Programmer to Programmer ™







C#

Web Services

Building Web Services with .NET Remoting and ASP.NET

Ashish Banerjee, Aravind Corera, Zach Greenvoss, Andrew Krowczyk,
Christian Nagel, Chris Peiris, Thiru Thangarathinam, Brad Maiani

wrox
Programmer to Programmer

Summary of Contents

.NET Remoting Architecture

In the last few chapters, we have seen how ASP.NET web services can be created and used. ASP.NET web services require the ASP.NET runtime as hosting environment. Using .NET Remoting directly, we can host a web service in any application we want. .NET Remoting allows much more flexibility because different transport protocols may be used, we can get a performance increase with different formatting options, and it's possible to host the server in different application types.

The next four chapters will deal with .NET Remoting. In this chapter, we will look at the architecture of .NET Remoting, and go into detail in the following areas:

- What is .NET Remoting?
- .NET Remoting Fundamentals
- Object Types
- Activation
- Marshaling
- Asynchronous Remoting
- Call Contexts

The next chapter will show different application types, transport protocols, and formatting options, and Chapters 8 and 9 will show an example of using .NET Remoting. Let's begin the chapter with a look at .NET Remoting.

> *As with the previous chapters, the code for the examples in this chapter can be downloaded from http://www.wrox.com.*

What is .NET Remoting?

.NET Remoting is the replacement for DCOM. As we have seen in the last chapters, ASP.NET web services are an easy-to use-technology to call services across a network. ASP.NET web services can be used as a communication link with different technologies, for example to have a COM or a Java client talk to web services developed with ASP.NET. As good as this technology is, however, it is not fast and flexible enough for some business requirements in intranet solutions, and ASP.NET web services requires the ASP.NET runtime. With .NET Remoting we get **Web Services Anywhere** that can run in every application type.

Web Services Anywhere

The term "Web Services Anywhere" means that web services can not only be used in any application, but any application can offer web services. ASP.NET web services require the IIS to run; web services that make use of .NET Remoting can run in any application type: console applications, Windows Forms applications, Windows services, and so on. These web services can use **any transport** with **any payload encoding**.

> *In the next chapter we will talk about when and how to use .NET Remoting with different transports (TCP and HTTP) and about different payload encoding mechanisms (SOAP and binary).*

CLR Object Remoting

The next part of .NET Remoting that we need to be aware of is **CLR Object Remoting.** With CLR Object Remoting we can call objects across the network, as if they were being called locally.

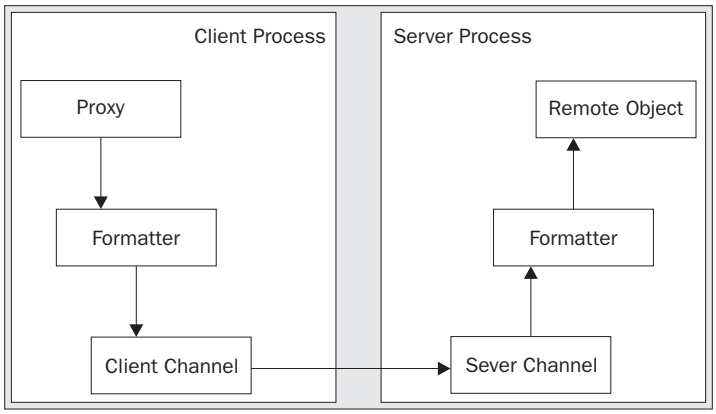
With CLR Object Remoting we have:

- **Distributed Identities** – Remote objects can have a distributed identity. If we pass a reference to a remote object, we will always access the same object using this reference.
- **Activation** – Remote objects can be activated using the `new` operator. Of course, there are other ways to activate remote objects, as we will see later.
- **Lease-Based Lifetime** – How long should the object be activated on the server? At what time can we assume that the client no longer needs the remote object? DCOM uses a ping mechanism that is not scalable to internet solutions. .NET Remoting takes a different approach with a lease-based lifetime that is scalable.
- **Call Context** – With the SOAP header, additional information can be passed with every method call that is not passed as an argument.

We will cover all of these CLR Object Remoting features in detail later on in this chapter.

.NET Remoting Fundamentals

The methods that will be called from the client are implemented in a remote object class. In the figure opposite we can see an instance of this class as the Remote Object. Because this remote object runs inside a process that is different from the client process – usually also on a different system – the client can't call it directly. Instead the client uses a **proxy**. For the client, the proxy looks like the real object with the same public methods. When the methods of the proxy are called, **messages** will be created. These are serialized using a **formatter** class, and are sent into a **client channel**. The client channel communicates with the server part of the channel to transfer the message across the network. The **server channel** uses a **formatter** to deserialize the message, so that the methods can be dispatched to the **remote object**:



In the simplest case, we have to create a remote object class and instantiate a channel for a .NET Remoting application. The formatter and the proxy will be supplied automatically. The architecture is very flexible in that different formatters and channels can be used. We cover the use of the TCP and HTTP channels, and the SOAP and binary formatters in the next chapter.

In the next section we'll start with the simplest case to develop a remoting object, server, and client. In this section we will not go into the details of the remoting architecture, as we will cover this later after we've finished a simple client and server.

Remote Object

A remote object is implemented in a class that derives from `System.MarshalByRefObject`. `MarshalByRefObject` defines methods for lifetime services that will be described later when we use the leasing features. A remote object is confined to the application domain where it is created. As we already know, a client doesn't call the methods directly; instead a proxy object is used to invoke methods on the remote object. Every public method that we define in the remote object class is available to be called from clients.

> *What is an **application domain**? Before .NET, processes were used as a security boundary so that one process couldn't crash another process because it used private virtual memory. With the help of the managed environment of .NET, the code of an application can be checked, and there's no way to crash the process. To reduce the overhead with different processes, the concept of an application domain was introduced with .NET. Multiple applications can run in the same process without influencing each other if they are called within different application domains. If one of these applications throws an exception that isn't handled, only the application domain is terminated, and not the complete process. To invoke a method in an object running in a different application domain, .NET remoting must be used.*

The following code sample shows a simple remote object class, and the code for this simple example can be found in the `SimpleTest` folder of the code download for this chapter, which is available from http://www.wrox.com. The method `Hello()` is declared public to make it available for a remoting client:

```
// MyRemoteObject.cs

using System;
namespace Wrox.Samples
{
```

```
    public class MyRemoteObject : System.MarshalByRefObject
    {
       public MyRemoteObject()
       {
          Console.WriteLine("Constructor called");
       }
       public string Hello()
       {
          Console.WriteLine("Hello called");
          return "Hello, .NET Client!";
       }
    }
}
```

It is useful to implement the class of the remote object in a different assembly from that of the remote server itself. This assembly can then be used in different server applications, and the client application can use it to get the metadata needed to build the proxy.

We build the assembly `MyRemoteObject` from the class `MyRemoteObject` as follows:

```
csc /t:library /out:MyRemoteObject.dll MyRemoteObject.cs
```

Server

The remote object needs a server process where it will be instantiated. This server has to create a channel and put it into listening mode so that clients can connect to this channel. In this chapter, we will use simple console applications as hosting servers, and in the next chapter we will look at the different application types.

Server Configuration File

The server channel can be configured programmatically or by using a configuration file.

Using configuration files for remoting clients and servers has the advantage that the channel and remote object can be configured without changing a single line of code and without the need to recompile. Another advantage is that the remoting code we have to write is very short.

Specifying the options programmatically has the advantage that we could get to the information during runtime. One way to implement this can be that the client uses a directory service to locate a server that has registered its long running objects there.

Configuration files have the advantage that the channel, endpoint name, and so on, can be changed without changing a single line of code and without doing a recompile. We will use configuration files first before we look at what happens behind the scenes when doing the configuration programmatically.

