Advanced .NET Remoting in VB .NET

INGO RAMMER

Apress™

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

Remoting in Action

IN THIS CHAPTER, I DEMONSTRATE the key techniques you'll need to know to use .NET Remoting in your real-world applications. I show you the differences between Singleton and SingleCall objects and untangle the mysteries of client-activated objects. I also introduce you to SoapSuds.exe, which can be used to generate proxy objects containing only methods' stubs. This chapter is somewhat code based, so prepare yourself to start VS .NET quite often!

Types of Remoting

As you have seen in the previous chapter's examples, there are two very different types of remote interaction between components. One uses serializable objects that are passed as a copy to the remote process. The second employs server-side (remote) objects that allow the client to call their methods.

ByValue Objects

Marshalling objects by value means to serialize their state (instance variables), including all objects referenced by instance variables, to some persistent form from which they can be deserialized in a different context. This ability to serialize objects is provided by the .NET Framework when you set the attribute <Serializable()> for a class or implement ISerializable.

When passing the Customer object in the previous chapter's validation example to the server, it is serialized to XML like this:

```
<a1:Customer id="ref-4">
<FirstName id="ref-5">Joe</FirstName>
<LastName id="ref-6">Smith</LastName>
<DateOfBirth>1800-05-12T00:00:00.0000+02:00</DateOfBirth>
</a1:Customer>
```

This XML document will be read by the server and an exact copy of the object created.

> **NOTE** *An important point to know about ByValue objects is that they are*
> *not remote objects. All methods on those objects will be executed locally*
> *(in the same context) to the caller. This also means that, unlike with*
> *MarshalByRefObjects, the compiled class has to be available to the client.*
> *You can see this in the preceding snippet, where "age" is not serialized but*
> *will be recalculated at the client using the* `getAge()` *method.*

When a ByValue object holds references to other objects, those have to
be either serializable or MarshalByRefObjects; otherwise, an exception will be
thrown, indicating that those objects are not remoteable.

MarshalByRefObjects

A MarshalByRefObject is a remote object that runs on the server and accepts
method calls from the client. Its data is stored in the server's memory and its
methods executed in the server's AppDomain. Instead of passing around a vari-
able that points to an object of this type, in reality only a pointer-like
construct—called an ObjRef—is passed around. Contrary to common pointers,
this ObjRef does not contain the memory address, rather the server name/IP
address and an object identity that identifies exactly *one* object of the many that
are probably running on the server. I cover this in depth later in this chapter.
MarshalByRefObjects can be categorized into two groups: *server-activated*
objects (SAOs) and *client-activated objects* (CAOs).

Server-Activated Objects

Server-activated objects are somewhat comparable to classic stateless Web
Services. When a client requests a reference to a SAO, no message will travel to
the server. Only when methods are called on this remote reference will the server
be notified.

Depending on the configuration of its objects, the server then decides
whether a new instance will be created or an existing object will be reused. SAOs
can be marked as either *Singleton* or *SingleCall*. In the first case, one instance
serves the requests of all clients in a multithreaded fashion. When using objects
in SingleCall mode, as the name implies, a new object will be created for each
request and destroyed afterwards.

In the following examples, you'll see the differences between these two kinds
of services. You'll use the same shared interface, client- and server-side imple-
mentation of the service, and only change the object mode on the server.

The shared assembly General.dll will contain the interface to a very simple remote object that allows the storage and retrieval of stateful information in form of an Integer value, as shown in Listing 3-1.

Listing 3-1. The Interface Definition That Will Be Compiled to a DLL

```
Imports System

Public Interface IMyRemoteObject
    Sub setValue(ByVal newval As Integer)
    Function getValue() As Integer
End Interface
```

The client that is shown in Listing 3-2 provides the means for opening a connection to the server and tries to set and retrieve the instance values of the server-side remote object. You'll have to add a reference to System.Runtime.Remoting.DLL to your Visual Studio .NET project for this example.

Listing 3-2. A Simple Client Application

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports General
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels

Module Client

    Sub Main()
        Dim channel As New HttpChannel()
        ChannelServices.RegisterChannel(channel)

        Dim obj As IMyRemoteObject = CType(Activator.GetObject( _
            GetType(IMyRemoteObject), _
            "http://localhost:1234/MyRemoteObject.soap"), _
            IMyRemoteObject)

        Console.WriteLine("Client.Main(): Reference to rem.obj. acquired")

        Dim tmp As Integer = obj.getValue()
```

```
            Console.WriteLine("Client.Main(): Original server side value: {0}", tmp)
            Console.WriteLine("Client.Main(): Will set value to 42")
            obj.setValue(42)
            tmp = obj.getValue()
            Console.WriteLine("Client.Main(): New server side value {0}", tmp)

            Console.ReadLine()
        End Sub

End Module
```

The sample client will read and output the server's original value, change it to 42, and then read and output it again.

SingleCall Objects

For SingleCall objects the server will create a single object, execute the method, and destroy the object again. SingleCall objects are registered at the server using the following statement:

```
RemotingConfiguration.RegisterWellKnownServiceType( _
    GetType(MyRemoteObject), _
    "MyRemoteObject.soap", _
    WellKnownObjectMode.SingleCall)
```

Objects of this kind can obviously not hold any state information, as all internal variables will be discarded at the end of the method call. The reason for using objects of this kind is that they can be deployed in a very scalable manner. These objects can be located on different computers with an intermediate multiplexing/load-balancing device, which would not be possible when using stateful objects. The complete server for this example can be seen in Listing 3-3.

Listing 3-3. The Complete Server Implementation

```
Option Explicit On
Option Strict On

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

```vbnet
Class MyRemoteObject
    Inherits MarshalByRefObject
    Implements IMyRemoteObject

    Private myvalue As Integer

    Public Sub New()
        Console.WriteLine("MyRemoteObject.Constructor: New Object created")
    End Sub 'New

    Public Sub New(ByVal startvalue As Integer)
        Console.WriteLine("MyRemoteObject.Constructor: .ctor called with {0}", _
            startvalue)
        myvalue = startvalue
    End Sub

    Public Sub setValue(ByVal newval As Integer) Implements IMyRemoteObject.setValue
        Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}", _
            myvalue, newval)
        myvalue = newval
    End Sub

    Public Function getValue() As Integer Implements IMyRemoteObject.getValue
        Console.WriteLine("MyRemoteObject.getValue(): current {0}", myvalue)
        Return myvalue
    End Function

End Class

Module ServerStartup

    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyRemoteObject), _
            "MyRemoteObject.soap", _
            WellKnownObjectMode.SingleCall)
```

```
            ' the server will keep running until keypress.
            Console.ReadLine()

        End Sub
End Module
```

When the program is run, the output in Figure 3-1 will appear on the client.
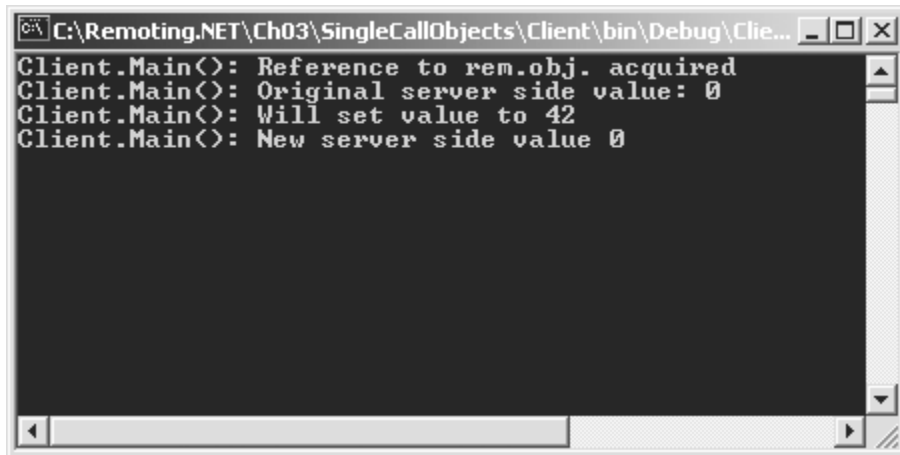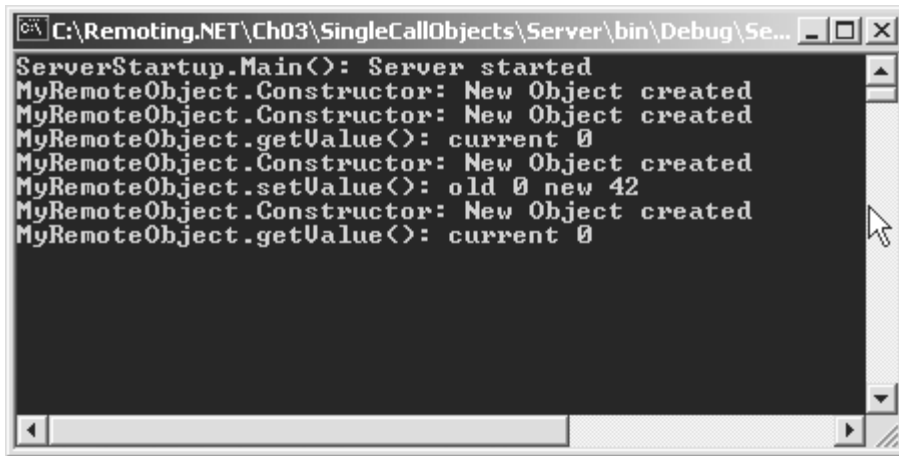
Figure 3-1. Client's output for a SingleCall object

What's happening is exactly what you'd expect from the previous description—even though it might not be what you'd normally expect from an object-oriented application. The reason for the server returning a value of 0 after setting the value to 42 is that your client is talking to a completely different object. Figure 3-2 shows the server's output.

This indicates that the server will really create one object for each call (and an additional object during the first call as well).

Figure 3-2. Server's output for a SingleCall object

Singleton Objects

Only one instance of a Singleton object can exist at any given time. When receiving a client's request, the server checks its internal tables to see if an instance of this class already exists; if not, this object will be created and stored in the table. After this check the method will be executed. The server guarantees that there will be exactly one or no instance available at a given time.

