

10

Queued Components

When most people think of Wall Street traders, they imagine a frenetic trading floor, countless arms outstretched, waving tickets under a green-dotted, scrolling marquee. But not all trading is like that. Take bond trading, for example. Bonds and bond trading are commonly referred to as Fixed Income (FI) trading on Wall Street. The FI trading environment is more reserved than you might expect. In many firms, FI traders sit quite closely to each other at long desks laden with electronic news feeds, telecommunications consoles, and something called a *Hoot and Holler* (used for hollering announcements throughout the office).

These days, proprietary monitors and news feeds are giving way to Windows desktops, and there is growing acceptance of Windows-based, message-oriented information systems. But at one investment bank I worked with, a major problem occurred regarding an inherent limitation of COM when it applies to messages. With this problem I introduce the concept of *asynchronous messaging* with Microsoft Message Queue (MSMQ), and ultimately, our discussion works its way up the evolutionary timeline to COM+ Queued Components.

In this chapter, we take a look at how to call methods asynchronously using Queued Components. As you'll see, Queued Components rely on MSMQ, and this chapter also includes as a detailed discussion of this their inter-relationship.

The Mystery of the Hanging News Feeder

As you can imagine, the prompt receipt and processing of a high volume of messages is critical for a trading system. Price corrections, small and large, need to be packaged, processed, and presented to the trader immediately through some sort of user interface.

One firm I consulted for developed a proprietary, centralized system to deliver messages to a number of clients. The system was based on traditional COM (not COM+; this was a couple years ago) and used a mechanism called *connection points.* Connection points are discussed in greater detail in Chapter 11, "Events," but in the simplest terms, connection points give rise to a scenario where a server COM object holds an interface to its client. This enables the server to call methods on the client's interface (a client can be or create a COM object for the purpose of receiving notification) or, to use other terminology, trigger events on the client.

My client's notification server was found to hang indefinitely on occasion and required someone to kill the process. The client asked me to troubleshoot the system, but I had a pretty good guess as to what was going on without looking at the code. I asked if any kind of modal dialog box was popping up in the event routine of their client application. A modal dialog box, I further explained, is one that suspends any further code execution until it is dismissed. I received a blank look, which was in itself revealing.

Basically, any COM method call is *synchronous.* This is because COM is based on traditional Distributed Computing Environment (DCE), an industry standard style Remote Procedure Calls (RPC) where a client application blocks until the server-side implementation of a function completes. In more specific terms, an RPC client thread always blocks—that is, freezes—while it is waiting for the server-side RPC thread to run the function implementation and return. If the server takes an arbitrarily long time to complete the function, the client waits just as long. If you turn the tables, you find that the same holds true for connection point clients. If the client-side event hangs somewhere in its implementation code after being called by the server, the server thread that called the event hangs as well. If the server is single threaded, the server is now blocked on the method call to the client, giving the impression that it is hung.

This, in fact, is exactly what was happening with my client's news feed server. The author of the VB client application had implemented a certain event such that it popped up a modal dialog box in response to certain messages. If, however, the trader did not dismiss the form, the server blocked on the call and ceased to deliver any further messages to this or any other client. So, by replacing the modal dialog box with a modeless one, the immediate problem was resolved.

Clearly, an architecture that can be held hostage by one rogue (or badly-written) client is not going to cut it in the enterprise. If, however, your remote method calls are synchronous, there is no convenient way of avoiding the hanging client problem. But what if your method calls could somehow be *asynchronous*? In other words, what if the

client did not need to block on a method call, but could return the moment after it called it? Microsoft offers two COM-based mechanisms toward this end:

- Queued Components (QC)
- Asynchronous COM

QC by far the easier of the two to use, is a COM+ core service, and does not suffer significant limitations of Asynchronous COM (mainly, COM+-configured components cannot use it!). This chapter, therefore, focuses on QC, but I discuss Asynchronous COM at the end.

For an asynchronous architecture to work, you need to forego the use of return values and output parameters in methods you want to call asynchronously. After all, if the client returns from a method call before the method executes, there is certainly no way output parameters can contain valid values.

Assuming you are willing to give up any output or return parameters, there remains the problem of delivery. Often, the return value of a function is used to indicate its success or failure, but if a client returns immediately after invoking a remote method and cannot rely on any kind of return values, it cannot be certain that the server even received the client's invocation. It is awkward to force the developer to write his own acknowledgement scheme, but it is just as awkward for the system to notify the client of a failed method call—how and when should it do so? Many different schemes have arisen to address failed asynchronous calls. Some schemes demand that the server-side implementation of the method be idempotent such that the system can call it multiple times without harm if needed. (We discuss idempotence in Chapter 9, "Compensating Resource Managers," in the section "Handling Recovery.") Other schemes simply ignore the problem altogether and say, "Call at your own risk—no guarantees of delivery."

Providing asynchronous method invocation services that also guarantee delivery is a tricky business, and Microsoft has taken up the challenge in small steps over the years, which has culminated in COM+ Queued Components. The first step, on which others are built, was introduced in November 1997 (as part of the NT 4 Option Pack), and it was called *Microsoft Message Queue (MSMQ)*.

Introducing Microsoft Message Queue

Simply put, MSMQ is a service that allows applications to send messages to one another asynchronously and can guarantee the successful delivery (or notification of the failure to deliver) of a given message. A *message* can be anything the developer wants it to be—arrays of bytes, strings, numbers, and so on—but it is important to realize that an asynchronous message is not the same thing as an asynchronous method call. MSMQ has no provision or charter to package the arguments of a method call and ferry it asynchronously from client and server. Rather, MSMQ is a relatively

simple C API and a collection of COM objects (presumably based on the C API) that developers can use to send information, again in the form of a message, from point A to point B asynchronously with a guarantee of delivery.

MSMQ as Middleware

MSMQ is itself an NT service that runs on Windows 2000. It is also a proper Resource Manager (RM) that allows messages to be sent in the context of a transaction. In fact, many developers classify MSMQ as *middleware* because it is not truly an innate service of the operating system, nor is it an application in its own right. It is a middle-tier message delivery mechanism that developers can leverage to send information from one application to another. Granted, developers can use TCP/IP sockets, named pipes, RPC, and a number of other established Inter-Process Communication (IPC) mechanisms, but MSMQ has the following advantages:

- Guaranteed message delivery (if requested).
- Asynchronous message delivery. The sender and recipient do not need to be running at the same time.
- Message logging services. A *journal* of all messages sent can be recorded and stored for later review.
- Security. Access permission can be set on queues (we talk about queues in the next section) and the message bodies themselves can automatically be encrypted.

I could add a number of additional bullets, but MSMQ is probably simplest to understand if you take a look at the code for the simplest possible MSMQ sender and receiver.

MSMQ Sender Implementation

MSMQ ships with a small collection of COM objects that wrap the underlying C API in a more friendly, language-independent abstraction. An example of a VB client that uses these objects follows in Listing 10.1.

Listing 10.1 **A Simple VB Client that Sends a Message to a Queue**

```
Dim qInfo As New MSMQQueueInfo
Dim msgSend As New MSMQMessage
Dim qDest As MSMQQueue

msgSend.Body = "Hello World"

qInfo.PathName = " darjeeling \private$\MSMQTest"
qInfo.Label = "Test Queue"
qInfo.Create
```

```
Set qDest = qInfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

msgSend.Label = "Test Message"
msgSend.Send qDest

qDest.Close
```

Creating and Opening a Queue

You might notice that *queues* seem to play a large part in MSMQ code. Intuitively, you might imagine that all messages must travel through a queue to reach their destination; it is more precise to realize that the queue is the destination. Much like a database table, a queue is a persistent repository for messages that can be read from any MSMQ recipient in an FIFO (First-In-First-Out) fashion. You will see such a recipient in the section "MSMQ Receiver Implementation," but let's continue our queue discussion for the moment.

There can be as many queues as you care to create, and they can be created by a sender, receiver, or any application. In Listing 10.1, it is the sender who creates a queue called `MSMQTest`. The following lines describe the location of, and then create, a queue:

```
Dim qInfo As New MSMQQueueInfo
qInfo.PathName = "darjeeling\private$\MSMQTest"
qInfo.Create
```

It is important to realize that a queue is not a connection nor is it an asynchronous path between two points. Queues do not have a sense of direction relative to the clients and servers that use them in the same way (for example, as TCP/IP sockets do). One end of the queue can be used by any client anywhere to *push* messages into the queue, and the other end can be read by any server anywhere to *pop* messages off the queue and interpret them in some way. A message queue is simply a persistent entity that acts as a repository and FIFO-style distribution center for messages. It is probably best to think of MSMQ as the RM it is. The MSMQ RM transactionally handles its own proprietary data store and provides methods allowing for the storing and retrieving of your messages to that store.

Now, back to queues. After a queue is created, it exists completely independently of the entity that created it and can (security permitting) be used by any client and server pair at any time for any purpose. If you want to see message queues that have been created and are currently in existence, you can open the Computer Management Microsoft Management Console (MMC) snap-in and look under the Message Queuing tree-view item, as shown in Figure 10.1.

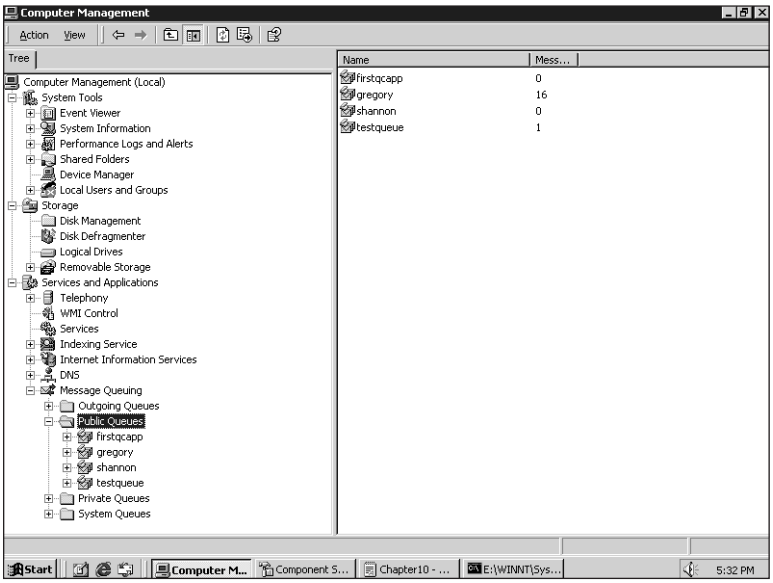


Figure 10.1 A list of message queues currently in existence, including the queue for a COM+ example application we will be using in this chapter, `FirstQCApp`.