For versioning and probing of assemblies, we can have application configuration files with the same name as the executable, but with a `.config` *file extension, such as* `SimpleServer.exe.config`*. The .NET Remoting configuration can be put into a different file or the same file. We have to read the .NET Remoting configuration explicitly, so it doesn't matter from a programming viewpoint. From the administrator viewpoint it would be useful to put the .NET Remoting configuration inside the same file as this can be used to configure the channel with the .NET Admin tool.*

When the client connects to the remote object it needs to know the URI of the object, that is, the name of the host where the remote object is running, the protocol and port number to connect to, the name of the server, and the name of the remote object. Such a connection string can look like this:

```
tcp://localhost:9000/SimpleServer/MyRemoteObject
```

With the exception of the host name we have to specify all these items in the configuration file.

In the following configuration file, `SimpleServer.exe.config`, all of the remoting configurations must be added as child elements to `<system.runtime.remoting>`. The `<application>` element specifies the name of the server with the `name` attribute. The application offers a service and requires the configureation of channels for the service. Correspondingly we have the `<service>` and `<channels>` elements. The service that is offered from the application must be listed as a child of `<service>`. This is the remote object itself. The remote object is specified with the `<wellknown>` element.

The remoting framework uses the information to create an object from the type specified. For instantiating the object the framework requires the name of the assembly to know where the type of this object can be loaded from. We can set this information with the XML attribute `type`. `type="Wrox.Samples.MyRemoteObject, MyRemoteObject"` defines that the type of the remote object is `MyRemoteObject` in the namespace `Wrox.Samples`, and it can be found in the assembly `MyRemoteObject`. The `mode` attribute is set to `SingleCall`. We will talk later about all the different object types and modes. With `objectURI` we set the endpoint name of the remote object that will be used from the client.

The name of the assembly is often confused with the name of the file in which the assembly is stored. The assembly name is `MyRemoteObject`*, while the file of the assembly is* `MyRemoteObject.dll`*. With method calls where an assembly name is needed as argument, never use the file extension.*

In the `<channels>` section we define the channels that will be used from the server. There are already some channels defined in the machine configuration file `machine.config` that we can use from our applications. Such a predefined channel can be referenced using the `ref` attribute of the `<channel>` element. Here we reference the predefined server channel using the TCP protocol: `tcp server`. We have to assign the port of this channel with the `port` attribute, as the server must have a well-known port number that the client must be aware of:

```
<configuration>
   <system.runtime.remoting>
      <application name="SimpleServer">
         <service>
            <wellknown
               mode="SingleCall"
               type="Wrox.Samples.MyRemoteObject, MyRemoteObject"
               objectUri="MyRemoteObject" />
```

```
            </service>
            <channels>
               <channel ref="tcp server" port="9000" />
            </channels>
         </application>
      </system.runtime.remoting>
   </configuration>
```

Machine.config

We can find predefined channels in the machine-wide configuration file `machine.config`. This configuration file is in the directory `%SystemRoot%\Microsoft.NET\Framework\<vx.x.x>\CONFIG`.

Six channels are predefined in this file, as we can see in the following XML segment. The `id` attribute defines the identifier of the channel that can be used with the `ref` attribute as we have done in the application configuration file to reference the `tcp server` channel. The `type` attribute defines the class and assembly name of the channel. With the `id` we can easily guess the protocol that is used by channel. The `id` also if the channel can be used on the client or server side. The `http` and `tcp` channels include both client and server functionality:

```
         <channels>
             <channel
                 id="http"
                 type="System.Runtime.Remoting.Channels.Http.HttpChannel,
                 System.Runtime.Remoting,
                 Version=1.0.3300.0, Culture-Neutral,PublicKeyToken-b77a5c561934e089" />
             <channel
                 id="http client"
                 type="System.Runtime.Remoting.Channels.Http.HttpClientChannel,
                 System.Runtime.Remoting,
                 Version=1.0.3300.0, Culture-Neutral,PublicKey Token-b77a5c561934e089" />
             <channel
                 id="http server"
                 type="System.Runtime.Remoting.Channels.Http.HttpServerChannel,
                 System.Runtime.Remoting,
                 Version=1.0.3300.0, Culture-Neutral,PublicKey Token-b77a5c561934e089" />
             <channel
                 id="tcp"
                 type="System.Runtime.Remoting.Channels.Tcp.TcpChannel,
                 System.Runtime.Remoting,
                 Version=1.0.3300.0, Culture-Neutral,PublicKey Token-b77a5c561934e089" />
             <channel
                 id="tcp client"
                 type="System.Runtime.Remoting.Channels.Tcp.TcpClientChannel,
                 System.Runtime.Remoting,
                 Version=1.0.3300.0, Culture-Neutral,PublicKey Token-b77a5c561934e089" />
             <channel
                 id="tcp server"
                 type="System.Runtime.Remoting.Channels.Tcp.TcpServerChannel,
                 System.Runtime.Remoting,
                 Version=1.0.3300.0, Culture-Neutral,PublicKey Token-b77a5c561934e089" />
         </channels>
```

Starting the Channel

All the server has to do is read the configuration file and activate the channel. This can be done with a single call to the static method `RemotingConfiguration.Configure()`.

Here the server is implemented in a console application. `RemotingConfiguration.Configure()` reads the configuration file `SimpleServer.exe.config` to configure and activate the channel. The creation of the remote object and communication with the client is done by the remoting infrastructure; we just have to make sure that the process doesn't end. We do this with `Console.ReadLine()` that will end the process when the user enters the return key:

```
// SimpleServer.cs

using System;
using System.Runtime.Remoting;
namespace Wrox.Samples
{
   class SimpleServer
   {
      static void Main(string[] args)
      {
         RemotingConfiguration.Configure("SimpleServer.exe.config");
         Console.WriteLine("Press return to exit");
         Console.ReadLine();
      }
   }
}
```

We compile the file `SimpleServer.cs` to a console application:

```
csc /target:exe SimpleServer.cs
```

> **We have to either copy the assembly of the remote object class to the directory of the server executable, or make a shared assembly and install it in the global assembly cache. The compiler doesn't complain that we are not referencing it because we didn't use the type `MyRemoteObject` in our server application. But the class will be instantiated from the remoting framework by reading the configuration file, so the assembly must be in a place where it can be found. If you get the exception System.Runtime.Remoting.RemotingException: cannot load type Wrox.Samples.MyRemoteObject while running the client application, remember this issue.**

Client

Creating the client is as simple as creating the server. Here we will create a client using a configuration file.

Client Configuration File

The client configuration file `SimpleClient.exe.config` uses the XML `<client>` element to specify the URL of the server using `protocol://hostname:port/application`. In this example we use `tcp` as the protocol, and the server runs on `localhost` with the port number `9000`. The application name of the server is defined with the `name` attribute of the `<application>` element in the server configuration file.

The <wellknown> element specifies the remote object we want to access. As in the server configuration file, the type attribute defines the type of the remote object and the assembly. The url attribute defines the path to the remote object. Appended to the URL of the application is the endpoint name MyRemoteObject. The channel that is configured with the client can again be found in the configuration file machine.config, but this time it is the client channel:

```
<configuration>
   <system.runtime.remoting>
      <application name="SimpleClient">
         <client url="tcp://localhost:9000/SimpleServer">
            <wellknown
               type="Wrox.Samples.MyRemoteObject, MyRemoteObject"
               url =
               "tcp://localhost:9000/SimpleServer/MyRemoteObject"
             />
         </client>
         <channels>
            <channel ref="tcp client" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Client Application

As in the server, we can activate the client channel by calling RemotingConfiguration.Configure(). Using configuration files we can simply use new to create the remote object. Next we call the method Hello() of this object:

```
// SimpleClient.cs

using System;
using System.Runtime.Remoting;
namespace Wrox.Samples
{
   class SimpleClient
   {
      static void Main(string[] args)
      {
         RemotingConfiguration.Configure("SimpleClient.exe.config");
         MyRemoteObject obj = new MyRemoteObject();
         Console.WriteLine(obj.Hello());
     }
   }
}
```

We can compile the client to a console application as we've done before with the server:

```
csc /target:exe /reference:MyRemoteObject.dll SimpleClient.cs
```

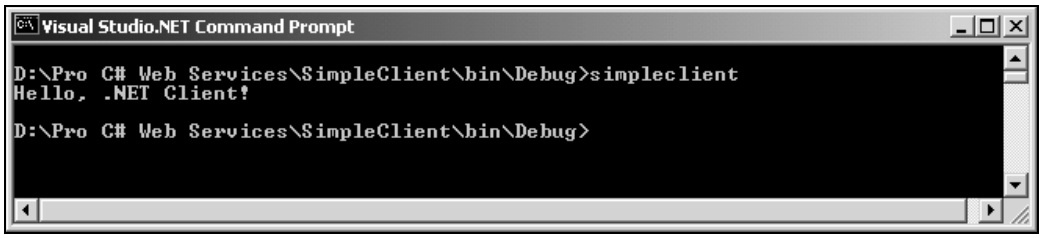
The assembly of the remote object must also be copied for the client application as well, but here it is clearer as we have to reference it explicitly in the compiler command.

Running the Server and the Client

Let's take a step back to look at the files we created so far. This table helps to summarize the files and assemblies and their purposes:

Class	Source File	Assembly File	Description
`MyRemote Object`	`MyRemote Object.cs`	`MyRemote Object.dll`	Remote object class. We offer the `Hello()` method.
`SimpleServer`	`Simple Server.cs`	`Simple Server.exe`	The server creates and registers a server channel. We use the configuration file `SimpleServer.exe.config` to configure the channel and the remote object.
`SimpleClient`	`Simple Client.cs`	`Simple Client.exe`	The client application. The configuration file `SimpleClient.exe.config` defines how to connect to the remote object.

Starting the server and then the client we get the following output from the client:



This is the output screen from the server. We can see that the constructor of the remote object is called twice. The remoting infrastructure instantiates the remote object once before the client activation takes place:

As we have seen it was an easy task to build a simple client and server: create a remote object class, and implement a client and a server. If we are using configuration files, only one remoting call is necessary to read the configuration file and start the channel.

There's a lot more that is provided by .NET remoting. In the next section we will look at more of the terms of this architecture, and discuss all the sub-namespaces and their purpose.

More .NET Remoting

As we have seen it was an easy task to build a simple client and server. We created a remote object that will run on the server, and registered a channel using a configuration file. For simple remoting applications that's all what is needed, but a lot more is possible with .NET Remoting. Let us discuss some terms of the architecture:

- **Remote Objects**

  As we have seen now, a remote object class must derive from the class `MarshalByRefObject`. Calling this object across application domains requires a proxy. .NET Remoting supports two types of remote objects: **well-known** and **client-activated**. For **well-known** objects a URI is used for identification. The client uses the URI to access such objects, which is why they are called well-known. For well-known objects we have two modes: **SingleCall** and **Singleton**. With a `SingleCall` object, the object is created new with every method call; it is a **stateless** object. A `Singleton` object is created only once. `Singleton` objects share state for all clients.

  The other object type that is supported with .NET Remoting is **client-activated**. This object type uses the type of the class for activation. The URI is created dynamically behind the scenes and returned to the client. In further calls to the remote object this URI is used automatically to call methods in the same instance that was created. Client-activated objects are **stateful**.

- **Activation**

  Well-known `SingleCall` objects are not created at the time when the client invokes the `new` method. Instead, they are created with every method call. They could also be called server-activated. Client-activated objects are created at the time when the client instantiates the object; that's why they are called client-activated. In both cases the client gets a **proxy** of the remote object. We have to differentiate a transparent proxy from a real proxy. The **transparent proxy** is created dynamically using the metadata of the remote object class. At proxy creation the methods and arguments of the public methods of the remote object are read from the metadata, and the proxy makes these methods available to the client. For the client the transparent proxy looks like the remote object class, but instead it calls methods of the **real proxy** to send the method calls across the network.

- **Marshaling**

  The process when data is sent across application domains is called marshaling. Sending a variable as argument of a remote method, this variable must be converted so that it can be sent across application domains. We differentiate two kinds of marshaling: Marshal-by-value (MBV) and Marshal-by-reference (MBR). With **Marshal-by-value** the data is serialized to a stream, and this stream is sent across. **Marshal-by-reference** creates a proxy on the client that is used to communicate with the remote object.

- **Interception**

  To fully understand the remoting architecture we have to discuss interception. With interception we can put some functionality into the method call chain. For example, if we call a method of an object, the interception layer could catch the call to convert the method call, or do some logging. Interception is used in every part of the call chain with .NET remoting.

  For interception **sink** objects are used. A sink can perform some actions with messages it receives, for example conversions or logging. Sinks are connected in sink chains; one sink passes the message to the next sink. We can differentiate formatter sinks, custom channel sinks, and transport sinks. **Formatter sinks** transform the messages into a message stream. The **transport sink** is the last sink in the chain in the client to send the message into the channel, and the first sink in the server to receive the message. Transport sinks are built into the channel. With **custom channel sinks** the interception mechanism can be used to add custom functionality to read and write the data received and implement logging, add additional headers, or to redirect messages, for example.

Now that we are more familiar with remoting terms we can look into the classes of the remoting namespace.

System.Runtime.Remoting Namespace

If we use configuration files, only one remoting call is necessary to read the configuration file and start the channel. The only class we used from the `System.Runtime.Remoting` namespace was `RemotingConfiguration`. Instead of using configuration, files the channel startup can be done programmatically. Before we do this let's get an overview of the classes that can be found in the remoting namespaces.

The `System.Runtime.Remoting` namespaces are contained in the assemblies `mscorlib` and `System.Runtime.Remoting`:

- In the namespace `System.Runtime.Remoting` some utility classes such as `RemotingConfiguration` and `RemotingServices` can be found. `RemotingConfiguration` is used to read configuration files and to get information about the registered channels; `RemotingServices` is used to publish remote objects.
- `System.Runtime.Remoting.Activation`: The channel uses the class `RemotingDispatcher` to dispatch method calls to the remote object. In this namespace, we can also find some interfaces for the activation of remote objects: `IActivator`, `IConstructionCallMessage`, and `IConstructionReturnMessage`.
- The namespace `System.Runtime.Remoting.Channels` has base classes for channels, and channel registration. An important class in this namespace is `ChannelServices`. This utility class can be used to register and to unregister a channel with the methods `RegisterChannel()` and `UnRegisterChannel()`, and it is also used to dispatch messages into a channel using `SyncDispatchMessage()` or `AsyncDispatchMessage()`.
- With the .NETF we get two channel types: TCP and HTTP. The TCP channels can be found in the namespace `System.Runtime.Remoting.Channels.Tcp`, where as the HTTP channels are in `System.Runtime.Remoting.Channels.Http`. In the next chapter we will look into the differences between these channels.
- The `System.Runtime.Remoting.Contexts` namespace not only has classes for the context management inside an application domain, but also defines `IContribute<XX>` interfaces to intercept remoting calls.

- The lifetime of remote stateful objects is defined by a leasing mechanism. The namespace `System.Runtime.Remoting.Lifetime` defines the classes `LifetimeServices` and `ClientSponsor`, and the interfaces `ISponsor` and `ILease`.
- The namespace `System.Runtime.Remoting.Messaging` defines some interfaces for messages, such as `IMessage`, `IMethodCallMessage`, and `IMethodReturnMessage`. Method calls are converted to messages that are passed between message sinks. Message sinks are interceptors that can change messages before they are passed into the channel.
- The `soapsuds` tool converts assembly metadata to WSDL. The classes that help with these actions can be found in the namespaces `System.Runtime.Remoting.Metadata` and `System.Runtime.Remoting.MetadataServices`. We will use the `soapsuds` tool in Chapter 9.
- The namespace `System.Runtime.Remoting.Proxies` contains classes that control and provide proxies.

The classes in the `System.Runtime.Remoting.Services` namespace provide services to the remoting framework. The class `EnterpriseServicesHelper` provides remoting functionality for COM+ services, `RemotingService` can be used to access ASP.NET properties like `Application`, `Session`, and `User` for web services running in Internet Information Services. `RemotingClientProxy` is a base class for a proxy that is generated from the `soapsuds` utility, and `TrackingServices` provides a way to register tracking handlers.

Let's look in more detail at how to use the .NET Remoting architecture.

Remote Objects

With remote objects we have to differentiate between **well-known** objects and **client-activated** objects.

> **Well-known objects are stateless; client-activated objects hold state.**

If we use client-activated objects that hold state we should bear in mind that every call across the network takes time. It is not a good idea to use properties to set some state of the remote object. Instead, it would be much faster to use method calls with more arguments and so use fewer calls that are sent across the network. With COM it was said that a COM object could similarly be used locally, or across the network. The reality showed that if a COM object was implemented with properties to be easy to use from the client, it was not efficient on the network, and if it was efficient on the network it was not easy to use from a scripting client. With .NET we already have the advantage that remote object classes must derive from `MarshalByRefObject`, and such objects should implemented in a way that is efficient when used across the network. Classes that don't derive from `MarshalByRefObject` can never be called across the network.

Well-Known Objects

For **well-known** objects, the client must know the endpoint of the object – that's why they are called well-known. The endpoint is used to identify the remote object. In contrast to that, with **client-activated** objects the type of the class is used to activate the remote object.

Instead of using a configuration file as we have done earlier to register a channel and define a well-known object, we can do this programmatically. To get the same result as with the configuration file we have to do some more coding steps:

- Create a server channel
- Register the channel in the .NET remoting runtime
- Register a well-known object

Removing the configuration file and the call to `RemotingConfiguration.Configure()`, the implementation of the server can look like the following sample. In the configuration file we specified the TCP Server channel in the `<channels>` section. Here, we create an instance of the class `TcpServerChannel` programmatically, and define `9000` as listening port in the constructor.

Next we register the channel in the .NET Remoting runtime with the static method `ChannelServices.RegisterChannel()`. `ChannelServices` is a utility class to help with channel registration and discovery.

Using the `RemotingConfiguration` class we register a well-known object on the server by calling `RegisterWellKnownServiceType()`. The helper class `WellKnownServiceTypeEntry` can be used to specify all the values that are required for well-known objects that are registered on the server. Calling the constructor of this class we specify the type of the remote object `MyRemoteObject`, the endpoint name `SimpleServer/MyRemoteObject`, and the mode that is one value of the enumeration `WellKnownObjectMode`: `SingleCall`:

```
// SimpleServer.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace Wrox.Samples
{
   class SimpleServer
   {
      static void Main(string[] args)
      {

         // Create and register the server channel

         TcpServerChannel channel = new TcpServerChannel(9000);
         ChannelServices.RegisterChannel(channel);

         // Register the remote object type and endpoint

         WellKnownServiceTypeEntry remObj = new
            WellKnownServiceTypeEntry(
               typeof(MyRemoteObject),
               "SimpleServer/MyRemoteObject",
               WellKnownObjectMode.SingleCall);
         RemotingConfiguration.RegisterWellKnownServiceType(remObj);
         Console.WriteLine("Press return to exit");
         Console.ReadLine();
      }
   }
}
```

With the compilation of the server we now have to reference the assembly of the remote object as the type is used in our code:

```
csc /target:exe /reference:MyRemoteObject.dll SimpleServer.cs
```

Nothing has really changed with the server and the remote object, so we can use the same client application we created earlier to communicate with the new server. To complete the picture, we will change the client code as can be seen below:

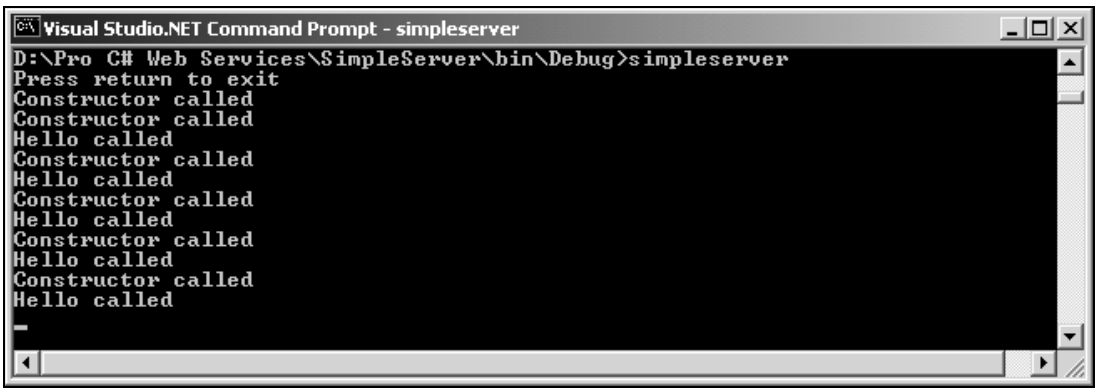
```
// SimpleClient.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace Wrox.Samples
{
   class SimpleClient
   {
      static void Main(string[] args)
      {

         // Create and register the client channel

         TcpClientChannel channel = new TcpClientChannel();
         ChannelServices.RegisterChannel(channel);

         // Register the name and port of the remote object

         WellKnownClientTypeEntry entry = new WellKnownClientTypeEntry(
            typeof(MyRemoteObject),
            "tcp://localhost:9000/SimpleServer/MyRemoteObject");
         RemotingConfiguration.RegisterWellKnownClientType(entry);
```

Now a client channel from the class `TcpClientChannel` is instantiated and registered. We use the `WellKnownClientTypeEntry` class to define the remote object. In contrast to the server version of this class, the complete path to the remote object must be specified here as we have to know the server name in the client, but the mode is not specified because this is defined from the server. The `RemotingConfiguration` class is then used to register this remote object in the remoting runtime.

As we saw earlier, the remote object can be instantiated with the `new` operator. To demonstrate that a well-known object gets activated with every method call, the method `Hello()` is now called five times in a for loop:

```
         MyRemoteObject obj = new MyRemoteObject();
         for (int i=0; i < 5; i++)
            Console.WriteLine(obj.Hello());
      }
   }
}
```

Running this program we can see in the output of the server (see screenshot opposite) that a new instance of the remote object is not created by calling the `new` operator in the client code, but with every method call instead. Well-known objects could also be called **server-activated** objects as compared to client-activated objects. With client-activated objects a new object gets instantiated on the server when the client invokes the `new` operator:



One important fact that we should remember with well-known objects:

> **A well-known object doesn't have state. A new instance is created with every method call.**

Activator.GetObject

Instead of using the `new` operator to instantiate remote objects, we can use the `Activator` class. Behind the scenes in the implementation of the `new` operator this class is used not only for remote but also for local object instantiations. For well-known objects the static method `GetObject()` returns a proxy to the remote object.

Using `Activator.GetObject()` instead of the `new` operator we no longer need to register the well known client type, because the URL to the remote object must be passed as argument to the `GetObject()` method:

```
// Create and register the client channel

TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel);
MyRemoteObject obj = (MyRemoteObject)Activator.GetObject(
   typeof(MyRemoteObject),
   "tcp://localhost:9000/SimpleServer/MyRemoteObject");
```

Whether we use the `new` operator or the `Activator.GetObject()` method is just a matter of choice. The `new` operator is easier to use and hides the fact that we deal with remote objects. `GetObject()` is nearer to the reality as the name of this method clearly shows that a new object does not get instantiated. We already know that with both versions when we use well-known objects a proxy is returned, and no connection to the server happens at this time.

Singletons

Well-known objects can be in `SingleCall` mode or `Singleton`. With the `SingleCall` mode an object is created with every method call; with a `Singleton` only **one** object is created in all. `Singleton` objects can be used to share information between multiple clients.

Using a configuration file the only change that must be done to the server is setting the `mode` attribute of the `<wellknown>` element to `Singleton`:

```
<wellknown
   mode="Singleton"
   type="Wrox.Samples.MyRemoteObject, MyRemoteObject"
   objectUri="MyRemoteObject" />
```

Without using a configuration file, the enumeration `WellKnownObjectMode.SingleCall` has to be replaced with `WellKnownObjectMode.Singleton`:

```
WellKnownServiceTypeEntry remObj = new WellKnownServiceTypeEntry(
   typeof(MyRemoteObject),
   "SimpleServer/MyRemoteObject",
   WellKnownObjectMode.Singleton);
RemotingConfiguration.RegisterWellKnownServiceType(remObj);
```

No changes are required for the client.

Multiple threads are used on the server to fulfill concurrent requests from clients. We have to develop this remote object in a thread-safe manner. For more about thread issues see the Wrox book Professional C#.

One more aspect of singletons that must be thought of is scalability. A `Singleton` object can't be spread across multiple servers. If scalability across multiple servers may be needed, don't use singletons.

Client-Activated Objects

Unlike well-known objects, client-activated objects can have state. A client-activated object is instantiated on the server when the client creates it, and not with every method call.

We can define properties in a client-activated object. The client can set values and get these values back, but we should be aware that every call across the network takes time. For performance reasons, it is better to pass more data with a single method call instead of doing multiple calls across the network.

Configuration Files

For client-activated objects the **server configuration file** must set the tag `<activated>` instead of `<wellknown>`. With the `<activated>` tag, only the `type` attribute with the class and assembly name must be defined. It is not necessary to define a URL for the remote object because it will be instantiated by its type. For well-known objects, a URL is required, but client-activated objects use the type for activation. The .NET Runtime automatically creates a unique URL to the remote object instance for client-activated objects. The URL that we defined for well-known objects is not unique for a single instance, but because a well-known instance is newly created with every method call a unique URL is not required:

```
<configuration>
   <system.runtime.remoting>
      <application name="SimpleServer">
         <service>
            <activated type="Wrox.Samples.MyRemoteObject,
                      MyRemoteObject" />
         </service>
         <channels>
            <channel ref="tcp server" port="9000" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

The client configuration file requires a similar change:

```
<configuration>
   <system.runtime.remoting>
      <application name="SimpleClient">
         <client url="tcp://localhost:9000/SimpleServer">
            <activated
                type="Wrox.Samples.MyRemoteObject, MyRemoteObject" />
         </client>
         <channels>
            <channel ref="tcp client" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Using the same code for the server and the client that we used before with the old configuration files, the constructor of the remote object is just called once for every calling of the `new` operator. Calling methods doesn't create new objects.

RemotingConfiguration

As we have seen before with well-known objects, we can register client-activated objects programmatically too. The server has to call `RemotingConfiguration.RegisterActivatedServiceType()`, and the client `RemotingConfiguration.RegisterActivatedClientType()` if we want to create and register the remote object programmatically. Similar to using the configuration file we have to set the type of the remote object using this method.

Non-default Constructor

With client-activated remote objects, it is possible to use a non-default constructor. This is not possible with well-known objects because a new object is created with every method call. We'll change the class `MyRemoteObject` in the file `MyRemoteObject.cs` to demonstrate non-default constructors and keeping state with client-activated remote objects:

```
public class MyRemoteObject : System.MarshalByRefObject
{
   public MyRemoteObject(int state)
   {
      Console.WriteLine("Constructor called");
      this.state = state;
```

```
        }
        private int state;
        public int State
        {
           get
           {
              return state;
           }
           set
           {
              state = value;
           }
        }
        public string Hello()
        {
           Console.WriteLine("Hello called");
           return "Hello, .NET Client!";
        }
     }
```

Now, in `SimpleClient.cs` we can invoke the constructor using the `new` operator as can be seen in this example:

```
        RemotingConfiguration.Configure("SimpleClient.exe.config");
        MyRemoteObject obj = new MyRemoteObject(333);
        int x = obj.State;
        Console.WriteLine(x);
```

Activator.CreateInstance

Instead of using the `new` operator, we can create client-activated objects with the `Activator` class. Well-known objects have to be created with the `GetObject()` method; client-activated objects require the method `CreateInstance()` because here the remote object really is instantiated on request of the client. With the method `CreateInstance()`it is also possible to invoke non-default constructors.

The following code example shows a client without a configuration file. The channel is created and registered as before. With `Activator.CreateInstance()` we instantiate a client-activated remote object. This method accepts activation attributes. One of these attributes must be the URL to the remote server. The class `System.Runtime.Remoting.Activation.UrlAttribute` can be used to define the URL of the remote object that is passed to the `CreateInstance()` method.

> *We can also use overloaded versions of `CreateInstance()` that don't accept activation arguments if we use a configuration file or the utility class `RemotingConfiguration` to define the URL to the remote object.*

The second argument of `CreateInstance` allows passing arguments to the constructor. To allow a flexible number of arguments, this parameter is of type `object[]` where we can pass any data type. In the `MyRemoteObject` class we now have a constructor that accepts a single `int` value, so we create an object array with one element, and assign an `int` value to that element of the array. The `attrs` array that is also an object array also has only one element. This is a `URLAttribute` where we pass the URL to the remote object to the constructor of this class. Both arrays are passed into the `CreateInstance()` method together with the type of the remote object to create the remote object in the right place using the appropriate constructor:

```
// SimpleClient.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Activation;

namespace Wrox.Samples
{
   class SimpleClient
   {
      static void Main(string[] args)
      {
         TcpClientChannel channel = new TcpClientChannel();
         ChannelServices.RegisterChannel(channel);
         object[] constr = new object[1];
         constr[0] = 333;
         object[] attrs = {
                  new UrlAttribute("tcp://localhost:9000/SimpleServer") };
         MyRemoteObject obj = (MyRemoteObject)Activator.CreateInstance(
                 typeof(MyRemoteObject), constr, attrs);
         int x = obj.State;
         Console.WriteLine(x);
      }
   }
}
```

Lease-Based Lifetime

A well-known single-call object can be garbage-collected after the method call of the client because it doesn't hold state. This is different for long-lived objects, which means both client-activated and well-known singleton objects. If the remote object is deactivated before the client stops using it, the client will get a `RemotingException` that the object has been disconnected with the next method call that is made from the client. The object has to be activated as long as the client needs it. What if the client crashes and if it cannot inform the server that the object is not needed any more? Microsoft's DCOM technology used a ping mechanism where the client regularly pings the server to inform it that the client is still alive and which objects it needs, but this is not scalable to internet solutions. Imagine thousands or millions of continuous ping requests going through the server. The network and the server would be loaded unnecessarily. .NET Remoting uses a **lease-based lifetime mechanism** instead. Comparing this mechanism to cars, if we lease a car we do not hear anything from the leasing company for months after the lease was instantiated. The object also remains activated until the lease time runs out with the .NET Remoting leasing facility.

> **If the exception `RemotingException`: Object <URI> has been disconnected or does not exist at the server occurs, this may be because of an expired lease time.**

There are some ways in which we can influence leasing times of objects. One way is to use configuration files, and another way is to do it programmatically for the server, the remote object, or in the client. Let us look at what can be configured first.

The remote object has a maximum time to lease which is defined with the `LeaseTime` option. If the client does not need the remote object for this time period, the object will be deactivated. Every time the client invokes a method with the remote object the leasing time **is incremented** by a value that is defined with `RenewOnCallTime`.

Sponsor

Let's compare.NET Remoting leasing again with the leasing of cars. If someone pays for the lease of the car, they are a sponsor. We have sponsors with .NET Remoting, too. If we don't want to rely on the client to invoke methods to extend the lifetime, we can use a **sponsor**. Using a sponsor is a good way to have long-running objects where we don't know how long the object will be needed for, or when the time intervals at which the client makes calls to the object are unforeseeable. The sponsor can **extend** the lease of the remote object. If the leasing time is expired, the sponsor is asked if it extends the lease. The default time until a call to a sponsor is timed out is defined with the `SponsorshipTimeout` option. If that time is reached, another sponsor can be asked. The option `LeaseManagerPollTime` defines the time the sponsor has to return a lease time extension.

The default values for the lease configuration options are listed in this table:

Lease Configuration	Default Value (seconds)
`leaseTime`	300
`renewOnCallTime`	120
`sponsorshipTimeout`	120
`leaseManagerPollTime`	10

LifetimeServices

There are several methods available to change the options for lifetime services. With the `System.Runtime.Remoting.Lifetime.LifetimeServices` class we can set default leasing time options that are valid *for all objects in the application domain* using static properties of this class.

Configuration Files

We can also write the lifetime configuration in the application **configuration file** of the server. This way this configuration is valid *for the complete application.* For the application configuration file we have the `<lifetime>` tag where `leaseTime`, `sponsorshipTimeout`, `renewOnCallTime`, and `pollTime` can be configured using attributes:

```
<configuration>
   <system.runtime.remoting>
      <application name="SimpleServer">
         <lifetime
            leaseTime = "15M"
            sponsorshipTimeOut = "4M"
            renewOnCallTime = "3M"
            pollTime = "30S" />
       </application>
   </system.runtime.remoting>
</configuration>
```

Overriding InitializeLifetimeService

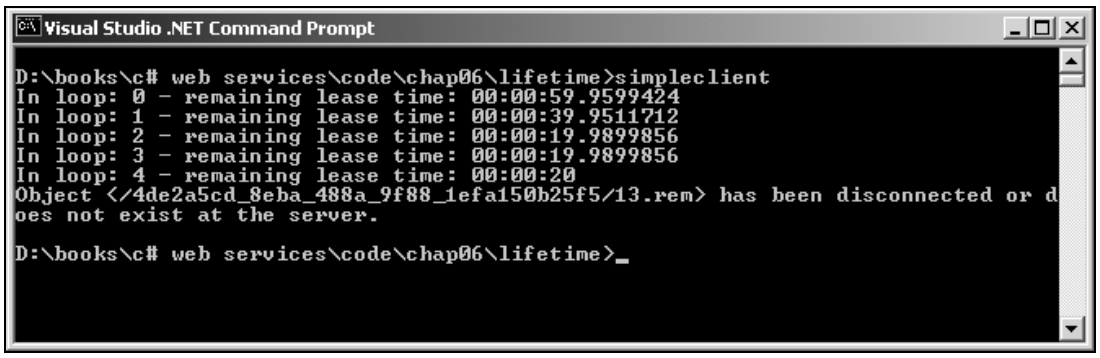
Setting the lifetime options in the configuration file is useful if we want to have the same lifetime management for all objects of the server. If we have remote objects with different lifetime requirements, it would be better to set the lifetime for the object programmatically. In the remote object class we can override the method `InitializeLifetimeService()` as we can see in the following example. The method `InitializeLifetimeService()` of the base class `MarshalByRefObject` returns a reference to the `ILease` interface that can be used to change the default values. Changing the values is only possible as long as the lease has not been activated, so that's why we check for the current state to compare it with the enumeration value `LeaseState.Initial`. We set the `InitialLeaseTime` to the very short value of one minute and the `RenewOnCallTime` to 20 seconds so that we soon see the results of the new behavior in the client application:

```
// MyRemoteObject.cs

using System;
using System.Runtime.Remoting.Lifetime;
namespace Wrox.Samples
{
   public class MyRemoteObject : System.MarshalByRefObject
   {
      public MyRemoteObject(int state)
      {
         Console.WriteLine("Constructor called");
         this.state = state;
      }
      public override object InitializeLifetimeService()
      {
         ILease lease = (ILease)base.InitializeLifetimeService();
         if (lease.CurrentState == LeaseState.Initial)
         {
            lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
            lease.RenewOnCallTime = TimeSpan.FromSeconds(20);
         }
         return lease;
      }
      private int state;
      public int State
      {
         get
         {
            return state;
         }
         set
         {
            state = value;
         }
      }
      public string Hello()
      {
         Console.WriteLine("Hello called");
         return "Hello, .NET Client!";
      }
   }
```

I'm changing the client code a little bit so that we can see the effects of the lease time. After the remote object is created, we do a loop five times to call the helper method `DisplayLease()` before we sleep for 20 seconds. The method `DisplayLease()` gets the `ILease` interface of the remote object by calling the method `GetLifetimeService()`. No exception is thrown if the object that is returned from `GetLifetimeService()` is not a `ILease` interface. This can be the case if the remote object is a well-known type that doesn't support the leasing mechanism. Calling the property `CurrentLeaseTime` we access the actual value of the lease to display it in the console.

```
// SimpleClient.cs

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Lifetime;
using System.Threading;
namespace Wrox.Samples
{
   class SimpleClient
   {
      static void Main(string[] args)
      {
         try
         {
            RemotingConfiguration.Configure("SimpleClient.exe.config");
            MyRemoteObject obj = new MyRemoteObject(33);
            for (int i=0; i < 5; i++)
            {
               DisplayLease("In loop: " + i, obj);
               Thread.Sleep(20000);
            }
            Thread.Sleep(30000);
            obj.Hello();
         }
         catch(RemotingException ex)
         {
            Console.WriteLine(ex.Message);
         }
      }
      public static void DisplayLease(string s, MarshalByRefObject o)
      {
         ILease lease = o.GetLifetimeService() as ILease;
         if (lease != null)
         {
            Console.WriteLine(s + " - remaining lease time: " +
                    lease.CurrentLeaseTime);
         }
      }
   }
}
```

Running the client application we now see the displays from the loop. The initial lifetime of the remote object is nearly 60 seconds because we have set the `InitialLeaseTime` property to one minute. The lifetime is reduced with the following loops. Starting with loop number two, the lifetime remains constant with 20 seconds. This is because we have set `RenewOnCallTime` property to 20 seconds. Every time we ask for the current value of the lifetime we do a new call on the remote object, so the lease time is set to 20 seconds. After we exit the loop we wait another 30 seconds, and this time the remote object is already garbage-collected because the lifetime has been exceeded, so we get a `RemotingException` the next time we call a method:



ClientSponsor

The third option to change the values for the lifetime services is by implementing a sponsor. A sponsor for an Indy cart team gives money to the team, so that the team can do its job. This is similar to remoting: a sponsor extends the lifetime of an object so that the object can do its job. Without a sponsor, the client would have to make method calls on the remote object to keep it alive.

The .NET Remoting runtime uses the `ISponsor` interface to extend the lifetime of remote objects using sponsoring. `ISponsor` defines the method `Renewal()`, which is called by the .NET remoting infrastructure to extend the lease time of the current object. This method has the following signature:

```
TimeSpan Renewal(ILease lease);
```

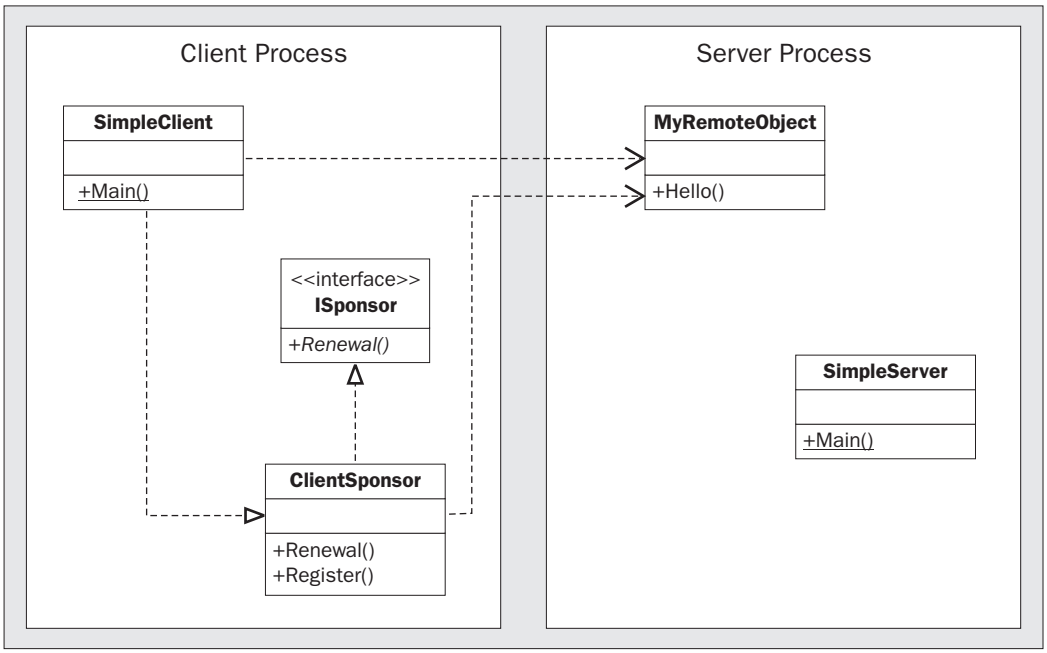
With the lease argument we can read the current configuration and the actual status of the lease time; with the return value we have to define the additional lease time for the object.

The class `ClientSponsor` provides a default implementation for a sponsor. `ClientSponsor` implements the interface `ISponsor`. `ClientSponsor` also derives from `MarshalByRefObject`, so it can be called across the network:

ClientSponsor Properties and Methods	Description
`RenewalTime`	This property defines the time that is used to extend the leasing time of the remote object.
`Register()`	With this method we can register a specified remote object with the sponsor, so that the sponsor answers requests for this object. The challenge with this method is that it must be called in the process of the remote object, because it adds the managed object to the sponsor table of this process.
`Unregister()`	Sponsorship is canceled. `Unregister()` removes the remote object from the sponsor table.
`Renewal()`	`Renewal()` is the method that is called by the runtime to extend the lifetime.
`Close()`	`Close()` clears the list of objects managed with this sponsor.

ClientSponsor Example

A useful place to put the sponsor is in the client process as shown in the next diagram. This has the advantage that the sponsor will not extend the lifetime of the remote object when the client is not reachable:



The client must create an instance of the `ClientSponsor` class, and call the `Register()` method where the remote object must be registered. As a result of this registration, the remoting runtime calls the `Renewal()` method that is defined with the `ISponsor` interface to renew the lifetime.

To demonstrate the calls to the sponsor in action I have created a `MySponsor` class that derives from `ClientSponsor` in the client application. The only thing that this class should do differently from the base class is answering `Renewal()` requests and writing Renew called to the console.

> *`Renewal()` is not declared virtual in the base class so we have to add an implements declaration of the `ISponsor` interface and implement this method explicitly. After writing "Renew called" to the console we just return the renewal time value that is defined by the `RenewalTime` property of the client sponsor.*