> **NOTE** *Singletons have an associated lifetime as well, so be sure to override the standard lease time if you don't want your object to be destroyed after some minutes. (More on this later in this chapter.)*

For registering an object as a Singleton, you can use the following lines of code:

```
RemotingConfiguration.RegisterWellKnownServiceType( _
    GetType(MyRemoteObject), _
    "MyRemoteObject.soap", _
    WellKnownObjectMode.SingleTon)
```

The module ServerStartup in your sample server will be changed accordingly:

```
Option Explicit On
Option Strict On
Module ServerStartup
    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyRemoteObject), _
            "MyRemoteObject.soap", _
            WellKnownObjectMode.Singleton)

        ' the server will keep running until keypress.
        Console.ReadLine()

    End Sub
End Module
```

When the client is started, the output will show a behavior consistent with the "normal" object-oriented way of thinking; the value that is returned is the same value you set two lines before (see Figure 3-3).
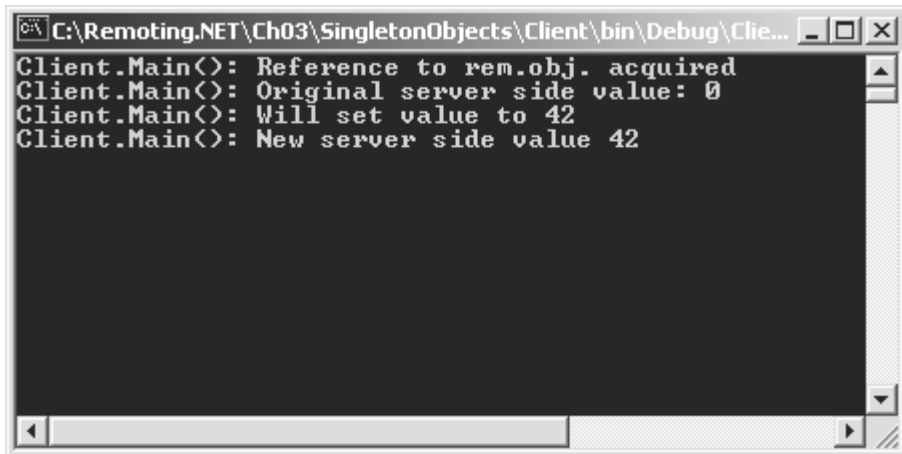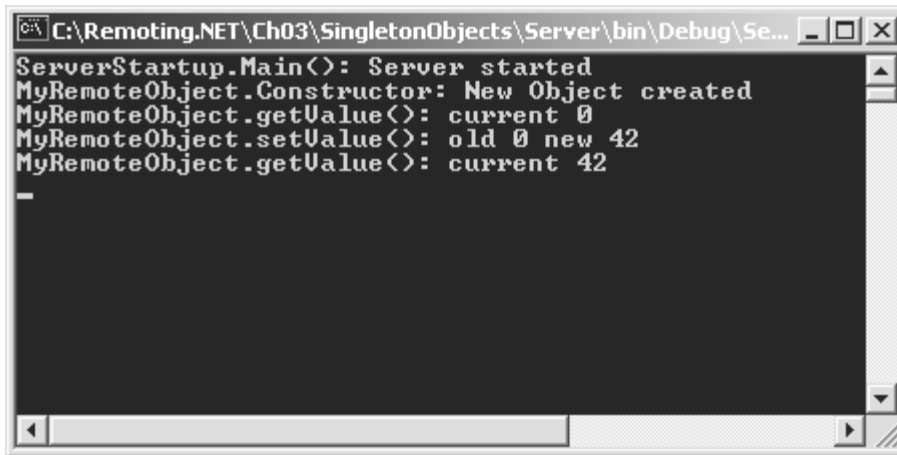

Figure 3-3. Client's output for a Singleton object

The same is true for the server, as Figure 3-4 shows.
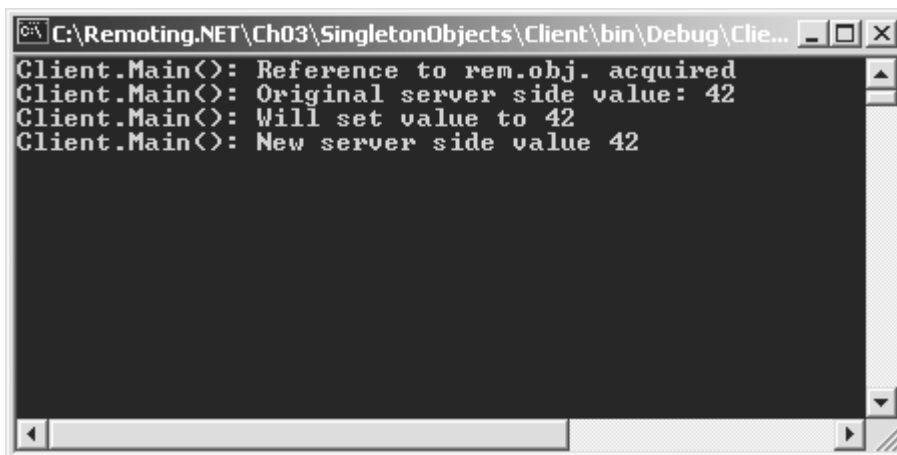
```
C:\Remoting.NET\Ch03\SingletonObjects\Server\bin\Debug\Se...
ServerStartup.Main(): Server started
MyRemoteObject.Constructor: New Object created
MyRemoteObject.getValue(): current 0
MyRemoteObject.setValue(): old 0 new 42
MyRemoteObject.getValue(): current 42
_
```

Figure 3-4. Server's output for a Singleton object
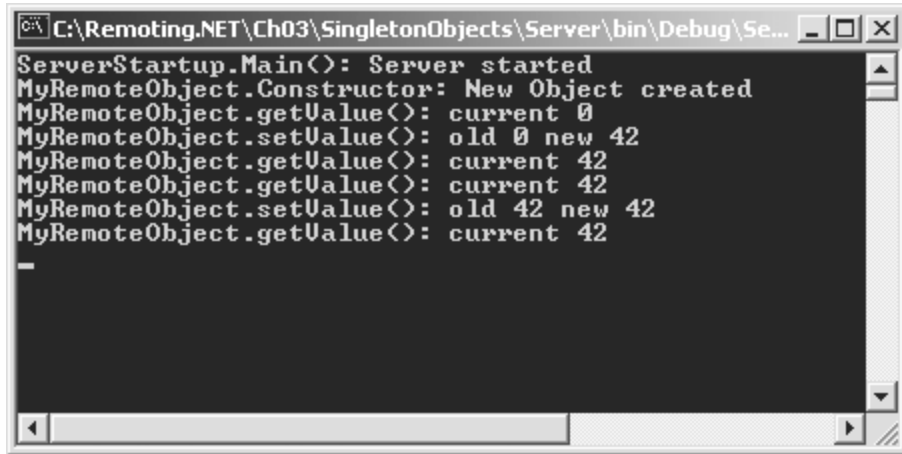
An interesting thing happens when a second client is started afterwards. This client will receive a value of 42 directly after startup without your setting this value beforehand (see Figures 3-5 and 3-6). This is because only one instance exists at the server, and the instance will stay alive even after the first client is disconnected.

> **TIP** *Use Singletons when you want to share data or resources between clients.*


```
C:\Remoting.NET\Ch03\SingletonObjects\Client\bin\Debug\Clie...
Client.Main(): Reference to rem.obj. acquired
Client.Main(): Original server side value: 42
Client.Main(): Will set value to 42
Client.Main(): New server side value 42
```

Figure 3-5. The second client's output when calling a Singleton object

Figure 3-6. Server's output after the second call to a Singleton object

Published Objects

When using either SingleCall or Singleton objects, the necessary instances will be created dynamically during a client's request. When you want to publish a certain object instance that's been precreated on the server—for example, one using a nondefault constructor—neither alternative provides you with a solution.

In this case you can use `RemotingServices.Marshal()` to publish a given instance that behaves like a Singleton afterwards. The only difference is that the object has to already exist at the server before publication.

```
Dim obj as YourObject = New YourObject(<your params for constr>)
RemotingServices.Marshal(obj,"YourUrl.soap")
```

The code in the ServerStartup module will look like this:

```
Option Explicit On
Option Strict On
Module ServerStartup

    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        Dim obj As MyRemoteObject = New MyRemoteObject(4711)
        RemotingServices.Marshal(obj, "MyRemoteObject.soap")
```

```
            ' the server will keep running until keypress.
            Console.ReadLine()


       End Sub
End Module
```

When the client is run, you can safely expect to get a value of 4711 on the first request because you started the server with this initial value (see Figures 3-7 and 3-8).
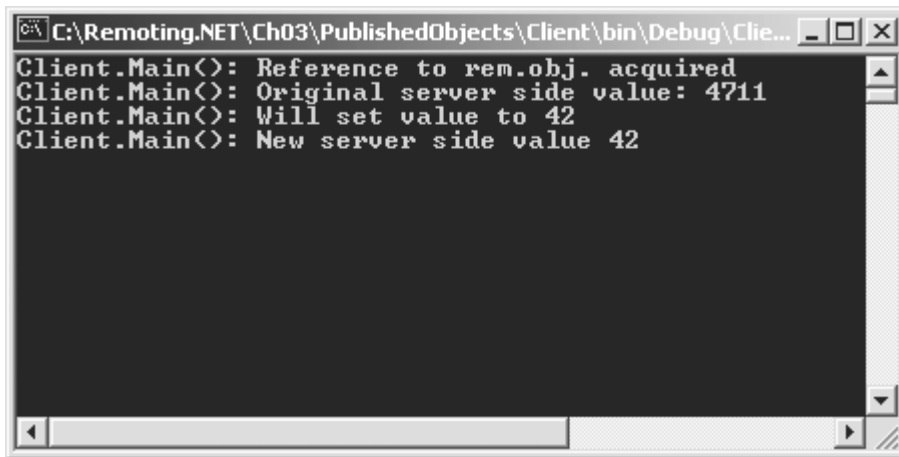

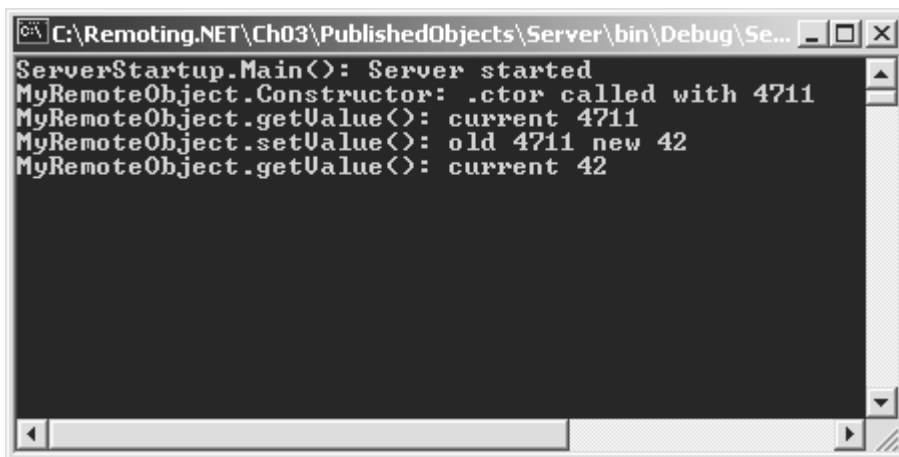*Figure 3-7. Client's output when calling a published object*

Figure 3-8. Server's output when publishing the object

Client-Activated Objects

A client-activated object (CAO) behaves mostly the same way as does a "normal" .NET object (or a COM object). When a creation request on the client is encountered (using `Activator.CreateInstance()` or the `New` operator), an activation message is sent to the server, where a remote object is created. On the client a proxy that holds the ObjRef to the server object is created like it is with SAOs.

A client-activated object's lifetime is managed by the same lifetime service used by SAOs, as shown later in this chapter. CAOs are so-called stateful objects; an instance variable that has been set by the client can be retrieved again and will contain the correct value.[1] These objects will store state information from one method call to the other. CAOs are explicitly created by the client, so they can have distinct constructors like normal .NET objects do.

Direct/Transparent Creation

The .NET Remoting Framework can be configured to allow client-activated objects to be created like normal objects using the `New` operator. Unfortunately, this manner of creation has one serious drawback: you cannot use shared interfaces or base classes. This means that you either have to ship the compiled objects to your clients or use SoapSuds to extract the metadata.

As shipping the implementation to your clients is neither feasible due to deployment and versioning issues nor in support of the general idea of distributed applications, I refrain from delving heavily into this approach here. Unfortunately, it's not currently possible to call nondefault constructors when using SoapSuds-generated metadata. When your application needs this functionality, you might choose the class factory–based approach (which is shown after this example) or rely on SoapSuds' -gc parameter to manually enhance the generated proxy (more on this parameter in Chapter 4).

In the following example, you'll use more or less the same class you did in the previous examples; it will provide your client with a `setValue()` and `getValue()` method to store and retrieve an Integer value as the object's state. The metadata that is needed for the client to create a reference to the CAO will be extracted with SoapSuds.exe, about which you'll read more later in this chapter.

The reliance on SoapSuds allows you to develop the server application without any need for up-front design of a shared assembly, therefore the server will simply include the CAOs implementation. You can see this in Listing 3-4.

---

[1]   The only exception from this rule lies in the object's lifetime, which is managed completely differently from the way it is in .NET generally or in COM.