Sending the Message

After a queue has been created, it can be opened for reading or writing. A sender of a message typically opens a queue for writing (though technically, it can open and read from the same queue if it wants) with the following line:

```
Set qDest = qInfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)
```

The arguments used in the preceding method call are typical, and we'll hold off on a more detailed discussion of these arguments; interested readers should consult Appendix E, "Queue Moniker Parameters." For now, note that the `Open` method of an `MSMQQueueInfo` object returns an `MSMQQueue` object. The `MSMQQueue` object represents a connection to a queue and is used to send and retrieve messages from the queue.

Not surprisingly, the message itself is represented by an object, one of type `MSMQMessage`. The following code demonstrates how to set the simplest of an `MSMQMessage` object's properties and then send the message:

```
msgSend.Body = "Hello World"
msgSend.Label = "Test Message"
msgSend.Send qDest
```

In this example, you are setting the `Body` property equal to the string `"Hello World"`. The `Body` property is really a variant, and so it can hold any type of Visual Basic data type or array. The question arises as to whether you can set the `Body` of a message to an instance of an object and send that object through a queue. The short answer is yes, provided the object supports the `IPersistStream` interface. We discuss this in more detail in the subsequent section "Persistence: Passing Objects Through Messages."

You might notice that the `MSMQMessage` object has a `Send()` method that takes the current queue connection (represented by the `MSMQQueue` object returned from an `MSMQQueueInfo`'s `Open` method) as an argument. Thus the following line of code sends the message represented by the `myMessage` object to the `myQueue` queue:

```
myMessage.Send myQueue
```

When this method returns, the message is sent. All the sender needs to do is close its connection to the queue. You do this by using the following code:

```
myQueue.Close
```

Remember, the preceding line does not destroy the queue. When created, a message queue stands on its own and can outlive its creating object or process. Queues must be explicitly deleted as shown in the next section.

MSMQ Receiver Implementation

The MSMQ receiving application is, as you might expect, the mirror image of the sending application (see Listing 10.2).

Listing 10.2 **A Simple VB Client that Receives a Message from the Queue**

```
Dim qInfo As New MSMQQueueInfo
Dim msgRec As MSMQMessage
Dim qSource As MSMQQueue

qInfo.PathName = "darjeeling\private$\MSMQTest"

Set qSource = qInfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
Set msgRec = qSource.Receive

MsgBox msgRec.Body
qSource.Close
```

Receiving a Message

All the basic objects, `MSMQQueueInfo`, `MSMQQueueMessage`, and `MSMQQueue`, are used in a similar fashion to the sender—the location of a queue is specified, a queue is opened, and a message is popped from the queue with the following line:

```
Set msgRec = qSource.Receive
```

By default, the `Receive` method is a blocking call. This means that if there are no messages in the queue, the calling thread freezes when making this method call. You can, however, specify a timeout period using one of `Receive`'s optional arguments, or you can ask MSMQ to notify you when a message comes in. In the C API, you can enable this feature by giving a callback function to MSMQ; in VB, you can implement the `MSMQEvent` interface in a form or class and pass it as an argument to the `EnableNotification()` method of the `MSMQQueue` object. See the companion source (`www.newrider.com/complus`) for greater detail.

Closing and Deleting a Queue

When a receiving application no longer wants to process messages, like the sender, it can simply call the following:

```
qSource.Close
```

Of course, the preceding line does not delete the queue, it only closes the receiver's connection to it. If you want to delete the queue, do so with the following line:

```
qInfo.Delete
```

This assumes, of course, that the caller's `myQueueInfo` is describing an existing queue and that the caller has the permissions to delete it. The ability to delete a queue, as well as privileges to send, receive, and peek (that is, look but don't remove) messages can be set on a per-queue basis.

From MSMQ to COM+ Queued Components

In this chapter, I flip quite a bit between MSMQ and QC, interpolating the two technologies until they ultimately converge. So, let's take a first look at COM+ QC.

Asynchronous Methods Calls with Queued Components

If a developer wants the methods of a specific interface to be asynchronous, it is simple to do with QC. Go to the Component Services snap-in, select the Properties and then the Queuing tab for an interface as shown in Figure 10.2. Then, click the Queued check box.

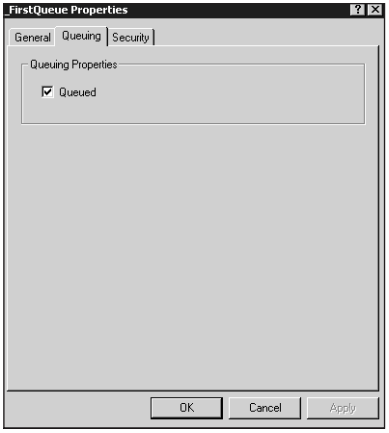


Figure 10.2 The QC dialog box.

Or, using the COM+ administrative objects, the code shown in Listing 10.3 does just as well.

Listing 10.3 **Marking an Application and Interface Queueable Using the COM+ *Admin* Objects**

```
'Error check omitted for brevity.  Consult the book's
'accompanying source code for fully-complete example

Dim Catalog As New COMAdminCatalog
Dim Application As COMAdminCatalogObject
Dim Component As COMAdminCatalogObject
Dim Interface As COMAdminCatalogObject

Dim colApp As COMAdminCatalogCollection  'Collection of Applications
Dim colComp As COMAdminCatalogCollection 'Collection of components
Dim colInt As COMAdminCatalogCollection  'Collection of components

'The interface we will attempt to make QUEUEABLE and
'the component and application it resides in:
Const APPLICATION_NAME = "Trading System"
Const COMPONENT_PROGID = "Current.Currency"
Const INTERFACE_NAME = "_Currency"

'obtain the Application collection from the Catalog:
Set colApp = Catalog.GetCollection("Applications")

'Search for the our application
colApp.Populate
For Each Application In colApp
    If Application.Value("Name") = APPLICATION_NAME Then Exit For
Next

'Did we find our Application?
If Application Is Nothing Then
    MsgBox "Did not find the Trading System Application!"
    End
End If

'Turn on queueing support for this application:
Application.Value("QueuingEnabled") = True
Application.Value("QueueListenerEnabled") = True
colApp.SaveChanges

'obtain a component collection of all the components in our application:
Set colComp = colApp.GetCollection("Components", _
                                   Application.Key)

'Search for our component:
colComp.Populate
```

continues

Listing 10.3 **Continued**

```
'For Each Component In colComp
If Component.Value("ProgID")=COMPONENT_PROGID Then Exit For
Next

'Did we find our component:
If Component Is Nothing Then
    MsgBox "Did not find the Currency Component!"
    End
End If

'obtain an interface collection of all the interfaces in the component
Set colInt = colComp.GetCollection( _
                    "InterfacesForComponent", Component.Key)

'Search for the interface we wish to make queueable
colInt.Populate
For Each Interface In colInt
    If Interface.Name = INTERFACE_NAME Then Exit For
Next

'Did we find our interface?
If Interface Is Nothing Then
    MsgBox "Did not find the desired Interface"
    End
End If

'Is queueing even supported on this interface?
If Interface.Value("QueuingSupported") = True Then
    'Queueing is supported on this interface, so turn it on:
    Interface.Value("QueuingEnabled") = True
End If

'Save changes to the Catalog.
colInt.SaveChanges
```

Or, if in the IDL declaration of your interface, you included the `QUEQUEABLE` tag thus (see Listing 10.4):

Listing 10.4 **Adding the *QUEQUEABLE* Tag in the Interface's IDL Declaration**

```
//Be sure to include "mtxattr.h" so that the QUEUEABLE tag will be recognized
[
    object,
    uuid(1E14A3F0-FEC5-4114-A3A6-796EA40D51A4),
    pointer_default(unique),
    QUEUEABLE
]
interface ITest : IDispatch
{
```

You might imagine that method calls made to an interface become asynchronous the moment you mark it as Queued. Not so. The object supporting this interface behaves identically, synchronously, as before. By marking an interface queueable, you are declaring its *candidacy* to receive asynchronous calls, but you'll soon see in the section "Creating a Queued Object Using a Moniker" that an object implementing a queued interface needs to be instantiated in a particular way, via a *moniker*, before it will exhibit queued behavior.

Assuming the object implementing the queued interface is instantiated appropriately, instead of synchronous RPC, MSMQ is used as the underlying transport. As you've seen, MSMQ does not demand or even expect that the client and server run at the same time, so you effectively de-couple the client using this interface from the object implementing it. A method call made to a queueable interface is turned into an MSMQ message that is delivered to the recipient object at some later time after being turned back into a method call.

So, MSMQ is an asynchronous messaging middleware accessed through an API (or a group of COM objects shipping with MSMQ that are layered on top of that API). Knowing that COM+ is leveraging MSMQ to provide asynchronous method calls, you can intuit the following:

- COM+ is providing some form of abstraction on top of MSMQ and hiding the details of MSMQ's involvement to clients and objects utilizing QCs.
- The existence of QCs implies the existence of MSMQ queues created and maintained by COM+.
- If MSMQ is a separate service, it must be installed on a machine for QCs to work. The possibility exists that it might not be installed, and therefore QCs might not work with a new Windows 2000 installation.
- QCs are ultimately subject to the strength and limitations of MSMQ.
- Methods of interfaces marked as queueable cannot have return or output parameters because the method call must be turned into an MSMQ message and sent asynchronously through a queue.
- Certain native MSMQ features are inaccessible and/or incompatible with the QC's paradigm.
- A certain degree of cooperation between pure MSMQ and QCs is possible.

The following sections explore each of these points.

Restrictions on Queued Interfaces

An interface can only be marked queued if its methods contain no output parameters and it uses only OLE automation compatible types—that is, VB-compatible types the OLE automation marshaler knows how to handle.

The Queued Components Abstraction: Introducing the Player, Listener, and Recorder

By now, you are probably tired of me telling you that QC relies on MSMQ. The question then becomes the following: How does COM+ leverage MSMQ to the degree that you can simply click a check box, and when done, your method calls are suddenly asynchronous without you ever creating a queue or calling the MSMQ APIs in any way?

To provide for MSMQ's hidden involvement, COM+ introduces a new paradigm involving the following three objects:

- The `Recorder`
- The `Listener`
- The `Player`

Before any of these objects come into play, however, a queueable object must first be instantiated via something called a *queue moniker*, as shown in the next section.

Creating a Queued Object Using a Moniker

Let's assume that you want to instantiate and obtain an interface (one designated as queued) from an object called `FirstQueue`. To create this object and obtain this interface, you need to go through the queue object via a moniker. This is demonstrated in Visual Basic in Listing 10.5.

