KIMMEL.book Page 221 Thursday, June 12, 2003 8:37 AM

CHAPTER 8

# Remoting

*The power of the passions, the force of the will,*
*the creative energy of the imagination, these make life.*

*—The Princess of Tivoli (Disraeli)*
*in Paul Smith,* Disraeli: A Brief Life

## Introduction

One of the splendid miseries of writing computer books is that you have to go out and get some practical experience about the subject. Many times this is possible, but occasionally it is not. Remoting is one of the few areas in this book where my experience is limited to nonproduction code, that is, sample applications. As I am writing this my colleagues and I are deliberating whether or not to use Web Services or .NET Remoting on a current project. Because we will unlikely be able to deploy clients or servers on the other end of the problem, Web Services will probably win. This brings us to the subject at hand. What is .NET Remoting and why should you care?

Remoting is the technology used to get two applications to interchange data. It is in the same category of problem and solution as CORBA and DCOM. Clearly Microsoft thinks that .NET Remoting is a best-of-category solution—otherwise, why would we need something other than DCOM?

To help you understand .NET Remoting in a general sense, I will share a story with you that perhaps will help you understand the concept behind .NET Remoting and its technology. Afterward I will include several examples that deal with the nuts and bolts of remoting.

## Understanding .NET Remoting

Around the year my youngest son was born—about 1996—I was working on a big project in Chicago. The problem was to manage information related to tracking labor in North America, which included Canada and Mexico. If you recall, around that time we had whopping speeds of 9,600 baud on dial-up connections. Our applications had to process huge amounts of data through tiny connections over wide areas.

The application was to be implemented in Delphi, C, and DB2. At that time C was used for stored procedures, IBM's Universal database was still called DB2, and there were no ODBC drivers for DB2. We had to write everything: stored procedures in C, client software for Windows, server software, and connections to DB2 servers using a Software Development Kit (SDK) from IBM. In essence we had to figure out how to write a rough supplement for an absent connectivity layer to solve the business problem, and we had to write a distributed application for Windows.

My smart friend Andrew Wozniewicz came up with a nice solution that people had difficulty understanding but that worked in practice. The solution was to define abstract classes (remember that this was pretty early in COM's history, and I don't recall that DCOM was a choice) and then share those abstract classes on client and server. Implementations would exist only on the server. The client would declare variables using the abstract classes, and a factory method would return an actual object to the abstract variable definition. Hence, we implemented thin clients containing only abstract classes and fatter servers containing implementations for those classes. From the server to the database server we used what Andrew referred to as "amorphous blobs of data." These amorphous blobs represented data that we created in a predetermined format to which the server application and server applications on the database server had agreed. This is pretty good multitier architecture considering the state of technology at the time. I am familiar with all of this because I implemented the proof-of-concept vertical slice from Andrew's description.

When we were finished we had a client with abstract classes, a middle-tier business layer with abstract classes, and completely implemented child classes in the middle tier, which sent amorphous blobs of binary data to the database server. On the database server the blobs were unpacked and the DB2 SDK was used to invoke stored procedures.

To Andrew's credit he figured some of this out by gleaning how Delphi's very advanced IDE—at the time—worked with the Visual Control Library (VCL) at design time to get controls from the designer onto a form. As I mentioned in Chapter 3, Anders Hejlsberg was instrumental to implementing Borland's Delphi and Microsoft's .NET. History provides perspective, and it is very likely that some ideas in .NET evolved from Anders' dozen or so years at Borland.

The problem Andrew and I and many others had still exists. How do we get applications to share data on a LAN or WAN? Worse, the problem is exacerbated now because the network includes the highly distributed, heterogeneous Internet. In addition to having to send data between client and server on a LAN and WAN, now programmers are expected to send data between multiple clients and multiple servers, potentially running different operating systems and different language-based implementations, across every kind of network. .NET Remoting is a solution to the problem of implementing highly distributed applications.

Sadly, Andrew's amorphous blobs aren't completely sufficient as a public standard. What Microsoft has done is to allow us to define interfaces or abstract classes on the client and implement classes on the server, and instead of amorphous blobs we get XML and SOAP. As open standards, XML and SOAP can be deciphered by any platform. In addition, Microsoft's .NET Remoting technology takes care of packaging the XML and SOAP blob back and forth—called *marshaling*—for us. For the most part we only have to worry about writing the business solution; .NET takes care of the infrastructure.

Of course, as is true with any subject, if you dig deep enough you can start customizing and extending the provided behavior. However, the ultimate end result is that .NET Remoting was defined to support a highly distributed world of TCP and HTTP networks by building on open standards and hiding the most difficult aspects of managing connections, marshaling data, and reading and writing XML and SOAP. As a result, if you can understand inheritance, declare and implement an interface, and use attributes, you are ready to begin using .NET Remoting.

## Marshaling Objects by Reference

Remoting handles data between client and server in two ways: (1) marshaling the data by reference and (2) marshaling the data by value. Marshaling by reference is analogous to having a pointer, and marshaling by value is analogous to having a copy. If we change a reference object, the original is changed, and changes to a copy have no effect on the original. To get your feet wet let's start with a quick marshal-by-reference example. (The Marshaling Objects by Value section later in this chapter talks about the other way data is moved back and forth.)

**NOTE:** Occasionally I will be accused of writing or saying something condescending. That is never my intent. That said, depending on your level of comfort with technical jargon, words like *marshal* may sound ominous. This is an advanced book, but if you are not comfortable with COM and DCOM, the word *marshal* may trouble you. An easier term might be *shepherd*, as in herding sheep. Because remoting moves data across a network, the data must be packaged and unpackaged in an agreed-upon format and shepherded between the application that has the data (the server) and the application that wants the data (the client).

Several graphics on the Web depict this relationship—marshaling between client and server—but I am not sure if they aid understanding or add to confusion. Rather than repeat those images, I encourage you to think of the code that does the shepherding as the responsibility of .NET. These codified shepherds are referred to as *proxies*. The `Remoting` namespace contains the proxy code.

## Hello, Remote World!

Rather than torture you with another Hello, World! application, I will use a sample application with a little more meat (not much but a bit more).

Suppose you work in the information technology department of a large insurance company. This company owns several broker dealers that sell mutual funds. As a result you are tasked with tracking all customer purchases of mutual funds, life and health products, and annuities. You can cobble together a solution that requires the remote broker dealer offices to run batch programs at night that upload data and combine the mutual fund trades with Universal Life payments, mixing and matching the client PC's database programs with your UDB, SQL Server, or Oracle databases. When you are finished you have VB6 applications on the client workstations running ObjectRexx dial-up scripts to FTP servers late at night. Or, you can use remoting and .NET to get everybody working together. Throw out the Perl, `.cmd`, `.bat`, and `.ftp` scripts; toss the various and sundry import and export utilities written in C, VB6, and Modula; and get everything working in real time.

Okay. We won't have enough time to tackle all of that in this section, but we can create a client application that requests a customer and a server application that simulates servicing that request. Because the code would take up a lot of space, we will simulate the client reading from the database. However, after you read Chapters 11, 12, and 16 on ADO.NET, you will be able to incorporate the code to read from the database too. Figure 8.1 shows a UML model of the design we will be using here. (I used Rational XDE, integrated into .NET, to create the UML class diagram.)

The class diagram accurately depicts the code that resides in the client and server. An assembly named `Interface` contains the two interfaces: `IFactory` and `ITrade`. The assembly named `Server` implements (*realizes* in the vernacular of the UML) `IFactory` and `ITrade` in `Factory` and `Trade`, respectively, and the assembly named `Client` is dependent on the two interfaces. Note that there is no dependency on the actual implementations of `IFactory` and `ITrade` in `Client`. If all the code on the server were on the client, then arguably the server would not be needed. (This isn't precisely true but logically makes sense.) Listings 8.1 and 8.2 contain the code for the `Interface` and `Server` assemblies, in that order.