Listing 3-4. A Server That Offers a Client-Activated Object

```
Option Explicit On
Option Strict On

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


Public Class MyRemoteObject
    Inherits MarshalByRefObject

    Private myvalue As Integer

    Public Sub New()
        Console.WriteLine("MyRemoteObject.Constructor: New Object created")
    End Sub 'New

    Public Sub New(ByVal startvalue As Integer)
        Console.WriteLine("MyRemoteObject.Constructor: .ctor called with {0}", _
            startvalue)
        myvalue = startvalue
    End Sub

    Public Sub setValue(ByVal newval As Integer)
        Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}", _
            myvalue, newval)
        myvalue = newval
    End Sub

    Public Function getValue() As Integer
        Console.WriteLine("MyRemoteObject.getValue(): current {0}", _
            myvalue)
        Return myvalue
    End Function
End Class

Module ServerStartup
    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")
```

```
        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.ApplicationName = "MyServer"
        RemotingConfiguration.RegisterActivatedServiceType( _
            GetType(MyRemoteObject))

        ' the server will keep running until keypress.
        Console.ReadLine()

    End Sub
End Module
```

On the server you now have the new startup code needed to register a chan-nel and this class as a client-activated object. When adding a Type to the list of activated services, you cannot provide a single URL for each object; instead, you have to set the `RemotingConfiguration.ApplicationName` to a string value that identifies your server.

The URL to your remote object will be http://<hostname>:<port>/ <ApplicationName>. What happens behind the scenes is that a general activation SAO is automatically created by the framework and published at the URL http://<hostname>:<port>/<ApplicationName>/RemoteActivationService.rem. This SAO will take the clients' requests to create a new instance and pass it on to the remoting framework.

To extract the necessary interface information, you can run the following SoapSuds command line in the directory where the server.exe assembly has been placed:

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

> **NOTE** *You should perform all command-line operations from the Visual Studio command prompt, which you can bring up by selecting Start ➢ Programs ➢ Microsoft Visual Studio .NET ➢ Visual Studio .NET Tools. This command prompt sets the correct "path" variable to include the .NET SDK tools.*

The resulting generated_metadata.dll assembly must be referenced by the client. The sample client also registers the CAO and acquires two references to (different) remote objects. It then sets the value of those objects and outputs them again, which shows that you really are dealing with two different objects.

As you can see in Listing 3-5, the activation of the remote object is done with the New operator. This is possible because you registered the Type as ActivatedClientType before. The runtime now knows that whenever your application creates an instance of this class, it instead should create a reference to a remote object running on the server.

Listing 3-5. The Client Accesses the Client-Activated Object
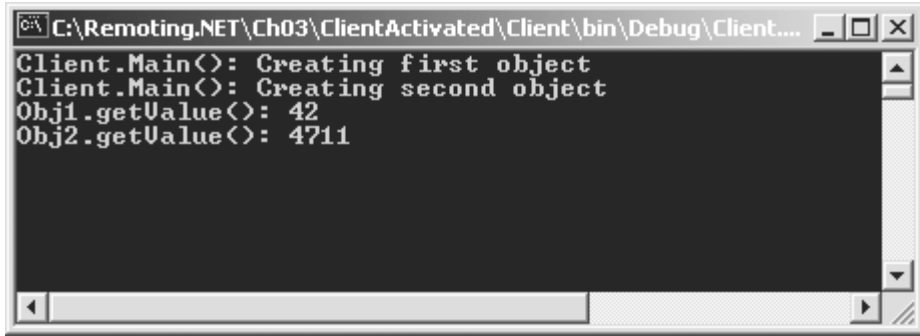
```
Option Explicit On
Option Strict On

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

Module Client
    Sub Main()
        Dim channel As New HttpChannel()
        ChannelServices.RegisterChannel(channel)

        RemotingConfiguration.RegisterActivatedClientType( _
            GetType(Server.MyRemoteObject), "http://localhost:1234/MyServer")

        Console.WriteLine("Client.Main(): Creating first object")
        Dim obj1 As New Server.MyRemoteObject()
        obj1.setValue(42)

        Console.WriteLine("Client.Main(): Creating second object")
        Dim obj2 As New Server.MyRemoteObject()
        obj2.setValue(4711)

        Console.WriteLine("Obj1.getValue(): {0}", obj1.getValue())
        Console.WriteLine("Obj2.getValue(): {0}", obj2.getValue())
        Console.ReadLine()
    End Sub
End Module
```

When this code sample is run, you will see the same behavior as when using local objects—the two instances have their own state (Figure 3-9). As expected, on the server two different objects are created (Figure 3-10).

Figure 3-9. Client-side output when using CAOs

Figure 3-10. Server-side output when using CAOs

Using the Factory Design Pattern

From what you've read up to this point, you know that SoapSuds cannot extract the metadata for nondefault constructors. When your application's design relies on this functionality, you can use a factory design pattern, in which you'll include a SAO providing methods that return new instances of the CAO.

> **NOTE** *You might also just ship the server-side implementation assembly to the client and reference it directly. But as I stated previously, this is clearly against all distributed application design principles!*

Here, I just give you a short introduction to the factory design pattern. Basically you have two classes, one of which is a *factory*, and the other is the real object you want to use. Due to constraints of the real class, you will not be able to construct it directly, but instead will have to call a method on the factory, which creates a new instance and passes it to the client.

Listing 3-6 shows you a fairly simple implementation of this design pattern.

Listing 3-6. The Factory Design Pattern

```
Imports System

Class SomeClass
End Class

Class MyFactory
    Public Function getNewInstance() As SomeClass
        Return New SomeClass()
    End Function
End Class

Module Client
    Sub Main()
        ' creation using "New"
        Dim obj1 As New SomeClass()

        ' creation using a factory
        Dim fac As New MyFactory()
        Dim obj2 As SomeClass = fac.getNewInstance()
    End Sub
End Module
```

When bringing this pattern to remoting, you have to create a factory that's running as a server-activated object (ideally a Singleton) that has a method returning a new instance of the "real class" (the CAO) to the client. This gives you a huge advantage in that you don't have to distribute the implementation to the client system or manually tweak the output from SoapSuds -gc.

> **NOTE** *Distributing the implementation to the client is not only a bad choice due to deployment issues, it also makes it possible for the client user to disassemble your object's codes using ILDASM or some other tool.*

You have to design your factory SAO using a shared assembly which contains the interface information (or abstract base classes) which are implemented by your remote objects. This is shown in Listing 3-7.

Listing 3-7. The Shared Interfaces for the Factory Design Pattern

```
Imports System

Public Interface IRemoteObject
    Sub setValue(ByVal newval As Integer)
    Function getValue() As Integer
End Interface

Public Interface IRemoteFactory
    Overloads Function getNewInstance() As IRemoteObject
    Overloads Function getNewInstance(ByVal initvalue As Integer) As
IRemoteObject
End Interface
```

On the server you now have to implement both interfaces and create a startup code that registers the factory as a SAO. You don't have to register the CAO in this case because every MarshalByRefObject can be returned by a method call; the framework takes care of the necessity to remote each call itself, as shown in Listing 3-8.

Listing 3-8. The Server-Side Factory Pattern's Implementation

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports General
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels

Class MyRemoteObject
    Inherits MarshalByRefObject
    Implements IRemoteObject
```

```vb
    Private myvalue As Integer

    Public Sub New()
        Console.WriteLine("MyRemoteObject.ctor() called")
    End Sub 'New

    Public Sub New(ByVal startvalue As Integer)
        Console.WriteLine("MyRemoteObject.ctor(Integer) called", _
            startvalue)
        myvalue = startvalue
    End Sub

    Public Sub setValue(ByVal newval As Integer) _
    Implements IRemoteObject.setValue
        Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}", _
            myvalue, newval)
        myvalue = newval
    End Sub

    Public Function getValue() As Integer _
    Implements IRemoteObject.getValue
        Console.WriteLine("MyRemoteObject.getValue(): current {0}", myvalue)
        Return myvalue
    End Function
End Class

Class MyRemoteFactory
    Inherits MarshalByRefObject
    Implements IRemoteFactory

    Public Sub New()
        Console.WriteLine("MyRemoteFactory.ctor() called")
    End Sub

    Public Function getNewInstance() As IRemoteObject _
    Implements IRemoteFactory.getNewInstance
        Console.WriteLine("MyRemoteFactory.getNewInstance() called")
        Return New MyRemoteObject()
    End Function
```

```vb
        Public Function getNewInstance(ByVal initvalue As Integer) As IRemoteObject _
        Implements IRemoteFactory.getNewInstance
            Console.WriteLine("MyRemoteFactory.getNewInstance(Integer) called")
            Return New MyRemoteObject(initvalue)
        End Function
End Class


Module ServerStartup
    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")
        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyRemoteFactory), _
            "factory.soap", _
            WellKnownObjectMode.Singleton)

        ' the server will keep running until keypress.
        Console.ReadLine()
    End Sub
End Module
```

The client, which is shown in Listing 3-9, works a little bit differently from the previous one as well. It creates a reference to a remote SAO using `Activator.GetObject()`, upon which it places two calls to `getNewInstance()` to acquire two different remote CAOs.

Listing 3-9. The Client Uses the Factory Pattern

```vb
Option Explicit On
Option Strict On

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

```
Module Client
    Sub Main()
        Dim channel As New HttpChannel()
        ChannelServices.RegisterChannel(channel)

        Console.WriteLine("Client.Main(): Creating factory")
        Dim fact As IRemoteFactory = CType(Activator.GetObject( _
            GetType(IRemoteFactory), _
            "http://localhost:1234/factory.soap"), _
            IRemoteFactory)

        Console.WriteLine("Client.Main(): Acquiring first object from factory")
        Dim obj1 As IRemoteObject = fact.getNewInstance()
        obj1.setValue(42)

        Console.WriteLine("Client.Main(): Acquiring second object from factory")
        Dim obj2 As IRemoteObject = fact.getNewInstance(4711)

        Console.WriteLine("Obj1.getValue(): {0}", obj1.getValue())
        Console.WriteLine("Obj2.getValue(): {0}", obj2.getValue())

        Console.ReadLine()
    End Sub
End Module
```
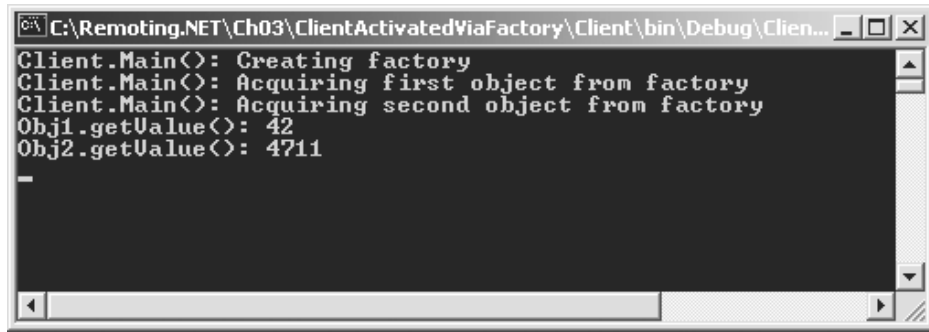
When this sample is running, you see that the client behaves nearly identically to the previous example, but the second object's value has been set using the object's constructor, which is called via the factory (Figure 3-11). On the server a factory object is generated, and each new instance is created using the overloaded getNewInstance() method (Figure 3-12).