Listing 10.5 **Examples of Creating Queued Objects Via Monikers**

```
//VB

Dim QC as IFirstQueue
Set QC = GetObject("queue:/new:FirstQC.FirstQueue")

//C++

IFirstQueue* pFirstQueue;
CoGetObject (L"queue:/new:FirstQC.FirstQueue", NULL, IID_IFirstQueue, (void**)
➥&pFirstQueue);
```

In the event you are not entirely comfortable with monikers, I will offer a brief explanation. A moniker string can allow you to create a COM object and pass it arguments in the process. The ordinary COM creation mechanisms (mainly `CoCreateInstance`, `New`) do not provide a way to actually send construction arguments to an object. Although monikers are certainly useful in helping you bypass this restriction, they don't seem especially impressive until you take into account that the arguments they let you send can be other COM objects. And if this still isn't compelling, the COM

objects specified in a moniker string can coordinate with one another and, in effect, cooperate to produce the object ultimately returned by a call to VB's `GetObject` (`CoGetObject` in C++). By way of example, examine the following moniker string:

```
queue:/new:FirstQC.FirstQueue
```

This moniker string has two COM classes in it, `queue` and `new`. If you look in the Registry, you will find both are COM classes with the ProgIDs of `queue` and `new`.

Both `queue` and `new` are ordinary COM objects, but they are also written to support certain interfaces (`IMoniker` being one) that make them monikers. An object wants to be a moniker if it wants to have its creation influenced by other monikers specified in a moniker string. The implementation details of a moniker are complex to describe, but from the client's perspective, it is pretty simple. When the preceding string is given to VB's `GetObject()` method, COM+ processes the string from left to right. It creates the queue object and feeds it the rest of the string. The queue object processes as much of the string as it can, but then gives the remainder back to COM, usually after detecting and chopping off some kind of character delimiter, a `/` in this case. COM then repeats this process with the remainder of the string by creating the `new` object and giving the portion of the string remaining to that object. This can be thought of as the *parsing process*, but another process, the binding process, remains.

As moniker objects are created left to right, a common context (called a `BindContext`) groups them together and relates them to one another. A moniker object is given an interface (`IMoniker`) to the moniker at its left (queue in this case); thus, the `new` moniker knows that a queue moniker is at the left of it and can interact with it in any fashion. This negotiation between monikers from right to left is often called *binding*. The binding process is like a snowball rolling down a hill, right to left, getting bigger as more creation information is amassed, ultimately resulting in a single object being created. By way of example, the moniker string `queue:/new:FirstQC.FirstQueue` can be described in the following conversation:

> COM: Create the object whose ProgID is queue and hand it the string `/new:FirstQC.FirstQueue`.
>
> `queue`: I am expecting some arguments that only I understand, but I don't see any supplied. All I see is a delimiter, `/`, which tells me that I am at the end of any string information intended for me. Therefore, I am going to chop off the `/` delimiter (COM defines a number of such delimiters; the `/` is only one) and hand the rest of the string back to COM.
>
> COM: Create the object whose ProgID is `new` and hand it the string `FirstQC.FirstQueue`.
>
> `new`: I see arguments. I understand. I will create a new instance of the object whose ProgId is `FirstQC.FirstQueue` because this is my purpose in life, to create new objects. But wait, I see through the binding context that I share with other

monikers that there is a queue object to my left. I am written to work with this type of object, so rather than create a new instance of `FirstQueue`, I will retrieve its GUID and pass that to the queue object.

`queue`: The `new` moniker on my right has given me a GUID. I will create an instance of the object specified by this GUID and return its default queueable interface to the caller of `GetObject()`.

The `queue` object mentioned in the moniker string ultimately produces an `IFirstQueue` interface. VB places a reference to this interface in the QC variable (as shown in Listing 10.5), and the client can call methods on it in a normal fashion. It is always true of moniker strings that no matter how complex, they always result in one single interface to one particular object.

Although creating objects via monikers might seem like an unnecessary step (after all, why not just create a queue object directly?), remember that monikers allow for arguments to be passed to objects. Although the moniker string `queue:/new:FirstQC.FirstQueue` is relatively simple, it is more common to pass additional information to the `queue` object in strings like the following:

```
queue: Priority=6,ComputerName=Darjeeling/new:FirstQC.FirstQueue
```

If you cannot pass arguments in the moniker string, the author of the queue object is forced to write the queue object such that it has properties corresponding to every possible argument. You end up with code like the following:

```
'We wouldn't want to do this:
Dim q as new Queue
q.Priority=6
q.ComputerName="Darjeeling"
'And every other possible property
```

Keep in mind that, although you are creating a queue object in the moniker, the `queue` object is ultimately returning `IFirstQueue`, an interface normally implemented by the `FirstQueue` object. Ultimately then, the `queue` object is merely a way station, a staging area for the creation of another object. As you see in the next section "The Recorder," the queue moniker does not return an actual RPC interface pointer to an instantiated `FirstQueue` object, it returns an MSMQ-based entity called a *Recorder*.

Before we move on to the next section, note that some other variations of the queue moniker are detailed and explained in Appendix E.

The Recorder

The Recorder is a COM+ system object that pretends to be a given, queueable object. When you create a queueable object via the queue moniker shown in the previous section, you get back an interface to a Recorder, not an interface to the object itself (which COM+ does not even bother to instantiate). The Recorder is a kind of proxy that stands in for the requested object and implements the interfaces of this object (those marked as Queued), such that the client is unaware of the deception. To

be fair, the Recorder cannot deceive the client completely; attempts by a client to QueryInterface (QI) for a non-queueable interface that the real object would ordinarily support is met with an error code. A Recorder is an imperfect proxy then, supporting only an object's queueable interfaces.

Proxies are certainly nothing new. Remember, a client application on Machine A can only have an interface proxy to an object on Machine B—that is, the client side of an RPC channel—not a pointer to the object itself. In this traditional case, there is a synchronous RPC connection between the two. The client can rely on the fact that when it calls a method on the interface proxy it holds, the call is immediately, synchronously delivered to a living object.

This is not the case with the Recorder. Think of the Recorder as exactly that, a recorder. The same way a tape recorder translates your spoken word to magnetic imprints and stores them on a tape to be played back to another party at some later time, the COM+ Recorder records method calls, translates them into messages, and stores them in a queue. When you record your voice on a tape recorder, you don't need to be around when the tape is played for someone else, and you certainly don't need to be around while someone is listening (processing) what you said on tape. To be fair, the person listening can play the tape recording whenever he is ready and prepared to, and the tape itself can outlive both you and the listener. If you think of the tape as an MSMQ queue and the listener and recorder as the COM+ Player and Recorder, you begin to get the idea.

The Player

The Player performs actions opposite the Recorder. It reads messages from a queue, turns them back into method calls, and invokes these method calls (plays them) on the queueable object. The Player can only play messages if the COM+ application in which the queueable object resides is running and listening for messages.

The next section demonstrates how to make a COM+ application listen, but basically an application listens to its queues through an instance of the `ListenerHelper` COM class. When a message arrives, the `ListenerHelper` object instantiates an instance of the Player and gives it the message(s) received. The Player, in turn, instantiates an instance of the recipient object, translates the message back into method invocations, and invokes them on the recipient object, security settings permitting.

As with the client's interaction with the Recorder, the recipient queueable object is unaware that the method invocations it is receiving on its interfaces are coming from the Player. They have the security context, privileges, and so on as though they are arriving via direct RPC invocation.

Creation Conditions for the Recorder

Note that a client comes into contact with the Recorder if and only if it instantiates a queueable object via the queue moniker. If it creates the COM object and/or QIs for the queued interface in the normal COM fashion (`CoCreateInstance` in C++, or VB's `New` keyword), it gets an ordinary, synchronous RPC connection to a real instance of the object.

Writing Queueable Objects

Queueable objects are ordinary COM objects. Aside from the fact that the methods of their interfaces cannot have output parameters and their method arguments can only take OLE automation compatible data types, they are not written any differently. Such objects can be used in the ordinary COM fashion, and it is only when they are instantiated via the queue moniker that their special nature becomes apparent. Keep in mind also that their queueableness can be taken away at any time by simply removing the check from the Queued check box for its interface(s), as illustrated in Figure 10.2. An attempt to instantiate a non-queueable object via this moniker results in an error.

Writing a queueable object is straightforward; any ActiveX Dynamic Link Library (DLL) will do (again provided that its interface methods do not have output parameters and it uses only OLE automation compatible types). In VB, after a new ActiveX DLL project has been created, adding even one subroutine, as in the following, is all you need:

```
'In VB ActiveX class FirstQC

Sub DisplayNumber(ByVal Number As Integer)
    MsgBox Number
End Sub
```

But in VB, beware! The following method renders the class unqueueable:

```
'Can't be queued
Sub DisplayNumber(Number As Integer)
    MsgBox Number
End Sub
```

The reason is that, in VB, all arguments are passed by reference (implicitly `ByRef`) unless otherwise stated by using the `ByVal` keyword. In VB, `ByRef` translates to an `[in,out]` parameter in the underlying type library, which is not allowed for a queueable interface.

Assuming our object is named `FirstQC`, after you compile this object (VB registers it during the compilation process), you then need to import it into COM+ through mechanisms described in Chapter 6, "The COM+ Catalog." At this point, you designate the `_FirstQC` interface as queued. (Note that VB prepends its default interfaces with a _.) One additional step is required. Even though `FirstQC` is now, technically, a queueable object, the COM+ application that `FirstQC` lives in must also have the appropriate settings. These settings will be discussed next.

Queueable Settings for the Application

One of the property pages that appear when you right-click on an application using Component Services is the Queuing page, show in Figure 10.3.

If you find that these boxes are grayed out, you might check to make sure that the application is a Server Application and not a Library Application. QC-based applications must always be Server Applications.

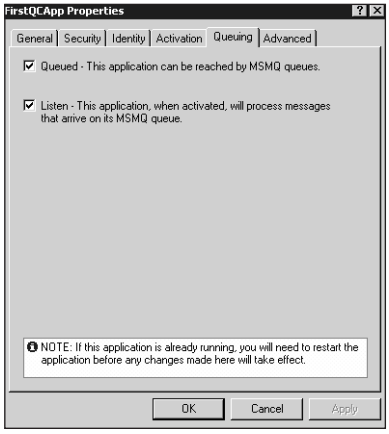


Figure 10.3 The Queuing tab for an application.

If you click the Queued check box, by default, COM+ creates seven queues. You can see them using the Computer Management snap-in, shown in Figure 10.4.

The first queue has the same name as the application, and it is followed by five similarly named queues numbered 0 to 4. There is one last queue, known as the *dead queue*. Basically, a message works its way through all these queues as COM+ repeatedly attempts to deliver the message to the receiver in the context of a transaction.