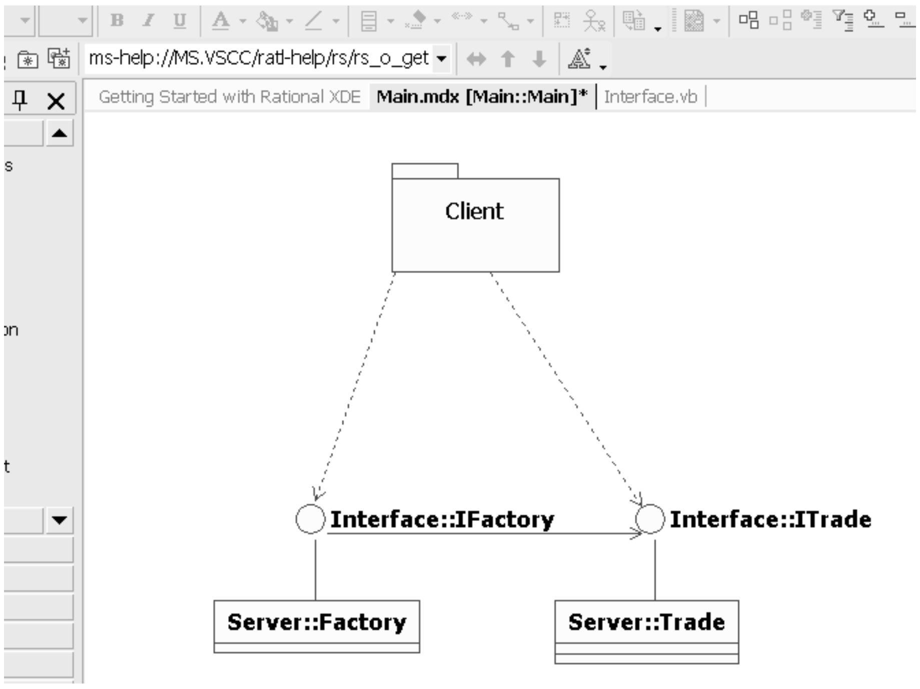


**Figure 8.1** The class diagram for our server application.

**Listing 8.1** The `Interface.vb` File Containing the `IFactory` and `ITrade` Interfaces

```
Public Interface IFactory

  Function GetTrade(ByVal customerId As Integer) As ITrade

End Interface

Public Interface ITrade

  Property NumberOfShares() As Double
  Property EquityName() As String
  Property EquityPrice() As Double
  ReadOnly Property Cost() As Double
  Property Commission() As Double
  Property RepId() As String

End Interface
```

**Listing 8.2** The `ServerCode.vb` File Containing the Implementation of `ITrade` and `IFactory`

```
Imports System
Imports [Interface]
Imports System.Reflection

Public Class Factory
  Inherits MarshalByRefObject
  Implements IFactory

  Public Function GetTrade( _
    ByVal customerId As Integer) As ITrade _
    Implements IFactory.GetTrade
    Console.WriteLine("Factory.GetTrade called")

    Dim trade As Trade = New Trade()
    trade.Commission = 25
    trade.EquityName = "DYN"
    trade.EquityPrice = 2.22
    trade.NumberOfShares = 1000
    trade.RepId = "999"
    Return trade
  End Function

End Class

Public Class Trade
  Inherits MarshalByRefObject
  Implements ITrade

  Private FCustomerId As Integer
  Private FNumberOfShares As Double
  Private FEquityName As String
  Private FEquityPrice As Double
  Private FCommission As Double
  Private FRepId As String

  Public Property NumberOfShares() As Double _
    Implements ITrade.NumberOfShares
  Get
    Return FNumberOfShares
  End Get
  Set(ByVal Value As Double)
    FNumberOfShares = Value
  End Set
  End Property

  Public Property EquityName() As String _
    Implements ITrade.EquityName
  Get
    Return FEquityName
  End Get
  Set(ByVal Value As String)
    Console.WriteLine("EquityName was {0}", FEquityName)
    FEquityName = Value
    Console.WriteLine("EquityName is {0}", FEquityName)
    Console.WriteLine([Assembly].GetExecutingAssembly().FullName)

  End Set
  End Property

  Public Property EquityPrice() As Double _
    Implements ITrade.EquityPrice
  Get
    Return FEquityPrice
  End Get
  Set(ByVal Value As Double)
    FEquityPrice = Value
  End Set
  End Property

  ReadOnly Property Cost() As Double _
    Implements ITrade.Cost
  Get
    Return FEquityPrice * _
      FNumberOfShares + FCommission
  End Get
  End Property

  Property Commission() As Double _
    Implements ITrade.Commission
  Get
    Return FCommission
  End Get
  Set(ByVal Value As Double)
    FCommission = Value
  End Set
  End Property

  Property RepId() As String _
    Implements ITrade.RepId
  Get
    Return FRepId
  End Get
  Set(ByVal Value As String)
    FRepId = Value
  End Set
  End Property

End Class
```

The code in both listings is pretty straightforward. Listing 8.1 defines the two interfaces `IFactory` and `ITrade`. Listing 8.2 provides an implementation for each of these interfaces.

After scanning the code you might assume that all we need to do is add a reference in the client to each of the two assemblies containing the code in Listings 8.1 and 8.2 and we're finished. And you'd be right if we were building a single application. However, we are building two applications: client and server.

Suppose for a moment that we did add a reference to the `Interface` and `Server` assemblies. .NET would load all three assemblies—client, interface, and server—into the same application domain (`AppDomain`), and the client could create `Trade` and `Factory` objects directly or by using the interfaces. This is a valid model of programming, but it is not distributed. It works because .NET uses `AppDomain` for application isolation. All referenced assemblies run in the same `AppDomain`. However, when we run a client application and a separate server, we have two applications, each running in its own `AppDomain`. .NET Remoting helps us get data across application domains.

In our distributed example, `Client.exe` is an executable with a reference to `Interface.dll`. Both of these assemblies run in the `AppDomain` for `Client.exe`. `Server.exe` also has a reference to `Interface.dll`, and `Server.exe` and `Interface.dll` run in the `AppDomain` for `Server.exe`. The code we have yet to add is the code that creates the object on the client by making a remote request to the server.

## Getting Client and Server Talking

Thus far we have written vanilla interface and class code. To get the client and server talking we have to use some code in the `System.Runtime.Remoting` namespace. The first step is to inherit from `MarshalByRefObject`. Listing 8.2 shows that both `Factory` and `Trade` inherit from `MarshalByRefObject`, which enables the classes to talk across application boundaries. The second piece of the puzzle is to tell the server to start listening, permitting the client to start making requests.

Listing 8.3 contains the code that instructs the server to start listening, and Listing 8.4 contains the code to get the client to start making requests. Both client and server are implemented as console applications (`.exe`) for simplicity. You can use .NET Remoting with a variety of hosting styles. (Refer to the Choosing a Host for Your Server subsection near the end of this chapter for more information.)

**Listing 8.3** Telling the Server Application to Begin Listening for Requests

```
1:  Imports System.Runtime.Remoting
2:  Imports System.Runtime.Remoting.Channels
3:  Imports System.Runtime.Remoting.Channels.Http
4:
5:  Public Class Main
```

```
6:
7:    Public Shared Sub Main(ByVal args() As String)
8:
9:      Dim channel As HttpChannel = New HttpChannel(9999)
10:     ChannelServices.RegisterChannel(channel)
11:     RemotingConfiguration.RegisterWellKnownServiceType( _
12:       GetType(Factory), "Factory.soap", _
13:       WellKnownObjectMode.Singleton)
14:
15:     RemotingConfiguration.RegisterWellKnownServiceType( _
16:       GetType(Trade), "Trade.soap", _
17:       WellKnownObjectMode.Singleton)
18:
19:     Console.WriteLine("Server is running...")
20:     Console.ReadLine()
21:     Console.WriteLine("Server is shutting down...")
22:   End Sub
23:
24: End Class
```

From the code and the shared `Main` method you can tell that Listing 8.3 comes from a .NET console application. Lines 1 through 3 import namespaces relevant to remoting.

