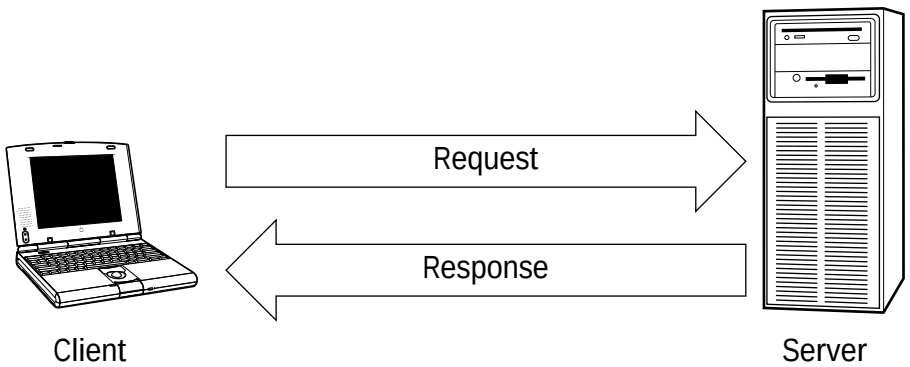


**Figure 19-1:** Processing a synchronous request

## Asynchronous processing

In some cases, it's useful for your program to issue a request for information and then perform other tasks while your program waits for the response. This is called *asynchronous processing*.

In asynchronous processing, the client program begins by sending a request to the server to perform a specific task and then resumes its normal processing. The server receives the request, processes it, and returns the result. When the result is received, the client program is notified that the result is available and ready for the client program to use (see Figure 19-2).


**Asynchronous processing made simple:** One way to understand how asynchronous processing works is to compare it to electronic mail. You can send a message to someone requesting a piece of information and then continue reading and sending other e-mail until you receive a response to your original message.

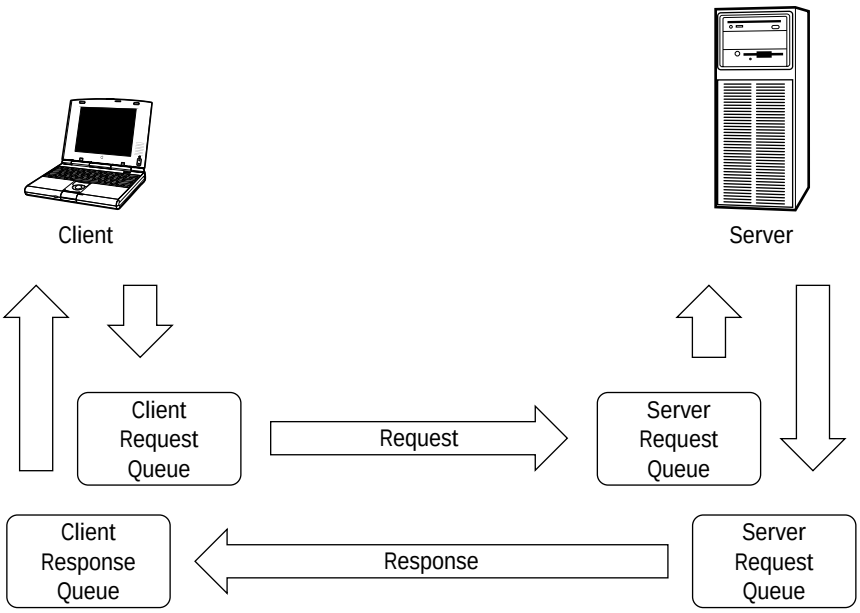


**Figure 19-2:** Processing an asynchronous request

Since the client is no longer tightly coupled to the server, the server has the freedom to process requests in the most efficient manner. Since it is quite possible that multiple clients will be sending requests, all of the messages are stored in a queue as they are received. As the server finishes processing one request, it will pull the next request from the queue. This ensures that all of the messages are processed in the order in which they were received.

## Benefits of message queuing

Using message queues will increase the complexity of your application. In addition, not all applications will benefit from using message queuing. However, the benefits of message queuing may outweigh the extra complexity.

### Component availability

Just because you have a full-time connection to the server doesn't mean that the server will always be available. The server's operating system or the application running on the server may have crashed. Networking problems are probably more common, especially if the server is located in another building or across the country from your client machines.

If you are unable to reach the server for any reason, all of the requests are buffered locally until the server can be reached. When the server becomes available, the information that was buffered locally will be transmitted to the server for processing. Also, any results that the server may have processed will be received into local storage for the client to process.

### Component performance

By queuing requests for processing, the user need not wait for the server to finish processing one request before starting on the next request. Thus, you can create your application to prioritize how the processing is done. This is important when your server becomes overloaded. Those tasks that need an immediate response, such as checking inventory levels, can be handled as a normal COM+ object. Those tasks that are not as time-sensitive, such as printing an invoice, can be deferred until the server is less busy by using message queuing.

### Component lifetimes

Delays caused by transmitting information over the network and interacting with the user extend the amount of time that an object actually exists inside the transaction server. By using message queuing, the exchange of information is handled outside the transaction server. This means that the object exists for far less time than it would without using message queuing. This translates into better performance in the transaction server, because fewer system resources are needed to process the actual transaction.

### Disconnected applications

If you create queues that go in both to the server and to the client, you may not need a full-time connection to the server. Requests can be queued up until a connection to the server is established. Then the requests are transmitted to the server for processing using the server queue. Any responses that the server has for the client can be queued in the client queue. When the connection is established, both queues will be transmitted to the other machine. If the connection remains up for enough time, the server may actually be able to respond to the requests sent by the client and return them via the reverse queue.

### Message reliability

Message queuing relies on a database to store the requests. This ensures that the requests are protected from system failures. If and when the database needs to be recovered, the requests in the queues would also be recovered. Since the time to transmit and add the request to a queue is much less than the time a normal object would exist, it is less likely that a failure would occur in the middle of a specific request.

## Server scheduling

Another advantage of message queuing is the ability to shift the work sitting in a queue to a time when the server is less busy. Thus, message queues have the ability to duplicate the batch-processing features of the mainframe world by making it easy to shift work to a less busy time.

