Database Programming with VB.NET

CARSTEN THOMSEN

Apress™

ISBN (pbk): 1-893115-29-1
Printed and bound in the United States of America 12345678910
Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Editorial Directors: Dan Appleman, Gary Cornell, Karen Watterson, Jason Gilmore
Technical Reviewer: Mark Dunn
Managing Editor: Grace Wong
Marketing Manager: Stephanie Rodriguez
Project Manager: Alexa Stuart
Developmental Editor: Valerie Perry
Copy Editor: Ami Knox
Production Editor: Kari Brooks
Page Composition: Impressions Book and Journal Services, Inc.
Artist: Allan Rasmussen
Indexer: Carol Burbo
Cover Design: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010
and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany
In the United States, phone 1-800-SPRINGER; orders@springer-ny.com; http://www.springer-ny.com
Outside the United States, contact orders@springer.de; http://www.springer.de; fax +49 6221 345229

For information on translations, please contact Apress directly at 901 Grayson Street, Suite 204, Berkeley, CA, 94710
Phone: 510-549-5938; Fax: 510-549-5939; info@apress.com; http://www.apress.com

The source code for this book is available to readers at http://www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 8

Message Queues

Using message queues for connectionless programming

Message queues are great whenever you don't have a permanent connection or in loosely coupled situations. They're also handy when you simply want to dump your input and return to your application without waiting for the output from whatever application or server will be processing your input. You can make message queues the basis of a fail-safe application that uses them whenever your regular database fails. This can help ensure that no data is lost and you have very little or no downtime at all, if you use transactional queues. Obviously not all applications can use message queues to such advantage, but for storing or accepting customer input it can be a great solution. One situation springs to mind in which using message queues would not be beneficial and that is a banking transaction that needs to be processed in real-time.

Microsoft's messaging technology is called *Message Queuing*. Mind you, in line with other traditional Microsoft marketing stunts, this technology has had other names before Windows 2000 was shipped. However, the release that ships with Windows 2000 is built upon a previous version of the technology and includes the Active Directory (AD) integration. AD integration is good because it is now easier to locate public message queues across your domain and it ensures easy Enterprise-wide backup.

Is MSMQ Dead?

Microsoft Message Queue Server (MSMQ) is the name Microsoft gave version 1.0 of its message queuing technology. It was originally included with the Enterprise Edition of Windows NT 4. With version 2.0 it became an add-on to any server version of Windows NT 4, and it was part of the Windows NT Option Pack. The Windows NT Option Pack could be freely downloaded from the Microsoft Web site. Since then the message queuing technology has been integrated into the OS, as is the case with Windows 2000 and the upcoming Windows 2002 and Windows XP. Not only has MSMQ been integrated into the

OS, it has changed its name to Message Queuing and been extended in numerous ways. One extension is the integration with Active Directory instead of being SQL Server based.

So to answer the question, "Is MSMQ Dead?" the answer is *yes* and *no*. Yes, because it is no longer a stand-alone product, and no because the features of the original MSMQ have been incorporated into the OS as a service called Message Queuing.

Connectionless Programming

Connectionless programming involves using a queue to hold your input to another application or server. Connectionless in this context means you don't have a permanent connection to the application or server. Instead, you log your input in a queue, and the other application or server then takes it from the queue, processes it, and sometimes puts it back in the same or a different queue.

The opposite of connectionless is *connection oriented,* and basically any standard database system these days falls into this category. You have probably heard these two terms in relation to TCP/IP (Transmission Control Protocol/Internet Protocol). The UDP (User Datagram Protocol) part of the protocol is connectionless, and the TCP part is connection oriented. However, there is a major difference between message queuing and the UDP protocol. Whereas the UDP protocol does not guarantee delivery, and a packet sent over the network using the UDP transport protocol can be lost, the message queue doesn't lose any packets. The reason why a message queue is said to be connectionless is that your client doesn't "speak" directly with your server. There is no direct connection, as there is with a standard database.

Now that you have an understanding of connectionless programming, let's move on to an overview of the **MessageQueue** class.

Taking a Quick Look at the MessageQueue Class

The **MessageQueue** class is part of the **System.Messaging** namespace. That namespace also holds the **Message** class, which is used in conjunction with the **MessageQueue** class. The **System.Messaging** namespace holds a number of classes that are exclusively used for accessing and manipulating message queues.

The **MessageQueue** class is a wrapper around **Message Queuing**, which is an optional component of Windows 2000 and Windows NT. With Windows NT it's part of the Option Pack. At any rate, you need to make sure you have installed Message Queuing on a server on your network before you can try the examples in this chapter.

When to Use a Message Queue

Use a message queue in your application if you need to perform any of these tasks:

- *Storing less-important information in a message queue while your DBMS is busy serving customers during the day:* The information in the message queue is then batch-processed when the DBMS is less busy. This is ideal for processing all real-time requests and storing all other requests in a message queue for later processing.

- *Connecting to a DBMS over a WAN, such as the Internet, and the connection is not always available for whatever reason:* You can do two things when you design your application. You can set up your application to access the DBMS the usual way, and if you get a time-out or another connection failure, you store the request in a message queue. You then start a component that checks for when the DBMS is available, and when it is, it will forward the requests that are stored in the message queue and return a possible result to your application. At this point your application can resume normal operation and shut down the message queue component. The other possibility is to design your application as an offline one, meaning that all DBMS requests are stored in a local message queue and forwarded by your message queue component to the DBMS.

- *Exchanging data with a mainframe system, like SAP:* You can use a message queue to hold SAP IDocs. These are then sent to and received from the SAP system by the queue manager/component. Anyone who ever worked with SAP will know that real-time communications with SAP is often a problem, because the server is often overloaded at specific intervals during the day. So basically you use the message queue for storing and forwarding requests to and from the SAP system. Recall that SAP is originally a mainframe system, but these days SAP is often found running on midsized computers or PCs. The five most popular platforms for R/3 are Windows NT, Sun Solaris, HP-UX, AS/400, and AIX.

I am sure you yourself can come up with a few applications that would benefit from message queuing based on your current or previous work experience. How about sales people in the field? Wouldn't it be good if they could save some info in a message queue and have it batch processed when they are back in the office?

Why Use a Message Queue and Not a Database Table?

A message queue can be of good use in many situations, but what makes it different from using a database table? When you call a DBMS, it is normally done synchronously, but with a message queue it is done asynchronously. This means it is generally faster to access a message queue than to access a database table. With synchronous access the client application has to wait for the server to respond, whereas with asynchronous access a query or the like is sent to the server, and the client application then continues its normal duties. When the server has finished processing the query, it will notify you and pass you the result.

When you use a database table, you must comply with a fairly strict format for adding data. There are certain fields that must be supplied, and you don't really have the ability to add extra information. Although the same can be said about message queues, when it comes to supplying certain fields, it is much easier to supply extra information that you have a hard time storing in a database table.

This approach is valid for Windows forms and Web forms, though it depends on the *application type*. Distributed applications, standard client-server applications on a local network, and stand-alone applications constitute some of the different application types. If your application is not distributed and you are accessing a standard DBMS on a local server or on your client, I don't see a reason for using a message queue. Actually, one reason might be that your DBMS is busy during the day, and hence you store your requests and messages in a message queue for batch processing when the DBMS is less busy.

In short, application infrastructure and business requirements in general determines whether you should use message queuing. Having said this, I realize that sometimes it is a good idea to work with a tool for a while so that you can get to know the situations when you should use it. So if you're new to message queuing, make sure you try the example code that follows later on in this chapter.

A message queue works using the First In, First Out (FIFO) principle, which means that the first message in the queue is the message you will retrieve if you use the **Receive** or **Peek** method of the **MessageQueue** class. See "Retrieving a Message" and "Peeking at Messages" later in this chapter for more information on these methods.

Because FIFO is the principle applied to how the messages are placed in the queue, a new message is placed at the bottom, or end, of the message queue when added, right? Well, this is correct, but there is more to it than just the arrival time in the queue. Each message also has a priority that determines its place in the queue. The priority takes precedence over the arrival time, so it is possible to place a message at the top of the message queue even if there are already other messages in the queue. If I were to make up a **SELECT** statement that describes how the messages are inserted into a queue and returned to you, it would look something like this:

```
SELECT * FROM MessageQueue ORDER BY Priority, ArrivalTime
```

Obviously this statement is just an example, but it does show you how the messages are ordered, first by priority and then by arrival time. Priority is not normally something you would set, so most messages would have the same priority, but there may be times when it is necessary to "jump" the queue. See "Prioritizing Messages," later in this chapter for more information on how to prioritize your messages.

How to Use a Message Queue

Of course, you need to set up the Message Queue before you can use it in your application. Maybe you already have a queue to connect to, but first I will show you how to set up a queue. Please note that you cannot view or manage public queues if you're in a workgroup environment. Public queues only exist in a domain environment. See your Windows 2000 or Windows NT documentation for installing Message Queuing. If you are running in workgroup mode, you cannot use Server Explorer to view a queue, not even private ones. I will get to private and public queues in the next section.

Private Queue vs. Public Queue

You need to know a distinction between the two types of queues you can create before continuing. There are private and public queues, and Table 8-1 compares some of the features of both types of message queues.

Table 8-1. Private vs. Public Message Queues

PRIVATE MESSAGE QUEUE	PUBLIC MESSAGE QUEUE
Not published in the Message Queue. Information Service (MQIS) database.	Published in the Message Queue Information Service (MQIS) database.
Can only be created on the local machine.	Must be registered with the Directory Service.
Can be created and deleted off-line.	Must be created and deleted while you're online.
Cannot be located by other message queuing applications, unless they are provided the full path of the message queue.	Can be located by other message queuing applications through the Message Queue Information Service (MQIS) database.
Is persistent, but backing up, although possible, is not an easy task.	Is persistent and can be backed up on an Enterprise level with the Active Directory.

Creating the Queue Programmatically

Chapter 4 covers how to create a message queue from the Server Explorer, but if you want to do it programmatically, please read on.

It is easy to create a message queue programmatically. Listings 8-1 and 8-2 show you two different ways of creating a private queue on the local machine, USERMANPC.

Listing 8-1. Creating a Private Message Queue Using the Machine Name

```
1 Public Sub CreatePrivateQueueWithName()
2    Dim queUserMan As New MessageQueue()
3
4    queUserMan.Create("USERMANPC\Private$\UserMan")
5 End Sub
```

Listing 8-2. Creating a Private Message Queue Using the Default Name

```
1 Public Sub CreatePrivateQueue()
2    Dim queUserMan As New MessageQueue()
3
4    queUserMan.Create(".\Private$\UserMan")
5 End Sub
```

In Listing 8-1 I specify the machine as part of the path argument, whereas I simply use a period in Listing 8-2. The period serves as a shortcut for letting the **MessageQueue** object know that you want it created on the local machine. The other thing you should notice is that I prefix the queue with `Private$`. This must be done to indicate that the queue is private. Exclusion of the prefix indicates you

are creating a public queue. The different parts of the path are separated using the back slash, as in a standard DOS file path:

```
MachineName\Private$\QueueName
```

A public queue is created more or less the same way, except you leave out the \Private$ part of the path. You can also use the period to specify that you want it created on the local machine, as shown in Listing 8-3.