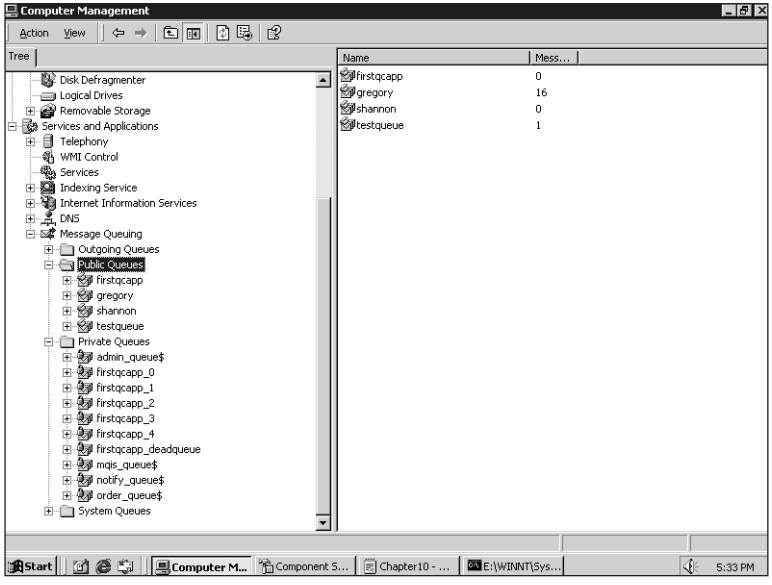


Figure 10.4 The Message Queuing branch of the Computer Management snap-in.

The fact that the message delivery is attempted as part of a transaction should not surprise you, given that MSMQ is a *bona fide* RM. To MSMQ, delivering a message to or from a queue is akin to a more traditional RM, such as a database receiving a SQL update. As far as MSMQ is concerned, the queue is the data store, and the main purpose of its transaction is to ensure that the message successfully gets placed in the queue or removed from it completely. It is also possible to use solitary non-transactional queues with QC, but we will delay the discussion of these until the section "Transactional and Non-Transactional Queues."

After the Queued check box is clicked and the queues are created, `FirstQC` can be instantiated via a queue moniker and can begin receiving messages. Keep in mind, however, that it is the Recorder standing in as a proxy for the object that is getting the method calls from the client, turning them into messages, and storing them in the queue for later retrieval.

This brings us to the other check box shown in Figure 10.3. The second check box, Listen, seems simple enough, especially with the helpful sentence imprinted on the dialog box. This check box causes the `ListenerHelper` to monitor the queues for the application. If the `ListenerHelper` detects a message, it hands it to the `ListenerHelper` object, which gives it to the Player, which creates a new instance of `FirstQC` and replays the method invocations made by the client.

One final, critical point to make is that `FirstQC`'s parent application must be running for `FirstQC` to receive any messages. To make certain that your application is running, simply right-click on it in the Component Services and select Start. Note that clients can still send messages via the Recorder even if the application is not running—they just aren't picked up and played to a new instance of the `FirstQC` until the application is next run. It might seem odd that the application needs to be explicitly run by the user or otherwise started using the COM+ administrative objects. After all, COM+ applications are normally run automatically when a client calls on one of its components. It is the asynchronous, monitoring nature of QC and MSMQ that make explicit startups a necessity.

QC Internals

Now that we've discussed the Player, Recorder, and Listener, let's explore implementation of a QC and track method calls/messages as they move through a COM+ application's queues.

Disabling Security

Security settings paired with your specific NT domain topology and other administrative issues add another, more complicated layer to this process, which we'll explore in Chapter 12, "Security." For the examples in this chapter, let's explore the simplest possible QC scenario first, where security and other network considerations are removed. To cut out the security layer altogether, go to the Properties page for the `FirstQCApp` application and click on the Security property tab. Adjust the settings such that Authentication Level for Calls is set to None, as shown in Figure 10.5.

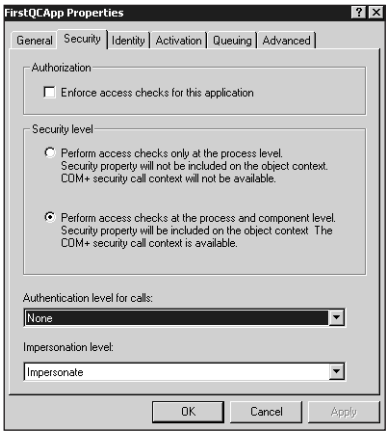


Figure 10.5 The Security settings tab for `FirstQCApp`.

Viewing Method Calls as Messages

You know that the Recorder translates method calls into MSMQ messages and puts them in an application's queue. If you are wondering exactly when the method calls are put into the queue by the Recorder, always remember that MSMQ is an RM. Therefore, by the rules outlined in Chapter 8, "Transactions," the message is placed in the queue when the transaction is committed—that is, when the root object is deactivated with its Happy bit set to `TRUE`. Given that your base client executable does not have a context and, therefore, does not have a transaction, COM+ automatically creates a new transaction when the Recorder is instantiated. If you check the transactional settings for the COM+ Recorder, you'll notice that it is set to Required. As you learned in Chapter 8, this setting means that the object inherits a transaction if its creator had one, or COM+ creates a new transaction for the object if it did not. Thus, all QC method calls are transactional, and the method calls are committed to the queue as messages when the Recorder object, the root object in this case, is deactivated and Happy.

Authentication Problems with Windows 2000 Workgroup Mode

Although I don't want to get into Windows 2000 administration too much at the moment, I want to save a few hours of frustration for you if I can. If you are experimenting with Windows 2000, perhaps you just have Windows 2000 Professional on your desktop but are not logged in by a *bona fide* Windows 2000 Server (or Advanced Server) domain controller, you must disable security entirely. This is because you are in *workgroup mode*, and in this mode, the Windows 2000 Active Directory services are not available to provide user authentication. Calls to create or otherwise use your queued components will fail.

Authentication will be discussed in greater detail in Chapter 12 where you will also find that role-based security can still operate, even if authentication is turned off.

Let's take a look at how method calls turn into messages. Imagine the following code exists in a simple VB client:

```
Dim QC As FirstQC.FirstQueue

Sub Form_Load()
Set QC = GetObject("queue:/new:FirstQC.FirstQueue")
End Sub

Sub Command1_Click
QC.DisplayNumber 4
End Sub
```

Further, imagine that you perform the following steps:

1. Run this client.
2. Click the button once.
3. Close the application.
4. Run the application again.
5. Click the button 10 times.
6. Close the application.

Keeping in mind that because `FirstQCApp` is not running (you haven't explicitly started it), you can go to this application's queues and take a look at the messages. Using the Computer Management Console, you can navigate to `FirstQCApp`'s queues, as shown in Figure 10.6.

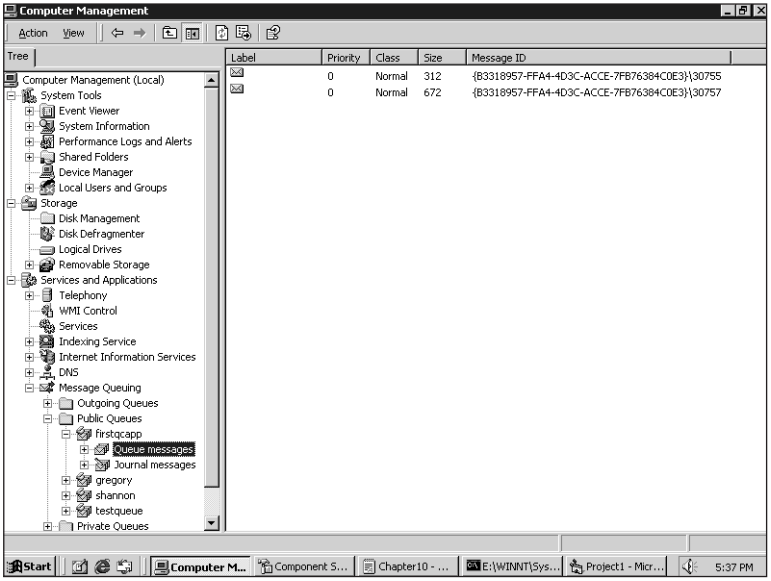


Figure 10.6 Two messages in the `FirstQCApp` queue.

You might have expected 11 messages because you made 11 method calls. But, if you think about it, you'll find it makes perfect sense. As an RM, MSMQ attempts to commit a transaction when the root object completes its work. In our example, our base client made one method call and was closed, thus deactivating the `Recorder` object (the root object of the transaction) and forcing the transaction to commit. The result of this operation was one message.

Next we reran the application, made 10 method calls, and again forced the transaction to complete. It seems logical and efficient then that the 10 method calls should be bundled into a singular message.

There is one large caveat, however. If any single method call of the 10 in the message causes the recipient to crash or call `SetAbort`, the message is retried (according to the rules laid out in the upcoming sidebar "Timeout Periods and Retry Attempts for Failed QC Messages"). This message, however, contains 10 method invocations, and they are all played to the recipient again, not just the one that failed. This implies that your method calls should be idempotent, or at the very least, any data modifications performed by queued methods should be under the umbrella of the recipient's transaction, which given that the modifications are undone in the event of an aborted transaction, amounts to the same thing, more or less. These messages can, of course, be viewed. Double-click on the first one (or solitary method call), and you will see something like Figure 10.7.

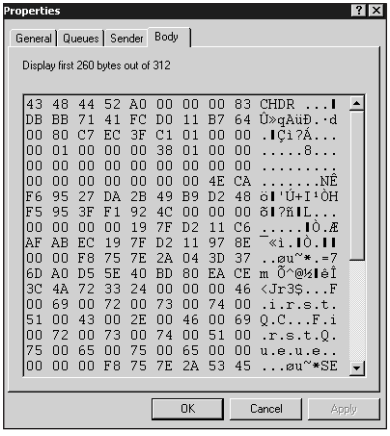


Figure 10.7 The contents of the message resulting from a method call to `FirstQC`.

You don't need to understand the contents of the message, only that this is the binary representation of your queued method call. In the section, "Manually Reading the Message," I demonstrate how to take the contents of this message and put it into a queue directly via an ordinary, non-COM+ MSMQ client. For the moment, however, let's continue our exploration.

Playing the Messages

Because `FirstQCApp` is not running, method calls made by clients requesting queueable objects from this application simply pile up in the queues. So, by the simple action of right-clicking the application in the Component Services Console and selecting the Start menu item, you are ultimately greeted by 11 message boxes, some of which are minimized, as shown in Figure 10.8.

If you dismiss the 11 message boxes and look into `FirstQCApp`'s queues, you find them empty. The messages have been detected by the Listener; and the Player was created, which then instantiated a `FirstQC` object and played the messages as method calls for the object. Messages are removed from the queue after they are successfully played for the recipient.