## Load leveling

One problem with most server-driven application systems is that their workload can vary greatly over time, which may have an adverse impact on performance. In a synchronous environment, a server can easily get overwhelmed for a few moments at a time. If this happens occasionally, the server can catch up and nobody will notice. On the other hand, if this happens frequently, the operating system and the server application will be forced to spend extra resources managing the extra work, which means that there are less resources to process the extra work.

This extra overhead can be eliminated by using message queuing (asynchronous processing). You would tune the server and the application to get the highest performance. Then you would stage the requests in a message queue until the server can process them. Most of the time, the requests would be pulled from the message queue as fast as they arrive. However, when the requests arrive faster than the server can process them, they will remain in the message queue until the server can process them.

In real terms, the slight penalty imposed by using message queuing will not make a big difference in the time it takes to process a request until the server becomes overloaded. Then, depending on how overloaded the server is, using message queuing will increase the time to process each request. However, because the application is continuing to process requests at optimal speed, the time the request spends in the message queue, plus the time it takes the application to process the request, will most likely be less than the time it takes to process the request while the server is overloaded.


**Too much is just right:** A long time ago, I ran into a situation where one of the computers I was responsible for was running at maximum capacity. The CPU utilization would stay at 100% for several minutes at a stretch. Response time was horrible. Upgrading to a CPU that was twice as fast solved the problem. However, the CPU utilization was about 40%, rather than the expected 50%. After a lot of digging and analysis, I determined that the old CPU was wasting nearly 20% of its capacity trying to manage the extra requests. With the faster CPU, the overload condition never arose and the CPU cycles were never wasted. While it is not always possible to double your CPU capacity, you may be able to use a tool like message queuing to ensure that the your CPU isn't wasting extra cycles trying to manage too much work.

# Microsoft Message Queuing

In order to use message queues in a Windows environment, you need a piece of software called Microsoft Message Queuing (often abbreviated MSMQ). It runs on Windows 9x, Windows NT, and Windows 2000, though you may need the appropriate patches to run it. It is considered a base component of these operating systems; thus, there isn't a separate charge for the software.

## Requests and responses

A request is sent to the server using an object containing all of the information needed to process the request. The object must be persistable and be less than four megabytes in length.

There are four basic responses to a request.

- **No response is necessary** – this means that the message sent by the client either doesn't need a response or the response will be returned using a different technique. For instance, the message sent by the client may request a report that will be returned by e-mail.
- **Trigger an event in the client application** – this means that the server application will fire an event in the client program when the server application has finished processing the message.
- **Use a synchronous method to check on the status** – this means that the server application may expose a property that a client application can access to determine if their message has been processed.
- **Use message queuing to receive a response** – this is probably the most common way to determine the outcome of processing a request. Consider the case of a salesperson that enters orders into a laptop computer while at a customer's site. At the end of the day, the salesperson could connect the laptop to the company's application server. Then MSMQ would transfer the local messages to the message queues on the server, while downloading any responses from the server into local storage for the salesperson to browse at his or her leisure.

## Types of queues

Queues are really just storage that is allocated to contain messages along with the information needed to manage the information in the queue. Different types of requests receive different types of responses. Also, in order to operate a message queue, the software on the client and server ends of the message queue need to exchange information. This information is typically sent using message queues.

There are six basic types of queues. The type of queue dictates how the queue will be used, but underneath, the same basic technology is used to support all of these types:

- **Message** – this is the most common type of queue and is used primarily by applications. Clients can send messages to the queue, while server applications can retrieve messages from the queue.
- **Administration** – this type of queue is used by an application to retrieve status information about messages in a message queue. For example, an acknowledgement message indicates that a message was received or retrieved from the application at the queue's destination.
- **Response** – this type of queue is used by client applications to receive responses from the server application.
- **Report** – this type of queue tracks the progress of the messages as they move through the system.
- **Journal** – this type of queue holds messages that have been retrieved at their destination.
- **Dead-letter** – application messages that can't be delivered are stored in this type of queue.

## Public and private queues

Queues can also be labeled as public or private. *Public queues* are registered in the Active Directory and can be located by anyone with access to the Active Directory. You can search for various properties of a queue in the Active Directory so that you can locate the correct one. Thus public queues can only be implemented on a Windows 2000 Server based system.

Combining the name of the computer with the name of the queue creates the name of a public queue. So, the queue `MyQueue` on the `Athena` would be known as `Athena\MyQueue`.


**Actively seeking directory information:** The Active Directory facility in Windows 2000 Server is a tool that stores information about the resources on the local computer, or resources that can be found elsewhere on the network. Windows 2000 makes extensive use of this facility to store information about the operating system, such as user names, like public queues, other specialized services, and so on. *Private queues* are registered on an individual computer and can be found by anyone who knows the names of the computer and the queue. You must also include the keyword `\Private$` in the name of the queue, so a private queue on Athena called `MyQueue` would be written as `Athena\Private$\MyQueue`.

By default, all queues are public. To make a queue private, you must include `\Private` in the name of the queue when it is created. You must have access to an Active Directory server to do this, however. Note that private queues can be created on Windows 98- and Windows NT-based machines.

## Message queuing and COM+ transactions

COM+ transactions using message queues work automatically with the COM+ transaction server and the `ObjectContext` object. Getting the `ObjectContext` object and performing the `SetComplete`, or `SetAbort`, method automatically rolls back any operations you made to the queues. In other words, no messages are removed from the queues or physically sent until the transaction is either committed or rolled back.

There is one big issue about using message queues and transactions. While inside a transaction, you can only send messages to transactional queues — queues that were created with the `IsTransactional` parameter of the `MSMQQueueInfo.Create` method set to `True`. Also, any messages that are sent automatically have their `MSMQMessage.Priority` value set to zero. This ensures that the messages are processed in the order in which they were received.

The message queues also include their own version of transaction support in case you don't want to use the COM+ transaction server or want to work outside its control. You need to use the `MSMQCoordinatedTransactionDispenser` to begin a transaction while under control of the COM+ transaction server. Use the `MSMQTransactionDispenser` object if you're not under control of the COM+ transaction server. Then use the `MSMQTransaction` object to either commit the transaction or abort the transaction.

# Message Queuing Object Model

Like most Windows tools, the MSMQ is accessed via a series of COM components. Figure 19-3 shows a brief overview of the objects and how they are related to each other.