Listing 8-3. Creating a Public Message Queue Using the Default Name

```
1 Public Sub CreatePublicQueue()
2    Dim queUserMan As New MessageQueue()
3
4    queUserMan.Create(".\UserMan")
5 End Sub
```

As stated, the period could have been substituted with the name of the local machine or indeed the name of a different machine on which you want to create the queue.

> **NOTE** *If you try to create an already existing message queue, a* ***QueExists*** *exception is thrown.*

There are two overloaded versions of the **MessageQueue** class's **Create** method, and you have seen the simpler one. The other one takes a second argument, a **Boolean** value indicating whether the message queue is transactional. I will go into message queue transactions in the "Making Message Queues Transactional" section later on in this chapter, but for now take a look at Listing 8-4, which shows you how to make the queue transactional:

Listing 8-4. Creating a Transactional Private Message Queue

```
1 Public Sub CreateTransactionalPrivateQueue()
2    Dim queUserMan As New MessageQueue()
3
4    queUserMan.Create(".\Private$\UserMan", True)
5 End Sub
```

In Listing 8-4 a private message queue named UserMan is created on the local machine with transaction support. If you run the example code in Listing 8-4, you might want to remove it afterwards, as it might otherwise have an impact on the rest of the examples in this chapter.

Displaying or Changing the Properties of Your Message Queue

Some of the properties of your message queue can be changed after you have created it. You can do this by using the Computer Management MMC snap-in that is part of the Administrative Tools under Windows 2000. When you have opened the Computer Management snap-in, do the following:

1. Expand the Services and Applications node.

2. Expand the Message Queuing node.

3. Expand the node with the requested queue and select the queue. The Computer Management MMC snap-in should now look like the one shown in Figure 8-1.

4. Right-click on the queue and select Properties from the pop-up menu. This brings up the queue Properties dialog box (see Figure 8-2).

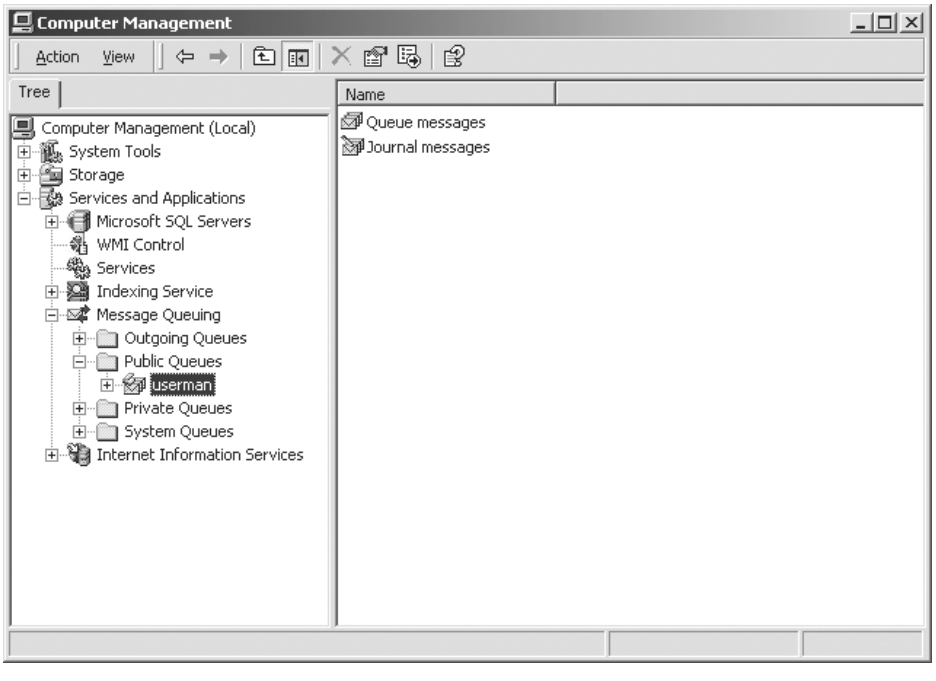

Figure 8-1. The Computer Management MMC snap-in with message queue selected

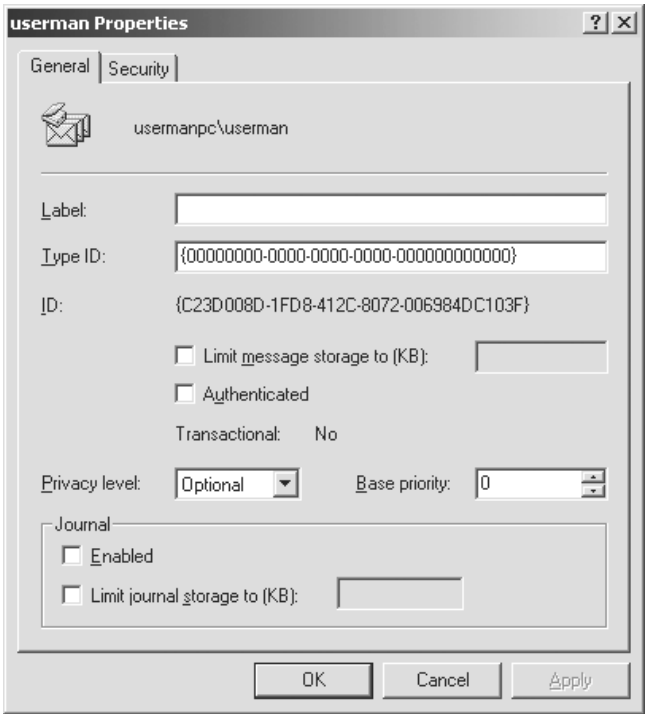

Figure 8-2. The Properties dialog box for the message queue

As you can see from Figure 8-2, one of the options that cannot be changed after the queue has been created is whether the queue is transactional or not. This makes it all the more important for you to know if you will need transaction support before you create the queue.

Assigning a Label to Your Message Queue

If you want to bind to your message queue using a label, you first need to assign a label to the queue. Looking back at Figure 8-2, notice you can enter the text for your label in the Label text box. One thing you have to remember is that it is not necessary for the label to be unique across all your message queues, but you are guaranteed problems if you try to bind and send messages to a queue using the same label. So the lesson learned here is make sure your label is unique!

You can also change the label of a message queue programmatically. See Listing 8-5 for example code.

Listing 8-5. Changing the Label of an Existing Queue

```
1 Public Sub ChangingQueueLabel()
2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3
4    queUserMan.Label = "USERMAN1"
5 End Sub
```

In Listing 8-5 the friendly name is used to bind to an existing private queue on the local machine. A friendly name is a name that makes it easier for humans to read and interpret it. In the case of the example code in Listing 8-5, the friendly name tells you that the queue is created on the local machine (.), it is private ((Private$), and the name of it is UserMan. You must use the friendly name to bind to the message queue if you want to change the label programmatically! Line 4 is where the actual changing of the label takes place simply by setting the label property.

Retrieving the ID of a Message Queue

If you want to bind to your public message queue using its ID, you can retrieve this using the Computer Management MMC snap-in. See "Displaying or Changing the Properties of Your Message Queue" later in this chapter for more information on how to display message queue properties. If you refer back to Figure 8-2, you can see the ID just below the Type ID text box! Mind you, this ID is only available if you are connected to the AD Domain Controller. If you're not connected, you cannot view the properties for a public message queue. If you try to view the ID for a private message queue, this space is simply blank.

You can also retrieve the ID of a message queue programmatically, as shown in Listing 8-6.

Listing 8-6. Retrieving the ID of an Existing Queue

```
1 Public Sub RetrieveQueueId()
2    Dim queUserMan As New _
3     MessageQueue(".\UserMan")
4    Dim uidMessageQueue As Guid
5
6    ' Save the Id of the message queue
7    uidMessageQueue = queUserMan.Id
8 End Sub
```

As you can see from Listing 8-6 the ID is a GUID, so you need to declare your variable to be data type **Guid** if you want to store the ID. You must be connected to the AD Domain Controller in order to retrieve the ID of a message queue,

otherwise an exception is thrown. If you try to retrieve the ID of a private message queue, an "empty" GUID is returned: **00000000-0000-0000-0000-000000000000**. You can display a GUID using the **ToString** method of the **Guid** class.

Binding to an Existing Message Queue

Once you have created a message queue or if you want to access an existing queue, you can now bind to it, as described in the following sections.

Binding Using Friendly Name

Listing 8-7 shows you three different ways of binding to an existing queue: using the New constructor, using a so-called friendly name (.\Private$\UserMan), or specifying the path of the queue and the name assigned to the queue when it was created.

Listing 8-7. Binding to an Existing Queue

```
1 Public Sub BindToExistingQueue()
2    Dim queUserManNoArguments As New MessageQueue()
3    Dim queUserManPath As New MessageQueue(".\Private$\UserMan")
4    Dim queUserManPathAndAccess As New MessageQueue(".\Private$\UserMan", _
5     True)
6
7    ' Initialize the queue
8    queUserManNoArguments.Path = ".\Private$\UserMan"
9 End Sub
```

Line 2 is the simplest method, but it requires an extra line of code because you haven't actually told the message queue object what queue to bind to. Line 8 takes care of specifying the queue using the **Path** property. On Line 4, I have specified the queue as well as the read-access restrictions. When the second argument is set to **True**, as is the case with the example in Listing 8-7, exclusive read access is granted to the first application that accesses the queue. This means that no other instance of the **MessageQueue** class can read from the queue, so be careful when you use this option.

Binding Using the Format Name

Since you cannot create or bind to a message queue on a machine in Workgroup mode, you need to access a queue on a machine that is part of Active Directory. However, this creates a problem when the message queue server you want to

access cannot access the primary domain controller. Because Active Directory is used for resolving the path, you won't be able to use the syntax shown in Listing 8-6 to bind to a message queue.

Thankfully there's a way around this. Instead of using the friendly name syntax as in the previous listings, you can use the format name or the label for this purpose. Listing 8-8 shows you how to bind to existing queues using the format name.

Listing 8-8. Binding to an Existing Queue Using the Format Name

```
1 Public Sub BindToExistingQueueUsingFormat()
2    Dim queUserManFormatTCP As New _
3     MessageQueue("FormatName:DIRECT=TCP:10.8.1.15\Private$\UserMan")
4    Dim queUserManFormatOS As New _
5     MessageQueue("FormatName:DIRECT=OS:USERMANPC\UserMan")
6    Dim queUserManFormatID As New _
7     MessageQueue("FormatName:Public=AB6B9EF6-B167-43A4-8116-5B72D5C1F81C")
8 End Sub
```

In Listing 8-8 I bind to the private queue named UserMan on the machine with the IP address 10.8.1.15 using the TCP protocol, as shown is on Lines 2 and 3. On Lines 4 and 5 I bind the public queue named UserMan on the machine with name USERMANPC. There is also an option of binding using the SPX protocol. If you want to use the SPX network protocol, you must use the following syntax: `FormatName:DIRECT=SPX:NetworkNumber;HostNumber\QueueName`.

You can connect to both public and private queues with all format name options, so in the example code in Listing 8-8 you could swap the public and private queue binding among the three different format names. The last format name shown in Listing 8-8 is on Lines 6 and 7 where I use the ID of the message queue as the queue identifier. The ID used in the sample code is fictive and it will look different on your network. The ID, which is a GUID, is generated at creation time by the MQIS. See "Retrieving the ID of a Message Queue" for more information on how to get the ID of your message queue.