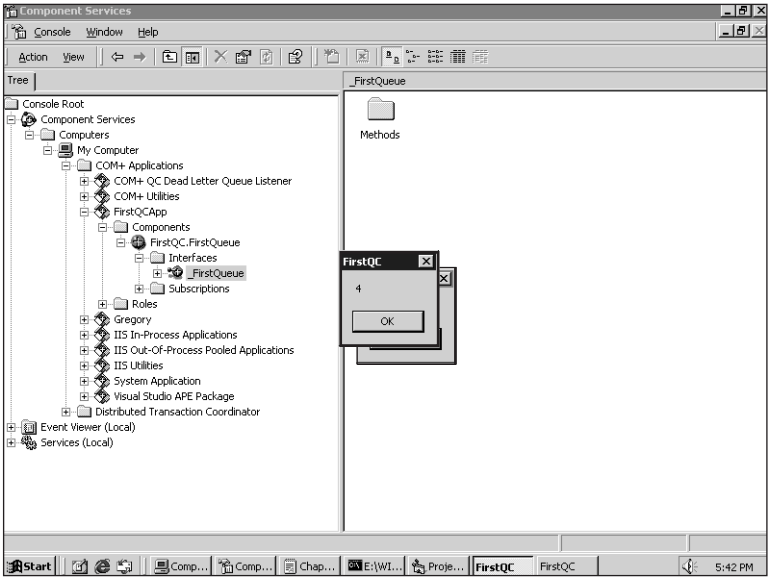


Figure 10.8 The methods are executed by the Player when `FirstQCApp` is run.

Transactional and Non-Transactional Queues

By default, when you click the Queued check box for a COM+ application, you get seven transactional queues. As COM+ tries repeatedly to deliver a transactional message to an object that keeps crashing, keeps calling `SetAbort()`, or is involved with some other RM that keeps failing, it moves through the numbered queues—lowest to highest. Ultimately, if the message cannot be delivered, it ends up in the dead message queue (or played to your exception class as discussed in the section, "Exception Classes") where a system administrator should probably take a look at the message. Note that each of the seven queues are transactional, but the movement of a message from one queue to a queue of a higher number as it fails to be delivered is not part of what it means to be a transactional queue. The upward movement through six queues and ultimately to a dead message queue is a convention of COM+—not of transactions or MSMQ. For more information on the way COM+ attempts to deliver messages, see the following sidebar, "Timeout Periods and Retry Attempts for Failed QC Messages."

It is possible to have a single queue that is transactional. You can even choose to create one such queue before creating your application, but COM+ quickly creates the other six queues as soon as you mark the application as Queued. Alternatively, you can create a single non-transaction queue, and if you later create an application of the same name, COM+ does not create six other queues. However, there's no good reason to do this unless you don't want COM+ to redeliver the message if the receiver calls `SetAbort()` or some other failure occurs.

Timeout Periods and Retry Attempts for Failed QC Messages

The protocol for retrying delivery and moving a message upward through the queues toward the dead message queue can be empirically discovered. Assuming that you have a client that crashes or calls `SetAbort()` continuously, you will find the message moves as follows:

- `BLOWUP_0` queue. COM+ waits 60 seconds before re-attempting delivery three times.
- `BLOWUP_1` queue. COM+ waits 120 seconds before re-attempting delivery three times.
- `BLOWUP_2` queue. COM+ waits 240 seconds before re-attempting delivery three times.
- `BLOWUP_3` queue. COM+ waits 480 seconds before re-attempting delivery three times.
- `BLOWUP_4` queue. COM+ waits 960 seconds before re-attempting delivery three times.
- `BLOWUP_dead` queue. The message should now be examined by a system administrator unless an exception class is set up to be notified. Exception classes are discussed in the subsequent section by that name.

Note that the retry times are individual—not cumulative. In other words, after 60 seconds of waiting on the first failure, COM+ waits 60 seconds after the second failure, and so on. At present, these times are not changeable by the system administrator.

In the context of COM+, QC transactional queues guarantee that a message will be delivered. However, although you might read in Microsoft publications that it is delivered only once, this is not necessarily true—remember that one message might translate to several method calls played for the client. As discussed in the previous section, if any of these method calls cause a crash or `SetAbort()` to occur in the recipient, the entire message is replayed and the nine good methods are executed again along with the failed tenth. It is important, then, to make sure that the method calls are idempotent and/or data modifications they make will be aborted and undone if the recipient's transaction should fail.

Transactional queues have some very desirable attributes, so the following question arises: Why would you not want to use a transactional queue? Perhaps performance might be far more important than reliability in some scenario, and you want to avoid the overhead of MSMQ transaction enlistment by using non-transactional queues. Alternatively, you might want to bypass the transactional `Recorder` and `ListenerHelper` (both are marked `transaction=Required`) and access the queues directly with non-transactional clients or objects.

Non-transactional queues do not, obviously, require transactions, nor do they support them. These queues do not guarantee delivery, and messages could, conceivably, be lost altogether.

The Different Queue Types: Private and Public

There are certain points where development efforts and NT system administration converge, and you cannot understand one fully without understanding the other. Private and public queues sit at such a point.

Basically, a *private queue* is one that an MSMQ application must explicitly know about. Senders and recipients must know the exact location and name of this queue. If you are familiar with IPCs mechanisms, queues have a very similar naming convention as named pipes. Although private queues are often created on the local machine and used when the sender and recipient are also on the local machine, the sender, receiver, and queue can all be on different machines. Private queues do not suffer any limitations except that they do not advertise their existence in any way.

Public queues, on the other hand, advertise their existence through the Active Directory. Active Directory was introduced with Windows 2000. You can think of it as a hierarchical database that is automatically replicated on all domain controllers (DCs) that are part of the same *forest*. A forest is a group of logically related domains that agree on a particular *schema*—that is, they share a common hierarchical view consisting of specific system and user-definable objects whose meaning and organization are relevant to them. Unfortunately, to give Active Directory its proper due and to explain schemas, forests, and so on in the appropriate depth would take a book of its own. For that reason, we are going to look at Active Directory from a high level.

The Active Directory database contains user account information for authentication purposes, as well as information about printers, fax machines, and, among other things, public queues. The existence of public queues are broadcast by Windows 2000 using Active Directory, and this makes them easy to find. A client might need only the name of the public queue and does not need to know where the queue was. This is not the case with a private queue.

Before we leave the topic of the Active Directory, know that it can also hold and replicate any form of hierarchical information that you care to enter into it programmatically via Active Directory Service Interfaces (ADSI). I will not get deep in a discussion about Active Directory because that would take us into the field of NT administration and is outside the scope of this book. For completeness, however, the companion code for this book includes a simple VB client that uses ADSI to find out information about public queues. Additional information on Active Directory can be found in New Riders's *Windows 2000 Active Directory*, by Edgar Brovick, Doug Hauger, and William C. Wade III. For additional information on ADSI scripting, see New Riders' *Windows NT/2000 ADSI Scripting for System Administration*, by Thomas Eck.

Other than their self-aggrandizing, showy nature, public queues are identical in functionality to private queues, and either type can be transactional or non-transactional. COM+ applications marked as Queued create their first, primary queue as public, and the six other queues are private because they are an implementation detail of QCs and are not meant to be accessed directly by MSMQ clients.

Interoperability Between MSMQ and QC

You know that QC is based on MSMQ. It stands to reason, then, that it must be possible for an MSMQ application to interact with a QC and *vice versa*. Although this is certainly the case, it is probably best to stick with one or the other. After all, QC uses a proprietary binary format when it translates method calls to messages, so although an MSMQ recipient can receive the message, in all likelihood, it cannot do anything meaningful with it.

To prove that MSMQ and QC are truly interoperable and to further explore the details of this symbiosis, I am going to demonstrate how to write an MSMQ client that masquerades as a QC client. Specifically, I'm going to simulate a method call to a QC object by writing a message directly to the queue using an ordinary MSMQ client, bypassing the Recorder altogether.

You know that the Recorder translates method calls made by a client on a queueable object (created by the client through the queue moniker) into messages and puts them in the object's application's queue. It is also true that the destination COM+ application might not be running/listening at that time the method call is made, and even if it is, the calling client and receiving object are uncoupled—the Receiver monitors the queue and does not know or care about how a message got into its queue, only that a message is in it.

Manually Reading the Message

The algorithm used by the Recorder to translate method calls to messages is not documented. Assume that the following code is executed by a QC client:

```
Dim QC As FirstQC.FirstQueue
Set QC = GetObject("queue:/new:FirstQC.FirstQueue")
QC.DisplayNumber 4
```

You can see the resulting message in `FirstQCApp`'s queue by navigating to `FirstQCApp`'s primary queue, viewing its messages section, and double-clicking the resulting message. You see a message similar to that shown in Figure 10.7.

We are going to simulate the preceding client-side code using an ordinary, non-transactional MSMQ client. For simplicity, I am assuming you are in Workgroup mode (that is, not using Active Directory). Follow these steps:

1. Because COM+ uses transactional queues by default and your MSMQ client is going to be an ordinary non-transactional EXE, you need to create a new, non-transactional private queue. This is easy to do, however. Simply choose Computer Management MMC, Services & Applications, Message Queuing, and right-click Private Queues. Then select New. The dialog box shown in Figure 10.9 appears.
2. Give the queue the name `SecondQCApp` and don't check the Transactional check box. This creates a private queue on the system called `SecondQCApp`.
3. Now create a COM+ application called `SecondQCApp` and make it queue-enabled (also, don't forget to turn authentication off). Instead of creating seven new queues as before, COM+ uses the existing non-transactional queue you just created for the queued components on this system.
4. Delete the `FirstQC` component from `FirstQCApp` and install it into `SecondQCApp`. Then, mark its interface as queued. The stage is now set—you have created a new application, `SecondQCApp`, that is using the new non-transactional queue you explicitly created for it. By installing `FirstQC` in `SecondQCApp`, queued calls you now make to `FirstQC` will reach `FirstQC` through the new, non-transaction queue.

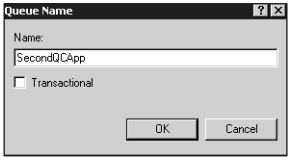


Figure 10.9 The New Queue dialog box.

5. Using the application described in the "Viewing Method Calls as Messages" section, try the following:

   Click the command button once.

   Close the application.

Now examine the contents of the queue as shown in Figure 10.10.

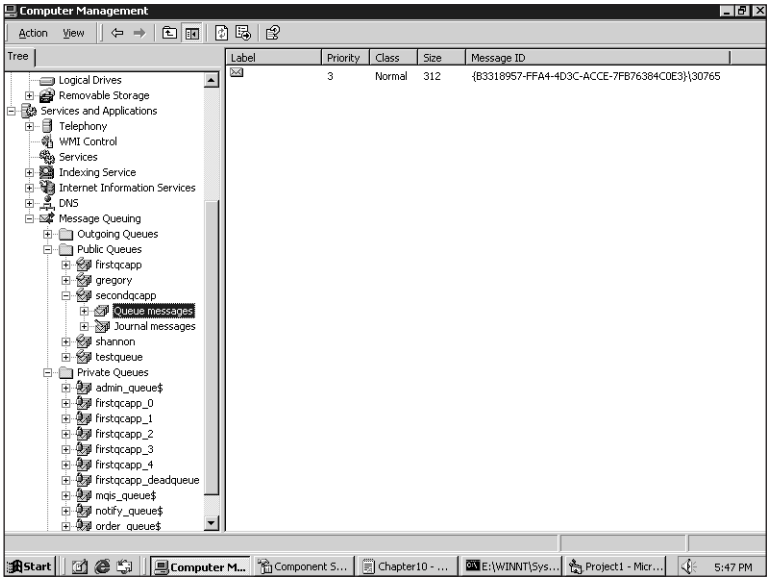


Figure 10.10 Contents of the new `SecondQCApp` queue.