- **MSMQQuery** – is used to search the Active Directory for public queues. It returns an `MSMQQueueInfos` collection containing the set of message queues that met the search criteria.
- **MSMQQueueInfos** – contains a set of `MSMQQueueInfo` objects. This component contains the results of an `MSMQQuery` operation.
- **MSMQQueueInfo** – is used to create and open queues as well as containing other information about a specific queue.

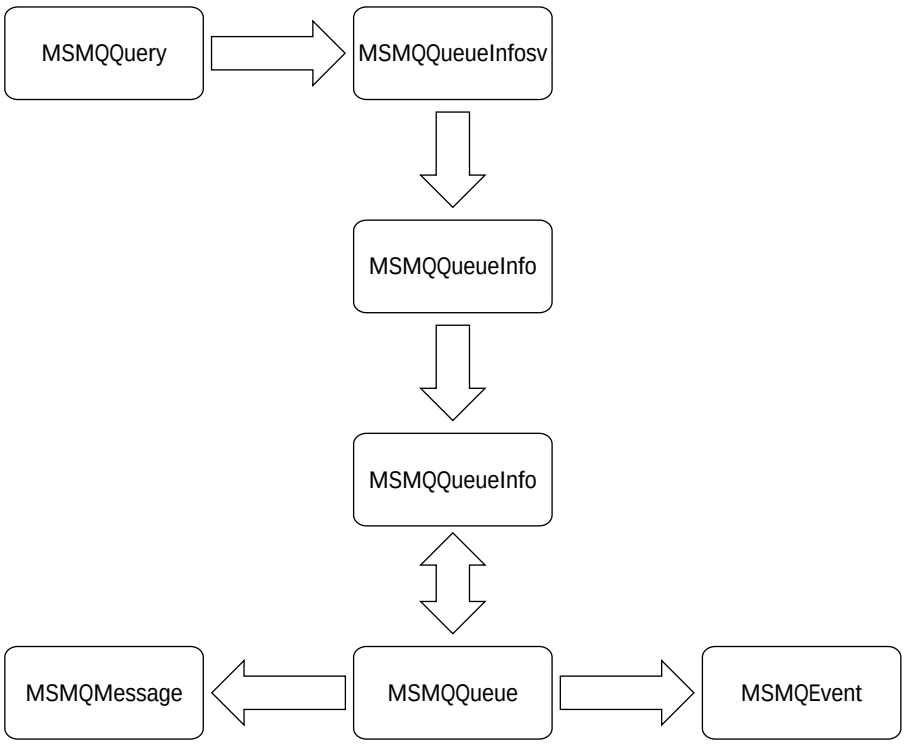


**Figure 19-3:** The Message Queuing Object Model

- **MSMQQueue** – is the base object in the object model. You use this object to access the individual messages in the queue.
- **MSMQMessage** – holds a message that is put in a queue or received from a queue.
- **MSMQEvent** – is used to define queuing events in your application. These events are fired as information arrives at your application.


**Making it simpler:** To simplify this discussion of message queues, I'm going to focus on private queues, which means that I'm going to ignore the `MSMQQuery` and `MSMQQueueInfos` objects. The only function lost by omitting the discussion of these objects is the ability to dynamically locate the proper queue by name only. To access a private queue, you must know the its name as well as the computer where it exists.

To add message queuing to your application, you need to select the Microsoft Message Queue 2.0 Object Library in the References dialog box (see Figure 19-4). To display this dialog box, choose Project ➪ References from the main menu.

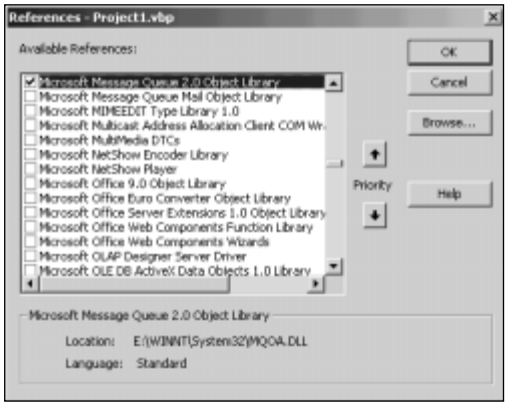


**Figure 19-4:** Selecting the Microsoft Message Queue Object Library

# The MSMQQueueInfo Object

The `MSMQQueueInfo` object is used to create or open a message queue. It can also set and return information about a particular queue.

## MSMQQueueInfo object properties

Table 19-1 lists the properties of the `MQMQQueueInfo` object.

**Table 19-1**
**Properties of the MSMQQueueInfo Object**

| *Property* | *Description* |
|---|---|
| `Authenticate` | A `Long` value that specifies whether the queue accepts only authenticated messages. |
| `BasePriority` | A `Long` that specifies a base priority for all messages sent to a public queue. |
| `CreateTime` | A `Date` value containing the date and time the public queue was created. |
| `IsTransactional` | A `Boolean` value when `True` means only messages from transactions will be accepted into the queue. |
| `IsWorldReadable` | A `Boolean` when `True` indicates that everyone can read the messages in the queue. |

| *Property* | *Description* |
|---|---|
| `Journal` | A `Long` when set to `MQ_JOURNAL` (0) means that the messages aren't saved; `MS_JOURNAL` (1) means that when a message is removed from the queue, it is stored in the journal queue. |
| `JournalQuota` | A `Long` containing the size in kilobytes of the journal queue. |
| `Label` | A `String` value containing up to 124 characters that describe the queue. |
| `ModifyTime` | A `Date` value containing the date and time when the properties of the queue were last modified. |
| `PathName` | A `String` containing the path name of the queue. This value is in the form of `\system\queue` or `\system\PRIVATE$\queue`. |
| `PathNameDNS` | A `String` containing the path name of the queue, where the system name is fully qualified. |
| `PrivLevel` | A `Long` when set to `MQ_PRIV_LEVEL_NONE` (0) means that the queue accepts only unencrypted messages; `MQ_PRIV_LEVEL_OPTIONAL` (1) means that the queue doesn't enforce privacy (default); or `MQ_PRIV_LEVEL_BODY` (2) accepts only encrypted messages. |
| `QueueGuid` | A `String` containing the GUID of the queue. |
| `Quote` | A `Long` containing the maximum size of the queue in kilobytes. |
| `ServiceTypeGuid` | A `String` containing the type of service provided by the queue. |

# MSMQQueueInfo object methods

The `MSMQQueueInfo` object contains methods for managing queues.