Binding Using the Label

One last way to bind to an existing message queue is to use the label syntax. This syntax cannot be used when you're offline, only when connected to the Active Directory Domain Controller. Listing 8-9 shows you how to connect to the message queue with the UserMan label. See "Assigning a Label to Your Message Queue" earlier in this chapter for more information on how to set the label of a message queue.

Listing 8-9. Binding to an Existing Queue Using the Label

```
Public Sub BindToExistingQueueUsingLabel()
    Dim queUserManLabel As New MessageQueue("Label:UserMan")
End Sub
```

That rounds up how to bind to a message queue. Now it gets interesting, because next I will show you how to send, retrieve, and otherwise deal with messages.

Sending a Message

Sending a message is obviously one of the most important aspects of a message queue. If you can't send a message, why have a message queue? Let's take a look at the simplest form of sending a message to a message queue. The **Send** method of the **MessageQueue** class is used for this purpose, as demonstrated in Listing 8-10.

Listing 8-10. Sending a Simple Message to a Message Queue

```
Public Sub SendSimpleMessage()
    Dim queUserMan As New _
     MessageQueue(".\Private$\UserMan")

    ' Send simple message to queue
    queUserMan.Send("Test")
End Sub
```

After binding to the private UserMan queue on the local machine, I send a **String** object containing the text "Test" to the queue. Obviously this is not exactly useful, but you can apply this example to testing whether something is actually sent to the queue. As you have probably guessed, the **Send** method is overloaded, and I have used the version that only takes one argument and is an object.

If you execute the code in Listing 8-10, you can see the resulting message using the Server Explorer or the Computer Management MMC snap-in. Expand the private UserMan message queue and select the Queue Messages node. Now the Computer Management snap-in should resemble what is shown in Figure 8-3; if you're using the Server Explorer, it should resemble Figure 8-4. Please note that in Server Explorer, you need to expand the Queue Messages node to see the messages in the queue, as done in Figure 8-4.

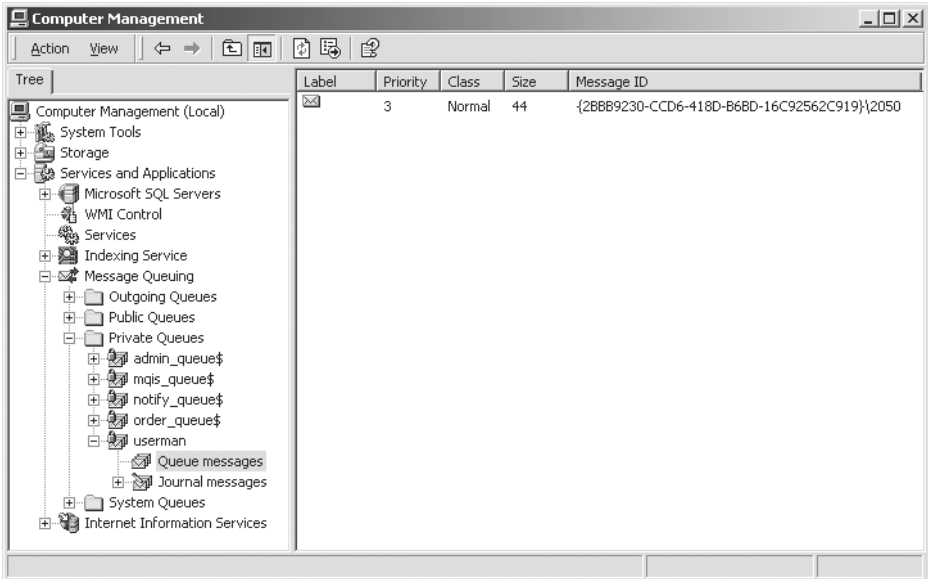

Figure 8-3. Computer Management with Queue Messages node selected

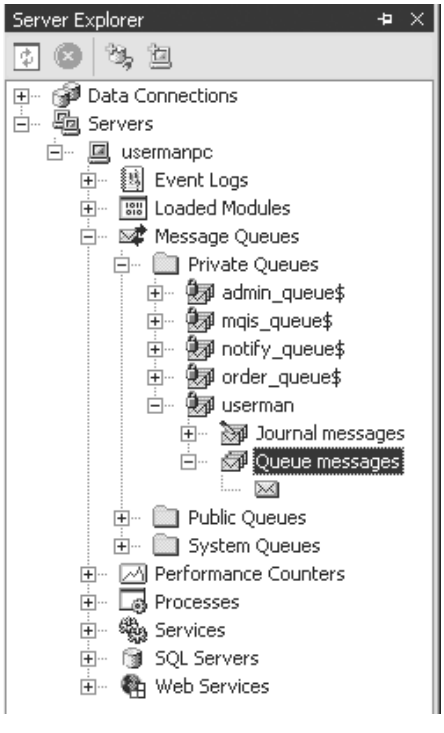

Figure 8-4. Server Explorer with Queue Messages node selected

NOTE *One major advantage Computer Management has over the Server Explorer when it comes to message queues is that you can use it when you're not connected to the AD Domain Controller. This is NOT possible with the Server Explorer, even if you want to look at private message queues!*

Retrieving a Message

Obviously it is a good idea to be able to retrieve the messages that have been posted to a message queue. Otherwise the messages simply stack up your message queue server and do absolutely no good to anyone. There are actually a number of ways you can retrieve a message from a message queue, but let's start out with retrieving the message sent in Listing 8-10. See Listing 8-11 for the code that retrieves the first message from the message queue.

Listing 8-11. Retrieving the First Message from a Message Queue

```
1 Public Sub RetrieveSimpleMessage()
2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3    Dim msgUserMan As Message
4
5    ' Retrieve first message from queue
6    msgUserMan = queUserMan.Receive()
7 End Sub
```

In Listing 8-11 I bind to the private UserMan message queue and retrieve the first message from the queue. When the message is retrieved from the message queue, it is also removed from the queue. If you need to read a message from the queue without removing it, you should take a look at "Peeking at Messages" later in this chapter for more information.

Although the example code in Listing 8-11 will retrieve the first message from the message queue, there isn't a lot you can do with it. If you try to access any of the properties of the msgUserMan message, an exception is thrown, because you have not set up a so-called formatter that can read the message. A message can take on a variety of different forms, so it is a must that you set up a formatter before you retrieve a message from the queue. See the next section, "Setting Up a Message Formatter," for more information on how to do this.

Receive and Peek

The **Receive** and **Peek** methods of the **MessageQueue** class are synchronous, meaning that they will block all other activity until the message has been retrieved. If there are no messages in the message queue, the **Receive** and **Peek** methods will wait until a message arrives in the queue. If you need asynchronous access you must use the **BeginReceive** method instead. Actually, you can specify a time-out value when you call the **Receive** or **Peek** methods to make sure you don't wait indefinitely. Both of these overloaded methods have a version that takes a **TimeSpan** argument. This means the method will return if a message is found in the queue or when the time specified in the **TimeSpan** argument has elapsed. An exception is thrown if the time elapses.

Setting Up a Message Formatter

Recall that you need to set up a formatter in order to be able to read the messages that you retrieve from your message queue. This is a fairly easy yet important task. The **Formatter** property of the **MessageQueue** class is used for this very purpose. When you instantiate an instance of the **MessageQueue** class, a default formatter is created for you, but this formatter cannot be used to read from the queue, only to write or send to the queue. This means you have to either change the default formatter or set up a new formatter.

See Listing 8-12 for example code that can retrieve and read the message sent in Listing 8-10. Mind you, if you have already retrieved that message from the queue, the queue is now empty, and you need to send another message before you try to receive again. If the message queue is empty, the **Receive** method will await the arrival of a new message in the queue.

Listing 8-12. Setting Up a New Formatter and Retrieving the First Message from a Queue

```
Public Sub RetrieveMessage()
   Dim queUserMan As New MessageQueue(".\Private$\UserMan")
   Dim msgUserMan As Message
   Dim strBody As String

   ' Set up the formatter
   queUserMan.Formatter = New XmlMessageFormatter(New Type() _
    {GetType(String)})

   ' Retrieve first message from queue
   msgUserMan = queUserMan.Receive
   ' Save the message body
   strBody = msgUserMan.Body.ToString
End Sub
```

In Listing 8-12 I specify that the formatter should be of type **XmlMessageFormatter**. This is the default formatter for a message queue, and it is used for serializing and deserializing the message body using XML format. I have also specified that the message formatter should accept message bodies of data type **String**. This is done on Lines 7 and 8. If I hadn't done this, I wouldn't be able to access the message body.

Generally you should use the same formatter for sending and receiving, but because this is a very simple example, the formatter is not needed for sending to the message queue. Now, let's see in which situations the **Formatter** property is really useful. In Listing 8-13 I set up the formatter so it will be able to read to different types of message bodies: A body of data type **String** and of data type **Integer**. This is done by adding both data types to the **Type** array as the only argument to the **XmlMessageFormatter** constructor.

Listing 8-13. Sending and Retrieving Different Messages from One Message Queue

```
 1 Public Sub SendDifferentMessages()
 2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
 3    Dim msgUserMan As New Message()
 4
 5    ' Set up the formatter
 6    queUserMan.Formatter = New XmlMessageFormatter(New Type() _
 7     {GetType(String), GetType(Integer)})
 8
 9    ' Create body as a text message
10    msgUserMan.Body = "Test"
11    ' Send message to queue
12    queUserMan.Send(msgUserMan)
13    ' Create body as an integer
14    msgUserMan.Body = 12
15    ' Send message to queue
16    queUserMan.Send(msgUserMan)
17 End Sub
18
19 Public Sub RetrieveDifferentMessages()
20    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
21    Dim msgUserMan As Message
22    Dim strBody As String
23    Dim intBody As Integer
24
25    ' Set up the formatter
26    queUserMan.Formatter = New XmlMessageFormatter(New Type() _
```

```
27     {GetType(String), GetType(Integer)})
28
29    ' Retrieve first message from queue
30    msgUserMan = queUserMan.Receive
31    strBody = msgUserMan.Body.ToString
32    ' Retrieve next message from queue
33    msgUserMan = queUserMan.Receive
34    intBody = msgUserMan.Body
35 End Sub
```

In Listing 8-13 I use the `SendDifferentMessages` procedure to send to two messages to the queue: one with a body of data type **String** and one with a body of data type **Integer**. In the `RetrieveDifferentMessages` procedure I retrieve these two messages in the same order they were sent to the message queue. The fact that you can send messages with a different body is what makes a message queue even more suitable than a database table for data exchange situations where the data exchanged has very different structures.

In the example code in Listing 8-13 I have used data types **String** and **Integer** as the body formats that can be deserialized when the messages are read from the queue, but you can in fact use any base data type as well as structures and managed objects. If you need to pass older COM/ActiveX objects, you should use the **ActiveXMessageFormatter** class instead. There is also the **BinaryMessageFormatter** class that you can use for serializing and deserializing messages using binary format, as opposed to the XML format used by the **XmlMessageFormatter** class.