Predictably, there is one message in the queue that corresponds to the method call you just made. But this queue can be read from or written to by MSMQ-aware applications, so it is legal to read and remove a QC-generated message using a pure MSMQ application. The code in Listing 10.6 demonstrates how this message can be read by an ordinary MSMQ application.

Listing 10.6 **Reading a Message and Dumping Its Contents into a Binary File**

```
Dim qInfo As New MSMQQueueInfo
Dim qSource As MSMQQueue
Dim msgRec As MSMQMessage
Dim vntBody As Variant

'Specify the queue we will be reading from, in this case
'the non-transactional queue we created before. Note that
'if this queue was transactional, an attempt to open it
'would fail since a client application has no transaction
'associated with it.

'Note that this code assumes you are in Workgroup mode
'in other words authentication via Active Directory is not being used

qInfo.PathName = ".\private$\MSMQTest"

'Open the queue and try to retrieve the message
Set qSource = qInfo.Open(MQ_RECEIVE_ACCESS, MQ_DENY_NONE)
Set msgRec = qSource.Receive(ReceiveTimeout:=1000)

If msgRec Is Nothing Then
    MsgBox "Nothing in Queue"
Else

    'dump the message body contents to a binary file:
    vntBody = msgRec.body
    Open "body.dat" For Binary As #1
    Put #1, , vntBody
    Close #1

    'dump the message extension contents to a binary file:
    vntBody = msgRec.Extension
    Open "extension.dat" For Binary As #1
    Put #1, , vntBody
    Close #1

End If

qSource.Close
```

The code in Listing 10.6 reads the message from the queue (thus removing it) and dumps the message body and extension into two binary files. The message extension is a new property in MSMQ 2.0 (the version that comes with Windows 2000) and is used to specify additional application-defined information that is associated with the message.

These two binary files completely describe the message that results in the method call you want. To illustrate this, you can construct a program that uses the binary files to package a message and put it back in the queue. Listing 10.7 demonstrates this process.

Listing 10.7 **Manually Placing a Message into a Queue**

```
Dim qInfo As New MSMQQueueInfo
Dim qDest As MSMQQueue
Dim msgSend As New MSMQMessage
Dim vntBody As Variant

qInfo.PathName = ".\private$\MSMQTest"
Set qDest = qInfo.Open(MQ_SEND_ACCESS, MQ_DENY_NONE)

'Construct the message by reading the binary files:

Open "body.dat" For Binary As #1
Get #1, , vntBody
Close #1
msgSend.body = vntBody

Open "extension.DAT" For Binary As #1
Get #1, , vntBody
Close #1
msgSend.Extension = vntBody

'Send the same message 5 times:
For k = 1 To 5
    msgSend.Send qDest
Next

qDest.Close
```

If the code in Listing 10.7 is run, an examination of `SecondQCApp`'s queue reveals five new messages. If you were expecting one large message, remember that you do not have a transactional queue, and this is not a transactional client, so each `Send` corresponds to exactly one message. That said, it should come as no surprise that when `SecondQCApp` is started, five message boxes appear as the `FirstQC` object is instantiated and played its messages.

Although it is possible to examine the contents of body.dat and try to understand the binary layout of QC messages, such an effort would probably bear little fruit. The layout can always change, and if authentication were on, the contents would be encrypted.

Persistence: Passing Objects Through Messages

A message's body property can store pretty much anything—any kind of variable and even binary data can be sent from sender to the recipient object. If you are wondering whether objects themselves can be passed through messages, the answer is sort of.

The terminology *passing an object* is a bit of a euphemism—that is, a friendly, simplified way of describing a more complicated, ugly reality. Objects are not passed in COM+; they do not travel. Rather, references to objects are passed in the

form of marshaled interfaces. You can certainly use an interface pointer as an argument to a method and hand out access to some Object A to other objects throughout the network for callback purposes. In this scenario, however, the fundamental Object A remains rooted on the machine in the process in which it was created. This paradigm, that of singleton server handing out references to multiple clients, simply doesn't fit in the QC framework.

Even if you could pass an interface to a living object as a message, there is no way to tell when the receiving application will be running. Therefore, you cannot tell when the recipient object in that application will be playing the message as a method call, and as such, you cannot be sure when it will be given this interface pointer as a method argument. By this time, the original client object referenced by the interface might have been destroyed, and the receiver might unmarshal an orphaned interface to a dead object. And even if the object somehow remained alive, data contained in a marshaled interface pointer is only valid for about five to six minutes.

Ideally, if you are going to send objects from sender to receiver using QC, you need to move the entire object from sender to recipient. And by supporting an interface called `IPersistStream`, an object can elect to take such a trip. An object that supports `IPersistStream` is indicating that it knows how to save its state into a binary stream of bytes and, upon request, gives this stream to any requesting client. Such an object is said to be *persistable,* and persistable objects also know how to reconstruct themselves if some other client instantiates a new instance then and hands them back this stream.

Listing 10.8 demonstrates a C++ implementation of `IPersistStream`. This listing shows a portion of the code for a COM class called `NumberClass`. `NumberClass` simply keeps an array of 1,000 integers, and by inheriting from and implementing `IPersistStream`, it can save and restore its state when asked.

Listing 10.8 **A Portion of C++ COM Component Demonstrating *IPersistStream***

```
// This is only a partial listing of the persistent component. Consult
// the book's accompanying source code for a complete example.

/*****************************************************************************
 IPersistStream implementation

 The implementation of the IPersisStream interface.  COM+ asks the component
 for this interface if this component is passed as a parameter to a method
 of a queued componenet.

 IPersistStream inherits from IPersist, and thus must support the one
 method of IPersist (GetClassID). In addition IPersistStream contains
 four other methods, all of which are explained below:

 *****************************************************************************/


// GetClassID is required by IPersist (and hence IPersistStream).
// For persistant objects, COM requires that we provide the component's
// CLSID, so that it can associate the component with the persistent data
// it will be writing/reading.
```

```
HRESULT NumberClass::GetClassID(CLSID *pClassID) {
    *pClassID = CLSID_NumberClass;
    return S_OK;
}


// Before COM asks an object to write its state to a stream,
// it calls the IsDirty method.  This method informs
// COM if the object's state has changed since the last save.
// If it is hasn't, then COM can forgo the write.
//
// In our case, inform COM to always perform the write:

HRESULT NumberClass::IsDirty(void) {
    return S_OK;
}

// The Load method is called when COM wants the object
// to restore its state from a stream.  If this component
// were passed as a parameter to a method of a queued component,
// this method would get called when the QC.Player unpacks
// the MSMQ message and instantiates the queued component.

HRESULT NumberClass::Load(IStream *pIStream) {

    ULONG lBytesRead;

    // Read the collection of numbers from the stream.  From the
    // Save method (below), we know that all 10000 array elements
    // were written to the stream regardless of how many of them
    // contain valid information (that information is kept
    // in the p_Numbers variable).

    for (int i=0; i < 10000; i++) {

        pIStream->Read((void*)&p_NumberCollection[i],
                       sizeof(p_NumberCollection[i]),&lBytesRead);

        if (lBytesRead != sizeof(p_NumberCollection[i])) {
            // For some reason, we didn't read proper number of
            // bytes from the stream:
            return E_FAIL;
        }

    }

    pIStream->Read((void*)&p_Numbers,sizeof(p_Numbers),&lBytesRead);
    if (lBytesRead != sizeof(p_Numbers)) return E_FAIL;

    pIStream->Read((void*)&p_Sum,sizeof(p_Sum),&lBytesRead);
    if (lBytesRead!= sizeof(p_Sum)) return E_FAIL;
```

continues

Listing 10.8 **Continued**

```
    return S_OK;
}


// The Save method is called when COM wants an object to
// save its state to the stream. If this component were
// passed as a method to a queued component, this method
// would be called when COM+ packages method calls made
// against the component into an MSMQ message — (most likely)
// when the application closes.

HRESULT NumberClass::Save(IStream *pIStream, BOOL fClearDirty) {

    ULONG           lBytesRead;
    HRESULT         hr;


    // This objects state consists of:
    //
    // 1.) The 10000 element array
    // 2.) The p_Numbers variable indicating how many #'s are in the collection
    // 3.) The p_Sum variable which keeps a running Sum of the numbers.

    for (int i=0; i < 10000; i++) {

        pIStream->Write((void*)&p_NumberCollection[i],
                        sizeof(p_NumberCollection[i]),&lBytesRead);

        if (lBytesRead != sizeof(p_NumberCollection[i])) {
            // For some reason we couldn't write the # of bytes
            // we wanted to the stream:
            return STG_E_CANTSAVE ;
        }

    }

    pIStream->Write((void*)&p_Numbers,sizeof(p_Numbers),&lBytesRead);
    if (lBytesRead != sizeof(p_Numbers)) return STG_E_CANTSAVE;

    pIStream->Write((void*)&p_Sum,sizeof(p_Sum),&lBytesRead);
    if (lBytesRead != sizeof(p_Sum)) return STG_E_CANTSAVE;

    return S_OK;

}


// GetSizeMax is called by COM to determine the maximum amount
// of bytes the object will need to save its state.  This method
```

```
// is commonly called before the Save method so COM can reserve
// space in the stream before an objects stores its state to the stream.

HRESULT NumberClass::GetSizeMax(ULARGE_INTEGER *pcbSize) {

    // The amount of space to save this object's state is determined
    // by the private member variables of our class.  These include:

    // Array of 10000 elements + p_Numbers + p_Sum.

    pcbSize->QuadPart = (DWORD)(sizeof(p_NumberCollection[0])
                        *10000+sizeof(p_Numbers)+sizeof(p_Sum));
    return S_OK;

}
```

Visual Basic's support for persistable components is far easier to use, if less versatile. First, your must set the VB class's `Persistable` property to `1` as shown in Figure 10.11. After this is done, the VB programmer is responsible for implementing two new methods as shown in Listing 10.9.

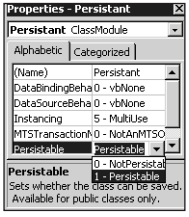


Figure 10.11 Making an object persistable in Visual Basic 6.