## Sub Create ([IsTransactional], [IsWorldReadable])

The `Create` method creates a new queue based on the properties defined in this object.

`IsTransactional` is a `Boolean` when `True` means only messages from transactions will be accepted into the queue.

`IsWorldReadable` is a `Boolean` when `True` indicates that everyone can read the messages in the queue.

### Sub Delete()

The `Delete` method deletes the queue associated with this object.

### Function Open(Access As Long, ShareMode As Long) As MSMQQueue

The `Open` method returns an object reference to `MSMQQueue` specified by the `PathName` property.

`Access` is a `Long` value specifying the level type of access to the queue. Multiple values may be combined by adding them together. Values are `MQ_RECEIVE_ACCESS` (1) means that messages can be received or peeked at in the queue; `MQ_SEND_ACCESS` (2) means that messages can be sent to the queue; and `MQ_PEEK_ACCESS` (32) means that you can peek at messages in the queue, but not remove them.

`ShareMode` is a `Long` specifying how the queue is shared. A value of `MQ_DENY_NONE` (0) means that the queue is shared with everyone (you must use this value if you specify `MQ_SEND_ACCESS` or `MQ_PEEK_ACCESS`); or `MQ_DENY_RECEIVE_SHARE` (1) prevents anyone except those in this process from accessing the queue (this value should be used when you specify `MQ_RECEIVE_ACCESS`).

### Sub Refresh()

The `Refresh` method gets a fresh copy of the properties associated with this queue.

### Sub Update()

The `Update` method updates the Active Directory for a public queue or the local computer with the current property values for this queue.

## The MSMQQueue Object

The `MSMQQueue` object is a fundamental object when dealing with message queuing. This object has the necessary properties and methods to access the information in the queue. The `MSMQQueueInfo` object is used to open the queue and return an object reference to this object.

## MSMQQueue object properties

Table 19-2 lists the properties of the MQMQQueue object.

Table 19-2
**Properties of the MSMQQueue Object**

| *Property* | *Description* |
|---|---|
| Access | A Long value indicating whether you can send messages, peek at messages, or receive messages from the queue. |
| Handle | A Long containing a handle of the open queue. |
| IsOpen | A Boolean value when True means that the queue is open. |
| QueueInfo | An object reference to a MSMQQueueInfo object containing additional information about the queue. |
| ShareMode | A Long that indicates whether the queue is available to everyone or whether only this process. |

## MSMQQueue object methods

The MSMQQueue object contains methods for examining the contents of the queue.

### Sub Close()

The Close method closes the queue.

### Sub EnableNotification(Event As MSMQEvent, [Cursor], [ReceiveTimeout])

The EnableNotification method instructs the message queuing software to trigger events using the MSMQEvent object. Note that this object needed to be declared using the WithEvents keyword in order to receive the events. You can specify that the event will be fired when there is a message in the queue, when a message is at the queue's current location, or when a message is at the queue's next location. Note that it may be possible that multiple messages may be in front of the message that triggered the event.

Event is an object reference to an MSMQEvent object that has been declared WithEvents, which will be fired as needed.

`Cursor` is an enumerated type that specifies the action of the cursor. A value of `MQMSG_FIRST` (0) means that the event will be fired when a message is in the queue. A value of `MQMSG_CURRENT` (1) means that the event will be fired when a message is at the current location of the cursor. A value of `MQMSG_NEXT` (2) means that the event will be fired when a message is at the new cursor location.

`ReceiveTimeout` is a `Long` containing the number of milliseconds that MSMQ will wait for a message to arrive.

### Function Peek([WantDestinationQueue], [WantBody], [ReceiveTimeout]) As MSMQMessage

The `Peek` method returns the first message in the queue without receiving the message from the queue. If the queue is empty, it will wait for a message to arrive.

`WantDestinationQueue` is a `Boolean` when `True` means that the `MSMQMessage.DestinationQueueInfo` property will be updated to contain the information about the destination queue. If not specified, a value of `False` will be assumed.

`WantBody` is a `Boolean` when `True` means that the body of the message should be returned (default). Set this value to `False` to reduce the amount of time to return the message.

`ReceiveTimeout` is a `Long` containing the number of milliseconds that MSMQ will wait for a message to arrive.

### Function PeekCurrent([WantDestinationQueue], [WantBody], [ReceiveTimeout], [WantConnectorType]) As MSMQMessage

The `PeekCurrent` method returns the current message in the queue without receiving it.

`WantDestinationQueue` is a `Boolean` when `True` means that the `MSMQMessage.DestinationQueueInfo` property will be updated to contain the information about the destination queue. If not specified, a value of `False` will be assumed.

`WantBody` is a `Boolean` when `True` means that the body of the message should be returned (default). Set this value to `False` to reduce the amount of time to return the message.

`ReceiveTimeout` is a `Long` containing the number of milliseconds that MSMQ will wait for a message to arrive.

`WantConnecterType` is a `Boolean` when `True` means that connector information will also be retrieved. If not specified, a value of `False` will be assumed.

### Function PeekNext([WantDestinationQueue], [WantBody], [ReceiveTimeout], [WantConnectorType]) As MSMQMessage

The `PeekNext` method returns the next method in the queue without receiving the message.

`WantDestinationQueue` is a `Boolean` value when `True` means that the `MSMQMessage.DestinationQueueInfo` property will be updated to contain the information about the destination queue. If not specified, a value of `False` will be assumed.

`WantBody` is a `Boolean` when `True` means that the body of the message should be returned (default). Set this value to `False` to reduce the amount of time to return the message.

`ReceiveTimeout` is a `Long` value containing the number of milliseconds that MSMQ will wait for a message to arrive.

`WantConnecterType` is a `Boolean` when `True` means that connector information will also be retrieved. If not specified, a value of `False` will be assumed.

### Function Receive([Transaction], [WantDestinationQueue], [WantBody], [ReceiveTimeout], [WantConnectorType]) As MSMQMessage

The `Receive` method returns the first message in the queue and removes it from the queue.