Peeking at Messages

Sometimes it is not desirable to remove a message from the queue after you have had a look at the contents, as will happen when you use the **Receive** method of the **MessageQueue** class. However, there is another way: You can use the **Peek** method instead. This method does exactly the same as the **Receive** method except for removing the message from the queue. In addition, repeated calls to the **Peek** method will return the same message, unless a new message with a higher priority is inserted into the queue. See "Prioritizing Messages" later in this chapter for more information on that topic.

Listing 8-14 is actually a nearly complete copy of Listing 8-12 except for the call to the **Peek** method on Line 11. So if you want to look at the first message in the queue, the **Peek** and **Receive** methods work exactly the same, except for the removal of the message. See "Clearing Messages from the Queue" for details about removing messages.

Listing 8-14. Peeking at a Message in the Queue

```
Public Sub PeekMessage()
Dim queUserMan As New MessageQueue(".\Private$\UserMan")
Dim msgUserMan As Message
Dim strBody As String

' Set up the formatter
queUserMan.Formatter = New XmlMessageFormatter(New Type() _
 {GetType(String)})

' Peek first message from queue
msgUserMan = queUserMan.Peek
' Save the message body
strBody = msgUserMan.Body.ToString
End Sub
```

Picking Specific Messages from a Queue

A message queue is designed with a stack-like structure in mind. It doesn't have any search facilities as such, meaning that messages are retrieved from the top of the stack, or rather the top of the message queue. However, it is possible to pick specific messages from your queue. To do so, use the **ReceiveById** method, which takes a message ID as its only argument. The message ID is generated by Message Queuing and it is assigned to an instance of the **Message class** when it is sent to the queue. Listing 8-15 shows you how to retrieve the ID from a message and later use that ID to retrieve this message, even though it is not at the top of the message queue.

Listing 8-15. Retrieving a Message from the Queue by Message ID

```
Public Sub RetrieveMessageById()
Dim queUserMan As New MessageQueue(".\Private$\UserMan")
Dim msgUserMan As New Message()
Dim strId As String

' Set up the formatter
queUserMan.Formatter = New XmlMessageFormatter(New Type() _
 {GetType(String)})

' Create body as a text message
msgUserMan.Body = "Test 1"
' Send message to queue
```

```
13    queUserMan.Send(msgUserMan)
14
15    ' Create body as a text message
16    msgUserMan.Body = "Test 2"
17
18    ' Send message to queue
19    queUserMan.Send(msgUserMan)
20    strId = msgUserMan.Id
21    ' Create body as a text message
22    msgUserMan.Body = "Test 3"
23    ' Send message to queue
24    queUserMan.Send(msgUserMan)
25
26    msgUserMan = queUserMan.ReceiveById(strId)
27    MsgBox("Saved Id=" & strId & vbCrLf & "Retrieved Id=" & msgUserMan.Id)
28 End Sub
```

In Listing 8-15 I bind to the existing private UserMan queue, set up the formatter to accept data type **String**, create three messages and send them to the queue. Just after the second message is sent to the queue, the message ID is saved, and then this ID is used to retrieve the associated message. This message is second in the message queue. On Line 27 I simply display the saved message ID and the retrieved message to show that they are the same.

If you use the **ReceiveById** or indeed the **PeekById** method, you must be aware that an **InvalidOperationException** exception is thrown if the message does not exist in the queue. So whenever you use one of these methods, you should use a structured error handler, as demonstrated in Listing 8-16.

Listing 8-16. Retrieving a Message by Message ID in a Safe Manner

```
 1 Public Function RetrieveMessageByIdSafe(ByVal vstrId As String) As Message
 2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
 3    Dim msgUserMan As New Message()
 4
 5    ' Set up the formatter
 6    queUserMan.Formatter = New XmlMessageFormatter(New Type() _
 7     {GetType(String)})
 8
 9    Try
10       msgUserMan = queUserMan.ReceiveById(vstrId)
11    Catch objE As InvalidOperationException
12       ' Message not found, return a Null value
13       RetrieveMessageByIdSafe = Nothing
14
15       Exit Function
```

```
    End Try

    ' Return message
    RetrieveMessageByIdSafe = msgUserMan
End Function
```

Listing 8-16 catches an attempt to locate a nonexisting message by ID and simply returns a null value to the caller. If the passed message ID can be found in the queue, the retrieved message is returned to the caller.

The ID of a message is of data type **String**, but it is internally a GUID as you can see from Listing 8-15. At any rate, the **Id** property of a message is read-only, so you cannot assign your own IDs to your messages. I guess this is one way of ensuring all messages are unique. Your only option is to save the ID of a particular message and then later use this ID to retrieve the message.

> **NOTE** *The **PeekById** method works the same as the **ReceiveById** method, except that the message is not removed from the message queue.*

Sending and Retrieving Messages Asynchronously

Sometimes it is desirable to be able to send and receive your messages in asynchronous fashion. This makes sure your application isn't held up until delivery or retrieval has been completed. In order to use asynchronous communication with a message queue, you need to set up an event handler that deals with the result of the asynchronous operation. This can done using the **AddHandler** statement. See Listing 8-17 for an example.

*Listing 8-17. Receive a Message Asynchronously Using **AddHandler***

```
Public Sub MessageReceiveCompleteEvent(ByVal vobjSource As Object, _
 ByVal vobjEventArgs As ReceiveCompletedEventArgs)
    Dim msgUserMan As New Message()
    Dim queUserMan As New MessageQueue()

    ' Make sure we bind to the right message queue
    queUserMan = CType(vobjSource, MessageQueue)

    ' End async receive
    msgUserMan = queUserMan.EndReceive(vobjEventArgs.AsyncResult)
End Sub

Public Sub RetrieveMessagesAsync()
```

```
14    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
15    Dim msgUserMan As New Message()
16
17    ' Set up the formatter
18    queUserMan.Formatter = New XmlMessageFormatter(New Type() _
19     {GetType(String)})
20
21    ' Add an event handler
22    AddHandler queUserMan.ReceiveCompleted, _
23     AddressOf MessageReceiveCompleteEvent
24
25    queUserMan.BeginReceive(New TimeSpan(0, 0, 10))
26 End Sub
```

In Listing 8-17 I have first set up the procedure that will receive the `ReceiveComplete` event. This procedure takes care of binding to the passed queue, which is the queue used to begin the retrieval, and then the asynchronous retrieval is ended by calling the **EndReceive** method. This method also returns the message from the queue.

In the `RetrieveMessagesAsync` procedure I set up the message queue, and on Line 22 I add the event handler that calls `MessageReceiveCompleteEvent` procedure on completion of the message retrieval. Finally the asynchronous message retrieval is started by calling the **BeginReceive** method on Line 25. I have specified that the call should time-out after 10 seconds using a new instance of the **TimeSpan** class.

Listing 8-17 makes use of the **BeginReceive** and **EndReceive** methods, but you can use the example code with only very few modifications if you want to peek instead of receive the messages from the queue (see Listing 8-18).

*Listing 8-18. Peeking Asynchronously Using **AddHandler***

```
 1 Public Sub MessagePeekCompleteEvent(ByVal vobjSource As Object, _
 2  ByVal vobjEventArgs As PeekCompletedEventArgs)
 3    Dim msgUserMan As New Message()
 4    Dim queUserMan As New MessageQueue()
 5
 6    ' Make sure we bind to the right message queue
 7    queUserMan = CType(vobjSource, MessageQueue)
 8
 9    ' End async peek
10    msgUserMan = queUserMan.EndPeek(vobjEventArgs.AsyncResult)
11 End Sub
12
13 Public Sub PeekMessagesAsync()
14    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
```

```
15    Dim msgUserMan As New Message()
16
17    ' Set up the formatter
18    queUserMan.Formatter = New XmlMessageFormatter(New Type() _
19     {GetType(String)})
20
21    ' Add an event handler
22    AddHandler queUserMan.PeekCompleted, _
23     AddressOf MessagePeekCompleteEvent
24
25    queUserMan.BeginPeek(New TimeSpan(0, 0, 10))
26 End Sub
```

Clearing Messages from the Queue

Messages can be removed from a message queue in two ways: You can remove them one by one or you can clear the whole queue in one go.

Removing a Single Message from the Queue

Removing a single message from the queue can only be done programmatically and not using either Server Explorer or Computer Management.

You can use the **Receive** method to remove messages from the queue. Although this method is generally used to retrieve messages from the queue, it also removes the message. See "Retrieving a Message" earlier in this chapter for more information.

You can also use the **ReceiveById** method to remove a message. Alternatively, you can use the **PeekById** method to find a specific message and then use **ReceiveById** to remove it. See "Picking Specific Messages from a Queue" earlier in this chapter for more information on how to use the **ReceiveById** and **PeekById** methods.

Removing All Messages from the Queue

Removing all messages from the queue can be done either manually using Server Explorer or Computer Management, or programmatically.

> **CAUTION** *Clearing all messages from a queue is an irreversible action, and once you confirm the deletion, the messages are permanently lost. So be careful when you perform this action!*

Removing All Messages Manually

If you want to manually remove all messages from a queue, you can use either the Server Explorer or Computer Management. Select the Queue Messages node (refer back to Figures 8-3 and 8-4), right-click on this node, and select All Tasks/Purge (Computer Management) or Clear Messages (Server Explorer) from the pop-up menu. Then click **Yes** or **OK** in the confirmation dialog box to clear all messages from the queue.

Removing All Messages Programmatically

If you want to clear all messages from a queue programmatically, you can use the **Purge** method for this purpose. Listing 8-19 shows you how to bind to a message queue and clear all messages from it.

Listing 8-19. Clearing All Messages from a Queue

```
1 Public Sub ClearMessageQueue()
2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3    Dim msgUserMan As Message
4
5    ' Clear all messages from queue
6    queUserMan.Purge()
7 End Sub
```

In Listing 8-19 I have used the **Purge** method of the **MessageQueue** class to clear all messages from the queue. This method works with any number of messages in the queue, meaning that it doesn't matter if the queue is empty when you call this method.

Prioritizing Messages

Every now and then it is important that a particular message is read ASAP. Normally when you send a message, it ends up at the end of the message queue, because messages are sorted by arrival time. However, since messages are first and foremost sorted by priority, you can specify a higher priority to make sure that this message is added to the message queue at the top.

You can set the priority of a message using the **Priority** method of the **Message** class. See Listing 8-20 for a code example that sends two messages to the queue, one with normal priority and one with highest priority.

Listing 8-20. Sending Messages with Different Priority

```
Public Sub SendPriorityMessages()
   Dim queUserMan As New MessageQueue(".\Private$\UserMan")
   Dim msgUserMan As New Message()

   ' Set up the formatter
   queUserMan.Formatter = New XmlMessageFormatter(New Type() _
    {GetType(String), GetType(Integer)})

   ' Create first body
   msgUserMan.Body = "First Message"
   ' Send message to queue
   queUserMan.Send(msgUserMan)

   ' Create second body
   msgUserMan.Body = "Second Message"
   ' Set priority to highest
   msgUserMan.Priority = MessagePriority.Highest
   ' Send message to queue
   queUserMan.Send(msgUserMan)
End Sub

Public Sub RetrievePriorityMessage()
   Dim queUserMan As New MessageQueue(".\Private$\UserMan")
   Dim msgUserMan As Message

   ' Set up the formatter
   queUserMan.Formatter = New XmlMessageFormatter(New Type() _
    {GetType(String)})

   ' Retrieve first message from queue
   msgUserMan = queUserMan.Receive
   ' Display the message body
   MsgBox(msgUserMan.Body.ToString)
End Sub
```

If you run the code in Listing 8-20, you will see that setting the priority of the second message to highest makes sure that it goes to the top of the message queue. The message box will display the text “Second Message”. Actually, this is only true if your message queue isn’t transactional. If it is, the text displayed will be “First Message.”