Figure 3-11. Client-side output when using a factory object

Figure 3-12. Server-side output when using a factory object

Managing Lifetime

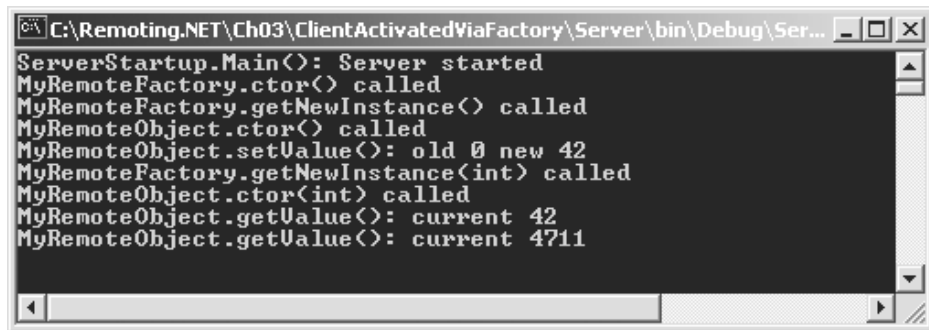One point that can lead to a bit of confusion is the way an object's lifetime is managed in the .NET Remoting Framework. Common .NET objects are managed using a garbage collection algorithm that checks if any other object is still using a given instance. If not, the instance will be garbage collected and disposed.

When you apply this schema (or the COM way of reference counting) to remote objects, it pings the client-side proxies to ensure that they are still using the objects and that the application is still running (this is mainly what DCOM did). The reason for this is that normally a client that has been closed unexpectedly or went offline due to a network outage might not have decremented the server-side reference counter. Without some additional measure, these server-side objects would in turn use the server's resources forever. Unfortunately, when your client is behind an HTTP proxy and is accessing your objects using SOAP remoting, the server will not be able to contact the client in any way.

This constraint leads to a new kind of lifetime service: the lease-based object lifetime. Basically this means that each server-side object is associated with

a lease upon creation. This lease will have a time-to-live counter (which starts at five minutes by default) that is decremented in certain intervals. In addition to the initial time, a defined amount (two minutes in the default configuration) is added to this time to live upon every method call a client places on the remote object.

When this time reaches zero, the framework looks for any sponsors registered with this lease. A *sponsor* is an object running on the server itself, the client, or any machine reachable via a network that will take a call from the .NET Remoting Framework asking whether an object's lifetime should be renewed or not (more on this in Chapter 6).

When the sponsor decides that the lease will not be renewed or when the framework is unable to contact any of the registered sponsors, the object is marked as timed out and then garbage collected. When a client still has a reference to a timed-out object and calls a method on it, it will receive an exception.

To change the default lease times, you can override `InitializeLifetimeService()` in the MarshalByRefObject. In the following example, you see how to change the last CAO sample to implement a different lifetime of only ten milliseconds for this object. Normally LeaseManager only polls all leases every ten seconds, so you have to change this polling interval as well.

```vbnet
Imports System.Runtime.Remoting.Lifetime
' more Imports as above...

Class MyRemoteObject
    Inherits MarshalByRefObject
    Implements IRemoteObject

    Public Overrides Function InitializeLifetimeService() As Object
        Console.WriteLine("MyRemoteObject.InitializeLifetimeService() called")
        Dim lease As ILease = CType(MyBase.InitializeLifetimeService(), ILease)

        If lease.CurrentState = LeaseState.Initial Then
            lease.InitialLeaseTime = TimeSpan.FromMilliseconds(10)
            lease.SponsorshipTimeout = TimeSpan.FromMilliseconds(10)
            lease.RenewOnCallTime = TimeSpan.FromMilliseconds(10)
        End If

        Return lease
    End Function
```

```
        ' rest of implementation...

End Class

Class MyRemoteFactory
    ' rest of implementation...
End Class

Module ServerStartup

    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")

        LifetimeServices.LeaseManagerPollTime = TimeSpan.FromMilliseconds(10)

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyRemoteFactory), _
            "factory.soap", _
            WellKnownObjectMode.Singleton)

        ' the server will keep running until keypress.
        Console.ReadLine()

    End Sub
End Module
```

On the client side, you can add a one-second delay between creation and the first call on the remote object to see the effects of the changed lifetime. You also need to provide some code to handle the RemotingException that will get thrown because the object is no longer available at the server. The client is shown in Listing 3-10.

Listing 3-10. A Client That Calls a Timed-Out CAO

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports General
```

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

Module Client

    Sub Main()
        Dim channel As New HttpChannel()
        ChannelServices.RegisterChannel(channel)

        Console.WriteLine("Client.Main(): Creating factory")
        Dim fact As IRemoteFactory = CType(Activator.GetObject( _
            GetType(IRemoteFactory), _
            "http://localhost:1234/factory.soap"), _
            IRemoteFactory)

        Console.WriteLine("Client.Main(): Acquiring object from factory")
        Dim obj1 As IRemoteObject = fact.getNewInstance()

        Console.WriteLine("Client.Main(): Sleeping one second")
        System.Threading.Thread.Sleep(1000)

        Console.WriteLine("Client.Main(): Setting value")
        Try
            obj1.setValue(42)
        Catch e As Exception
            Console.WriteLine("Client.Main(). EXCEPTION " + vbCrLf + e.Message)
        End Try

        Console.ReadLine()
    End Sub

End Module
```
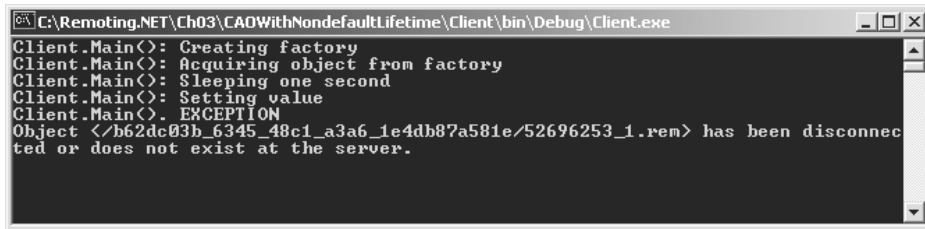
Running this sample, you see that the client is able to successfully create
a factory object and call its getNewInstance() method (Figure 3-13). When calling
setValue() on the returned CAO, the client will receive an exception stating the
object has timed out. The server runs normally (Figure 3-14).

51

Figure 3-13. The client receives an exception because the object has timed out.

Figure 3-14. The server when overriding `InitializeLifetimeService()`

Types of Invocation

The .NET Framework provides three possibilities to call methods on remote objects (no matter if they are Singleton, SingleCall, or published objects). You can execute their methods in a synchronous, asynchronous, or asynchronous one-way fashion.

Synchronous calls are basically what I showed you in the preceding examples. The server's remote method is called like a common method, and the client blocks (waits) until the server has completed its processing. If an exception occurs during execution of the remote invocation, the exception is thrown at the line of code in which you called the server.

Asynchronous calls are executed in a two-step process. (Asynchronous calls are discussed in more detail in Chapter 6.) The first step triggers the execution but does not wait for the method's response value. The program flow continues on the client. When you are ready to collect the function's response, you have to call another function that checks if the server has already finished processing your request; if not, it blocks until finalization. Any exception thrown during the call of your method will be rethrown at the line of code where you collect the response. Even if the server has been offline, you won't be notified beforehand.

The last kind of function is a little different from the preceding ones. With asynchronous one-way methods, you don't have the option of receiving return values or getting an exception if the server has been offline or otherwise unable to fulfill your request. The .NET Remoting Framework will just *try* to call the methods on the remote server and won't do anything else.

Synchronous Calls

As I've mentioned, synchronous calls are the usual way of calling a function in the .NET Framework. The server will be contacted directly and, except when using multiple client-side threads, the client code will block until the server has finished executing its method. If the server is unavailable or an exception occurs while carrying out your request, the exception will be rethrown at the line of code where you called the remote method.

Using Synchronous Calls

In the following series of examples for the different types of invocation, you use a common server and a shared assembly called General.dll (you'll see some slight modifications in the last part). This server just provides you with a Singleton object that stores an Integer as its state and has an additional method that returns a String. You'll use this later to demonstrate the collection of return values when using asynchronous calls.

Defining the General.dll

In Listing 3-11, you see the shared General.dll in which the necessary interface is defined.

Listing 3-11. The Shared Assembly's Source Code

```
Imports System

Public MustInherit Class BaseRemoteObject
    Inherits MarshalByRefObject

    Public MustOverride Sub setValue(ByVal newval As Integer)
    Public MustOverride Function getValue() As Integer
    Public MustOverride Function getName() As String
End Class
```

Creating the Server

The server, shown in Listing 3-12, implements the defined methods with the addition of making the setValue() and getName() functions long-running code. In both methods, a five-second delay is introduced so you can see the effects of long-lasting execution in the different invocation contexts.

Listing 3-12. A Server with Some Long-Running Methods

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels
Imports System.Collections
Imports System.Threading
Imports General

Class MyRemoteObject
    Inherits BaseRemoteObject
    Private myvalue As Integer

    Public Sub New()
        Console.WriteLine("MyRemoteObject.Constructor: New Object created")
    End Sub

    Public Overrides Sub setValue(ByVal newval As Integer)
        Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}", _
            myvalue, newval)

        ' we simulate a long running action
        Console.WriteLine("  .setValue() -> waiting 5 sec before setting value")
        Thread.Sleep(5000)

        myvalue = newval
        Console.WriteLine("  .setValue() -> value is now set")
    End Sub


    Public Overrides Function getValue() As Integer
        Console.WriteLine("MyRemoteObject.getValue(): current {0}", myvalue)
        Return myvalue
    End Function
```

```
    Public Overrides Function getName() As String
        Console.WriteLine("MyRemoteObject.getName(): called")

        ' we simulate a long running action
        Console.WriteLine("  .getName() -> waiting 5 sec before continuing")
        Thread.Sleep(5000)

        Console.WriteLine("  .getName() -> returning name")
        Return "John Doe"
    End Function
End Class


Module ServerStartup

    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyRemoteObject), "MyRemoteObject.soap", _
            WellKnownObjectMode.Singleton)

        ' the server will keep running until keypress.
        Console.ReadLine()
    End Sub
End Module
```

Creating the Client

The first client, which is shown in Listing 3-13, calls the server synchronously, as in all preceding examples. It calls all three methods and gives you statistics on how long the total execution took.

Listing 3-13. The First Client Calls the Methods Synchronously

```vb
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Proxies
Imports General

Module Client

    Sub Main()
        Dim start As DateTime = System.DateTime.Now

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

        Dim obj As BaseRemoteObject = CType(Activator.GetObject( _
            GetType(BaseRemoteObject), _
            "http://localhost:1234/MyRemoteObject.soap"), BaseRemoteObject)

        Console.WriteLine("Client.Main(): Reference to rem.obj. acquired")

        Console.WriteLine("Client.Main(): Will set value to 42")
        obj.setValue(42)

        Console.WriteLine("Client.Main(): Will now read value")
        Dim tmp As Integer = obj.getValue()
        Console.WriteLine("Client.Main(): New server side value {0}", tmp)

        Console.WriteLine("Client.Main(): Will call getName()")
        Dim name As String = obj.getName()
        Console.WriteLine("Client.Main(): received name {0}", name)

        Dim finished As DateTime = System.DateTime.Now
        Dim duration As TimeSpan = finished.Subtract(start)
        Console.WriteLine("Client.Main(): Execution took {0} seconds.", duration.Seconds)