`Transaction` is an `MSMQTransaction` object or one of these values: `MQ_NO_TRANSACTION` (0), which means that call is not part of a transaction; `MQ_MTS_TRANSACTION` (1), which means that the call is made as part of the current MTS or COM+ transaction; `MQ_XA_TRANSACTION` (2), which means that the call is part of an externally coordinated, XA-compliant transaction; or `MQ_SINGLE_MESSAGE` (3), which means that the call retrieves a single message.

`WantDestinationQueue` is a `Boolean` when `True` means that the `MSMQMessage.DestinationQueueInfo` property will be updated to contain the information about the destination queue. If not specified, a value of `False` will be assumed.

`WantBody` is a `Boolean` when `True` means that the body of the message should be returned (default). Set this value to `False` to reduce the amount of time to return the message.

`ReceiveTimeout` is a `Long` containing the number of milliseconds that MSMQ will wait for a message to arrive.

`WantConnecterType` is a `Boolean` when `True` means that connector information will also be retrieved. If not specified, a value of `False` will be assumed.

### Function ReceiveCurrent([Transaction], [WantDestinationQueue], [WantBody], [ReceiveTimeout], [WantConnectorType]) As MSMQMessage

The `ReceiveCurrent` method returns the current message in the queue and removes it from the queue.

`Transaction` is an `MSMQTransaction` object or one of these values: `MQ_NO_TRANSACTION` (0), which means that call is not part of a transaction; `MQ_MTS_TRANSACTION` (1), which means that the call is made as part of the current MTS or COM+ transaction; `MQ_XA_TRANSACTION` (2), which means that the call is part of an externally coordinated, XA-compliant transaction; or `MQ_SINGLE_MESSAGE` (3), which means that the call retrieves a single message.

`WantDestinationQueue` is a `Boolean` when `True` means that the `MSMQMessage.DestinationQueueInfo` property will be updated to contain the information about the destination queue. If not specified, a value of `False` will be assumed.

`WantBody` is a `Boolean` when `True` means that the body of the message should be returned (default). Set this value to `False` to reduce the amount of time to return the message.

`ReceiveTimeout` is a `Long` containing the number of milliseconds that MSMQ will wait for a message to arrive.

`WantConnecterType` is a `Boolean` when `True` means that connector information will also be retrieved. If not specified, a value of `False` will be assumed.

### Sub Reset()

The `Reset` method moves to the current message cursor to the start of the queue.

# The MSMQMessage Object

The `MSMQMessage` object contains the message that is sent and received using message queuing.

## MSMQMessage object properties

Table 19-3 lists the properties of the `MSMQMessage` object.


**Set not for the body of a message:** Don't use the `Set` statement to assign an object to the `Body` property of an `MSMQMessage` object. The `Set` statement creates a reference to an object. You should use an assignment statement to copy the entire contents of the object into the `Body` property.

Table 19-3
**Properties of the MSMQMessage Object**

| *Property* | *Description* |
|---|---|
| `Ack` | A `Long` value that specifies the type of acknowledgement that is returned. |
| `AdminQueueInfo` | An object reference to the `MSMQQueueInfo` object that is used for acknowledgement messages. |
| `AppSpecific` | A `Long` value containing application-specific information. |
| `ArrivalTime` | A `Date` value containing the date and time the message arrived at the queue. |
| `AuthenticationProviderName` | A `String` value containing the name of the cryptographic provider used to generate the digital signature for the message. |
| `AuthenticationProviderType` | A `Long` value containing the type of the cryptographic provider. |
| `AuthLevel` | A `Long` value that specifies whether or not the message should be authenticated when received. |
| `Body` | A `Variant` value containing the message to be sent. Typically, this will be either a `String` or apersistent COM component. |
| `BodyLength` | A `Long` value containing the number of bytes in the message. |
| `Class` | A`Long` value containing the type of message being sent. |

*Continued*

Table 19-3 *(continued)*

| *Property* | *Description* |
|---|---|
| `ConnectorTypeGUID` | A `String` value containing the GUID associated with the component that was used to externally set some of the message properties that are typically set by MSMQ. |
| `CorrelationId` | A `Variant` value containing a 20-byte application-defined value that can be used to link messages together. |
| `Delivery` | Specifies how the message is delivered. Possible values are `MQMSG_DELIVERY_EXPRESS` (0), which is default and specifies a normal delivery process where it may be possible to lose the message in case of system failure; or `MQMSG_DELIVERY_RECOVERABLE` (1), which means that a more reliable system is used. |
| `DestinationQueueInfo` | An object reference to an `MSMQQueueInfo` object that will be used as the destination queue. |
| `DestinationSymmetricKey` | A `Variant` value containing the symmetric key used to encrypt messages. |
| `EncryptAlgorithm` | A `Long` value, which specifies the encryption algorithm used to encrypt the body of the message. |
| `Extension` | A `Variant` value containing additional application-specific information associated with the message. |
| `HashAlgorithm` | A `Long` value containing the hash algorithm used to authenticate a message. |
| `Id` | A `Variant` value containing the identifier for this message. This value is automatically generated by MSMQ. |
| `IsAuthenticated` | An `Integer` value that specifies whether the local queue manager authenticated the message. |
| `IsFirstInTransaction` | An `Integer` value that specifies whether the message is the first message sent in a transaction. |
| `IsLastInTransaction` | An `Integer` value that specifies whether the message is the last message sent in a transaction. |

| *Property* | *Description* |
|---|---|
| `Journal` | A `Long` value that specifies whether a copy of the message was stored in the journal queue. |
| `Label` | A `String` value containing an application-specific value describing the message. |
| `MaxTimeToReachQueue` | A `Long` value containing the maximum number of seconds that can elapse before the message must reach the queue before it will be canceled and an error message returned to the sender. |
| `MaxTimeToReceive` | A `Long` value containing the maximum number of seconds that can elapse before the message must be received before it will be canceled and an error message returned to the sender. |
| `MsgClass` | A `Long` value containing the message type. |
| `Priority` | A `Long` value containing the relative priority of the message. |
| `PrivLevel` | A `Long` value containing how the message is encrypted. |
| `ResponseQueueInfo` | An object reference to an `MSMQQueueInfo` object that specifies the response queue used to send response information. |
| `SenderCertificate` | A `Byte()` array containing the security certificate information. |
| `SenderId` | A `Byte()` array containing the name of the user who sent the message. |
| `SenderIdType` | A `Long` value that specifies whether the `SenderId` value was included in the message. |
| `SenderVersion` | A `Long` value containing information about the version of MSMQ that was used to send the message. |
| `SendTime` | A `Date` value containing the date and time the message was sent. |
| `Signature` | A `Byte()` containing the digital signature used to authenticate the message. |
| `SourceMachineGuid` | A `String` array value containing the GUID of the computer, which sent the message. |

*Continued*

Table 19-3 *(continued)*

| *Property* | *Description* |
|---|---|
| `Trace` | A `Long` value that specifies where a message will be returned to the sender in the report queue for each hop taken by the message in the delivery process. |
| `TransactionId` | A `Byte()` array that identifies the transaction that sent the message. |
| `TransactionStatusQueueInfo` | An object reference to a transaction status queue used for transactional messages. |

## MSMQMessage object methods

The `MSMQMessage` object contains methods for sending a message to a queue.