The priority of the second message is set on Line 17. When you set the **Priority** property of a **Message** object, it must be set to a member of the **MessagePriority** enum. The default is **Normal**.

Locating a Message Queue

Sometime you don't know the path to a particular queue or you just need to verify that a specific queue still exists. Let's start with the easy task: how to check if a particular queue exists (see Listing 8-21).

Listing 8-21. Checking If a Message Queue Exists

```
1 Public Function CheckQueueExists(ByVal vstrPath As String) As Boolean
2    CheckQueueExists = MessageQueue.Exists(vstrPath)
3 End Function
```

In Listing 8-21 I use the **Exists** method of the **MessageQueue** class. Because this is a public shared function, you don't actually have to instantiate a message queue object in order to use this method.

The `CheckQueueExists` procedure will return **True** or **False** depending on if the passed `vstrPath` argument matches an existing message queue.

Now this is pretty simple, because you already know the path. However, sometimes you don't know the path, just the name of the machine where message queuing is installed. See Listing 8-22 for an example of how to retrieve a list of all the private queues on a specific machine.

Listing 8-22. Retrieve All Private Queues on a Machine

```
 1 Public Sub BrowsePrivateQueues()
 2    Dim arrquePrivate() As MessageQueue = _
 3     MessageQueue.GetPrivateQueuesByMachine("USERMANPC")
 4    Dim queUserMan As MessageQueue
 5
 6    ' Display the name of all the private queues on the machine
 7       For Each queUserMan In arrquePrivate
 8          MsgBox(queUserMan.Label)
 9       Next
10 End Sub
```

In Listing 8-22 I retrieve all the message queues on the USERMANPC machine and display their label. As you can see from Line 2, all the queues are saved in an array of **MessageQueue** class instances. If you need the public queues instead, simply replace the **GetPrivateQueuesByMachine** method call with a call to the **GetPublicQueuesByMachine** method.

The public queues can also be located on the whole network instead of just one machine. There are three different methods that can be used to locate public queues network wide:

- **GetPublicQueues**
- **GetPublicQueuesByCategory**
- **GetPublicQueuesByLabel**

See Listings 8-23, 8-24, and 8-25 for code that shows you how to use these methods.

Listing 8-23. Retrieve All Public Queues on a Network

```
Public Sub BrowsePublicQueuesNetworkWide()
   Dim arrquePublic() As MessageQueue = _
    MessageQueue.GetPublicQueues()
   Dim queUserMan As MessageQueue

   ' Display the name of all the public queues on the network
   For Each queUserMan In arrquePublic
      MsgBox(queUserMan.QueueName)
   Next
End Sub
```

Listing 8-23 uses the **GetPublicQueues** method to retrieve all public queues on the network and then displays the name of each queue.

Listing 8-24 retrieves all public queues on a network by category using the **GetPublicQueuesByCategory** method. For each returned message queue in the "00000000-0000-0000-0000-000000000001" category, I display the queue name. The category being referred to is the same as the Type ID in Figure 8-2 and the **Category** property of the **MessageQueue** class. This is a way for you to group your message queues, particularly for administrative purposes.

Listing 8-24. Retrieving All Public Queues on a Network by Category

```
Public Sub BrowsePublicQueuesByCategoryNetworkWide()
   Dim arrquePublic() As MessageQueue = _
    MessageQueue.GetPublicQueuesByCategory( _
    New Guid("00000000-0000-0000-0000-000000000001"))
   Dim queUserMan As MessageQueue

   ' Display the name of all the public queues
   ' on the network within a specific category
   For Each queUserMan In arrquePublic
```

```
10       MsgBox(queUserMan.QueueName)
11    Next
12 End Sub
```

Listing 8-25 shows you how to find all the public queues on the network that have the "userman" label. The name of the machine where each queue is located is then displayed one by one.

Listing 8-25. Retrieve All Public Queues on a Network by Label

```
 1 Public Sub BrowsePublicQueuesByLabelNetworkWide()
 2    Dim arrquePublic() As MessageQueue = _
 3     MessageQueue.GetPublicQueuesByLabel("userman")
 4    Dim queUserMan As MessageQueue
 5
 6    ' Display the name of all the public queues
 7    ' on the network with a specific label
 8    For Each queUserMan In arrquePublic
 9       MsgBox(queUserMan.MachineName)
10    Next
11 End Sub
```

Removing a Message Queue

Now and then you might need to remove a message queue from a machine, and for this purpose you have several options. A message queue can be deleted either manually using Server Explorer or Computer Management or programmatically.

> **CAUTION** *Clearing a message queue is an irreversible action, and once you confirm the deletion, the queue and all its messages are permanently lost!*

Removing a Message Queue Manually

If you want to manually remove a queue, you can use either the Server Explorer or Computer Management. Select the queue node (refer back to Figures 8-1 and 8-5), right-click on this node, and select **Delete** from the pop-up menu. Then click **Yes** in the confirmation dialog box to remove the message queue and all its messages.

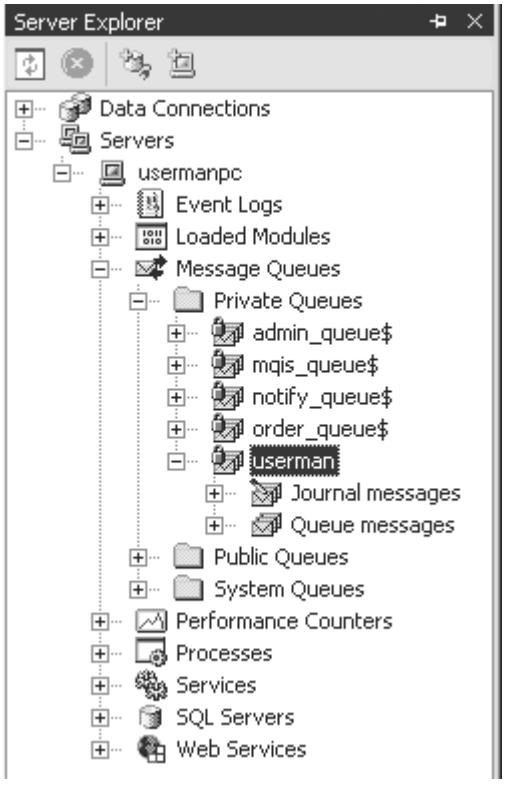

Figure 8-5. The Server Explorer with message queue selected

Removing a Message Queue Programmatically

If you want to clear all messages from a queue programmatically, you can use the **Purge** method for this purpose. Listing 8-26 shows you how to bind to a message queue and clear all messages from it.

Listing 8-26. Removing a Message Queue

```
1 Public Sub RemoveMessageQueue(ByVal vstrPath As String)
2    MessageQueue.Delete(vstrPath)
3 End Sub
```

In Listing 8-26 I have used the **Delete** method of the **MessageQueue** class to remove the queue with the `vstrPath` path. Keep in mind that if the queue doesn't exist, an exception is thrown, so I can only recommend that you don't use the example code in Listing 8-26 without being sure the queue actually exists. See how to use the **Exists** method in the section "Locating a Message Queue" later in this chapter. Alternatively, you can set up a structured error handler like the one in Listing 8-27.

Listing 8-27. Removing a Message Queue in a Safe Manner

```
1 Public Sub RemoveMessageQueueSafely(ByVal vstrPath As String)
2    Try
3       MessageQueue.Delete(vstrPath)
4    Catch objE As Exception
5       MsgBox(objE.Message)
6    End Try
7 End Sub
```

In Listing 8-27 I catch any exception being thrown when I try to remove the message queue specified using the **vstrPath** argument. I am only displaying the error message, but I am sure you can take it from there.

Making Message Queues Transactional

As with normal database access, it is possible to make your message queues transactional. This means you can send several messages as a single transaction, and then act upon the result of the transmission of these messages by determining if you should commit or rollback the messages. Whether a message queue is transactional or not is determined at creation time. You cannot change this after the queue has been created.

There are two types of transactions when it comes to message queues: internal and external. Please see the following sections for a brief description of the two types of message queue transactions.

Internal Transactions

Internal transactions refers to transactions that you manage manually or explicitly and only involves messages sent between a client and a message queue. This means that no other resources, such as database manipulation, can be part of an internal transaction and that you must explicitly begin and then commit or roll back a transaction. Internal transactions are handled by the Message Queuing's Transaction Coordinator.

Internal transactions are faster than external transactions, which are discussed next.

External Transactions

External transactions are exactly what the name suggests: external. Resources other than message queue resources are part of external transactions. This could be database access, Active Directory access, and so on. Furthermore, external

transactions are NOT handled by the Message Queuing's Transaction Coordinator, but by a coordinator such as the Microsoft Distributed Transaction Coordinator (MS DTC), and external transactions are implicit or automatic.

External transactions are slower than internal transactions. In this chapter I will only be covering internal transactions, since external transactions fall outside the scope of this book and could by themselves be the subject of an entire book.

Creating a Transactional Message Queue

If you are creating your queue programmatically, see Listing 8-4 for an example of how to make it transactional. If you want to create your transactional message queue using Server Explorer, Chapter 4 gives you the details. Of course it is also possible to use the Computer Management MMC snap-in. Open the snap-in and follow these steps:

1. Expand the Services and Applications node
2. Expand the Message Queuing node
3. Select either the Public Queues or the Private Queues node.
4. Right-click on the node and select New/Public Queue or New/Private Queue from the pop-up menu. This brings up the Queue Name dialog box, as shown in Figure 8-6.

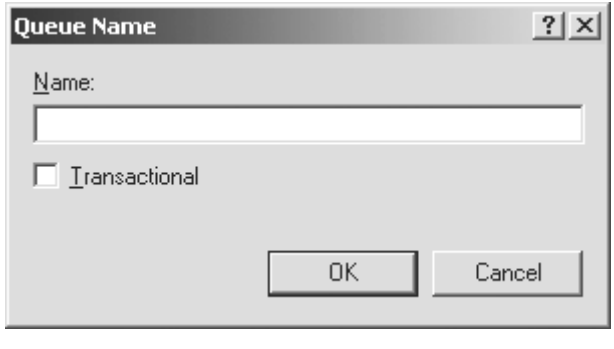

*Figure 8-6. The **Queue Name** dialog box*

Give the message queue a name by typing one in the Name text box, and make sure you check the Transactional check box before you click **OK** to create the message queue.