The first thing we need to do is declare a channel. I elected to use the HTTP protocol, and the `HttpChannel` constructor takes a port number. This is the port number on which the server will listen. If you want the server to automatically choose an available port, send `0` to the `HttpChannel` constructor. There are about 65,500 ports. If you want to specify a port number, just avoid obvious ports that are already in use like 80 (Web server), 23 (Telnet), 20 and 21 (FTP), and 25 (mail). Picking a port that is being used by another application will yield undesirable results. After we have elected a channel we need to call the shared method `RegisterChannel` (line 10).

Next we register the server as a well-known service type. (Inside the CLR there is a check to make sure that the service inherits from `MarshalByRefObject`.) We pass the `Type` object of the type to register, the Uniform Resource Identifier (URI) for the service, and the way we want the service instantiated. When you read *URI*, think *URL*. The URI identifies the service; by convention we use the class name and `.soap` or `.rem` for the URI. You can use any convention, but Internet Information Services (IIS) maps the `.soap` and `.rem` extensions to .NET Remoting. This is important when hosting remote servers in IIS. (Refer to the Choosing a Host for Your Server subsection near the end of this chapter.) You can pass the `WellKnownObjectMode.Singleton` or `WellKnownObjectMode.SingleCall` enumerated values to the registration method. `Singleton` is used to ensure that one object is used to service requests, and `SingleCall` will cause a new object to be created to service each request. (`SingleCall` causes a remoted server to respond like a Web application. The server has no knowledge of previous calls.)

The `Factory` type is registered in lines 11 through 13 and the `Trade` type in lines 15 through 17. After the server types are registered we use `Console.ReadLine` to prevent the server from exiting. To quit the server application, set the focus on the console running the server and hit the carriage return; the server will respond until then. Listing 8.4 contains the code that prepares the client to send requests to the server.

**Listing 8.4** Preparing the Client Application to Begin Making Requests

```
1:  Private Sub Form1_Load(ByVal sender As System.Object, _
2:      ByVal e As System.EventArgs) Handles MyBase.Load
3:
4:      Dim channel As HttpChannel = New HttpChannel()
5:      ChannelServices.RegisterChannel(channel)
6:
7:      Dim instance As Object = _
8:        Activator.GetObject(GetType(IFactory), _
9:          "http://localhost:9999/Factory.soap")
10:
11:     Dim factory As IFactory = _
12:       CType(instance, IFactory)
13:
14:     Dim trade As ITrade = _
15:       factory.GetTrade(1234)
16:
17: End Sub
```

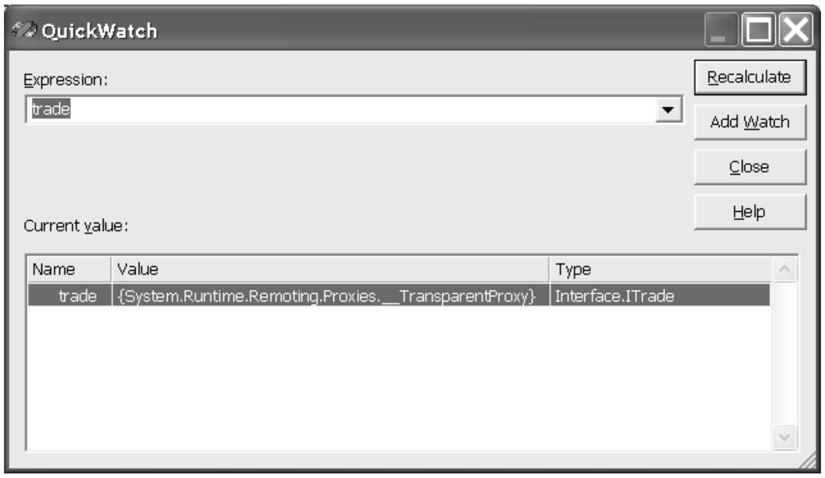
The client declares, creates, and registers a channel in lines 4 and 5. We don't need the port here when we register the channel; we will indicate the port when we request an instance of the object from the server. Lines 7 through 9 use the shared `Activator.GetObject` class to request an instance of the `Factory` class defined in the server. The URL (line 9) indicates the domain and port of the server and the name we registered the server with. Lines 11 and 12 convert the instance type returned by `Activator` to the interface type we know it to be, and lines 14 and 15 use the `factory` instance to request a `Trade` object.

To see that the value of the `trade` object (line 14) is actually a proxy, place a breakpoint in line 17 and use QuickWatch to examine the value of the `trade` variable (Figure 8.2).

## Using Server-Activated Objects

In the example above we created what is known as a *server-activated object* (SAO). When you construct an SAO—for example, with `Activator.GetObject`—only a proxy of the object is created on the client. The actual object on the server isn't created until you invoke an operation on that type via the proxy. (The proxy is transparent; thus the invocation occurs in the background when you call a method or access a property.) The lifetime of an SAO is controlled by the server, and only default constructors are called.

In a production application it is more than likely that you will want to permit the operator to manage the configuration of the server without having to recompile the server application. This can be handled in an application configuration file. `Example3\Client.sln` defines an application configuration



trade

| Name | Value | Type |
|---|---|---|
| trade | {System.Runtime.Remoting.Proxies.__TransparentProxy} | Interface.ITrade |

**Figure 8.2** The local variable `trade` is an instance of the `TransparentProxy` class, indicating the unusual remoted relationship between client and server.

file for `server.vbproj`. You can add an application configuration file by accessing the File|Add New Item menu in Visual Studio .NET and selecting the Application Configuration File template from the Add New Item dialog. Listing 8.5 contains the externalized XML settings used to register the server. The revision to the `Main` class in Listing 8.3, which accommodates the application configuration file, is provided in Listing 8.6.

**Listing 8.5** An Application Configuration That Externalizes Server Registration Settings

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="8080" />
      </channels>
      <service>
        <wellknown mode="Singleton"
          type="Server.Factory, Server"
          objectUri="Factory.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

The first statement describes the XML version and the text encoding (8-bit Unicode in the example). The `configuration` element indicates this is a configuration file. Typically, XML elements have an opening tag—for example, `<configuration>`—and a matching closing tag with a whack (`/`) inside the tag. Sometimes this is abbreviated to `/>`, as demonstrated with the `wellknown` element in Listing 8.5.

The third element indicates the relevant namespace, `system.runtime.remoting`. The `channel` element indicates the channel type and port. The `wellknown` element indicates the `WellKnownObjectMode` (`Singleton` in the example), the type information for the type we are registering (`Server.Factory`, in Listing 8.6), and the URI (`Factory.soap`). This is precisely the same information we provided in Listing 8.3, programmatically. Now, however, if we find that port 8080 is in use by a proxy server or another HTTP server, we can reconfigure the channel without recompiling.

Having modified the server application to store the server registration information in the `.config` file, we can modify Listing 8.3 to simplify the registration of `WellKnownServiceType`. Listing 8.6 shows the shorter, revised code.

**Listing 8.6** Revised Code after Moving Registration Settings to `Server.exe.config`

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Http

Public Class Main

  Public Shared Sub Main(ByVal args() As String)

    RemotingConfiguration.Configure("Server.exe.config")

    Console.WriteLine("Server is running...")
    Console.ReadLine()
    Console.WriteLine("Server is shutting down...")
  End Sub