### Sub AttachCurrentSecurityContext()

The `AttachCurrentSecurityContext` method retrieves the security information needed to attach a certificate to the message when requesting authentication.

### Sub Send(DestinationQueue As MSMQQueue, [Transaction])

The `Send` method transmits the message to the specified queue.

`DestinationQueue` is an object reference to an open `MSMQQueue` object.

`Transaction` is a `Variant` which can be a reference to a `MSMQTransaction` object or one of these constants: `MQ_NO_TRANSACTION` (0), which means that this call isn't part of a transaction; `MQ_MTS_TRANSACTION` (1), which means that this call is part of the current MTS or COM+ transaction; `MQ_XA_TRANSACTION`, which means that this call is part of an externally coordinated, XA-compliant transaction; or `MQ_SINGLE_TRANSACTION` (3), which means the message comprises a single transaction.

# MSMQEvent object events

The `MSMQEvent` object is used to define the events in your program that will be fired when a message arrives in the queue. These events are defined by the `EnableNotification` method of the `MSMQQueue` object.

### Event Arrived (Queue As Object, Cursor As Long)

The `Arrived` event is triggered when an `MSMQMessage` object arrives in the associated queue.

`Queue` is an object reference to the queue containing the newly arrived message.

`Cursor` is a `Long` value for `Cursor` as specified in the `EnableNotification` method.


**It's not automatic:** You must call the `EnableNotification` method after processing the newly arrived message in order to be notified when the next message arrives.

### Event ArrivedError (Queue As Object, ErrorCode As Long, Cursor As Long)

The `ArrivedError` event is triggered when an error occurs.

`Queue` is an object reference to the queue containing the newly arrived message.

`ErrorCode` is a `Long` value containing the cause of the error.

`Cursor` is a `Long` value for `Cursor` as specified in the `EnableNotification` method.


**Timeout for an error:** The most common error returned in the ArrivedError event is a timeout, where no messages were received in the specified amount of time. As with the `Arrived` event, you must use the `EnableNotification` method again to receive the next event.

## Accessing Message Queues

While the object model for message queuing looks complicated, it isn't that difficult to master, especially if you use private queues. To demonstrate message queuing, I built a pair of programs. One originates requests, while the other responds to them. The client program (see Figure 19-5) retrieves information about a customer, while the server program (see Figure 19-6) processes the requests. I decided to use the COM+ transaction to get the information because it demonstrates how the various pieces can fit together into a robust application design.

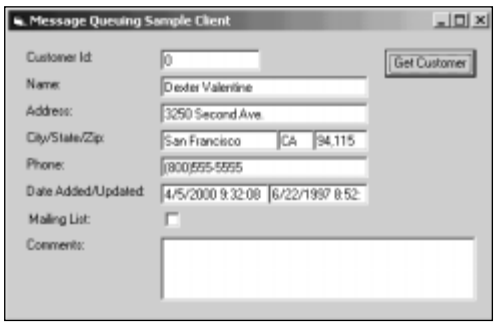


**Figure 19-5:** Running the message queuing client


**Figure 19-6:** Running the message queuing server

## Building the client program

The client program is a relatively simple Visual Basic program that communicates with the server using two message queues. The first queue is the request queue, which is used to pass information to the server, while the second queue is the response queue, which is used to receive information from the server.

### Starting the client

When the client program loads, I initialize both the request queue (`ReqQueue`) and the response queue (`RespQueue`) (see Listing 19-1). The process to open both queues is the same. I create a new `QueueInfo` object, specify the `PathName` of the queue, and then use the `Open` method to get access to the queue.

### Listing 19-1: **The Form_Load event in MSMQ Client**

```
Private Sub Form_Load()

Set ReqInfo = New MSMQQueueInfo
ReqInfo.PathName = ".\Private$\VB6DB19Req"
Set ReqQueue = ReqInfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

Set RespInfo = New MSMQQueueInfo
RespInfo.PathName = ".\Private$\VB6DB19Resp"
Set RespQueue = RespInfo.Open(MQ_RECEIVE_ACCESS, _
MQ_DENY_RECEIVE_SHARE)

Set RespEvent = New MSMQEvent
RespQueue.EnableNotification RespEvent, , 1000

End Sub
```

When I open the request queue, I specify `MQ_DENY_NONE` to allow other programs to share the request. However, when I open the response queue, I don't permit anyone else to receive data from it by specifying `MQ_DENY_RECEIVE_SHARE`. In general, each user should have their own unique response queue. This permits the server to send information to multiple users, even though the users may not be around to receive the information.

Once the response queue is opened, I create an `MSMQEvent` object, whose `Arrived` event will be fired when a message arrives in the response queue. I use the `EnableNotification` method to associate the Event object with the response queue. I also specify a timeout value of 1000 milliseconds (or 1 second). If no messages arrive in this amount of time, the `ArrivedError` event will be fired.

## Requesting a customer's information

After the user enters a CustomerId value and presses the Get Customer button, the `Command1_Click` event will be triggered (see Listing 19-2). This routine verifies that a numeric value was entered into the `Text1` text box, and then constructs and sends a message to the server.

I begin by creating a new `Customer` object and assigning the value of the text box to the `CustomerId` property. Then I create a new `MSMQMessage` object. I set the `ResponseQueueInfo` property to point to the `QueueInfo` object associated with the response queue.