Starting a Transaction

Because internal transactions are explicit or manual, you must begin a transaction from code before you start sending and retrieving messages that are to be part of the transaction. Actually, the very first thing you should check is if your message queue is transactional. This can be done using the **Transactional** property, which returns a **Boolean** value indicating if the queue is transactional or not:

```
If queUserMan.Transactional Then
```

The next thing you need to do is to create an instance of the **MessageQueueTransaction** class. The constructor for this class is not overloaded, so it's created like this:

```
Dim qtrUserMan As New MessageQueueTransaction()
```

Now it's time to start the transaction, and this is done using the **Begin** method of the **MessageQueueTransaction** class, as follows:

```
qtrUserMan.Begin()
```

You may be wondering how you would reference this transaction object from the message queue object. I do understand if you are confused, but it's really not that difficult. You simply pass the transaction object when you send or retrieve a message, like this:

```
msgUserMan = queUserMan.Receive(qtrUserMan)
```

or

```
queUserMan.Send(msgUserMan, qtrUserMan)
```

As you can see from these short examples, you do as you normally would when you write your code without transactions, with the exception that you pass the transaction object when you perform an operation that needs to be part of the transaction. This has some advantages over having the transaction coupled directly with the message queue: You decide what operations are part of the transaction, therefore you can send or receive messages that are part of the transaction and send or receive other messages that are not. You can also use your transaction with more than one message queue.

Ending a Transaction

When you have started a transaction, you must also end it. That's pretty logical if you ask me. However, how the transaction should be ended is not quite so direct. If you didn't run into any problems during the operations that are part of the transaction, you would normally commit the transaction or apply the changes to the message queue(s). If you do run into a problem with any of the transactional operations, you would normally abort the transaction as soon as the problem occurs. When I refer to problems I am not only talking about exceptions that are thrown by the **MessageQueue** object, but also other operations external to the message queue operations that can make you abort the transaction.

Committing a Transaction

Committing a transaction is straightforward, and it is done using the **Commit** method of the **MessageQueue** class, as follows:

```
qtrUserMan.Commit()
```

The **Commit** method is not overloaded and it doesn't take any parameters. However, a **MessageQueueException** exception is thrown if you try to commit a transaction that hasn't been started (**Begin** method).

Aborting a Transaction

In situations where you have to abort a transaction, you can use the **Abort** method of the **MessageQueue** class, like this:

```
qtrUserMan.Abort()
```

The **Abort** method is not overloaded and it doesn't take any parameters. However, like the **Commit** method, a **MessageQueueException** exception is thrown if you try to abort a transaction that hasn't been started (**Begin** method).

Using the MessageQueueTransaction Class

I have just gone over all the methods that are important to managing a transaction, but there is one property that I haven't mentioned and that is the **Status** property. This read-only property returns a member of the **MessageQueueTransactionStatus** enum that tells you the status of

the transaction. The **Status** property can be read from the time you create your instance of the **MessageQueueTransaction** class and until the object has been destroyed. You can see the **MessageQueueTransactionStatus** enum members in Table 8-1.

*Table 8-1. Members of the **MessageQueueTransactionStatus** Enum*

MEMBER NAME	DESCRIPTION
Aborted	The transaction has been aborted. This can be done by the user with the **Abort** method.
Committed	The transaction has been committed. This is done using the **Commit** method.
Initialized	The transaction object has been instantiated, but no transaction has yet been started. In this state you should not pass the transaction object to the message queue methods.
Pending	The transaction has been started and is now pending an **Abort** or **Commit**. When the transaction is pending, you can pass the transaction object to the message queue methods.

See Listing 8-28 for an example of how to use transactions combined with a structured exception handler. The example code can only be run if you have created a private, transactional message queue named UserMan on the local machine.

Listing 8-28. Using Message Queue Transactions

```
Public Sub UseMQTransactions()
   Dim qtrUserMan As New MessageQueueTransaction()
   Dim queUserMan As New MessageQueue(".\Private$\UserMan")
   Dim msgUserMan As New Message()

   ' Set up the queue formatter
   queUserMan.Formatter = New XmlMessageFormatter(New Type() _
    {GetType(String)})

   ' Clear the message queue
   queUserMan.Purge()
   ' Start the transaction
   qtrUserMan.Begin()

   Try
      ' Create message body
      msgUserMan.Body = "First Message"
      ' Send message to queue
```

```
19        queUserMan.Send(msgUserMan, qtrUserMan)
20
21        ' Create message body
22        msgUserMan.Body = "Second Message"
23        ' Send message to queue
24        queUserMan.Send(msgUserMan)
25
26        ' Retrieve message from queue
27        msgUserMan = queUserMan.Receive()
28        ' Display message body
29        MsgBox(msgUserMan.Body)
30
31        ' Commit transaction
32        qtrUserMan.Commit()
33
34        ' Retrieve message from queue
35        msgUserMan = queUserMan.Receive()
36        ' Display message body
37        MsgBox(msgUserMan.Body)
38    Catch objE As Exception
39        ' Abort the transaction
40        qtrUserMan.Abort()
41    End Try
42 End Sub
```

In Listing 8-28 I have demonstrated how you can send messages with and without transactions to the same message queue. When you run the code, you will see the text "Second Message" displayed before the "First Message" text, because when the first message is retrieved from the queue on Line 27, the first message has not yet been committed to the queue. The transaction is then committed and the first message sent to the queue.

Looking at System-Generated Queues

So far I have only been looking at user-created queues, but there is another group of queues that needs attention: system-generated queues. These queues are maintained by **Message Queuing**. The following queues are system queues:

- Dead-letter messages
- Journal messages
- Transactional dead-letter messages

The two dead-letter message queues, seen in the Server Explorer in Figure 8-7 on the System Queues node, are used for storing messages that cannot be delivered. One queue holds the nontransactional messages and the other one holds the transactional ones. See the next section for an explanation of the Journal messages queues.

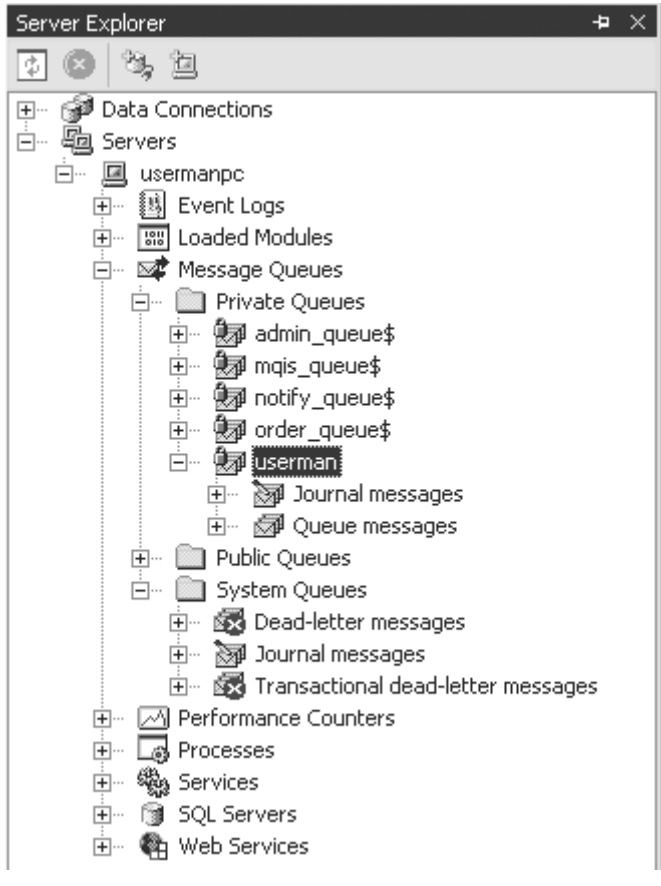

Figure 8-7. The system and journal queues

Using Journal Storage

Journal storage is a great facility for keeping copies of messages that are sent to or removed from a message queue. It can be very helpful if for some reason you need to resend a message. The message delivery might have failed, and you receive a negative acknowledgement. Because an acknowledgement does not contain the message body, you cannot use this message to resend the original

message. However, you can use it to find a copy of the original message in the Journal messages queue.

Every message queuing client (machine) has a global journal queue called the system journal. The system journal stores copies of all messages sent from the message queuing client. This is true whether the message is delivered or not.

Besides the system journal, all message queues have their own journal queue. The journal queue associated with each message queue stores copies of all messages removed from the message queue, but only if journaling is enabled on the message queue. See the next section, "Enabling Journaling on a Message Queue," for more information on how to enable journaling on a message queue.

The last way to use journal storage is to programmatically enable journaling on a per-message basis. If you do this, copies of messages sent from your system will be saved in the system journal on the machine that sends the message.

Enabling Journaling on a Message Queue

Journaling can be enabled on a message queue either manually or programmatically. If you want to do it manually, you can do it from the Computer Management MMC snap-in. In Figure 8-2 you can see the Properties dialog box for an existing queue or a queue that is about to be created. If you want to enable journaling you should check the Enabled check box in the Journal group and click on Apply.

You can also programmatically enable journaling on an existing queue by setting the **UseJournalQueue** property of an existing message queue to **True**, like this:

```
Dim queUserMan As New MessageQueue(".\Private$\UserMan")

queUserMan.UseJournalQueue = True
```

Enabling Journaling on a Per-Message Basis

It is also possible to use journaling on a per-message basis, and this is done programmatically, by setting the **UseJournalQueue** property of an existing message to **True** before sending it to the queue, as shown in Listing 8-29.

Listing 8-29. Enabling Message Journaling

```
 1 Public Sub EnableMessageJournaling()
 2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
 3    Dim msgUserMan As New Message()
 4
 5    ' Set up the formatter
 6    queUserMan.Formatter = New XmlMessageFormatter(New Type() _
 7     {GetType(String)})
 8
 9    ' Create message body
10    msgUserMan.Body = "Message"
11    ' Enable message journaling
12    msgUserMan.UseJournalQueue = True
13    ' Send message to queue
14    queUserMan.Send(msgUserMan)
15 End Sub
```

In Listing 8-29 I enable message journaling on the `msgUserMan` message on Line 12, just before I send it to the queue. This saves a copy of the message in the system journal on the local machine. Well, actually it won't be saved until the message is sent to the queue (Line 14).

Retrieving a Message from Journal Storage

Now, enabling journaling is easy enough, but how do you actually retrieve the messages from the journal when needed? Well, you can access these journal queues and other system queues by specifying the correct path to the queue. For example, the following code accesses the journal storage for the private message queue UserMan on the local machine:

```
Dim queUserMan As New MessageQueue(".\Private$\UserMan\Journal$")
```

Securing Your Message Queuing

So far in this chapter you have been shown how to create private and public queues, both transactional and nontransactional ones, and you have seen how you send and receive messages to and from a queue, including peeking at messages without removing them from the queue. However, I have yet to mention message queuing security, which is what I am going to cover in this section.

Message Queuing uses built-in features of the Windows OS for securing messaging. This includes the following:

- Authentication
- Encryption
- Access control
- Auditing

Using Authentication

Authentication is the process by which a message's integrity is ensured and the sender of a message can be verified. This can be achieved by setting the message queue's **Authenticate** property to **True**. The default value for this property is **False**, meaning no authentication is required. When this property is set to **True**, the queue will only accept authenticated messages—nonauthenticated messages are rejected. In other words, the message queue on the server requires messages to be authenticated, not just the message queue object you use to set the **Authenticate** property. Basically this means that when you set the property, you affect all other message queue objects that are working on the same message queue. You can programmatically enable authentication as it's done in Listing 8-30.