End Class
```

In the example we have removed the `channel` construction and calls to the shared method `RemotingConfiguration.RegisterWellKnownServiceType` that appeared in Listing 8.3. All we need to do now is pass the name of our `.config` file to the `RemotingConfiguration.Configure` method in Listing 8.6.

Keep in mind that when you add the Application Configuration File template to your project you will see an `App.config` file in the Solution Explorer with the rest of your source. When you compile your application, the *`applicationname`*`.exe.config` file is written to the directory containing the executable. While in the debug configuration mode, for example, you will see the `Server.exe.config` file written to the `.\bin` directory.

## Using Client-Activated Objects

Client-activated objects (CAOs) are registered and work a bit differently than server-activated objects. A CAO is created on the server as soon as you create an instance of the CAO, which you can do by using `Activator.CreateInstance` or the `New` constructor. The biggest difference between SAOs and CAOs is that CAOs do not use shared interfaces; rather, a copy of the shared code must exist on both the client and the server. Deploying code to client and server will mean more binaries on the clients, a more challenging deployment, and possible versioning problems.

To preclude re-reading all the code, I have reused the same `Factory` and `Trade` classes for our CAO example. However, I have gotten rid of the interfaces, placed the `Factory` class in the client (since we don't really need two server-side classes to demonstrate CAO), and shared the `Trade` class between client and server. Instead of literally sharing the `Trade` class in a third DLL assembly, I defined the `Trade` class in the `Server` assembly and used `soapsuds.exe` (a utility that ships with VS .NET) to generate the shared DLL. We'll go through each of these steps in the remaining parts of this section. (The code for this section can be found in the `Example2\Client.sln` solution.)

### *Implementing the Server for the CAO Example*

The `Server.vbproj` file contains the same `Trade` class shown in Listing 8.3, so I won't relist that code here. The `Factory` class has been moved to the client (see the Implementing the Client subsection below). What's different about the server is how we register it. The revision to the `Main` class is shown in Listing 8.7.

**Listing 8.7** Registering a Server for Client Activation

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Http

Public Class Main

  Public Shared Sub Main(ByVal args() As String)

    Dim channel As HttpChannel = New HttpChannel(9999)
    ChannelServices.RegisterChannel(channel)

    ' Code needed for client activation
    RemotingConfiguration.ApplicationName = "Server"
    RemotingConfiguration. _
      RegisterActivatedServiceType(GetType(Trade))
```

```
16:
17:     Console.WriteLine("Server is running...")
18:     Console.ReadLine()
19:     Console.WriteLine("Server is shutting down...")
20:   End Sub
21:
22: End Class
```

Registration for client activation is much simpler. We provide a name for the application and register the type we will be remoting. The application name is provided in line 13 and the `Trade` class is registered in lines 14 and 15 using the shared method `RemotingConfiguration.RegisterActivatedServiceType`, passing the type of the class to register. Recall that we actually have the implementation of the type—`Trade`—defined on the server.

That's all we need to do to the server's `Main` class—change the registration code.

### *Exporting the Server Metadata for the `Trade` Class*

To construct an instance of a class in the client using the new operator, we need a class. Calling `New` on an interface—as in `Dim T As ITrade = New ITrade()`—won't work because interfaces don't have code. You can create a third assembly and share that code in both the client and server, or you can use the `soapsuds.exe` utility to generate a C# source code or a DLL that can be referenced in your client application. I implemented a batch file `mysoapsuds.bat` in the `Example2\Server\bin` directory that will create a DLL named `server_metadata.dll`. Here is the single command in that batch file.

```
soapsuds -ia:server -nowp -oa:server_metadata.dll
```

In this code, `soapsuds` is the name of the executable. The `–ia` switch is the name of the input assembly. (Note that the assembly extension—`.exe` for this example—is left off.). The `–nowp` switch causes `soapsuds` to stub out the implementations, permitting a dynamic transparent proxy to handle the method calls. The `–oa` switch indicates the output assembly name. In the example an assembly named `server_metadata.dll` will be generated. Next we will add a reference to this assembly in our client application.

### *Implementing the Client*

The client application needs a definition of the interface and the type for client activation. We can actually share the code between client and server and use parameterized constructors for client activation; or, in our example, we use `soapsuds.exe` to generate a metadata DLL and give up parameterized constructors for remoted objects.

On the user's PC we need some kind of application as well as remoting registration code, and we can use a factory on the client to simulate constructor parameterization (if we are using `soapsuds`-generated metadata.) As a general rule it is preferable to use `soapsuds` to generate metadata and a factory for convenience, as opposed to shipping the server executable to every client. Listing 8.8 shows a Windows Forms implementation of the CAO client and a factory for the `Trade` class.

**Listing 8.8** Implementing a Client-Activated Object and a Factory

```
Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Activation
Imports System.Reflection
Imports Server

Public Class Form1
  Inherits System.Windows.Forms.Form

[ Windows Form Designer generated code ]

 Private Generator As Generator

 Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load

  Dim channel As HttpChannel = New HttpChannel()
  ChannelServices.RegisterChannel(channel)

  ' Client-activated object code
  RemotingConfiguration.RegisterActivatedClientType( _
   GetType(Trade), _
   "http://localhost:9999/Server")

```

```
27:     Dim Factory As Factory = New Factory()
28:     Dim Trade As Trade = Factory.GetTrade(5555)
29:     Trade.Commission = 25
30:     Trade.EquityName = "CSCO"
31:     Trade.EquityPrice = 11.0
32:     Trade.NumberOfShares = 2000
33:     Trade.RepId = 999
34:
35:     Generator = New Generator(Me, _
36:       GetType(Trade), Trade)
37:     Generator.AddControls()
38:
39:   End Sub
40:
41: End Class
42:
43: Public Class Factory
44:
45:   Public Function GetTrade( _
46:     ByVal customerId As Integer) As Trade
47:     Console.WriteLine("Factory.GetTrade called")
48:
49:     Dim trade As Trade = New Trade()
50:     trade.CustomerId = 555
51:     trade.Commission = 25
52:     trade.EquityName = "DYN"
53:     trade.EquityPrice = 2.22
54:     trade.NumberOfShares = 1000
55:     trade.RepId = "999"
56:
57:     Return trade
58:   End Function
59:
60: End Class
```

Listing 8.8 contains two classes: the Windows Forms class `Form1` and the `Factory` class. The `Form1` class creates and registers a channel in lines 19 and 20. Instead of using the `Activator` class to create the remote object, we call the shared method `RemotingConfiguration.RegisterActivatedClientType`, passing the type to register and the URI of the server-registered type (lines 23 through 25).

After the type that can be activated on the server is registered, we use the `Factory` class to create an instance of that type (lines 27 and 28). To provide you with additional calls to the server, I changed the values set by the `Factory` class. There is no requirement here, just extra code.

The `Generator` class used on lines 35 through 37 is extra code I added to create a Windows user interface. This code is included with the downloadable remoting example and creates a simple user interface comprised of text boxes and labels, created by reflecting the remote type.

## Marshaling Objects by Value

Marshal-by-value objects are not remote objects. By-value objects are marked with the `SerializableAttribute` or implement the `ISerializable` interface. When a serializable object is requested from a remote server, the object is serialized into an XML or binary format and transmitted to the requester. Only data is shipped. Thus, if the serialized object has methods that need to be invoked, the code must exist on the recipient device. This is similar to how Web Services work. A Web Service returns an XML representation of an object that is comprised of data only. If you want to invoke operations on the data, you need the assembly that contains the methods. (This approach is used to return serialized data sets from an XML Web Service.)

In this section I offer another example using the `Factory` and `Trade` classes. The `Factory` class is a `MarshalByRefObject` that returns a by-value object, an instance of the serialized `Trade` class. In this example I need to share the implementation of the `Trade` class between client and server. The server will serialize a representation of the `Trade` class, and the client will deserialize the representation. Implicitly the deserialized data will be mapped to the shared implementation of the `Trade` class. The end result is that we get the data from the server—passing just the data—and reconstitute the actual object on the client.

On top of the basic example, I will provide an example of a customer version of the `Trade` class that implements `ISerializable`.

### Employing By-Value Classes

In our revised example the `Factory` class is a marshal-by-reference class. (Recall this means that it inherits from `MarshalByRefObject`.) Further, I have converted the implementation of the `Trade` class to be a marshal-by-value class. Both classes inherit from their respective interfaces: `Trade` implements `ITrade`, and `Factory` implements `IFactory`. `Interface.dll`, containing the interfaces `ITrade` and `IFactory`, is shared by both client and server. On the server we configure and register the `Factory` class as remotable by using a `.config` file (as demonstrated in Listing 8.6) to manage registration of the server-activated class. As a result, only the `Trade` class contains modification. Listing 8.9 shows the complete, revised listing of the `Trade` class as defined on the server. The absence of `MarshalByRefObject` inheritance and the `SerializableAttribute` indicates that the `Trade` class is a by-value object in this listing.