Listing 10.9 **A Persistable Component Written in Visual Basic 6**

```
'An example of a persistable component written in Visual Basic.
'This is a new feature provided by Visual Basic 6.

'This component implements a simple object that provides
'statistical information for a collection of numbers.
'The state of the object is thus the collection of numbers, stored
'in the private class array. The visual basic example does not have
'the 10000 collection limit of its C++ counterpart.

Option Explicit

'Private member variables:
Private m_vntNumberCollection As Variant
```

continues

Listing 10.9 **Continued**

```
Private m_vntNumbers As Variant
Private m_dSum As Double

'Class constructor. Initialize private class variables:
Private Sub Class_Initialize()
    m_vntNumbers = 0
    m_dSum = 0
    ReDim m_vntNumberCollection(0)
End Sub

'AddNumber allows the user to add a number to the collection.
'Redimension the array to accomodate for this newly added number
Sub AddNumber(ByVal Number As Double)
    m_vntNumbers = m_vntNumbers + 1
    ReDim m_vntNumberCollection(m_vntNumbers)
    m_vntNumberCollection(m_vntNumbers) = Number
    m_dSum = m_dSum + Number
End Sub

'Average returns the average of all the numbers in the collection.
'Since we keep a running total of the sum of the numbers this is easy:
Function Average() As Double
    Average = m_dSum / m_vntNumbers
End Function

'Sum returns the Sum of all the numbers in the collection:
Function Sum() As Double
    Sum = m_dSum
End Function

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'Persistable objects in Visual Basic must implement the
'following two procedures:

'The ReadProperties procedure in Visual Basic is like
'the IPersistStream::Load method in C++. It allows
'an object to restore its state from disk.

Private Sub Class_ReadProperties(PropBag As PropertyBag)
    Dim iNumber As Integer

    'The property bag where we write the object's state to
    '(much like the stream in C++):
    m_vntNumbers = PropBag.ReadProperty("Numbers")
    m_dSum = PropBag.ReadProperty("Sum")

    'Dimension the array to the amount of numbers
    'that were previously saved:
```

```
    ReDim m_vntNumberCollection(m_vntNumbers)

    'Read in each element of the array:
    For iNumber = 0 To m_vntNumbers - 1
        m_vntNumberCollection(iNumber) = PropBag.ReadProperty("Number" & _
                                                              iNumber)
    Next

End Sub

'The WriteProperties procedure in Visual Basic is like
'the IPersistStream::Save method in C++. It allows
'an object to store its state from disk.

Private Sub Class_WriteProperties(PropBag As PropertyBag)
    Dim iNumber As Integer

    PropBag.WriteProperty "Numbers", m_vntNumbers
    PropBag.WriteProperty "Sum", m_dSum

    'The property bag CANNOT write arrays, so we must iterate
    'through it ourselves:
    For iNumber = 0 To m_vntNumbers - 1
        PropBag.WriteProperty "Number" & iNumber, m_vntNumberCollection(iNumber)
    Next

End Sub
```

Notification and Callbacks

By default, you can never be sure that a method call made to a queueable object will ultimately be processed by that object. Although it is true that you can be sure that the message will be placed in a queue and queued for processing (provided the object's application is using transactional queues), you cannot be certain that the object's application will ever be run or that, if running, it is listening for new messages.

Certain mission-critical applications generate messages like `BUY` or `SELL` that need some form of acknowledgement. The acknowledgement might also need to contain additional information like what an item was sold or purchased for. Fortunately, QC provides for such a mechanism in the form of *callbacks*.

As convenient as it might seem, you cannot simply instantiate an object locally on the sending machine and pass an interface via a method call argument to a QC object. As we've discussed, by the time the recipient even gets the method call, the `callback` object probably has perished along with the sender. Persistent objects don't help you either—the recipient is getting a completely new instantiation of the object that would run locally. The QC framework does, however, provide a callback solution. It is allowable for a queueable object to exist on the sender's machine and, after being instantiated via a `queue` moniker, can be passed as an argument to the recipient object. In other words, you can pass a Recorder as an argument to another Recorder.

There is a kind of duality involved in dealing with recorders in that you sometimes treat them as living callback objects and other times as references to objects that will be instantiated some time in the future. The latter treatment is closer to the truth, but the former is more easily understood. It might even be fair to say that you shouldn't care about the fact that you get a Recorder and not a real interface to a queued object. Due to the declarative nature of COM+, it is certainly true that your code should never make the assumption to write your server's objects as if they really are getting true interface pointers and not Recorders. It is COM+'s job to make sure everything works.

Writing an interface with a method that can take a callback object as an argument is simple. For the purpose of example, imagine that you have two queueable objects, `CallBackObject` and `ServerCallback`. The former object belongs to the client and provides the client with callback notifications from `ServerCallback`. An interface of `CallBackObject` (a Recorder) is passed as an argument to a method of `ServerCallback` so that `ServerCallback` can use it to send acknowledgements back to the client. The VB code in Listing 10.10 demonstrates how to implement `ServerCallback`.

Listing 10.10 **Implementation of *ServerCallback* ActiveX Class**

```
'Remember the ByVal keyword is important!
Sub ServerMethod(ByVal CallBackObject As CallBack.CallBackObj, ByVal Message)
    On Error GoTo problem
    MsgBox "Server received the following message: " & Message
    CallBackObject.CallBackMethod Message

    Exit Sub
problem:
    MsgBox Err.Description & " - " & Hex$(Err.Number)

End Sub
```

Assume the subroutine in Listing 10.10 is located in a VB class called `ServerCallback` in an ActiveX DLL project. Remember that in VB, a class is really an interface. So according to VB's conventions, there is a coclass called `ServerCallback` that supports an interface called `CallBackObject`. The following is the generated IDL:

```
interface _ServerCallback : IDispatch {
        [id(0x60030000)]
        HRESULT ServerMethod(
                        [in] _CallBackObj* CallBackObject,
                        [in] BSTR SomeString);
    };
```

If a client application executes the code in Listing 10.11, the client is, in effect, creating two Recorder objects—one representing the `ServerCallback` object and another representing `CallBackObject` on the client machine.

Listing 10.11 **Client-Side Code Demonstrating Callbacks Using *CallBackObj***

```
Dim callBck As callback.CallBackObj
Dim srvr As server.ServerCallback

'Use the queue moniker to get a recorder to the local client component:
Set callBck = GetObject("queue:/new:Callback.CallBackObj")

'Use the queue moniker to get a recorder which points the remote machine:
Set srvr = GetObject("queue:CompuerName=Sajid,QueueName=Gregory" & _
                     "/new:Server.ServerCallBack")


'Call a method
srvr.ServerMethod callBck, "This is a message for the server!"
```

Because `CallBackObject` has one method, `ServerMethod()`, that can take a `CallBackObject` interface as an argument, the client can send its local `CallBackObject` to the server, which can use it to send notifications back to the client.

You might imagine that passing a Recorder is like passing a persistable object, but it isn't. When you pass a Recorder, you are really passing little more than a moniker string that tells the recipient what object in what location to create. For example, if you examine the method/message recorded to the queue for the `ServerMethod` method call, you will find something like the following:

```
computername=MSMQHEAD,queuename=Gregory,formatname=PUBLIC=92df4d1c-a62a-48c0-ae43-
➥690e9d46a5cc,authlevel=1Æ¯«ì_Ò_—Zøu~*V_∫π˝AyI‰*'( ¯÷t(Callback.CallBackObj)
```

Additional information might be automatically added to the moniker string by COM+, such as the sender's queue's GUID if public queues are used. You are not, however, passing an object interface or Recorder, just a string and a little extra contextual information. The recipient uses this string and automatically instantiates a new Recorder back to the sender.

In that case, the following client code, although seemingly correct, does not work unless `CallBackObj` is persistable:

```
Set c = new CallBack.CallBackObj 'Won't work, did not create with a queue moniker
➥s.ServerMethod c, "This is a message for the server!"
```

Of course, if it is persistable, a new instance of `CallBackObj` is instantiated on the recipient machine. But this is not what you want because this new object has no ties back to the client, thus invalidating its use as a callback mechanism. So, although they look syntactically similar, passing a Recorder and passing an object interface are very different things.

One final point worth mentioning is that the client might want callbacks to be sent, not to itself, but to another machine altogether. The client can create a Recorder via a `queue` moniker that points to a callback object on another machine and pass that to some recipient. There is no rule that says the callback object must be on the same

machine as the original sender. The asynchronous nature of QC grants an enormous amount of flexibility in terms of acknowledgements because you are not passing interfaces, you are passing locations, addresses in the form of a string.

Exception Classes

Callbacks and acknowledgements are one thing, but failure is another. If you want to be explicitly notified when a queued method call fails to be delivered, you need only declare an *exception class*. The Properties dialog box of any COM+ component has a tab labeled Advanced. It looks like the one shown in Figure 10.12.

Type the ProgID (or ClassID) of an object that you want COM+ to call in the event of a failed message delivery. Most commonly, you set this property on both the sending and receiving applications on different machines. This is because COM+ triggers the exception class on the sending machine if it cannot reach the destination machine at all (perhaps the network is down); it triggers the exception on the destination machine if it can reach the destination, but ultimately it cannot play the message for the recipient.

In the event of a failed delivery, COM+ instantiates your exception class and then QIs for an interface called `IPlayBackControl`. If your object implements this interface, COM+ calls one of its two methods, which follow:

- `FinalClientRetry`. COM+ calls this method on the exception object on the sending client if the message cannot be delivered to the server at all. This most likely occurs if the server is down or otherwise unreachable. Although this method has no arguments, upon receiving this call, a client knows to check its dead queue.
- `FinalServerRetry`. COM+ calls this method on your exception object on the receiving server after repeated attempts to play the message fail. This method is called when the failed message enters the dead message queue for the application.

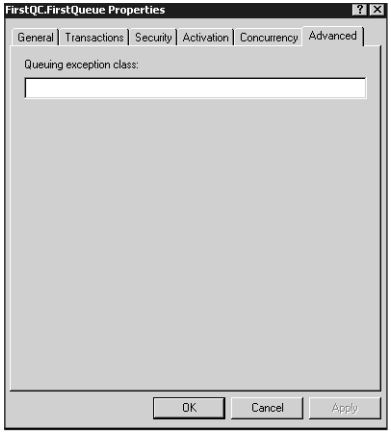


Figure 10.12 The Advanced tab.

If you don't feel like peeking into a queue and analyzing the failed message, you can request that COM+ play the failed method call to your exception class. All your exception class needs to do is simply implement the queued interface containing the failed method call in question. In other words, you can write your exception class such that it is, in a sense, the recipient object's evil twin; if the method call fails to be played to the recipient object, then your exception class will receive the method call instead, just as if it were the originally targeted object. Of course, by receiving method calls originally slated for the recipient object, your implementation of these method calls would probably do little more than announce "This function was attempted but failed!"

For example, imagine some interface, `IExample`, is declared queued in some recipient object, and it has one method called `Test()`. If a client calls `IExample->Test()`, and the message resulting from that call fails to successfully play for the recipient, COM+ instantiates the event class you established for that object, and after QIing for `IPlaybackControl`, it then QIs for `IExample`. Predictably, the Player for your exception class invokes the `Test()` method on this interface of your exception class. So, when authoring such an event class and implementing `IExample`, you can write code to take whatever actions you deem appropriate given that you know exactly which method(s) failed to be delivered to the recipient.