        Console.ReadLine()
    End Sub
End Module
```
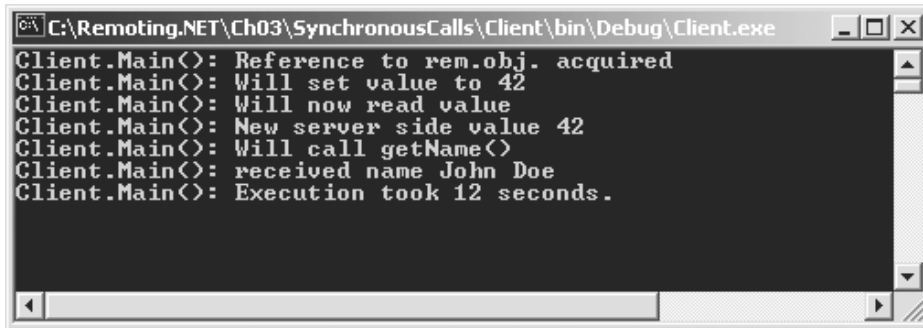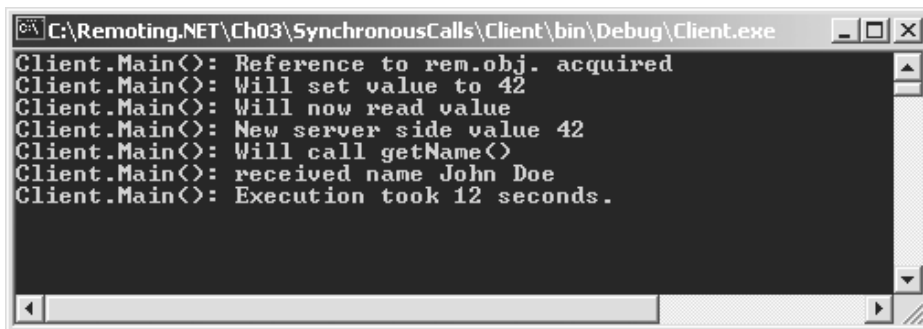
As the calls to the long-running methods getName() and setValue() are expected to take roughly five seconds each, and you have to add a little overhead for .NET Remoting (especially for the first call on a remote object), this example will take more than ten seconds to run.

You can see that this assumption is right by looking at the client's output in Figure 3-15. The total client execution takes 12 seconds. When looking at the server's output in Figure 3-16, note that all methods are called synchronously. Every method is finished before the next one is called by the client.

Figure 3-15. Client's output when using synchronous calls

Figure 3-16. Server's output when called synchronously

Asynchronous Calls

In the synchronous calls example, you saw that waiting for every method to complete incurs a performance penalty if the calls themselves are independent; the second call doesn't need the output from the first. You could now use a separate thread to call the second method, but even though threading is quite simple in .NET, it would probably render the application more complex if you use a distinct thread for any longer lasting remote function call. The .NET Framework provides a feature, called *asynchronous delegates,* that allows methods to be called in an asynchronous fashion with only three lines of additional code.

Delegate Basics

A *delegate* is, in its regular sense, just a kind of an object-oriented function pointer. You will initialize it and pass a function to be called when the delegate is invoked. In .NET Framework, a delegate is a subclass of `System.MulticastDelegate`, but Visual Basic .NET provides an easier way to define a delegate instead of declaring a new `Class`.

Declaring a Delegate

The declaration of a delegate looks quite similar to the declaration of a method:

```
Delegate [Function|Sub] <name> ([parameters]) As <ReturnType>
```

As the delegate will call a method at some point in the future, you have to provide it with a declaration that matches the method's signature. When you want a delegate to call the following method:

```
Public Function doSomething(ByVal myValue As Integer) As String
```

you have to define it as follows:

```
Delegate Function doSthDelegate(ByVal myValue As Integer) As String
```

> **NOTE** *The delegate's parameter and return types have to match those of the method.*

Remember that the delegate is in reality just another class, so you cannot define it within a method's body, only directly within a namespace, another class, or another module!

Asynchronously Invoking a Delegate

When you want to use a delegate, you first have to create an instance of it, passing the address of the method to be called as a constructor parameter:

```
Dim del As New doSthDelegate(AddressOf doSomething)
```

> **NOTE** *When passing the method to the AddressOf Operator, be sure not to include an opening or closing parenthesis — (or) —as in* `doSomething()`. *The previous example uses a method* `doSomething` *in the same module. When using shared methods of other classes, you have to pass* `SomeClass.someMethod`, *and for instance methods, you pass* `SomeObject.doSomething`.

The asynchronous invocation of a delegate is a two-step process. In the first step, you have to trigger the execution using `BeginInvoke()`, as follows:

```
Dim ar As IAsyncResult = del.BeginInvoke(42, Nothing, Nothing)
```

`BeginInvoke()` then returns an IAsyncResult object that will be used later to retrieve the method's return values. When ready to do so, you call `EndInvoke()` on the delegate passing the IAsyncResult as a parameter. The `EndInvoke()` method will block until the server has completed executing the underlying method.

```
Dim res As String = del.EndInvoke(ar)
```

Creating an Example Delegate

In Listing 3-14, a delegate is used to asynchronously call a local function and wait for its result. The method returns a String built from the passed Integer parameter.

Listing 3-14. Using a Delegate in a Local Application

```
Option Explicit On
Option Strict On

Imports System

Module DelegateDemo

    Delegate Function doSthDelegate(ByVal myValue As Integer) As String

    Public Function doSomething(ByVal myValue As Integer) As String
        Return "HEY:" + myValue.ToString()
    End Function

    Sub Main()
        Dim del As New doSthDelegate(AddressOf doSomething)
        Dim ar As IAsyncResult = del.BeginInvoke(42, Nothing, Nothing)
        '...do something different here
        Dim res As String = del.EndInvoke(ar)

        Console.WriteLine("Got result: '{0}'", res)

        ' wait for return to close
        Console.ReadLine()
    End Sub

End Module
```

As expected, the application outputs "HEY:42" as you can see in Figure 3-17.
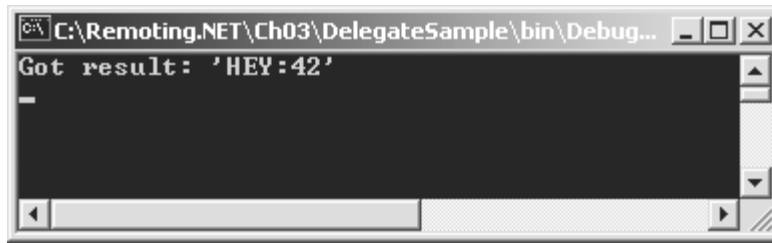
Figure 3-17. The sample delegate

Implementing the New Client

In the new remoting client, shown in Listing 3-15, you see how to change the calls to getName() and setValue() to use delegates as well. Your client then invokes both delegates and subsequently waits for their completion before synchronously calling getValue() on the server. In this instance, you use the same server application as in the preceding example.

Listing 3-15. The New Client Now Using Asynchronous Delegates

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Proxies
Imports General

Module Client

    Delegate Sub SetValueDelegate(ByVal value As Integer)
    Delegate Function GetNameDelegate() As String

    Sub Main()
        Dim start As DateTime = System.DateTime.Now

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

```vbnet
        Dim obj As BaseRemoteObject = CType(Activator.GetObject( _
            GetType(BaseRemoteObject), _
            "http://localhost:1234/MyRemoteObject.soap"), BaseRemoteObject)

        Console.WriteLine("Client.Main(): Reference to rem.obj. acquired")

        Console.WriteLine("Client.Main(): Will call setValue(42)")
        Dim svDelegate As New SetValueDelegate(AddressOf obj.setValue)
        Dim svAsyncres As IAsyncResult = svDelegate.BeginInvoke(42, Nothing, _
            Nothing)
        Console.WriteLine("Client.Main(): Invocation done")

        Console.WriteLine("Client.Main(): Will call getName()")
        Dim gnDelegate As New GetNameDelegate(AddressOf obj.getName)
        Dim gnAsyncres As IAsyncResult = gnDelegate.BeginInvoke(Nothing, Nothing)
        Console.WriteLine("Client.Main(): Invocation done")

        Console.WriteLine("Client.Main(): EndInvoke for setValue()")
        svDelegate.EndInvoke(svAsyncres)
        Console.WriteLine("Client.Main(): EndInvoke for getName()")
        Dim name As String = gnDelegate.EndInvoke(gnAsyncres)

        Console.WriteLine("Client.Main(): received name {0}", name)

        Console.WriteLine("Client.Main(): Will now read value")
        Dim tmp As Integer = obj.getValue()
        Console.WriteLine("Client.Main(): New server side value {0}", tmp)

        Dim finished As DateTime = System.DateTime.Now
        Dim duration As TimeSpan = finished.Subtract(start)
        Console.WriteLine("Client.Main(): Execution took {0} seconds.", _
            duration.Seconds)

        Console.ReadLine()
    End Sub
End Module
```
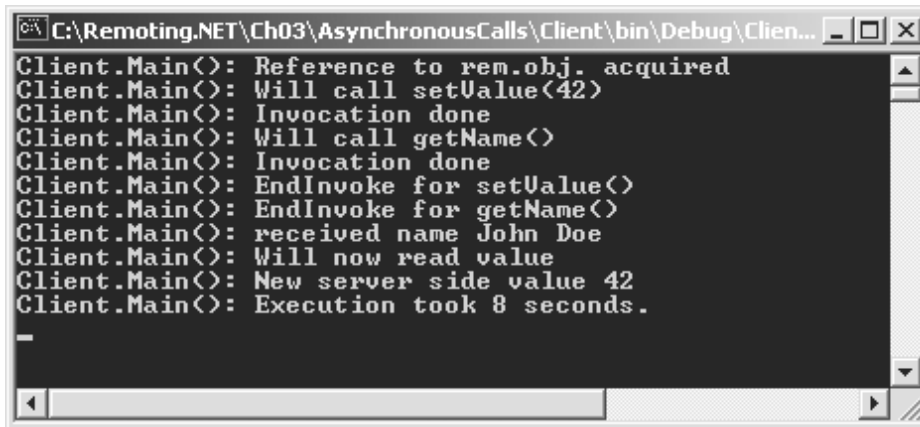
When looking in the client's output in Figure 3-18, you can see that both long-running methods have been called at nearly the same time. This results in improved runtime performance, taking the execution time down from 12 seconds to 8 at the expense of making the application slightly more complex.
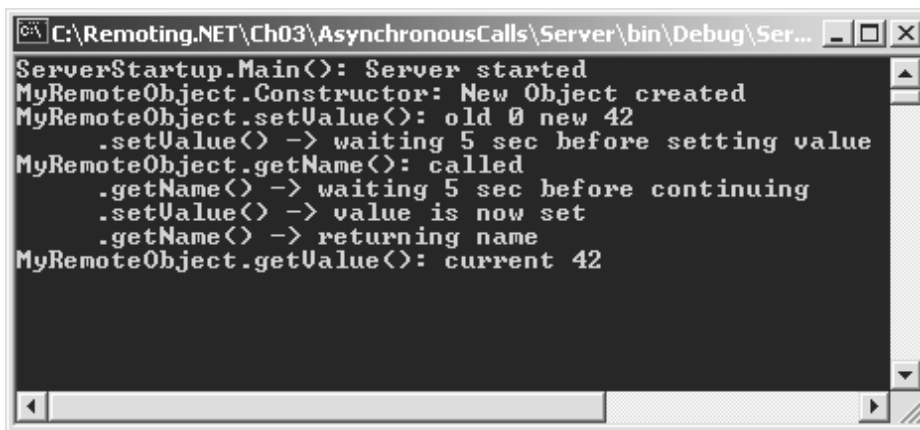
Figure 3-18. Client output when using asynchronous calls

The server output in Figure 3-19 shows that both methods have been entered on the server at the same time without blocking the client.

Figure 3-19. Server's output when called asynchronously

Asynchronous One-Way Calls

One-way calls are a little different from asynchronous calls in the respect that the .NET Framework does not guarantee their execution. In addition, the methods used in this kind of call cannot have return values or out parameters. You also use delegates to call one-way methods, but the EndInvoke() function will exit immediately without checking if the server has finished processing yet.

No exceptions are thrown, even if the remote server is down or the method call is malformed. Reasons for using these kind of methods (which aren't guaranteed to be executed at all) can be found in uncritical logging or tracing facilities, where the nonexistence of the server should not slow down the application.

Demonstrating an Asynchronous One-Way Call

You define one-way methods using the <OneWay()> attribute. This happens in the defining metadata (in the General.dll in these examples) and doesn't need a change in the server or the client.

Defining the General.dll

The attribute `<OneWay()>` has to be specified in the interface definition of each method that will be called this way. As shown in Listing 3-16, you change only the `setValue()` method to become a one-way method; the others are still defined as earlier.

Listing 3-16. The Shared Interfaces DLL Defines the One-Way Method

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting.Messaging

Public MustInherit Class BaseRemoteObject
    Inherits MarshalByRefObject

    <OneWay()> _
    Public MustOverride Sub setValue(ByVal newval As Integer)
    Public MustOverride Function getValue() As Integer
    Public MustOverride Function getName() As String
End Class
```

Implementing the Client

On the server side, no change is needed, so you can directly look at the client. In theory, no modification is needed for the client as well, but extend it a little here to catch the eventual exception during execution, as shown in Listing 3-17.