### Listing 19-2: **The Command1_Click event in MSMQ Client**

```
Private Sub Command1_Click()

Dim msg As New MSMQMessage

If IsNumeric(Text1.Text) Then
   Set c = New Customer
   c.CustomerId = CLng(Text1.Text)
   Set msg = New MSMQMessage
   Set msg.ResponseQueueInfo = RespQueue.QueueInfo
   msg.Body = c
   msg.Send ReqQueue

End If

End Sub
```

The next statement is very critical. You must assign the object to the message's `Body` property. This saves a copy of the object in the `Body` property rather than merely saving a pointer to the object. Once you've populated the message, you can use the `Send` method to place the message in a queue.


**Set not:** By now, if you see an assignment with an object on the right side of the equal sign (=), you expect to see a `Set` statement before the variable on the left side. If you try to use the `Set` statement with the `Body` property of a message, your information will be lost as soon as the message is sent. Message queuing saves the persistent information from an object and then destroys it. The persistent information is then sent to the destination, where it is recreated when the program receives the object.

## Getting the results

Because message queuing is asynchronous in nature, you don't know if the message will be returned in milliseconds, seconds, hours, or days. Thus, it is useful for an event to be fired when the message arrives. The `Arrived` event is triggered when a message is ready to be received (see Listing 19-3). If no messages arrive before the timeout limit is reached, the `ArrivedError` event will be fired (see Listing 19-4).

The `Arrived` event begins by receiving the first message from the queue. The queue that the message arrives in is specified by the `Queue` parameter and the `Receive` method is used to retrieve the message. I specify a value of 1000 milliseconds to retrieve the message before returning an error.

### Listing 19-3: **The RespEvent_Arrived event in MSMQ Client**

```
Private Sub RespEvent_Arrived(ByVal Queue As Object, _
      ByVal Cursor As Long)

Dim r As Customer
Dim msg As MSMQMessage

Set msg = Queue.Receive(, , , 1000)
If Not (msg Is Nothing) Then
   Set r = msg.Body
   Text2.Text = r.Name
   Text3.Text = r.Street
   Text4.Text = r.City
   Text5.Text = r.State
   Text6.Text = FormatNumber(r.Zip, 0)
   Text7.Text = r.Phone
   Text8.Text = FormatDateTime(r.DateAdded, vbGeneralDate)
   Text9.Text = FormatDateTime(r.DateUpdated, vbGeneralDate)
   Check1.Value = CLng(r.MailingList) * -1
   Text10.Text = r.Comments

End If

RespQueue.EnableNotification RespEvent, , 1000

End Sub
```

Once I have the message, it is a very simple matter to get a reference to the `Customer` object from the `Body` property and assign its values to each of the fields on the form.


**What happened to all my requests?:** When receiving messages in the `Arrived` event, it is possible to process the messages too quickly. In this example, I assumed that the user would submit one request at a time, but there is nothing to prevent the user from requesting information on multiple customers before the response to the first message is received. If this happens, it is quite possible that the server will retrieve the next response before the user has finished viewing the first response. In this case, it may be useful to save the messages into a `Collection` object as they are received. Another approach would be to examine the information in each response using the `Peek` methods, and only receive the message from the message queue when you're ready to delete it.

Before I leave this event, I need to reset the trigger so that the next message is received. This is done using the same `EnableNotification` method that I originally used to enable this event.

**But it worked once:** If you can receive the first message, but your program doesn't process any other messages after that, verify that you re-enabled the response event after processing a message.

The `ArrivedError` event is fired when a message doesn't arrive in the message queue before its timeout value. If you are expecting a steady stream of messages, you may want to re-enable the event in the `ArrivedError` event (see Listing 19-4).

### Listing 19-4: The RespEvent_ArrivedError event in MSMQ Client

```
Private Sub RespEvent_ArrivedError(ByVal Queue As Object, _
      ByVal ErrorCode As Long, ByVal Cursor As Long)

RespQueue.EnableNotification RespEvent, , 1000

End Sub
```

## Building the server program

The server program is designed to receive requests from a client in the request queue, call the `GetCustomer` transaction I wrote in Chapter 18, and return the data to the client in the response queue. It also includes tools to create and delete the queues that the client program will use.

### Creating queues

When the user presses the Build Queues button, the `Command1_Click` event will be triggered (see Listing 19-5). For each queue, this routine creates a new `QueueInfo` object, specifies the `PathName` of the queue, and then calls the `Create` method to define the queue.

**Deleting is simple:** To delete a queue, use the same logic as shown in Listing 19-5, but substitute `Delete` for `Create`.

### Listing 19-5: The Command1_Click event in MSMQ Server

```
Private Sub Command1_Click()

Dim RespInfo As MSMQQueueInfo

Set RespInfo = New MSMQQueueInfo
RespInfo.PathName = ".\Private$\VB6DB19Resp"
RespInfo.Create

Set ReqInfo = New MSMQQueueInfo
ReqInfo.PathName = ".\Private$\VB6DB19Req"
ReqInfo.Create

Set respinfo = Nothing
Set ReqInfo = Nothing

End Sub
```

## Starting the server

The server will not process any messages until the user presses the Run Server button. This triggers the `Command2_Click` event, as shown in Listing 19-6. This routine begins by writing that the server is active to the text box on the form, and then it opens the request queue and enables notification using the `ReqEvent` object. At this point, the server will respond to messages as they are sent, using the `ReqEvent_Arrival` events.

### Listing 19-6: The Command2_Click event in MSMQ Server

```
Private Sub Command2_Click()

Text1.Text = FormatDateTime(Now, vbGeneralDate) & _
   " Server active." & vbCrLf
Set ReqInfo = New MSMQQueueInfo
ReqInfo.PathName = ".\Private$\VB6DB19Req"
Set ReqQueue = ReqInfo.Open(MQ_RECEIVE_ACCESS,
MQ_DENY_RECEIVE_SHARE)

Set ReqEvent = New MSMQEvent
ReqQueue.EnableNotification ReqEvent, , 60000

End Sub
```

### Processing messages

The heart of the server is the `ReqEvent_Arrived` event (see Listing 19-7). This event is responsible for receiving the request, calling the COM+ transaction, and returning the data to the user via the response queue. This routine begins by appending an entry to the text box with the date and time that the message was received. This information is useful in monitoring the server's activities while testing the program, though it should be eliminated in a production application.