Listing 8-30. Enable Authentication on a Messsage Queue

```
1 Public Sub EnableQueueAuthentication()
2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3    Dim msgUserMan As New Message()
4
5    ' Enable queue authentication
6    queUserMan.Authenticate = True
7 End Sub
```

You can also request authentication by setting this property from the Computer Management MMC snap-in. See Figure 8-2 and the section "Displaying or Changing the Properties of Your Message Queue" for more information on how to set the properties of an existing message queue. In Figure 8-2 you can see the Authenticated check box, which you need to enable before clicking OK to request that messages on this queue are authenticated.

If a message is not authenticated, it is rejected and therefore lost. However, you can specify that the rejected message should be placed in the dead-letter queue, as shown in Listing 8-31.

Listing 8-31. Rejecting a Nonauthenticated Message

```
Public Sub RejectNonauthenticatedMessage()
    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
    Dim msgUserMan As New Message()

    ' Enable queue authentication
    queUserMan.Authenticate = True
    ' Set up the queue formatter
    queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})

    ' Create message body
    msgUserMan.Body = "Message Body"
    ' Make sure a rejected message is placed in the dead-letter queue
    msgUserMan.UseDeadLetterQueue = True

    ' Send message to queue
    queUserMan.Send(msgUserMan)
```

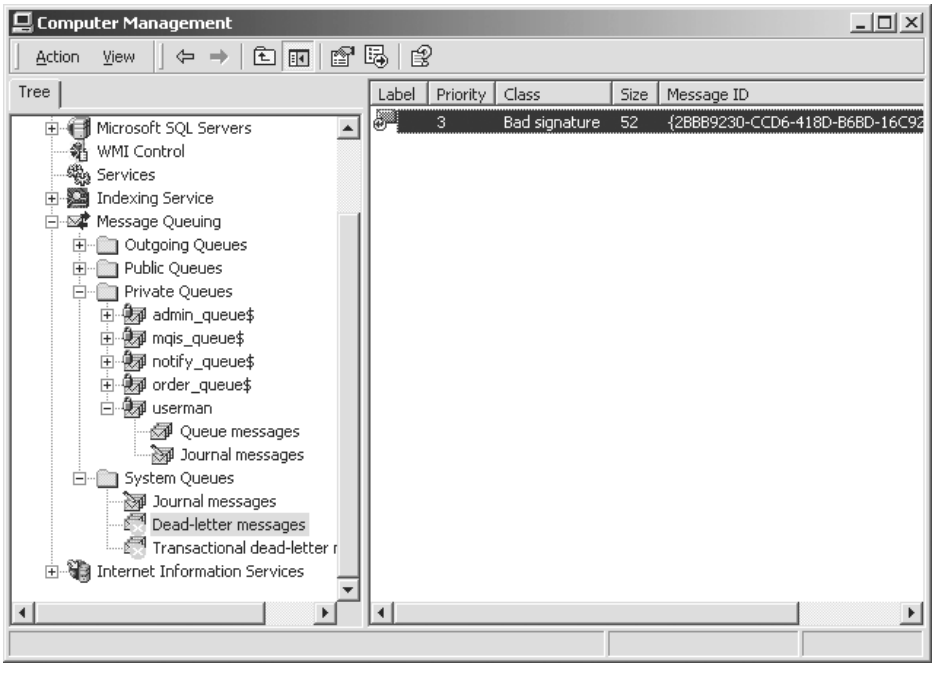

Figure 8-8. A rejected nonauthenticated message in the dead-letter queue

```
17 End Sub
```

In Listing 8-31 the **UseDeadLetterQueue** property is set to **True**, meaning that if the message is rejected, it is placed in the dead-letter queue (see Figure 8-8).

In Figure 8-8 you can see that the message in the dead-letter queue has been rejected because of a bad signature. Another way of dealing with rejected messages is to request notification of the message rejection. Setting the **AcknowledgeType** and **AdministrationQueue** properties does this (see Listing 8-32).

Listing 8-32. Receiving Rejection Notification in the Admin Queue

```
 1 Public Sub PlaceNonauthenticatedMessageInAdminQueue()
 2    Dim queUserManAdmin As New MessageQueue(".\Private$\UserManAdmin")
 3    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
 4    Dim msgUserMan As New Message()
 5
 6    ' Enable queue authentication
 7    queUserMan.Authenticate = True
 8    ' Set up the queue formatter
 9    queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})
10
11    ' Create message body
12    msgUserMan.Body = "Message Body"
13    ' Make sure a rejected message is placed in the admin queue
14    msgUserMan.AdministrationQueue = queUserManAdmin
15    ' These types of rejected messages
16    msgUserMan.AcknowledgeType = AcknowledgeTypes.NotAcknowledgeReachQueue
17
18    ' Send message to queue
19    queUserMan.Send(msgUserMan)
20 End Sub
```

In Listing 8-32 I have set up the private queue UserManAdmin to receive notification of rejection. Please note that you need to have created this message queue as a nontransactional queue before you run the example code.

As is the case with the code in Listing 8-31, the code in Listing 8-32 rejects the messages that you send. That seems okay, because you requested that all messages should be authenticated, but you "forgot" to authenticate your message.

Setting the **AttachSenderId** property to **True** does this, because it results in Message Queuing setting the **SenderId** property. However, this is not enough, because you also need to set the **UseAuthentication** property to **True**. This

ensures the message is digitally signed before it is sent to the queue. Likewise, the digital signature that Message Queuing has assigned is used for authenticating the message when it is received. Listing 8-33 shows you how to send an authenticated message.

Listing 8-33. Sending an Authenticated Message

```
1 Public Sub AcceptAuthenticatedMessage()
2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3    Dim msgUserMan As New Message()
4
5    ' Enable queue authentication
6    queUserMan.Authenticate = True
7    ' Set up the queue formatter
8    queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})
9
10   ' Make sure a rejected message is placed in the dead-letter queue
11   msgUserMan.UseDeadLetterQueue = True
12   ' Make sure that message queuing attaches the sender id and
13   ' is digitally signed before it is sent
14   msgUserMan.UseAuthentication = True
15   msgUserMan.AttachSenderId = True
16   ' Create message body
17   msgUserMan.Body = "Message Body"
18
19   ' Send message to queue
20   queUserMan.Send(msgUserMan)
21 End Sub
```

In Listing 8-33 I set the **UseAuthentication** and **AttachSenderId** properties of the **Message** object to **True** in order to ensure that the message is digitally signed and can be authenticated by Message Queuing. In this situation the message queue, unlike in the examples in Listings 8-31 and 8-32, will not reject the message, even if it only accepts authenticated messages. If authentication has already been set up on the message queue, you can leave out Lines 5 and 6.

Now it's obviously good to know if a message has been authenticated or not, but you don't actually have to check anything to find out. If you have set up authentication on the message queue, all messages in the queue have been authenticated when they arrived in the queue, meaning you can trust any message you receive from the queue.

Using Encryption

Encryption is another way of securing messages sent between message queuing computers. With encryption, anyone trying to spy on the traffic on the network between the message queuing computers will receive encrypted messages. Now, while someone might be able to decrypt your messages, encryption certainly makes it harder to obtain sensitive information.

There is some overhead involved when you encrypt your messages at the sending end and decrypt them at the receiving end, but if your network is public and you are sending sensitive information, you should definitely consider using encrypted messages.

As is the case with authentication, encryption requirements can be applied to the queue, meaning that all nonencrypted messages will be rejected by the queue. Setting the **EncryptionRequired** property of the **MessageQueue** object to **EncryptionRequired.Body**, as in Listing 8-34, does this.

Listing 8-34. Making Sure Nonencrypted Messages Are Rejected by the Queue

```
1 Public Sub EnableRequireBodyEncryption()
2    Dim queUserMan As New MessageQueue(".\Private$\UserMan")
3    Dim msgUserMan As New Message()
4
5    ' Enable body encryption requirement
6    queUserMan.EncryptionRequired = EncryptionRequired.Body
7 End Sub
```

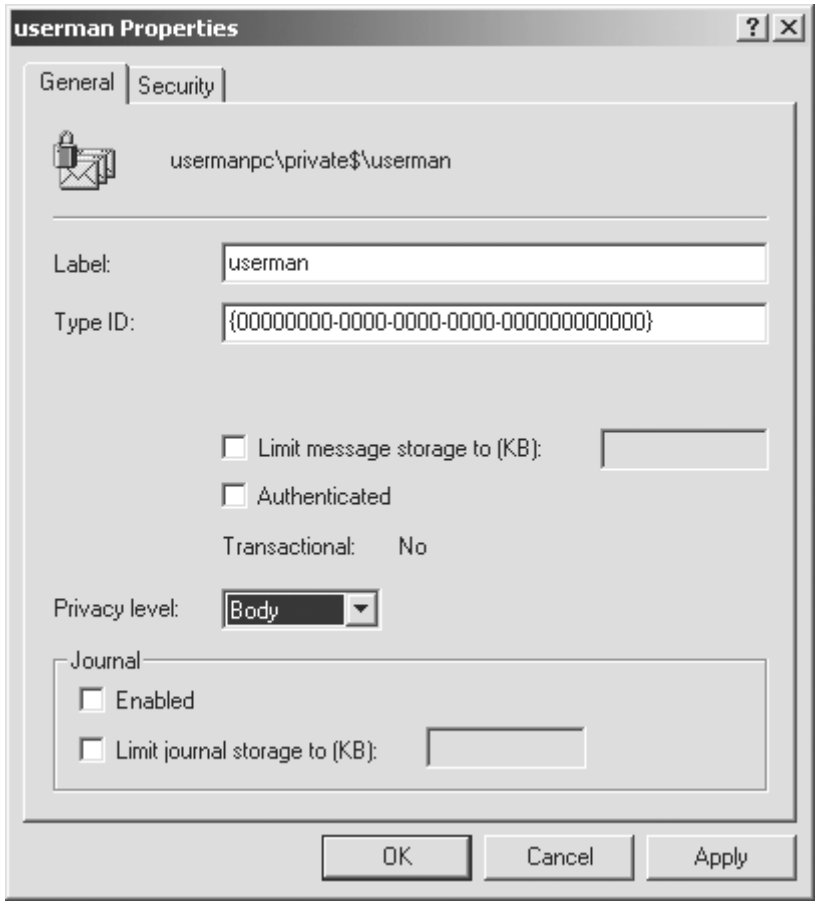

Figure 8-9. The privacy level set to Body

In Listing 8-34 I set the **EncryptionRequired** property of the **MessageQueue** object to **EncryptionRequired.Body**. As a result, the body of any message sent to the queue must be encrypted; otherwise it is rejected. Message Queue encryption is also called privacy, meaning a private message, not to be confused with a private message queue, is an encrypted message. Instead of setting the encryption property of the message queue programmatically, you can also set it from the Computer Management MMC snap-in (see Figure 8-9).

In Figure 8-9 Body has been selected from the Privacy level drop-down list, which has the same results as running the example code in Listing 8-34. Please note that I have unchecked the Authentication check box in this example. This doesn't mean that you can't use encryption and authentication at the same time; I have simply done so to simplify the code in the following listings.