Listing 3-17. Try/Catch Blocks Are Added to the Client

```vb
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Proxies
Imports General

Module Client

    Delegate Sub SetValueDelegate(ByVal value As Integer)

    Sub Main()
        Dim start As DateTime = System.DateTime.Now

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

        Dim obj As BaseRemoteObject = CType(Activator.GetObject( _
            GetType(BaseRemoteObject), _
            "http://localhost:1234/MyRemoteObject.soap"), BaseRemoteObject)

        Console.WriteLine("Client.Main(): Reference to rem.obj. acquired")

        Console.WriteLine("Client.Main(): Will call setValue(42)")
        Dim svDelegate As New SetValueDelegate(AddressOf obj.setValue)
        Dim svAsyncres As IAsyncResult = svDelegate.BeginInvoke(42, Nothing, _
            Nothing)

        Console.WriteLine("Client.Main(): Invocation done")

        Console.WriteLine("Client.Main(): EndInvoke for setValue()")
        Try
            svDelegate.EndInvoke(svAsyncres)
            Console.WriteLine("Client.Main(): EndInvoke returned successfully")
        Catch e As Exception
            Console.WriteLine("Client.Main(): EXCEPTION during EndInvoke")
        End Try
```

```
                ' wait...
                Console.ReadLine()
        End Sub
End Module
```

When this client is started, you will see the output in Figure 3-20 *no matter whether the server is running or not.*
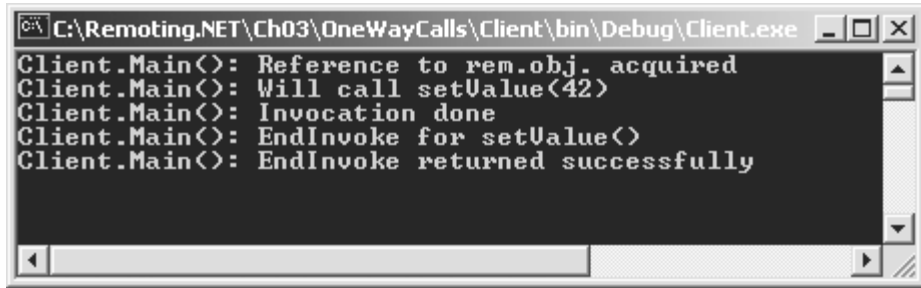
Figure 3-20. Client output when using one-way methods

As shown in Listing 3-18, you can now change the method in General.dll back to a standard method (non–one-way) by commenting out the `<OneWay()>` attribute.

Listing 3-18. Removing the <OneWay()> Attribute

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting.Messaging

Public MustInherit Class BaseRemoteObject
    Inherits MarshalByRefObject

    ' No more one-way attribute <OneWay()>
    Public MustOverride Sub setValue(ByVal newval As Integer)
    Public MustOverride Function getValue() As Integer
    Public MustOverride Function getName() As String
End Class
```

Recompilation and a restart of the client (still without a running server) yields the result in Figure 3-21: an exception is thrown and a corresponding error message is output.

Figure 3-21. Client output when removing the <OneWay()> attribute

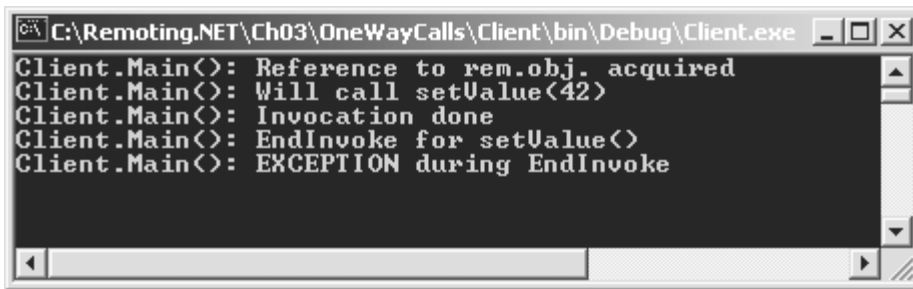When you now start the server (and restart the client), you get the output shown in Figure 3-22, no matter if you used the <OneWay()> attribute or not. The interesting thing is that when using <OneWay()>, the call to EndInvoke() finishes *before* the server completes the method. This is because in reality the client just *ignores* the server's response when using one-way method calls.

> **CAUTION**   *Always remember that the client* ignores *the server's output and doesn't even check if the server is running when using one-way methods!*

Figure 3-22. Output on the server—independent of <OneWay()> attribute

Multiserver Configuration

When using multiple servers in an application in which remote objects on one server will be passed as parameters to methods of a second server's object, there are a few things you need to consider.

Before talking about cross-server execution, I show you some details of remoting with MarshalByRefObjects. As the name implies, these objects are marshaled by reference—instead of passing a copy of the object over the network, only a pointer to this object, known as an ObjRef, will travel. Contrary to common pointers in languages like C++, ObjRefs don't reference a memory address but instead contain a network address (like a TCP/IP address and TCP port) and an object ID that's employed on the server to identify which object instance is used by the calling client. (You can read more on ObjRefs in Chapter 7.) On the client side these ObjRefs are encapsulated by a proxy object (actually, by two proxies, but you also get the chance to read more on those in Chapter 7).

After creating two references to client-activated objects on a remote server, for example, the client will hold two TransparentProxy objects. These objects will both contain an ObjRef object, which will in turn point to one of the two distinct CAOs. This is shown in Figure 3-23.
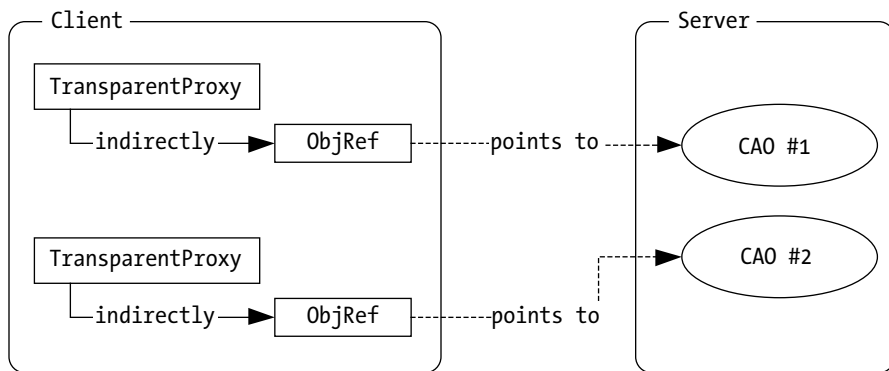
Figure 3-23. ObjRefs are pointing to server-side objects.

When a variable referencing a MarshalByRefObject is passed as a parameter to a remote function, the following happens: the ObjRef is taken from the proxy object, gets serialized (ObjRef is `<Serializable()>`), and is passed to the remote machine (the second server in this example). On this machine, new proxy objects are generated from the deserialized ObjRef. Any calls from the second machine to the remote object are placed *directly* on the first server without any intermediate steps via the client.

> **NOTE**   *As the second server will contact the first one directly, there has to be a means of communication between them; that is, if there is a firewall separating the two machines, you have to configure it to allow connections from one server to the other.*

Examining a Sample Multiserver Application

In the following example, I show you how to create a multiserver application in which Server 1 will provide a Singleton object that has an instance variable of type Integer. The client will obtain a remote reference to this object and pass it to a "worker object" located on a secondary server. This worker object is a SingleCall service providing a doSomething() method, which takes an instance of the first object as a parameter. Figure 3-24 shows the Unified Modeling Language (UML) diagram for this setup.
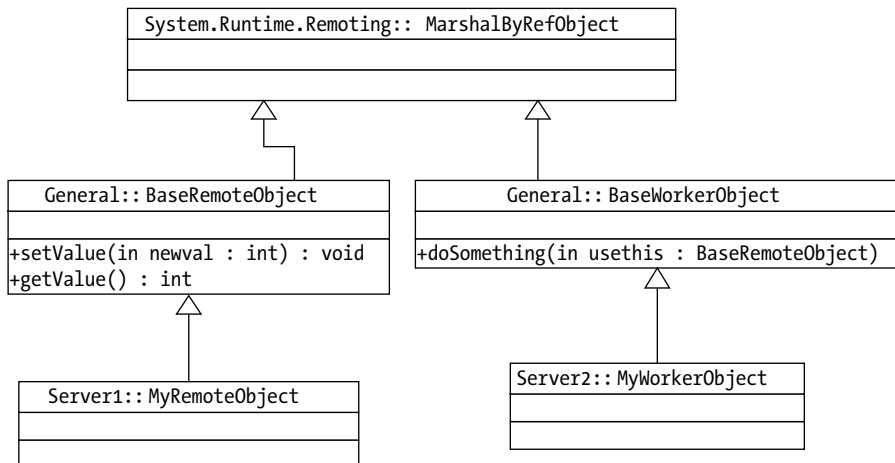
Figure 3-24. UML diagram of the multiserver example

> **NOTE** *For this example, I change the approach from using interfaces in General.dll to using abstract base classes. The reason for the change is that, upon passing a MarshalByRefObject to another server, the ObjRef is serialized and deserialized. On the server side, during the deserialization, the .NET Remoting Framework will generate a new proxy object and afterwards will try to downcast it to the correct type (cast from MarshalByRefObject to BaseRemoteObject in this example). This is possible because the ObjRef includes information about the type and its class hierarchy. Unfortunately, the .NET Remoting Framework does not also serialize the interface hierarchy in the ObjRef, so these interface casts would not succeed.*

Figures 3-25 to 3-27 illustrate the data flow between the various components. In Figure 3-25, you see the situation after the first method call of the client on the first server object. The client holds a proxy object containing the ObjRef that points to the server-side Singleton object.

> **NOTE** *I use IDs like MRO#1 for an instance of MyRemoteObject not because that's .NET-like, but because it allows me to more easily refer to a certain object instance when describing the architecture.*
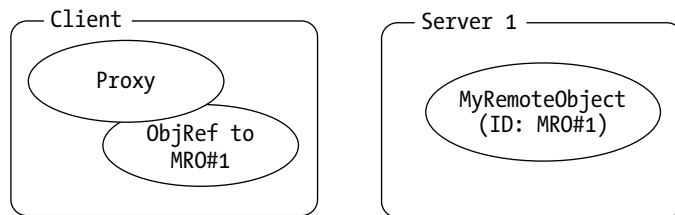
Figure 3-25. Client and single server

In the next step, which you can see in Figure 3-26, the client obtains a reference to the MarshalByRefObject called MyWorkerObject on the second server. It calls a method and passes its reference to the first server's object as a parameter. The ObjRef to this object (MRO#1) is serialized at the client and deserialized at the server, and a new proxy object is generated that sits on the second server and points to the object on the first (Figure 3-27). When MWO#1 now calls a method on MRO#1, the call will go directly from Server 2 to Server 1.
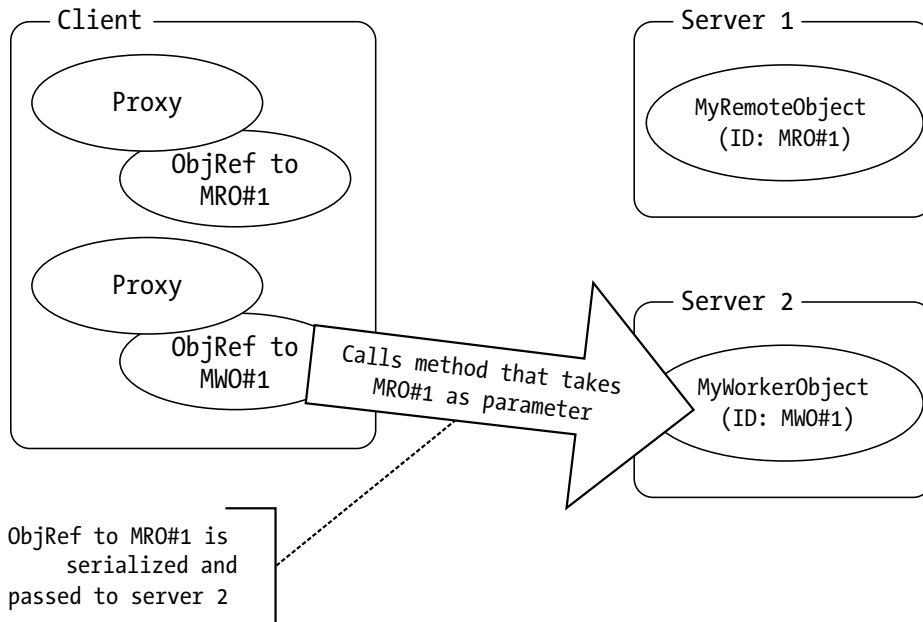
Figure 3-26. Client calls a method on the second server with MRO#1 as parameter.
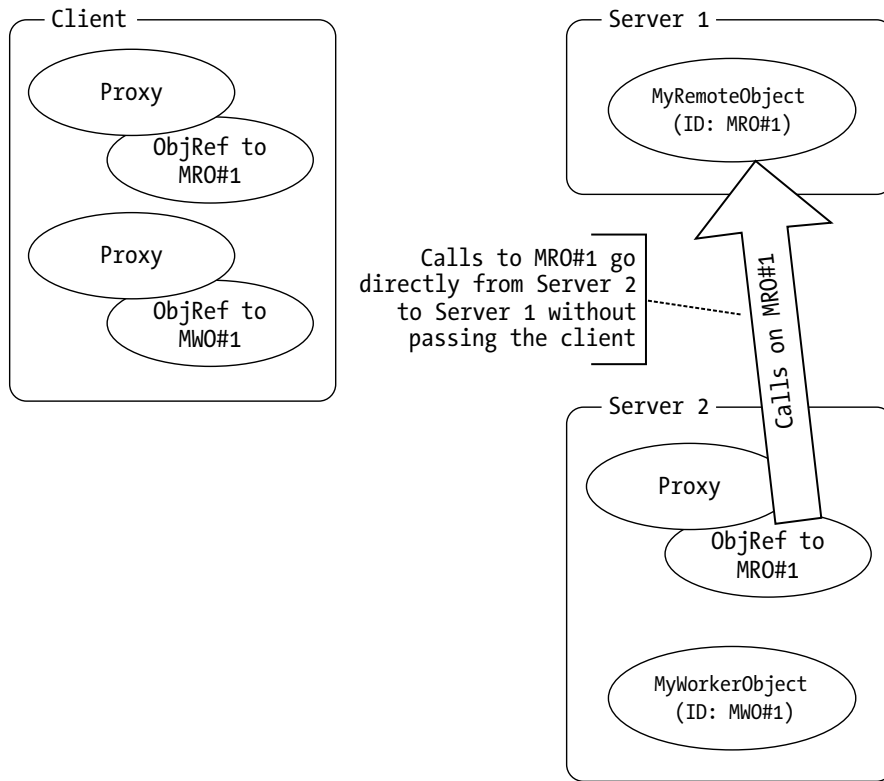
Figure 3-27. Calls to the first server will go there directly without passing the client.

Implementing the Shared Assembly

In the shared assembly, which is shown in Listing 3-19, you have to change the approach from using interfaces (which have been used in the prior examples) to abstract base classes because of the reasons stated previously. These are the superclasses of the classes you will implement in the two servers, therefore they have to descend from MarshalByRefObject as well.

Listing 3-19. Using Abstract Base Classes in the Shared Assembly