An event class might support the `IPlaybackControl`, the queueable interface of the recipient, or both.

Callbacks and Events

At this point, you might be envisioning schemes that allow the client executable to receive acknowledgements while it is running. Maybe you're thinking about writing a trade-entry front end that allows the user to submit a trade via QC and soon after get back an acknowledgement that perhaps populates some control on the currently running form. For this type of application, however, COM+ *events* are better suited. As you see in the next chapter, COM+ events can use either synchronous Distributed COM (DCOM) or MSMQ as an asynchronous transport but provide a generic, extensible architecture for real-time acknowledgements.

Some Subtle Requirements for QC: Parallel Application Configurations Required for Sender and Receiver

Originating with terminals and mainframes, the idea of thin clients is once again in vogue (that is, after a brief interlude with client/server in the early 1990s). Its maxim: Configure the server to handle the work, not the client.

In QC, you would expect the recipient machine receiving a queued method call to have a configured COM+ application, complete with queues, in which the destination component had been imported. You would further expect the interfaces of that component to be marked as Queued. It is surprising then to learn that the client must also have a fully configured COM+ application, complete with queues, in which the destination component is likewise locally installed and configured.

This kind of forced symmetry seems analogous to installing a brick oven in your home and donning an apron to order out for pizza. According to the rules of traditional COM, a client machine need only have the type library for the server object registered locally (and even then, only if the client wants to use early binding instead of late binding). Requiring an entire configured application complete with queues, however, seems a bit much. In addition, if the recipient application's queues are transactional, your client-side queues need to be transactional as well. And if you imagine that the sender's local queues are somehow used to temporarily stage a message before sending it to the recipient queues, you will be disappointed—the local queues on the sender's machine are not used.

At the time of this book's writing, Microsoft documentation doesn't specifically mention this client-side symmetry as a requirement. However, empirical research indicates this symmetry is necessary; it seems likely that COM+ uses the locally configured application, queue, and component settings to determine exactly how a message should be sent to a recipient. If, for example, the recipient application's queues are transactional, the sender must know this so that it adds transaction information as part of the message when sending it. The existence of similarly configured local queues can give the sender this information, even if they are not otherwise used. If your client COM+ application is configured to use non-transactional queues but the recipient application uses transactional queues, your QC method call will appear to work. However, if you check the event log, you will find that the `QC.Recorder` logs an error message like that shown in Listing 10.12.

Listing 10.12 **Error Message Reported by the Recorder When the Locally Configured Queue of the Client Application is Not Transactional but the Recipient Queue is Transactional**

```
An unexpected error was returned by the MSMQ API function indicated. The following
➥error message was retrieved from MSMQ.
MQSendMessage : Transaction usage is invalid.
Server Application ID: {E3D43E4D-D73C-4AC1-8318-36DF1BEFDF69}
Server Application Name: shannon
Error Code = 0xc00e0050 :
COM+ Services Internals Information:
File: .\channelmanager.cpp, Line: 388
```

Forgetting about queues for the moment, a locally configured COM+ application also makes it possible to specify exception classes on the sender, as well as the receiver. This requires a configured, sender-side COM+ application.

Asynchronous COM

Sometimes two (or more) different technologies originate to solve the same problem, but they take different evolutionary paths. Eventually, one might dominate and force the other into obscurity and, perhaps, kill it altogether. This is perhaps the case with QC and another technology *Asynchronous COM*. However, regardless of what the future brings or whichever technology wins out, QC and Asynchronous COM are both around—documented and clamoring for use at present. So a discussion of Asynchronous COM is helpful if only so you recognize it when you see it described or in use, and you are able to distinguish it from QC.

QC is a native COM+ service. QCs can leverage MSMQ transparently to make asynchronous method calls and can utilize other COM+ services such as transactions and role-based security. Asynchronous COM, though far more complicated to implement, can do almost none of these things except provide for asynchronous method calls. Asynchronous COM components cannot even be configured COM+ components, so they can't use any COM+ services.

Asynchronous COM Implementation

If you think of QC as an inherent COM+ service and Asynchronous COM as an RPC enhancement, you are not far from the truth. Simply put, Asynchronous COM begins at the IDL level. To demonstrate, imagine you have the IDL shown in Listing 10.13, but with the additional attribute shown in bold.

Listing 10.13 **Simple IDL Declaration of an Interface with an *async_uuid* Attribute**

```
 [
object,
uuid(8183C0C6-8AB3-4445-9D31-D2F7A575F7FB),
helpstring("ITester Interface"),
async_uuid(340C09D1-E06B-41f3-85BD-70FF4D205807),
pointer_default(unique)
]
interface ITester : IUnknown
{
[helpstring("method Info")] HRESULT TestMethod([in] int x, [out] int *  y);

};
```

The `async_uuid` attribute causes an additional interface prototype to be created when MIDL.EXE is run on this IDL file. If you examine the resulting header file, you find that shown in Listing 10.14.

Listing 10.14 **MIDL-Generated Prototype for *AsyncITester***

```
MIDL_INTERFACE("340C09D1-E06B-41f3-85BD-70FF4D205807")
AsyncITester : public IUnknown
    {
public:
virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE Begin_TestMethod(
/* [in] */ int x) = 0;
virtual /* [helpstring] */ HRESULT STDMETHODCALLTYPE Finish_TestMethod( /* [out]
➥*/  int
 __RPC_FAR *y) = 0;
    };
```

MIDL has created an entirely new interface, one not specified in the IDL file. This new interface has the same name as the original, `ITester`, but is prepended with the tag `Async:` result `AsynchITester`. An Asynchronous COM client must use this interface if it wants to make method calls asynchronously, but as you see in a few paragraphs, the client does not QI for it.

You might also notice that the single method, `TestMethod()`, which contained both an `in` and an `out` parameter has been split into two methods, `Begin_TestMethod()` and `Finish_TestMethod()`.

The `Begin` method contains all the `in` parameters. The Asynchronous COM client calls this method, and it returns immediately, *asynchronously,* allowing the client to continue with other work. Some time later, however, the client needs to call the `Finish` method (which contains all the `out` parameters) unless it wants to cancel the call altogether. Obviously, the client needs to know when the server is finished, so the server supports an interface called `ISynchronize` which the client can use to determine when the server is finished.

Clearly, Asynchronous COM imposes a more complex protocol that the developer of the client and server object must adhere to. The server author, in particular, has to support the `ISynchronize` interface, and it normally does so by aggregating one of the COM system synchronization objects, `CLSID_StdEvent` or `CLSID_ManualResetEvent`. Don't look for these objects in the Registry, and you won't find them especially well-described in Microsoft documentation (at least at the time of this writing); they exist entirely in the COM sub-system. The platform SDK demonstrates the use of these objects in the context of implementing an Asynchronous COM server object, so you might want to check out the Async examples that come with the SDK if you are interested in greater detail.

You might imagine, then, that all the client needs to do is create an instance of the object, QI for `AsynchITester` and `ISynchronize`, and then call the `Begin` method to execute the `async` function call. After the `async` call is made, the client should use the methods of `ISynchronize` to determine when the server is finished and then call `Finish` when `ISynchronize` says it is okay to retrieve any `out` parameters.

Well, there is one additional step. The server object must also support an interface called `ICallFactory` that is used to create the `call` object. It is through the `call` object that the client must get its `AsynchITester` interface. Examine Listing 10.15.

Listing 10.15 **Client-Side Code Using Asynchronous COM**

```
ISynchronize * pSync;
ICallFactory * pCallF;
AsyncITester * pAsynchTest;
ITester * pTester;
int result;

//Not shown: COM Inititalization

CoCreateInstance(CLSID_Test, NULL, IID_ITester, (void **)&pTester);
pTester->QueryInterface(IID_ICallFactory, (PVOID*)&pCallF);
pCallF->CreateCall(IID_AsyncITest, NULL, IID_AsyncITester, (LPUNKNOWN*)
➥&pAsyncTest);
pAsyncTest->Begin_ TestMethod(1);
pAsyncTest->QueryInterface(IID_ISynchronize, reinterpret_cast<void**>(&pSync));

while ((hr = pSync->Wait(0, 1000)) == RPC_S_CALLPENDING)
    {
        //do anything you like while the server is still processing
    }

hr = pAsyncTest->Finish_TestMethod(&result);

//Not Shown: Release all interfaces
```

Asynchronous COM Drawbacks

The complexity outlined in the previous section makes it difficult to use Asynchronous COM with clients not written in C++. When compared with the simplicity of marking an interface as Queued and creating an object through a moniker, Asynchronous COM doesn't have a lot going for it. Add to this the following:

- Asynchronous COM does not work with interfaces that inherit from `IDispatch`. Therefore, Asynchronous COM cannot be used with late binding or implemented by an object supporting only dual interfaces. This precludes its use from scripting environments, such as ASP.
- Asynchronous COM does not work with configured components—that is, components that are part of a COM+ application. Thus, Asynchronous COM servers cannot take part in transactions, role-based security, or any other COM+ service. This incompatibility between Asynchronous COM and COM+ was announced at the Microsoft Professional Developers Conference (PDC 1999), and at the time of this writing, no plans have been announced to make Asynchronous COM COM+-compatible.

When you take into account the preceding points, it is hard to make a compelling argument for the use of Asynchronous COM. Given the existence of QC, it is puzzling to contemplate why Asynchronous COM was introduced at all. My feeling is that the Asynchronous COM paradigm fits into the DCOM camp. DCOM was an evolutionary step that allowed COM to cross network boundaries via RPC, but the considerations that weigh on DCOM developers (mainly, writing their own out-of-process servers and devising their own services) no longer concern a COM+ developer. So, where Asynchronous COM is attractive to the DCOM developer, it might be considered redundant and needlessly complex by the COM+ developer.

You can go further and claim that Asynchronous COM stands in direct contradiction to the COM+ declarative model. Components should not explicitly be written to support specific services; rather, a system administrator should be able to declare his intent to use them. But I've been hard enough on Asynchronous COM. I will not go into greater detail on Asynchronous COM because it is outside of COM+. Keep in mind that it is there, and now that you know its capabilities and limitations, you can be the judge regarding if and when it should be used.

Summary

Traditional COM uses synchronous RPC to connect client applications and running objects. This mechanism demands that both the client and server object are running at the same time; it has no provision for recovering from failed method calls, and a poorly coded server can cause the calling client to hang.

QC removes the limitations inherent in RPC by using MSMQ as a transport for method calls. By marking an interface as Queued, a client that obtains this interface via a queue moniker gets a special proxy for the recipient object called a Recorder. From the client's perspective, the Recorder appears to be the actual object, yet all method calls return immediately. COM+ translates the method calls into MSMQ messages and places them in the message queues of the recipient object's application. When the recipient application is next run (and if it is configured to listen), COM+ creates an object called the Player, handing it the messages. The Player then instantiates the recipient object (if it is not already running), translates the messages back into method calls, and invokes them on the object, security permitting.