Unlike with authentication, it is possible to use encryption even if the message queue doesn't require it, but only if the privacy level is set to Optional. This

can be done from the Computer Management MMC snap-in or programmatically, as follows:

```
' Set message body encryption to optional
queUserMan.EncryptionRequired = EncryptionRequired.Optional
```

If you specify Optional as the privacy level, you can send encrypted and nonencrypted messages to the message queue. However, if you set the privacy level to None, you can only send nonencrypted messages to the queue. If you send an encrypted message to a queue with a privacy level of None, the message is rejected.

The example in Listing 8-35 shows you how to send and receive an encrypted message.

Listing 8-35. Sending and Receiving Encrypted Messages

```
Public Sub SendAndReceiveEncryptedMessage()
   Dim queUserMan As New MessageQueue(".\Private$\UserMan")
   Dim msgUserMan As New Message()

   ' Require message body encryption
   queUserMan.EncryptionRequired = EncryptionRequired.Body
   ' Set up the queue formatter
   queUserMan.Formatter = New XmlMessageFormatter(New Type() {GetType(String)})

   ' Make sure that message is encrypted before it is sent
   msgUserMan.UseEncryption = True
   ' Create message body
   msgUserMan.Body = "Message Body"

   ' Send message to queue
   queUserMan.Send(msgUserMan)

   ' Retrieve message from queue
   msgUserMan = queUserMan.Receive()
   ' Show decrypted message body
   MsgBox(msgUserMan.Body.ToString)
End Sub
```

In Listing 8-35 I encrypt a message, send it to the queue, and receive it from the queue. During the transport to and from the queue, the message body is encrypted, but the message is automatically decrypted by the receiving **MessageQueue** object. This means decryption of an encrypted message is always performed automatically when the message is received.

Using Access Control

Controlling access to the message queue is probably the best way to secure your messages. As with most other Windows operations, such as creating new users, the reading and writing of messages can be access controlled. Access control happens when a user tries to perform an operation, such as reading a message from a queue. Each user under Windows 2000 or Windows NT has an Access Control List (ACL), which contains all the operations the user can perform. The ACL is checked when the user tries to read a message. If the user has read access, the user can read the message from the queue. However, if the user isn't allowed to read from the queue, the read is disallowed.

Access control can be applied at the message queue level or even at the message level, but it can also be applied at the Message Queuing level, meaning that all message queues in the Active Directory will abide by the permissions you set. Take a look at Figure 8-10, where you can see the Access Control List for the UserMan user of the UserMan domain. The permissions shown are for the UserMan private message queue.

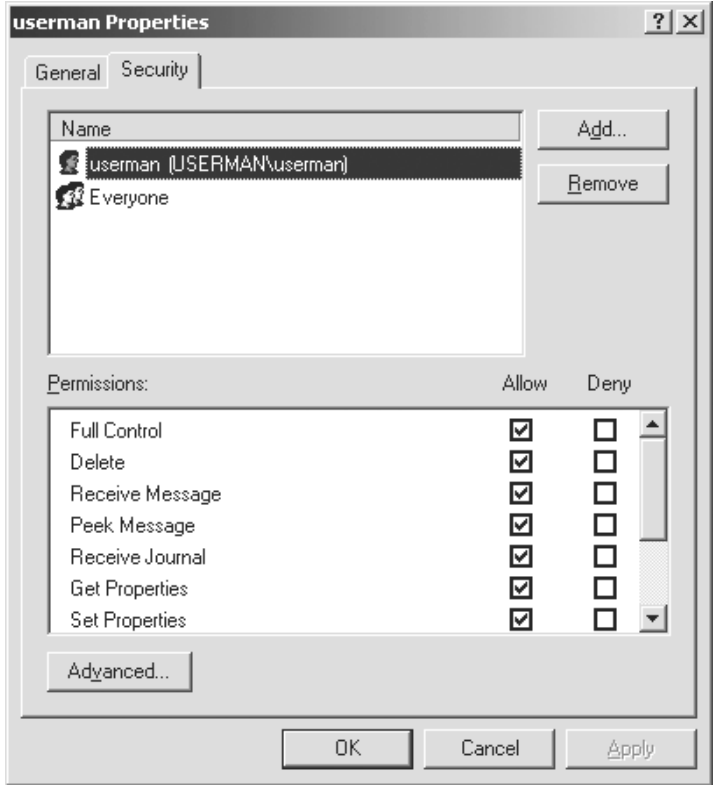

Figure 8-10. The user Properties dialog box with Security tab selected

You can bring up the ACL by opening the Computer Management MMC snap-in, selecting the private UserMan queue, right-clicking on the queue, and selecting Properties from the pop-up menu. This brings up the Properties dialog box, in which you must click Security to get to the ACL (see Figure 8-10).

As you can see from the ACL in Figure 8-10, there are permissions for just about any thing you can do with the message queue, such as write, read, and peek at messages. Next to the operation there are two check boxes, one for allowing the operation and one for denying the operation. All you have to do is select the user or group you want to set the permissions for in the list box in top part of the dialog box. If the user or group is not listed, you can add it by clicking the Add button. If you need to remove a user or group, you can click the Delete button.

You need to be especially careful when you set the permissions for groups, because as always with ACLs in Windows, the more restrictive permission takes precedence. For instance, if you set the Delete operation to Allow for the UserMan user and then set the same permission for the Everyone group to Deny, the UserMan user is denied the right to delete the message queue. All users in the domain are part of the Everyone group, so you need to be especially careful with this option. Setting group and user permissions are really outside the scope of this book, so if you need more information on how ACLs work and permissions are set with respect to users and groups, you should look in the documentation that comes with your Windows OS.

Using the SetPermissions Method

Instead of setting the user permissions from Computer Management, you can also perform this task programmatically using the **SetPermissions** method of your **MessageQueue** object. Listing 8-36 shows you how to do it.

Listing 8-36. Setting User Permissions Programmatically

```
Public Sub SetUserPermissions()
   Dim queUserMan As New MessageQueue(".\Private$\UserMan")
   Dim msgUserMan As New Message()
   Dim aclUserMan As New AccessControlList()

   ' Give UserMan user full control over private UserMan queue
   queUserMan.SetPermissions("userman", MessageQueueAccessRights.FullControl)
   ' Give UserMan user full control over private UserMan queue
   queUserMan.SetPermissions(New MessageQueueAccessControlEntry( _
    New Trustee("userman"), MessageQueueAccessRights.FullControl))
   ' Deny UserMan deleting the private UserMan queue
   queUserMan.SetPermissions("userman", MessageQueueAccessRights.DeleteQueue, _
```

```
13     AccessControlEntryType.Deny)
14    ' Deny UserMan all access rights on the private UserMan queue
15    aclUserMan.Add(New AccessControlEntry(New Trustee("userman"), _
16      GenericAccessRights.All, StandardAccessRights.All,
        AccessControlEntryType.Deny))
17    queUserMan.SetPermissions(aclUserMan)
18 End Sub
```

In Listing 8-36 I have used all four overloads of the **SetPermissions** method. The first two (Lines 7 and 9) take the name of the user or group and the rights to assign to this user or group. They are basically the same and can only be used to assign rights to a user or group, not revoke rights. The next overloaded version of the method (Line 12) can be used to allow, deny, revoke, or set a specific permission for a particular user or group. The last version of the method (Lines 15 through 17) can be used to allow, deny, revoke, or set generic and standard access rights for a user or group. Actually, the last overloads can be used for changing permissions for several users and/or groups in one go. You simply need to add as many Access Control Entries (ACE) as necessary to the ACL (Line 15), before you set the permission using the ACL (Line 17).

I have only constructed the example code in Listing 8-36 for you to see how all four of the overloaded versions of the **SetPermissions** method can be used. You shouldn't be using them at the same time. If a conflict occurs between the rights you assign or revoke, only the last rights assigned or revoked will count.

Using Auditing

In the context of this chapter, auditing is used for logging access to a message queue. This means allows you to check the Event Log to see who has been accessing it, or even better, who has been trying to access it and been denied the access. Auditing really falls beyond the scope of this book, as it is a generic tool used for logging access to any kind of service or object in the Windows OS.

You can set up auditing from the Computer Management MMC snap-in. Select the message queue you want to audit or indeed all of Message Queuing by selecting the node by this name. Right-click on the selected node and select Properties from the pop-up menu. This brings up the Properties dialog box, in which you should select the Security tab. On the Security tab click the Advanced button. This brings up the Access Control Settings dialog box. Select the Auditing tab to bring up the dialog box shown in Figure 8-11. Normally there are no Auditing Entries on the list, but in Figure 8-11 I have added an entry that logs when any user (Everyone group) fails to write a message to the message queue.

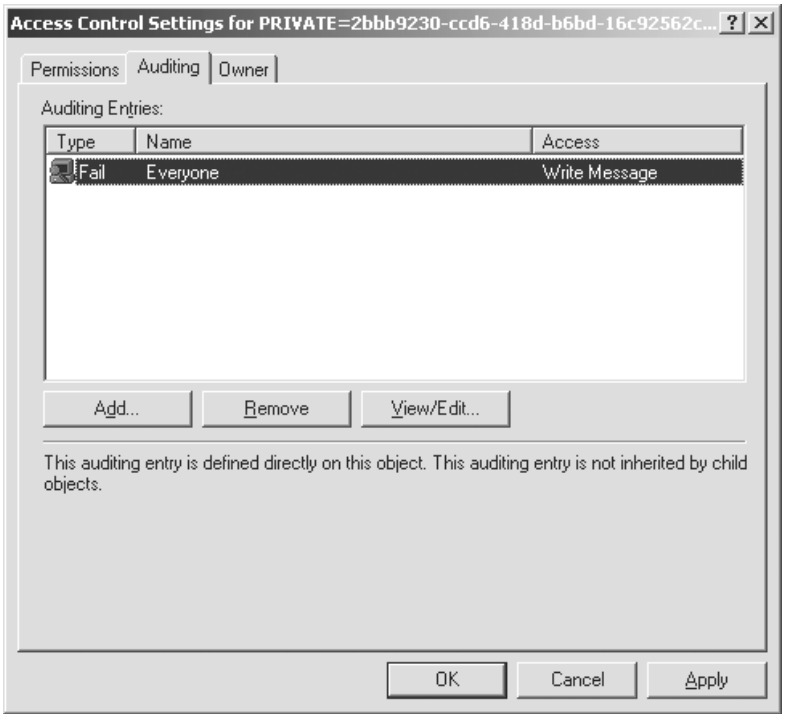

Figure 8-11. The Access Control Settings dialog box

There are two types of audits: Success and Failure. Try playing around with these audits and don't forget to check the entries added to Event Log (the Application Log). You can view the Event Log using the Event Viewer.

If you want further information about auditing or just Message Queuing security in general, I suggest you read the documentation for Windows 2000.

Summary

This chapter introduced you to the concept of connectionless programming using message queues. I showed you how to create message queues from the Server Explorer and the Computer Management MMC snap-in, and how to do it programmatically.

I discussed how you can locate message queues on the network; how message queues work with transactions; why you should use a message queue and not a database table; how the messages are ordered or sorted in a queue; and finally how you work with the various properties and methods of both the **MessageQueue** and **Message** classes.

In the next chapter I will show you how to wrap your data access functionality in classes. I will be covering OOP fundamentals such as polymorphism very briefly and show you how to use them in your UserMan application.