```
Option Explicit On
Option Strict On

Imports System
```

```
Public MustInherit Class BaseRemoteObject
    Inherits MarshalByRefObject
    Public MustOverride Sub setValue(ByVal newval As Integer)
    Public MustOverride Function getValue() As Integer
End Class


Public MustInherit Class BaseWorkerObject
    Inherits MarshalByRefObject
    Public MustOverride Sub doSomething(ByVal usethis As BaseRemoteObject)
End Class
```

The BaseRemoteObject's descendant is a Singleton located on the first server, and it allows the client to set and read an Integer as state information. The BaseWorkerObject's implementation is placed in Server 2 and provides a method that takes an object of type BaseRemoteObject as a parameter.

Implementing the First Server

The first server very closely resembles the servers from the other examples. The only difference is that MyRemoteObject is no direct child of MarshalByRefObject, but instead is a descendant of BaseRemoteObject, defined in the shared assembly.

This object, implemented as a Singleton, is shown in Listing 3-20.

Listing 3-20. The First Server

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports General
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels

Class MyRemoteObject
    Inherits BaseRemoteObject
    Private myvalue As Integer

    Public Sub New()
        Console.WriteLine("MyRemoteObject.Constructor: New Object created")
    End Sub
```

```
    Public Overrides Sub setValue(ByVal newval As Integer)
        Console.WriteLine("MyRemoteObject.setValue(): old {0} new {1}", _
            myvalue, newval)

        myvalue = newval
    End Sub

    Public Overrides Function getValue() As Integer
        Console.WriteLine("MyRemoteObject.getValue(): current {0}", myvalue)
        Return myvalue
    End Function
End Class



Module ServerStartup

    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server [1] started")

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)
        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyRemoteObject), _
            "MyRemoteObject.soap", _
            WellKnownObjectMode.Singleton)

        ' the server will keep running until keypress.
        Console.ReadLine()
    End Sub
End Module
```

Implementing the Second Server

The second server works differently from those in prior examples. It provides a SingleCall object that accepts a BaseRemoteObject as a parameter. The SAO will contact this remote object, read and output its state, and change it before returning.

The server's startup code is quite straightforward and works the same as in the preceding examples. It opens an HTTP channel on port 1235 and registers the well-known object. This second server is shown in Listing 3-21.

> **NOTE**   *When running two servers on one machine, you have to give the servers different port numbers. Only one application can occupy a certain port at any given time. When developing production-quality applications, you should always allow the user or system administrator to configure the port numbers in a configuration file, via the registry or using a GUI.*

Listing 3-21. The Second Server

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports General
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels
Imports System.Collections

Class MyWorkerObject
    Inherits BaseWorkerObject

    Public Sub New()
        Console.WriteLine("MyWorkerObject.Constructor: New Object created")
    End Sub 'New

    Public Overrides Sub doSomething(ByVal usethis As BaseRemoteObject)
        Console.WriteLine("MyWorkerObject.doSomething(): called")
        Console.WriteLine(("MyWorkerObject.doSomething(): Will now call" & _
            "getValue() on the remote obj."))

        Dim tmp As Integer = usethis.getValue()
        Console.WriteLine("MyWorkerObject.doSomething(): current value of " & _
            "the remote obj.: {0}", tmp)

        Console.WriteLine("MyWorkerObject.doSomething(): changing value to 70")
        usethis.setValue(70)
    End Sub
End Class
```

```
Module ServerStartup

    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server [2] started")

        Dim chnl As New HttpChannel(1235)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(MyWorkerObject), _
            "MyWorkerObject.soap", _
            WellKnownObjectMode.SingleCall)

        ' the server will keep running until keypress.
        Console.ReadLine()
    End Sub
End Module
```

Running the Sample

When the client is started, it first acquires a remote reference to MyRemoteObject
running on the first server. It then changes the object's state to contain the
value 42 and afterwards reads the value from the server and outputs it in
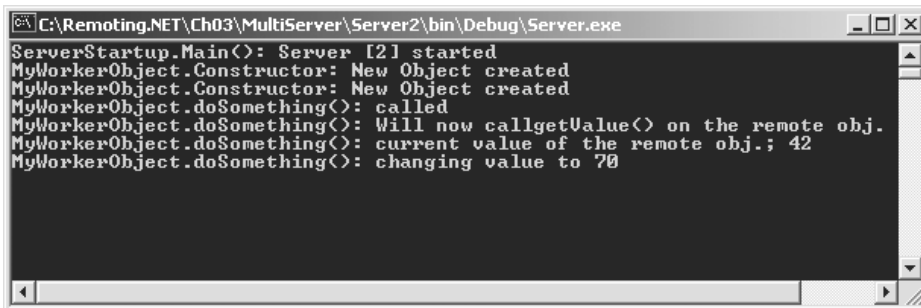the console window (see Figure 3-28).

Figure 3-28. The client's output

Next it fetches a remote reference to MyWorkerObject running on the second server. The client calls the method doSomething() and passes its reference to MyRemoteObject as a parameter. When Server 2 receives this call, it contacts Server 1 to read the current value from MyRemoteObject and afterwards changes it to 70. (See Figures 3-29 and 3-30.)


```
C:\Remoting.NET\Ch03\MultiServer\Server1\bin\Debug\Server.exe
ServerStartup.Main(): Server [1] started
MyRemoteObject.Constructor: New Object created
MyRemoteObject.setValue(): old 0 new 42
MyRemoteObject.getValue(): current 42
MyRemoteObject.getValue(): current 42
MyRemoteObject.setValue(): old 42 new 70
MyRemoteObject.getValue(): current 70
```

Figure 3-29. The first server's output