**Listing 8.9** Implementing the `Trade` Class as a By-Value Object

```
<Serializable()>_
Public Class Trade

  Private FCustomerId As Integer
  Private FNumberOfShares As Double
  Private FEquityName As String
  Private FEquityPrice As Double
  Private FCommission As Double
  Private FRepId As String

  Public Property CustomerId() As Integer
  Get
    Return FCustomerId
  End Get
  Set(ByVal Value As Integer)
    FCustomerId = Value
  End Set
  End Property

  Public Property NumberOfShares() As Double
  Get
    Return FNumberOfShares
  End Get
  Set(ByVal Value As Double)
    FNumberOfShares = Value
  End Set
  End Property

```

```
Public Property EquityName() As String
Get
  Return FEquityName
End Get
Set(ByVal Value As String)
  Console.WriteLine("EquityName was {0}", FEquityName)
  FEquityName = Value
  Console.WriteLine("EquityName is {0}", FEquityName)
  Console.WriteLine([Assembly].GetExecutingAssembly().FullName)

End Set
End Property

Public Property EquityPrice() As Double
Get
  Return FEquityPrice
End Get
Set(ByVal Value As Double)
  FEquityPrice = Value
End Set
End Property

ReadOnly Property Cost() As Double
Get
  Return FEquityPrice * _
    FNumberOfShares + FCommission
End Get
End Property

Property Commission() As Double
Get
  Return FCommission
End Get
Set(ByVal Value As Double)
  FCommission = Value
End Set
End Property

Property RepId() As String
Get
  Return FRepId
End Get
Set(ByVal Value As String)
  FRepId = Value
```

```
73:   End Set
74:   End Property
75:
76: End Class
```

After a quick observation you will see that only the first couple of lines have changed and all the interface implementation code has been removed. Line 1 shows the use of the `SerializableAttribute` (defined in the `System` namespace), and I removed the statement `Inherits MarshalByRefObject`. This is all you need to do to indicate that an object can be sent back and forth in a serialized form, such as an XML document.

Additionally, the `IFactory` interface (not shown here, see Listing 8.1) has been modified to return a `Trade` value rather than the `ITrade` interface, and the `ITrade` interface—no longer required—has been removed from the `Interface.vb` source file.

## Revising the Client to Use the By-Value Object

The client has to change very little to accommodate the marshal-by-value object. `Factory` is the remote object and it returns the `Trade` type, which we have referenced in both the client and the server. Because we are using a server-activated object—`Factory`—we only need to get an instance of the factory and invoke the `GetTrade` method. .NET automatically serializes the `Trade` object, and our locally declared `Trade` variable can handle the deserialized instance. We do not have to manage serialization on the server or deserialization on the client; this is automatic. Listing 8.10 contains the client code for the marshal-by-value `Trade` object.

**Listing 8.10** Client Code for the By-Value `Trade` Object

```
Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Http
Imports System.Reflection
Imports [Interface]

Public Class Form1
    Inherits System.Windows.Forms.Form

```

```
11: [ Windows Form Designer generated code ]
12:
13:   Private Generator As Generator
14:   Private trade As Trade
15:
16:   Private Sub Form1_Load(ByVal sender As System.Object, _
17:     ByVal e As System.EventArgs) Handles MyBase.Load
18:
19:     Dim channel As HttpChannel = New HttpChannel()
20:     ChannelServices.RegisterChannel(channel)
21:
22:     Dim Instance As Object = _
23:       Activator.GetObject(GetType(IFactory), _
24:       "http://localhost:8080/Factory.soap")
25:
26:     Dim Factory As IFactory = CType(Instance, IFactory)
27:     trade = Factory.GetTrade(1234)
28:
29:     trade.EquityName = "MSFT"
30:     Debug.WriteLine(trade.Cost.ToString())
31:     Generator = New Generator(Me, _
32:       GetType(Trade), trade)
33:     Generator.AddControls()
34:
35:   End Sub
36:
37: End Class
```

Note that in the example the `trade` variable (line 14) is declared as a `Trade` type. We are actually getting a serialized form of the `Trade` object from the server, and the client is automatically deserializing the object returned by the `Factory` method and reconstituting it as a `Trade` object. Because we have an implementation of the `Trade` class shared between client and server, this works nicely.

The balance of the code registers the server-activated `Factory` and uses the `Generator` class I defined to create a user interface. You can download `Example4\Client.sln` to experiment with this code.

## Implementing `ISerializable`

The default behavior of the `SerializableAttribute` is to serialize all public properties. In a serialized form they are transmitted as public fields.

However, because we have the binary code on both the client and the server, the deserialized object can be reconstituted as a complete object. Completeness, here, means that we have properties, fields, methods, attributes, and events.

Generally this default behavior is sufficient. However, it may be insufficient if you want to serialize additional data that may not be part of the public properties but is beneficial to the class or intensive to calculate. Whenever you need extra data serialized you can get it by implementing the `System.Runtime.Serialization.ISerializable` interface. The help documentation tells you that you need to implement `GetObjectData`, which is the serialization method. What is implied is that you need the symmetric deserialization behavior. Deserialization is contrived in the form of a constructor that initializes an object based on serialized data.

In order to demonstrate custom serialization in the `Example4\Client.sln` file I added a contrived value to the `Trade` class used for debugging purposes. This contrived field, `DateTime`, holds the date and time when the object was serialized. When a `Trade` object is serialized, I include the current `DateTime` value. When the object is deserialized, the `DateTime` value is written to the Debug window. To affect the custom serialization I needed to change only the shared class we have been using all along. The complete listing of the `Trade` class is shown in Listing 8.11 with the revisions (compared with Listing 8.9) in bold font. (The actual source is contained in `Example4\Interface\Interface.vb`.)

**Listing 8.11** Implementing Custom Serialization for .NET Remoting

```
<Serializable()>_
Public Class Trade
  Implements ISerializable

  Private FCustomerId As Integer
  Private FNumberOfShares As Double
  Private FEquityName As String
  Private FEquityPrice As Double
  Private FCommission As Double
  Private FRepId As String

  Public Sub New()
  End Sub

  Public Sub New(ByVal info As SerializationInfo, _
    ByVal context As StreamingContext)

    Debug.WriteLine("Started deserializing Trade")
    FCustomerId = CType(info.GetValue("CustomerId", _
      GetType(Integer)), Integer)
    FNumberOfShares = CType(info.GetValue("NumberOfShares", _
      GetType(Double)), Double)

    FEquityName = CType(info.GetValue("EquityName", _
      GetType(String)), String)

    FEquityPrice = CType(info.GetValue("EquityPrice", _
      GetType(Double)), Double)

    FCommission = CType(info.GetValue("Commission", _
      GetType(Double)), Double)

    FRepId = CType(info.GetValue("RepId", _
      GetType(String)), String)

    Dim SerializedAt As DateTime _
      = CType(info.GetValue("SerializedAt", _
      GetType(DateTime)), DateTime)

    Debug.WriteLine(String.Format( _
      "{0} was serialized at {1}", _
      Me.GetType.Name(), SerializedAt))

    Debug.WriteLine("Finished deserializing Trade")
  End Sub

  Protected Sub GetObjectData( _
    ByVal info As SerializationInfo, _
    ByVal context As StreamingContext _
    ) Implements ISerializable.GetObjectData

    Console.WriteLine("Started serializing Trade")

    info.AddValue("CustomerId", FCustomerId)
    info.AddValue("NumberOfShares", FNumberOfShares)
    info.AddValue("EquityName", FEquityName)
    info.AddValue("EquityPrice", FEquityPrice)
    info.AddValue("Commission", FCommission)
    info.AddValue("RepId", FRepId)
    info.AddValue("SerializedAt", DateTime.Now)

    Console.WriteLine("Finished serializing Trade")
  End Sub


  Public Property CustomerId() As Integer
  Get
    Return FCustomerId
  End Get
  Set(ByVal Value As Integer)
    FCustomerId = Value
  End Set
  End Property

  Public Property NumberOfShares() As Double
  Get
    Return FNumberOfShares
  End Get
  Set(ByVal Value As Double)
    FNumberOfShares = Value
  End Set
  End Property

  Public Property EquityName() As String
  Get
    Return FEquityName
  End Get
  Set(ByVal Value As String)
    Console.WriteLine("EquityName was {0}", FEquityName)
    FEquityName = Value
    Console.WriteLine("EquityName is {0}", FEquityName)
    Console.WriteLine([Assembly].GetExecutingAssembly().FullName)
  End Set
  End Property

  Public Property EquityPrice() As Double
  Get
    Return FEquityPrice
  End Get
  Set(ByVal Value As Double)
    FEquityPrice = Value
  End Set
  End Property
```

```
104:
105:  ReadOnly Property Cost() As Double
106:  Get
107:    Return FEquityPrice * _
108:      FNumberOfShares + FCommission
109:  End Get
110:  End Property
111:
112:  Property Commission() As Double
113:  Get
114:    Return FCommission
115:  End Get
116:  Set(ByVal Value As Double)
117:    FCommission = Value
118:  End Set
119:  End Property
120:
121:  Property RepId() As String
122:  Get
123:    Return FRepId
124:  End Get
125:  Set(ByVal Value As String)
126:    FRepId = Value
127:  End Set
128:  End Property
129:
130: End Class
```

**TIP:** You need to include the `SerializableAttribute` even when you are implementing the `ISerializable` interface. Remember to add an `Imports` statement for `System.Runtime.Serialization`, or use the completely qualified name for the `ISerializable` interface when performing custom serialization.