**Listing 19-7: The ReqEvent_Arrival event in MSMQ Server**

```
Private Sub ReqEvent_Arrived(ByVal Queue As Object, _
      ByVal Cursor As Long)

Dim c As Customer
Dim ci As CustInfo.CustomerInfo
Dim msg As MSMQMessage
Dim xc As Customer
Dim xmsg As MSMQMessage
Dim xqueue As MSMQQueue

Text1.Text = Text1.Text & FormatDateTime(Now, vbGeneralDate) _
   &      " Message received." & vbCrLf

Set msg = Queue.Receive(, , , 1000)
If Not (msg Is Nothing) Then
   Set c = msg.Body

   Set ci = CreateObject("CustInfo.CustomerInfo")
   Set xc = ci.GetCustomer(c.CustomerId)
   Set ci = Nothing

   Set xmsg = New MSMQMessage
   xmsg.Body = xc

   Set xqueue = msg.ResponseQueueInfo.Open(MQ_SEND_ACCESS, _
      MQ_DENY_NONE)
   xmsg.Send xqueue
   Set xqueue = Nothing
   Set xc = Nothing

End If

ReqQueue.EnableNotification ReqEvent, , 60000

End Sub
```

Next, I receive the message using the `Queue` parameter. Once I have the message, I can extract the `CustomerId` value from the message's body and call the `CustInfo.CustomerInfo` transaction to get the requested information. Then I create a new `MSMQMessage` object and save the `Customer` object into the `Body` property. Note that I didn't use a `Set` statement, since I want a copy of the document rather than a reference for the object.

To send the message, I open the response queue contained in the original message and use the `Send` method to put the message into the response queue. Then I reset the message notification so that the next time a message arrives, I will be ready to process it. The program also includes a `ReqEvent_ArrivedError` similar to the one used in the MSMQ Client program to allow the server to continue to receive messages after a timeout condition.

# Viewing Message Queue Information

You may find it useful to have an independent tool to see messages as they arrive in a message queue for processing. If you are running Windows 2000 Server, you can use the Computer Management utility (Start ➪ Programs ➪ Administrative Tools ➪ Computer Management) to examine the contents of a message queue.

After starting this program, you'll see the typical Microsoft Management Console with a tree of icons on the left side with details about the currently selected icon on the right (see Figure 19-7). By expanding the Services and Applications icon, you can expose the Message Queuing icon. Then by drilling down, under Private Queues, you can see the queues that are used by the MSMQ Client and MSMQ Server programs.

Within each queue, you can select Queue messages to see the messages that are in the queue waiting to be processed. The easiest way to try this, using the sample programs from this chapter, is to run the client program without running the server. Any messages sent from the client program will accumulate in the queue, and if you then run the server program, you can see the messages quickly disappear—returned to the client program via the response queue.


**Where did my message go?:** If your program isn't working properly, this utility will help you discover whether or not the program is sending the messages properly.

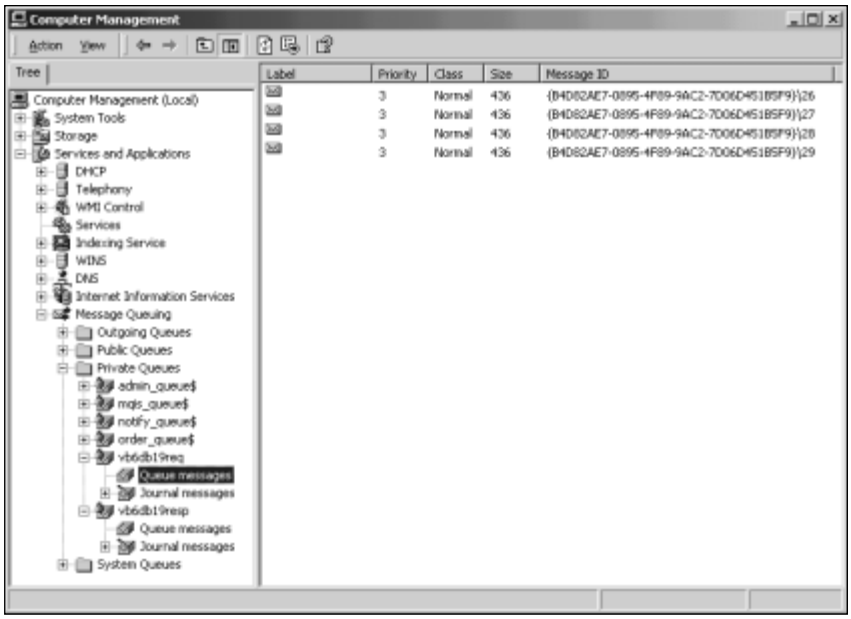


**Figure 19-7:** Running the Computer Management utility

## Thoughts on Message Queuing

This chapter is just a brief introduction to what you can do with message queuing. I find message queuing a fascinating tool that makes it easy to implement asynchronous processing.

Consider this example. You have a group of salespeople out in the field with laptop computers. Since it's not practical for them to stay in constant contact with the home office, you build a stand-alone application that keeps track of the items in the sales catalog and allows them to enter orders that can be printed out and submitted later.

If you were to include message queuing in the application, you could modify the application so that it automatically queues orders to be sent to the home office the next time the salesperson connects to the organization's network. The application could also receive updates to the local sales database, as well as updates to customer orders that have already been placed.

Another situation where you might find message queuing useful is when you have long-running tasks like report generators, or complex database queries where an immediate response isn't necessary. By using message queuing, you can collect these requests and then process them as time and resources permit. In the mainframe world, this is referred to as batch processing. However, I suggest that you refer to this process by another name so that your users don't think you're a relic from the olden days—like me.

# Summary

In this chapter you learned the following:

- ✦ You can use the Microsoft Message Queuing to send information asynchronously from one computer to another.
- ✦ You can use message queues to smooth the workload in a heavily loaded server.
- ✦ You can use public queues only on Windows 2000 servers, while private queues can be used on Windows 98 and NT as well as Windows 2000.
- ✦ You can open a queue with the `MSMQQueueInfo` object.
- ✦ You can send messages using the `MSMQQueue` object.
- ✦ You can receive messages via the `MSMQEvent` object.
- ✦ You can view the messages in a message queue by using the Computer Management utility to verify if your application is properly sending messages.

✦ ✦ ✦