```
C:\Remoting.NET\Ch03\MultiServer\Server2\bin\Debug\Server.exe
ServerStartup.Main(): Server [2] started
MyWorkerObject.Constructor: New Object created
MyWorkerObject.Constructor: New Object created
MyWorkerObject.doSomething(): called
MyWorkerObject.doSomething(): Will now callgetValue() on the remote obj.
MyWorkerObject.doSomething(): current value of the remote obj.; 42
MyWorkerObject.doSomething(): changing value to 70
```

Figure 3-30. The second server's output

When the call from client to the second server returns, the client again contacts MyRemoteObject to obtain the current value, 70, which shows that your client really has been talking to the same object from both processes.

Shared Assemblies

As you've seen in this chapter, .NET Remoting applications need to share common information about remoteable types between server and client. Contrary to other remoting schemas like CORBA, Java RMI, and J2EE EJBs, with which you don't have a lot of choice for writing these shared interfaces, base classes, and metadata, the .NET Framework gives you at least four possible ways to do so, as I discuss in the following sections.

Shared Implementation

The first way to share information about remoteable types is to implement your server-side objects in a shared assembly and deploy this to the client as well. The main advantage here is that you don't have any extra work. Even though this might save you some time during implementation, I really recommend against this approach. Not only does it violate the core principles of distributed application development, but it also allows your clients, which are probably third parties accessing your ERP system to automate order entry, to use ILDASM or one of the upcoming MSIL-to-high-level-language decompilers to disassemble and view your business logic. Unfortunately, this approach is shown in several MSDN examples.

Nevertheless, there are application scenarios that depend on this way of sharing the metadata. When you have an application that can be used either connected or disconnected and will access the same logic in both cases, this might be the way to go. You can then "switch" dynamically between using the local implementation and using the remote one.

Shared Interfaces

In the first examples in this book, I show the use of shared interfaces. With this approach, you create an assembly that is copied to both the server and the client. The assembly contains the interfaces that will be implemented by the server. The disadvantage to using this process of sharing the metadata is that you won't be able to pass those objects as parameters to functions running in a different context (either on the same or another server or on another client) because the resulting MarshalByRefObject cannot be downcast to these interfaces.

Shared Base Classes

Instead of sharing interfaces between the client and the server, you can also create abstract base classes in a shared assembly. The server-side object will inherit from these classes and implement the necessary functionality. The big advantage here is that abstract base classes are, contrary to the shared interfaces, capable of being passed around as parameters for methods located in different AppDomains. Still, this approach has one disadvantage: you won't be able to use those objects without `Activator.GetObject()` or a factory. Normally when the .NET Framework is configured correctly on the client side, it is possible to use the `New` operator to create a reference to a remote object. Unfortunately, you can never create a new instance of an abstract class or an interface, so the compiler will block this functionality.

SoapSuds-Generated Metadata

As each of the other approaches has a drawback, let's see what SoapSuds can do for you. This program's functionality is to extract the metadata (that is, the type definition) from a running server or an implementation assembly and generate a new assembly that contains only this meta information. You will then be able to reference this assembly in the client application without manually generating any intermediate shared assemblies.

Calling SoapSuds

SoapSuds is a command-line utility, therefore the easiest way to start it is to bring up the Visual Studio .NET Command Prompt by selecting Start ➢ Programs ➢ Microsoft Visual Studio .NET ➢ Visual Studio .NET Tools. This command prompt will have the path correctly set so that you can execute all .NET Framework SDK tools from any directory.

Starting SoapSuds without any parameters will give you detailed usage information. To generate a metadata DLL from a running server, you have to call SoapSuds with the -url parameter:

```
soapsuds -url:<URL> -oa:<OUTPUTFILE>.DLL -nowp
```

> **NOTE** *You normally have to append ?wsdl to the URL your server registered for a SOA to allow SoapSuds to extract the metadata.*

To let SoapSuds extract the information from a compiled DLL, you use the -ia parameter:

```
soapsuds -ia:<assembly> -oa:<OUTPUTFILE>.DLL -nowp
```

Wrapped Proxies

When you run SoapSuds in its default configuration (without the -nowp parameter) by passing only a URL as an input parameter and telling it to generate an assembly, it will create what is called a *wrapped proxy*. The wrapped proxy can only be used on SOAP channels and will directly store the path to your server. Normally you do not want this.

> **NOTE** *This behavior is useful when you want to access a third-party Web Service whose application URL you happen to have.*

I normally recommend using wrapped proxies only when you want to quickly test a SOAP remoting service. As an example, in the next section I show you how to implement a server without previously specifying any shared interfaces or base classes.

Implementing the Server

The server in this example will be implemented without any up-front definition of interfaces. You only need to create a simplistic SAO and register an HTTP channel to allow access to the metadata and the server-side object, as shown in Listing 3-22.

Listing 3-22. Server That Presents a SAO

```
Option Explicit On
Option Strict On

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

Class SomeRemoteObject
    Inherits MarshalByRefObject

    Public Sub doSomething()
        Console.WriteLine("SomeRemoteObject.doSomething() called")
    End Sub
End Class

Module ServerStartup
    Sub Main()
        Console.WriteLine("ServerStartup.Main(): Server started")

        Dim chnl As New HttpChannel(1234)
        ChannelServices.RegisterChannel(chnl)

        RemotingConfiguration.RegisterWellKnownServiceType( _
            GetType(SomeRemoteObject), _
```

```
            "SomeRemoteObject.soap", _
            WellKnownObjectMode.SingleCall)

        ' the server will keep running until keypress.
        Console.ReadLine()
    End Sub
End Module
```

Generating the SoapSuds Wrapped Proxy

To generate a wrapped proxy assembly, use the SoapSuds command line shown in Figure 3-31. The resulting meta.dll should be copied to the client directory, as you will have to reference it when building the client-side application.
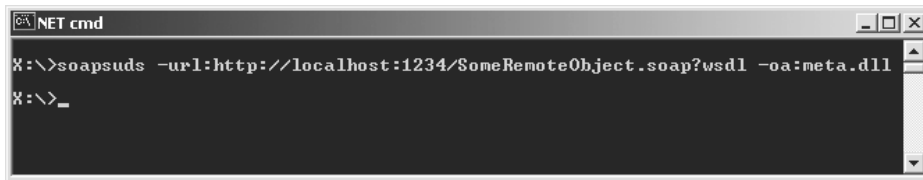

Figure 3-31. SoapSuds command line used to generate a wrapped proxy

Implementing the Client

Assuming you now want to implement the client application, you first have to set a reference to the meta.dll in the project's References dialog box in VS .NET or employ the /r:meta.dll parameter to the command-line compiler. You can then use the Server namespace and directly instantiate a SomeRemoteObject using the New operator, as shown in Listing 3-23.

Listing 3-23. Wrapped Proxies Simplify the Client's Source Code

```
Option Explicit On
Option Strict On

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

```
Module Client

    Sub Main()
        Console.WriteLine("Client.Main(): creating rem. reference")
        Dim obj As New SomeRemoteObject()
        Console.WriteLine("Client.Main(): calling doSomething()")
        obj.doSomething()
        Console.WriteLine("Client.Main(): done ")

        Console.ReadLine()
    End Sub
End Module
```

Even though this code looks intriguingly simply, I recommend against using a wrapped proxy for several reasons: the server's URL is hard coded, and you can only use an HTTP channel and not a TCP channel.

When you start this client, it will generate the output shown in Figure 3-32. Check the server's output in Figure 3-33 to see that doSomething() has really been called on the server-side object.

Figure 3-32. Client's output when using a wrapped proxy

Figure 3-33. The server's output shows that doSomething() *has been called.*

Wrapped Proxy Internals

Starting SoapSuds with the parameter -gc instead of -oa:<assemblyname> will generate C# code in the current directory. You can use this code to manually compile a DLL or include it directly in your project. However, it's unfortunately not possible to generate Visual Basic. NET code with SoapSuds.

Looking at the code in Listing 3-24 quickly reveals why you can use it without any further registration of channels or objects. (I strip the SoapType attribute, which would normally contain additional information on how to remotely call the object's methods.)

Listing 3-24. A SoapSuds-Generated Wrapped Proxy

```csharp
using System;
using System.Runtime.Remoting.Messaging;
using System.Runtime.Remoting.Metadata;
using System.Runtime.Remoting.Metadata.W3cXsd2001;

namespace Server {

    public class SomeRemoteObject :
            System.Runtime.Remoting.Services.RemotingClientProxy
    {
        // Constructor
        public SomeRemoteObject()
        {
            base.ConfigureProxy(this.GetType(),
                "http://localhost:1234/SomeRemoteObject.soap");
        }

        public Object RemotingReference
        {
            get{return(_tp);}
        }

[SoapMethod(SoapAction="http://schemas.microsoft.com/clr/nsassem/
Server.SomeRemoteObject/Server#doSomething")]
        public void doSomething()
        {
            ((SomeRemoteObject) _tp).doSomething();
        }

    }
}
```

What this wrapped proxy does behind the scenes is provide a custom implementation/extension of RealProxy (which is the base for RemotingClientProxy) so that it can be used transparently. This architecture is shown in detail in Chapter 7.

Nonwrapped Proxy Metadata

Fortunately, SoapSuds allows the generation of nonwrapped proxy metadata as well. In this case, it will only generate empty class definitions, which can then be used by the underlying .NET Remoting TransparentProxy to generate the true method calls—no matter which channel you are using.

This approach also gives you the huge advantage of being able to use configuration files for channels, objects, and the corresponding URLs (more on this in the next chapter) so that you don't have to hard code this information. In the following example, you can use the same server as in the previous example, only changing the SoapSuds command and implementing the client in a different way.

Generating the Metadata with SoapSuds

As you want to generate a metadata-only assembly, you have to pass the -nowp parameter to SoapSuds to keep it from generating a wrapped proxy (see Figure 3-34).
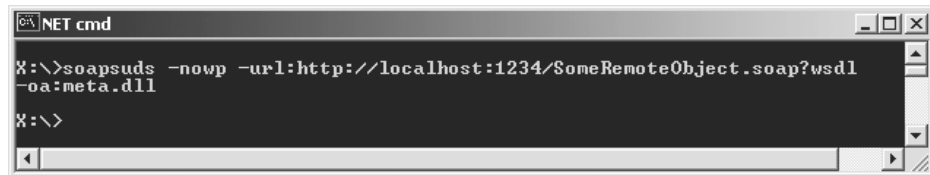


```
C:\NET cmd                                                    _ □ ×
X:\>soapsuds -nowp -url:http://localhost:1234/SomeRemoteObject.soap?wsdl
-oa:meta.dll

X:\>
```

Figure 3-34. SoapSuds command line for a metadata-only assembly

Implementing the Client

When using metadata-only output from SoapSuds, the client looks a lot different from the previous one. In fact, it closely resembles the examples I show you at the beginning of this chapter.

First you have to set a reference to the newly generated meta.dll from the current SoapSuds invocation and indicate that your client will be using this namespace. You can then proceed with the standard approach of creating and registering a channel and calling `Activator.GetObject()` to create a reference to the remote object. This is shown in Listing 3-25.

Listing 3-25. The Client with a Nonwrapped Proxy

```
Option Explicit On
Option Strict On

Imports System
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels.Http
Imports System.Runtime.Remoting.Channels
Imports Server

Module Client

    Sub Main()
        Dim chnl As New HttpChannel()
        ChannelServices.RegisterChannel(chnl)

        Console.WriteLine("Client.Main(): creating rem. reference")

        Dim obj As SomeRemoteObject = CType(Activator.GetObject( _
            GetType(SomeRemoteObject), _
            "http://localhost:1234/SomeRemoteObject.soap"), SomeRemoteObject)

        Console.WriteLine("Client.Main(): calling doSomething()")
        obj.doSomething()

        Console.WriteLine("Client.Main(): done ")
        Console.ReadLine()
    End Sub
End Module
```

When this client is started, both the client-side and the server-side output will be the same as in the previous example (see Figures 3-35 and 3-36).
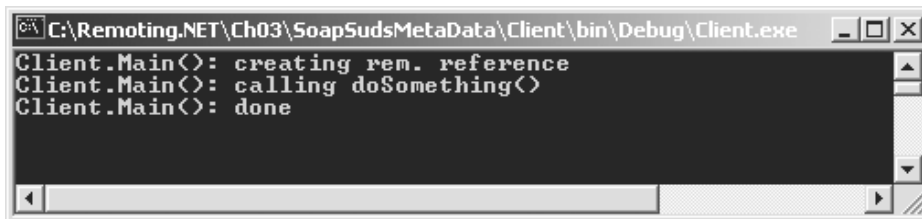

Figure 3-35. The client's output when using a metadata-only assembly

Figure 3-36. The server's output is the same as in the previous example.

Summary

In this chapter you read about the basics of distributed .NET applications using .NET Remoting. You now know the difference between ByValue objects and MarshalByRefObjects, which can be either server-activated objects (SAO) or client-activated objects (CAO). You can call remote methods asynchronously, and you know about the dangers and benefits of one-way methods. You also learned about the different ways in which a client can receive the necessary metadata to access remote objects, and that you should normally use the -nowp parameter with SoapSuds.

It seems that the only thing that can keep you from developing your first real-world .NET Remoting application is that you don't yet know about various issues surrounding configuration and deployment of such applications. These two topics are covered in the following chapter.