Serialization and deserialization in Listing 8.11 are constrained to lines 15 through 63. The recurring pattern is a constructor and a method named `GetObjectData`. Both the constructor and `GetObjectData` take `SerializationInfo` and `StreamingContext` arguments. The constructor reads the streamed field values, and the serialization method, `GetObjectData`, writes the fields to be streamed. Since you will be writing both the serializer and deserializer you will know the order and type of the arguments streamed.

To serialize an object, write the fields using the SerializationInfo object in the GetObjectData method. Call SerializationInfo.SetValue, passing a name for the value and the value itself. For example, line 59 passes the literal "RepId" and the value of the field FRepId. When you deserialize the object in the constructor, use the SerializationInfo argument and call GetValue. Pass the name used to serialize the object and the type information for that value. It is a good practice to perform an explicit type conversion on the return value since GetValue returns an Object type. For example, lines 33 and 34 of Listing 8.11 call GetValue, passing the literal "RepId" and the Type object for the String class, and perform the cast to String.

Line 60 demonstrates how we can serialize an arbitrary value, SerializedAt. Lines 36 through 38 demonstrate how we can deserialize that same value, perform the type conversion, and assign the value to a local variable (or a field). In lines 40 through 42 I use the value SerializedAt to indicate when the client was serialized. Perhaps such a value could be used as a rough measure of latency. If you compared the SerializedAt time with the current time, you would know how long the serialization and deserialization behavior took in a single instance.

## Comparing By-Reference to By-Value Objects

There are two ways to pass objects around: by value and by reference. By-reference objects are remoted objects that inherit from MarshalByRefObject. This means that they actually exist on the remote server. By-value objects are copied to the client and use the SerializableAttribute. It's important to decide when to use either technique.

Pass objects by reference to prevent a large object from clogging network bandwidth. You will also have to pass objects by reference when the object refers to resources that exist in the server domain. For example, C:\winnt\system32 on the client is a completely different folder than C:\winnt\system32 on the server.

Consider passing objects by value when the data on the client does not need to be maintained on the server. For example, if we are simply reporting on trade information, we don't necessarily need a reference to a Trade object on the server. Using a by-value Trade object will reduce round-trips to the server since the code resides on the client.

Think of by-value objects as similar to the data returned by a Web application: It is disconnected. Think of by-reference objects as the connected model of programming.

KIMMEL.book Page 249 Thursday, June 12, 2003 8:37 AM

## Writing to the Event Log

Thus far I have been using console applications to simplify the examples. However, you are more likely to use .NET Remoting for WinForms, WebForms, or NT Service applications. For debugging and tracing information for these applications you can use the event log. Chapter 17 gives more information on using the `EventLog` class for logging application events, but I'll mention it here briefly.

The easiest way to log application events is to invoke the shared method `EventLog.WriteEntry`, passing the event source and message to write. The event source is a unique name across all event logs, and the message is whatever text you want to appear in the log entry. By default, information will be written to the Application log. (Refer to Chapter 17 to read about creating custom logs and writing log entries to remote machines.)

## Handling Remote Events

Microsoft included an example of using events in .NET Remoting in the help documentation at `ms-help://MS.VSCC/MS.MSDNVS/cpguide/html/cpconremotingexampledelegatesevents.htm`. (You can open this help topic by browsing to the referenced link in Internet Explorer or the URL control of the Web toolbar in the VS .NET IDE.) The example is a simple console-based, chat example that permits clients to communicate through a `Singleton` object. Rather than repeat that code here (and because I thought the example was fun), I include a WinForms-based version that is slightly more advanced and a lot of fun to play with. Here is the basic idea.

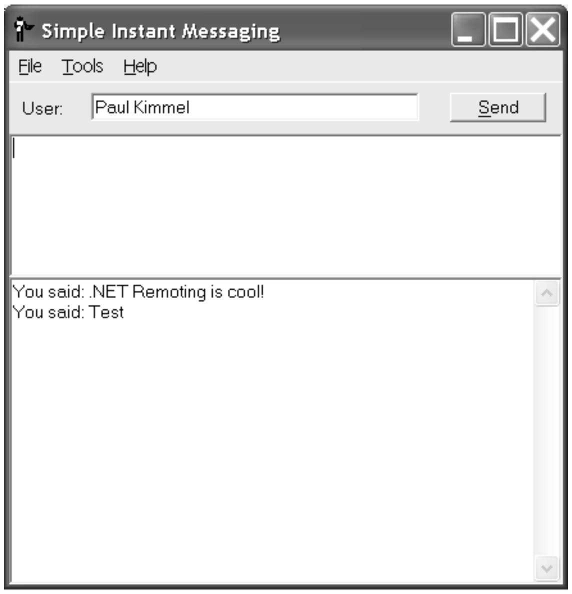
Recall that we talked about `Singleton` remoted objects. When we create a `Singleton MarshalByRefObject`, every client will get a transparent proxy—think "super pointer"—to the exact same object on the server. By exposing an event each client can add an event handler to the event. Now mix in delegates. Delegates are multicast in .NET. This means that the event is sent to all handlers. Thus, if one client raises an event, every client that has a handler in that object's invocation list will receive an event message. Combined with a `Singleton` Remote object reference, each client will be adding a handler to one object's multicast delegate invocation list. Voilà! A simplified chat application.

## Understanding Remote Event Behavior

We have thought of remoting so far as clients having transparent proxy reference to an object on the server. However, when we need to raise an event on the server, the server is actually calling back to the client; the client handler becomes a server, and the server becomes the client. Consequently the client has a reference to the server, and when the roles are reversed, the server needs a reference to the client. We can solve this predicament by sharing code between client and server.

## Invoking Remote Events

The example is comprised of three projects all contained in the `\Chapter 8\Events\Chat\Chat.sln` file. The solution includes the `Chat.vbproj` client, `ChatServer.vbproj`, and the shared class library `General`. The `ChatServer.exe` is a console application that has a reference to `General.dll` and configures `ChatServer.ChatMessage` as a well-known `Singleton` object using an application configuration file. The `Chat.exe` server is a Windows Forms application (see Figure 8.3) that has a reference to `General.dll`. Each instance of `Chat.exe` requests a reference to



```
You said: .NET Remoting is cool!
You said: Test
```

**Figure 8.3** The simple instant messaging example.

the `ChatMessage` object created on the remote server. The server returns the same instance to every client that requests a `ChatMessage` object on the same channel from the same server. After the client gets the `ChatMessage` wrapper back, it assigns one of its event handlers to an event defined by the wrapper class. When any particular client sends a message to the server, a `ChatMessage` object raises an event and all clients get the event. As a result we can selectively echo the original message (or not) to the sender and notify each client of a message.

The server class simply uses a configuration file to register a `Singleton` instance of a `ChatMessage` wrapper object. You can see the code for the server in `\Chapter 8\Events\Server\Server.vb`. The shared `General.dll` assembly (which contains the wrapper) and the client that sends and handles events provide the most interesting functionality. We will go over most of that code next.

### *Implementing the Shared Event Wrapper*

The code containing the shared event wrapper class is defined in `\Chapter 8\Events\General\Class1.vb`. `Class1.vb` defines three classes and a delegate. Listing 8.12 contains all the code for `Class1.vb`; a synopsis of the code follows the listing.

**Listing 8.12** The Shared Classes That Manage Events between Client and Server

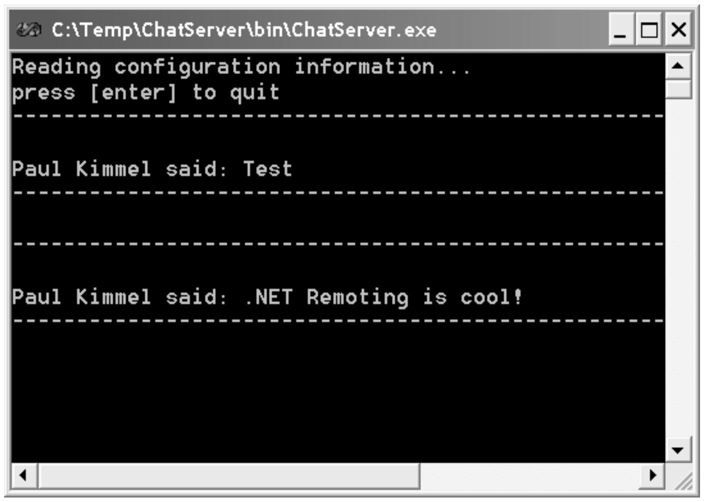
```
Option Strict On
Option Explicit On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Messaging

Imports System.Collections

<Serializable()>_
Public Class ChatEventArgs
  Inherits System.EventArgs

  Private FSender As String
  Private FMessage As String

  Public Sub New()
    MyBase.New()
  End Sub

  Public Sub New(ByVal sender As String, _
    ByVal message As String)
    MyClass.New()
    FSender = sender
    FMessage = message
  End Sub

  Public ReadOnly Property Sender() As String
  Get
    Return FSender
  End Get
  End Property

  Public ReadOnly Property Message() As String
  Get
    Return FMessage
  End Get
  End Property
End Class

Public Delegate Sub MessageEventHandler(ByVal Sender As Object, _
  ByVal e As ChatEventArgs)

Public Class ChatMessage
  Inherits MarshalByRefObject

  Public Event MessageEvent As MessageEventHandler

  Public Overrides Function InitializeLifetimeService() As Object
    Return Nothing
  End Function

  <OneWay()> _
  Public Sub Send(ByVal sender As String, _
    ByVal message As String)

    Console.WriteLine(New String("-"c, 80))
    Console.WriteLine("{0} said: {1}", sender, message)
    Console.WriteLine(New String("-"c, 80))

    RaiseEvent MessageEvent(Me, _
      New ChatEventArgs(sender, message))
  End Sub

End Class


Public Class Client
  Inherits MarshalByRefObject

  Private FChat As ChatMessage = Nothing

  Public Overrides Function InitializeLifetimeService() As Object
    Return Nothing
  End Function

  Public Sub New()
    RemotingConfiguration.Configure("Chat.exe.config")

    FChat = New ChatMessage()

    AddHandler FChat.MessageEvent, _
      AddressOf Handler
  End Sub

  Public Event MessageEvent As MessageEventHandler

  Public Sub Handler(ByVal sender As Object, _
    ByVal e As ChatEventArgs)
    RaiseEvent MessageEvent(sender, e)
  End Sub

  Public Sub Send(ByVal Sender As String, _
    ByVal Message As String)
    FChat.Send(Sender, Message)
  End Sub

  Public ReadOnly Property Chat() As ChatMessage
  Get
    Return FChat
  End Get
  End Property

End Class
```

Lines 12 through 41 define a new type of event argument, `ChatEventArgs`. `ChatEventArgs` inherits from `System.EventArgs` and introduces two new members: `Message` and `Sender`. `Message` is the content of the message sent by a client, and `Sender` is a user name. `ChatEventArgs` is an example of an object that the client needs for information purposes only; hence it was designated as a by-value object.

Lines 43 and 44 define a new delegate named `MessageEventHandler`. Its signature accepts the new event argument `ChatEventArgs`.

Lines 46 through 67 define the by-reference object `ChatMessage` that is the `Singleton` object shared by all clients. Every client on the same channel and originating from the same server will be referring to the same instance of this class. The class itself is easy enough, but it demonstrates some old concepts and introduces some new ones. Line 47 indicates that `ChatMessage` is a by-reference type. Line 49 exposes a public event; this is how all clients attach their event handlers to the `ChatMessage Singleton`. Lines 51 through 53 override the `MarshalByRefObject.InitializeLifetimeService` method. `InitializeLifetimeService` can be overridden to change the lifetime of a Remote object. `Return Nothing` sets the lifetime to infinity. (Refer to the Managing a Remoted Object's Lifetime subsection later in this chapter for more information.) Lines 55 through 65 define the `Send` message. Clients use `Send` to broadcast messages. All `Send` does is raise `MessageEvent`. Note that `Send` is adorned with the `OneWayAttribute`, which causes the server to treat `Send` as a "fire and forget" method. `Send` doesn't care whether the recipients receive the message or not. This handles the case of a client dropping off without disconnecting its handler. (`Send` also displays trace information on the server application; see Figure 8.4.) That's all the `ChatMessage` class is: a class shared between client and server that wraps the message invocation.

Finally, we come to the `Client` class in lines 70 through 106. The `Client` class plays the role of the executable code that is remotable and shared between client and server. If you examine it closely you will see that it mirrors the `ChatMessage` class except that `Client` is responsible for allowing the server to call back into the client application. The `Client` class in `General.dll` plays the role of client-application-on-the-server when the roles between client and server are reversed. If we didn't have a remotable class shared between client and server, we would need to copy the client application into the directory of the server application. Remember that for clients to run code defined on a server, we need an interface or shared code in order to have something to assign the shared object to. When the roles between client and server are reversed—client becomes server during the callback—the server would need an interface or shared code to the client to



```
Reading configuration information...
press [enter] to quit
------------------------------------------------

Paul Kimmel said: Test
------------------------------------------------

------------------------------------------------

Paul Kimmel said: .NET Remoting is cool!
------------------------------------------------
```

**Figure 8.4** Trace information being written to the server console.

talk back to it. Thus for the same reason that we share code between client and server, we also share code between server and client.

**TIP:** For a comprehensive discussion of event sinks and .NET Remoting, Ingo Rammer [2002] has written a whole book, *Advanced .NET Remoting.*

Listing 8.13 contains the `Chat.exe.config` file that describes the configuration information to the well-known object registered on the server and the back channel to the client used when the client calls the server back.

**Listing 8.13** The Configuration File for the Client Application

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel
          ref="http"
          port="0"
        />
      </channels>
      <client>
        <wellknown
          type="ChatServer.ChatMessage, General"
          url="http://localhost:6007/ChatMessage.soap"
        />
      </client>
    </application>
  </system.runtime.remoting>

  <appSettings>
    <add key="user" value="Your Name Here!" />
    <add key="echo" value="true" />
  </appSettings>
</configuration>
```

The `<channels>` element describes the back channel used by server to client. By initializing the `port` attribute with `0` we allow the port to be dynamically selected. The `<client>` element registers the reference to the well-known `ChatMessage` class on the client. This allows us to create an instance of the `ChatMessage` class on the client using the `New` operator, getting a transparent proxy instance rather than the literal `ChatMessage` class also defined in the client. Without the `<client>` element we would need to use the Activator or we'd end up with a local instance of `ChatMessage` rather than the remote instance.

Finally, the `<appSettings>` element is used by the `ConfigurationSettings.AppSettings` shared property to externalize general, nonremoting configuration information.

### *Implementing the Client Application*

The client application creates an instance of the `Client` class. `Client` represents the assembly shared by both client and server, allowing server to talk back to client. The client application (shown in Figure 8.3) actually registers its events with the `Client` class. Listing 8.14 provides the relevant code for the client application that responds to events raised by the remote `ChatMessage` object. (The `Client.vb` source contains about 400 lines of Windows Forms code not specifically related to remoting. Listing 8.14 contains only that code related to interaction with the remote object. For the complete listing, download `\Chapter 8\Events\Client\Client.vb`.)

**Listing 8.14** An Excerpt from the Client Application Related to Remoting

```
Option Strict On
Option Explicit On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Http
Imports Microsoft.VisualBasic
Imports System.Configuration

Public Class Form1
    Inherits System.Windows.Forms.Form
    . . .

   Public Sub Handler(ByVal sender As Object, _
     ByVal e As ChatEventArgs)

     If (e.Sender <> User) Then
       Received = GetSenderMessage(e.Sender, e.Message) + Received
     ElseIf (Echo) Then
       Received = GetSendeeMessage(e.Message) + Received
     End If

   End Sub

   Private ChatClient As Client

   Private Sub Form1_Load(ByVal sender As Object, _
     ByVal e As System.EventArgs) Handles MyBase.Load

     Init()

     ChatClient = New Client()

     AddHandler ChatClient.MessageEvent, _
       AddressOf Handler
   End Sub

   Private Sub Send()
     If (ChatClient Is Nothing = False) Then
       ChatClient.Send(User, Sent)
       Sent = ""
```

```
313:     End If
314:   End Sub
315:
316:   Private Sub Button1_Click(ByVal sender As System.Object, _
317:     ByVal e As System.EventArgs) Handles ButtonSend.Click, _
318:     MenuItemSend.Click
319:
320:     Send()
321:
322:   End Sub
323:
324:   Private Sub Form1_Closed(ByVal sender As Object, _
325:     ByVal e As System.EventArgs) Handles MyBase.Closed
326:
327:     If (ChatClient Is Nothing) Then Return
328:     RemoveHandler ChatClient.MessageEvent, _
329:       AddressOf Handler
330:   End Sub
331:  . . .
344:
345:   Private Sub Init()
346:     User = ConfigurationSettings.AppSettings("user")
347:     Echo = (ConfigurationSettings.AppSettings("echo") = "true")
348:   End Sub
349:  . . .
396: End Class
```

Listing 8.14 contains snippets from `Client.vb`. Parts that are basic to Windows Forms or programming in general were removed to shorten the listing. Lines 285 through 294 define an event handler named `Handler`. As you can see from the listing, this handler looks like any other event handler. Note that there are no special considerations made for remoting (although there should be; more on this in a moment).

Line 296 declares the shared instance of the `Client` object. `Client` is the remotable object that the server treats like a server when it needs to communicate back with us.

Lines 298 through 307 define the form's `Load` event handler. `Load` initializes the application settings (line 301), creates a new instance of the `Client` class, and associates the form's event handler with the `Client` class's event handler. `Client` is the actual object called back by `ChatMessage`.

`Button1_Click` in lines 316 through 322 calls a local `Send` method that invokes `Client.Send`. `Form1_Closed` (lines 324 through 330) removes the event handler. If for some reason this code isn't called, the server will try to call this instance of the client application for as long as the server is running. If we hadn't used the `OneWayAttribute`, removing the client application without removing the event would cause exceptions. Using the `OneWayAttribute` avoids the exceptions but could potentially send out tons of calls to dead clients. (An alternative is to skip using the `OneWayAttribute` on the server and remove delegates that cause an exception on the server.) The `Init` method (lines 345 through 348) demonstrates how to read configuration settings from an application `.config` file.

### *Remoting and Threads*

.NET Remoting is easier than DCOM and other technologies, but writing distributed applications is still not a trivial exercise. Recall that reference to something missing from the event handler in Listing 8.14 in lines 285 through 294? What's missing is a discussion of threads.

When the event handler is called, it actually comes back on a different thread than the one that Windows Forms controls are in. Recall that in Chapter 6, Multithreading, I said that Windows Forms is not thread-safe. This means that it is not safe to interact with Windows Forms controls across threads. To resolve this predicament we need to perform a synchronous invocation to marshal the call to the event handler—a thread used by remoting—to the same thread the Windows Forms controls are on. In short, we need to add a delegate and call `Form.Invoke` to move the data out of the event handler onto the same thread that the form and its controls are on.

## Other Remoting Subjects

Writing distributed applications well is one of those things that rests at the upper echelon of advanced topics. This chapter will get you started. However, if you are going to deploy a distributed application that employs remoting, you should explore in detail such other subjects as management of a remoted object's lifetime, asynchronous behavior and remoting, security issues related to remoting, and implementation of Remote behavior for a variety of host applications. This material will ultimately reside in several books. Ingo Rammer [2002] wrote one of the more comprehensive books currently available on remoting. I have included below a quick overview of these subjects to help guide further exploration.

## Managing a Remoted Object's Lifetime

DCOM managed object lifetime by pinging the client to see if the client was still hanging out. This causes a lot of extra network traffic. .NET Remoting uses an approach similar to Java by supplying an object a lease. The basic idea is that an object has a default amount of five minutes of time to live (TTL). After five minutes the object is destroyed. If a remote object is accessed, the TTL is reset to the lease time (in the case of the default setting, five minutes). As long as the object is accessed within the TTL, it stays alive.

You can adjust the lifetime by adding a `<lifetime>` element to the `.config` file or implementing a sponsor. The sponsor answers the question, "My lease time is up; should I go away?" Whereas the lifetime is a static value, the sponsor can be dynamic, based on some programmatic logic. You implement a sponsor as a remotable object by implementing the `System.Runtime.Remoting.Lifetime.ISponsor` interface, by programmatically returning an `ILease` object from an overridden `InitializeLifetimeService` method, or by codifying the lifetime in a `.config` file. Listing 8.15 offers an example lease.

**Listing 8.15** Changing the Default Lifetime by Using the Configuration File

```
<lifetime
  leaseTime="2M"
  sponsorshipTimeout="2M"
  renewOnCallTime="2M"
  leaseManagerPollTime="5S"
/>
```

This example sets the lease time, sponsor time-out, and renew time to two minutes. The polling interval is set to five seconds. (The units of measure are `D` for days, `H` for hours, `M` for minutes, `S` for seconds, and `MS` for milliseconds.) You cannot combine intervals. For example, you cannot define a lease time of `"2M5S"`.

In Listing 8.15, `leaseTime` represents the object's lease on life; `sponsorshipTimeout` represents the time-out period of a sponsor; and `renewOnCallTime` represents the extended amount of lease time on an object (this value is not accumulative). Finally, `leaseManagerPollTime` specifies how long the lease manager waits between polling intervals.

### Asynchronous Remoting

.NET Remoting supports both synchronous and asynchronous method invocation. Use the `BeginInvoke` and `EndInvoke` methods combined with delegates to invoke remote operations asynchronously. Read Chapter 6 for more information on asynchronous processing.

### Remoting Security Issues

You can employ authentication and encryption with .NET Remoting and use secure HTTP (HTTPS) to make distributed applications more secure. For secure HTTP you need a certificate. You can acquire a certificate from VeriSign (*http://www.netsol.com*) for free. For more information on remoting and security, read information on HTTPS, certificates, encryption, and authentication.

### Choosing a Host for Your Server

The examples in this chapter host all the server applications as console applications, but you are not limited to console applications. You can also use .NET Remoting in Windows Forms, Web (hosting the remote server in IIS), and Service (Windows NT Service) applications. To create a remote server using any of these host types, use the project template to create the project for the particular type. Each style of host will have individuated requirements.

## Summary

.NET Remoting is probably one of the most advanced subjects. In addition to replacing DCOM and being relevant to distributed application development (which you may not do every day) remoting involves threading, `Singletons`, security, networking, Reflection, `AppDomains`, the differences between marshaling by reference and by value, SOAP, XML, serialization, interfaces, and more.

Chapter 8 introduced .NET Remoting fundamentals. We discussed the difference between marshaling objects by value and by reference, server-activated objects and client-activated objects, `Singleton` and `SingleCall` remote objects, configuration files, custom serialization, and how to raise events from remote objects. These are the key elements of all .NET Remoting and will aid you in experimenting with distributed applications and further exploration.