```
// SimpleClient.cs

// ...

public class MySponsor: ClientSponsor, ISponsor
{
   TimeSpan ISponsor.Renewal(ILease lease)
   {
      Console.WriteLine("Renewal called");
      return this.RenewalTime;
   }
}
```

The following code example shows the `Main()` method of the client that creates a sponsor and sets the `RenewalTime` of the sponsor:

```
// SimpleClient.cs

// ...

static void Main(string[] args)
{
   RemotingConfiguration.Configure("SimpleClient.exe.config");
   MyRemoteObject obj = new MyRemoteObject(333);
   MySponsor sponsor = new MySponsor();
   sponsor.RenewalTime = TimeSpan.FromMinutes(2);
   sponsor.Register(obj);
```

The sponsor is a remote object created in the client process. Here, the server acts as a client when doing the renewal, and the client acts as server when the sponsor is called. To support this, the channel for the client must support a client and a server channel, and the same is also true for the channel running on the server.

To see the sponsor in action we'll add a `for` loop to the `Main()` method in the file `SimpleClient.cs`. The time to sleep is longer than the configured `RenewalTime`, so we should see calls to the sponsor:

```
   for (int i=0; i < 20; i++)
   {
      DisplayLease("In loop: " + i, obj);
      obj.Hello();
      Thread.Sleep(25000);
   }
```

We have to change the configuration files and use the `TcpChannel` instead of the `TcpClientChannel` in the client configuration file `SimpleClient.exe.config`, because this channel has the functionality of both a client and server channel. If we start both the client and server application on the same system, we also have to configure a different port. Here I'm using port 9002, so we can run the application on a single system:

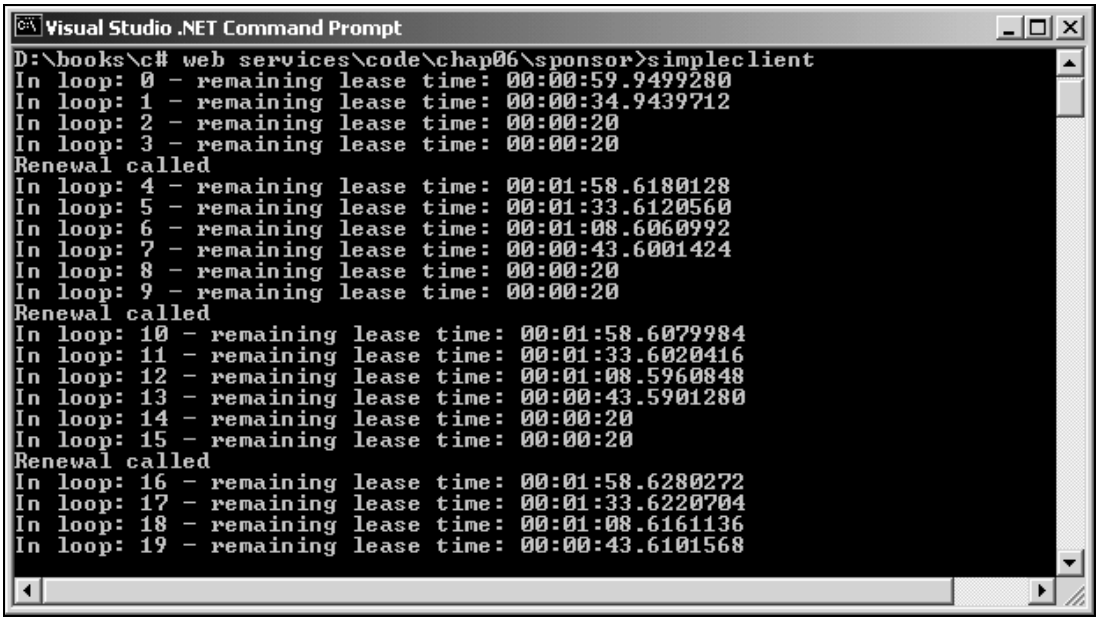
```
<channels>
   <channel ref="tcp" port="9002" />
</channels>
```

The server configuration file `SimpleServer.exe.config` needs a similar change:

```
<channels>
   <channel ref="tcp" port="9000" />
</channels>
```

Running the Example

The console output of the client application in the screenshot overleaf lists the lease times of the remote object. The first line shows a lease time value that is near to 60 seconds; this was the initial lease time. 25 seconds later the second iteration of the loop happened. The garbage collection doesn't happen immediately after the time 0 is reached, so the remote object was lucky that the lease time was extended twice to 20 seconds after a wait (loops 2 and 3), but the `Renewal()` method still wasn't called. Before loop 4, the `Renewal()` method was called for the first time, and here the lease was extended to the value that the sponsor supports, 2 minutes. The lease time again is decremented, before `Renewal()` is called again before loop 10:



Remote Object Types Summary

We have now seen the types of the remote objects and their purpose. Let's summarize the features of these objects in the following table.

Object Type	Time of Object Activation	State	Leasing
Well-known `SingleCall`	With every method call	Stateless	No
Well-known `Singleton`	Only once with the first method call	State shared for all clients	Yes
Client-Activated	When the client instantiates the remote object	State	Yes

Activation

So far we have seen different object types and how these remote objects can be activated using the `new` operator and the `Activator` class. Depending whether the object type is a well-known object or a client-activated object we can use `GetObject()` or `CreateInstance()`. Let us look further into the client side to see what happens on the client that we can also make use of.

RemotingServices.Connect

Instead of using the `new` operator or the `Activator` class it is also possible to use the static method `Connect()` of the class `RemotingServices` for well-known objects. This method doesn't really instantiate the connection as we maybe would assume from its name. A connection is not needed at this point, but all the information passed here will be used to build up the connection when the first method call is made.

In this code example below, we create a proxy by calling `RemotingServices.Connect()`. This method requires the type of the remote object as we have seen with the `Activator` class, and the URL string to the remote object:

```
static void Main(string[] args)
{
   RemotingConfiguration.Configure("SimpleClient.exe.config");
   MyRemoteObject obj =
      (MyRemoteObject)RemotingServices.Connect(
      typeof(MyRemoteObject),
      "tcp://localhost:9000/SimpleServer/MyRemoteObject");
```

Error Messages

If the connection string that is used with the `Connect()` method is wrong, we get an error the first time that a remote method is called.

The possible error messages we can get are listed here:

- SocketException: No such host is known.

  This error message occurs when the host name cannot be resolved.

- SocketException: No connection could be made because the target machine actively refused it.

  If the specified port number is incorrect or the server is not started, we will get this error message.

- RemotingException: Object <Name> has been disconnected or does not exist at the server.

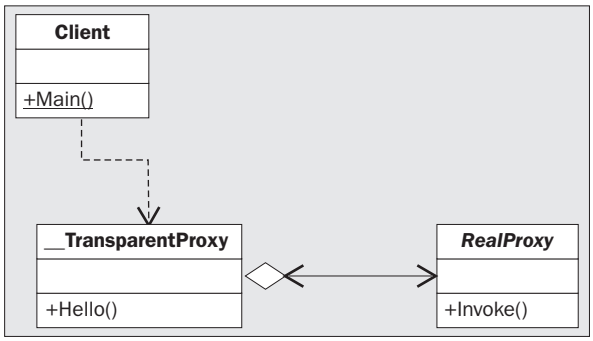
  This error message occurs if the name of the endpoint is incorrectly specified, or when the leasing time has expired when using a client-activated object.

Proxy

`RemotingServices.Connect()` defines the connection and **creates a proxy**. No matter which options are used to create remote objects, it is always a proxy that is created and returned by the creation methods. The proxy looks just like the real object as it has the same public methods; but the proxy just converts the method call to a message so that it can be sent across the network.

We actually create two objects in this case: the **transparent proxy** and the **real proxy**. The transparent proxy is the one used by the client. It looks like the remote object with the same public methods. This proxy is created dynamically by reading the metadata of the assembly from the remote object. The transparent proxy itself uses the real proxy to create messages from method calls. The transparent proxy is an instance of the class `System.Runtime.Remoting.Proxies.__ TransparentProxy`. This class is internal to the `mscorlib` assembly, so we cannot derive custom proxies from it, and we won't find it in the MSDN documentation.

In the picture below we can see that the client uses the `__TransparentProxy`. The transparent proxy has the method `Hello()` because this method is available from the remote object. The transparent proxy creates an object that implements the interface `IMessage` to pass it to the `Invoke()` method of the `RealProxy`. `IMessage` contains the method name and parameters that can be accessed through the property `IMessage.Properties`. The real proxy transforms this into an `IMethodCallMessage` and sends it to the remote object through the channel.



With the utility class `RemotingServices` we can check if the object really is a proxy to the remote object and not the remote object itself with the static method `IsTransparentProxy()`. The method `RemotingServices.GetRealProxy()` returns a reference to the `RealProxy` class:

```
MyRemoteObject obj = new MyRemoteObject();
if (RemotingServices.IsTransparentProxy(obj))
{
   Console.WriteLine("a transparent proxy!");
   RealProxy proxy = RemotingServices.GetRealProxy(obj);
}
```

An example where the method `IsTransparentProxy()` returns `false` is the case where the remote object is instantiated locally in the application domain. This would be the case if we do a `new` of the remote object class inside the server application. No proxy is needed for the server, of course, because it won't need to invoke the object on a different system.

Messages

The proxy deals with messages. All message objects implement the interface `IMessage`. `IMessage` contains a dictionary of the method name and parameters that can be accessed through the property `IMessage.Properties`.

In the namespace `System.Runtime.Remoting.Messaging` more message interfaces and classes are defined. The interfaces are specialized versions for passing and returning methods for example `IMethodMessage`, `IMethodCallMessage`, and `IMethodReturnMessage`. These interfaces have specialized properties that give faster access to method names and parameters. With these interfaces, it is not necessary to deal with the dictionary to access this data.

Marshaling

Marshaling is the term used when an object is converted so that it can be sent across the network (or across processes or application domains). Unmarshaling creates an object from the marshaled data.

With **marshal-by-value (MBV)** the object is serialized into the channel, and a copy of the object is created on the other side of the network. The object to marshal is stored into a stream, and the stream is used to build a copy of the object on the other side with the unmarshaling sequence. **Marshaling-by-reference (MBR)** creates a proxy on the client that is used to communicate with the remote object. The marshaling sequence of a remote object creates an `ObjRef` instance that itself can be serialized across the network.

Objects that are derived from `MarshalByRefObject` are always marshaled by reference – as the name of the base class says. To marshal a remote object the static method `RemotingServices.Marshal()` is used.

`RemotingServices.Marshal()` has these overloaded versions:

```
public static ObjRef Marshal(MarshalByRefObject obj);
public static ObjRef Marshal(MarshalByRefObject obj, string objUri);
public static ObjRef Marshal(MarshalByRefObject obj, string objUri,
       Type requestedType);
```

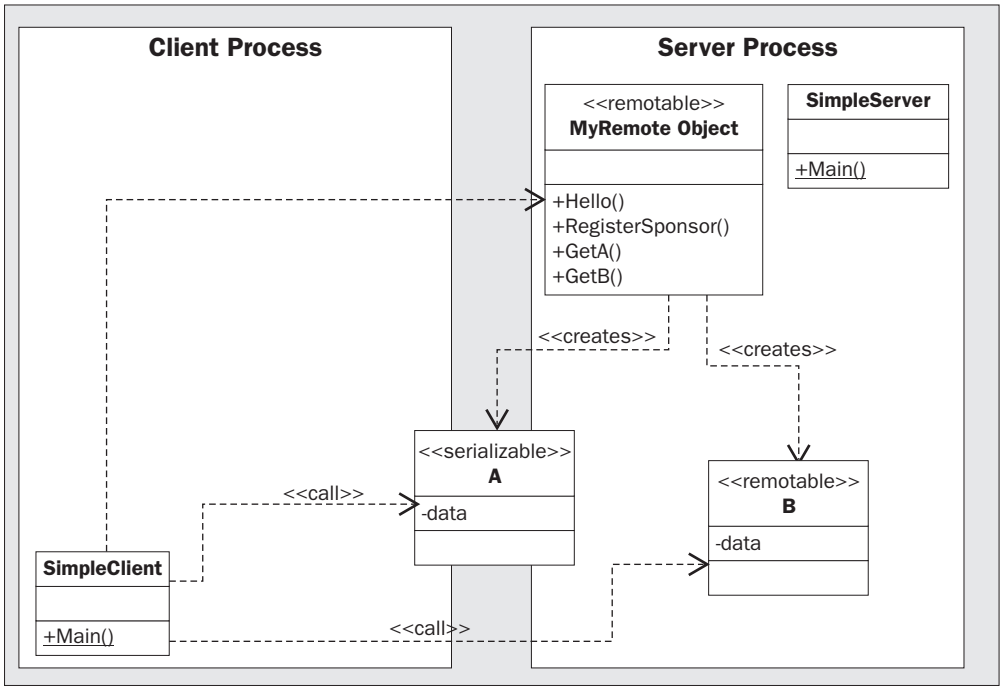
The first argument `obj` specifies the object to marshal. The `objUri` is the path that is stored within the marshaled object reference; it can be used to access the remote object. The `requestedType` can be used to pass a different type of the object to the object reference. This is useful if the client using the remote object shouldn't use the object class but an interface that the remote object class implements instead. In this scenario the interface is the `requestedType` that should be used for marshaling.

ObjRef

With all these `Marshal()` methods an `ObjRef` is returned. The `ObjRef` is serializable because it implements the interface `ISerializable`, and can be marshaled by value. The `ObjRef` knows about the location of the remote object: the host name, the port number, and the object name. `RemotingServices.Unmarshal()` uses the `ObjRef` to create a proxy that will be used by the client.

Passing Objects

With methods of remote objects we can pass basic data types and classes that we define. Whether an object is passed by value or by reference depends on the class declaration, as we will see next. We will add two methods to the class `MyRemoteObject` that return objects of class `A` and `B`. Class `A` will be serializable and passed using MBV to the client, `B` will be remotable and passed using MBR.



Let's start by adding two methods to the remote object class `MyRemoteObject` in the file `MyRemoteObject.cs` that returns objects of class `A` and `B` with the methods `GetA()` and `GetB()`. We will create the classes `A` and `B` in the next section:

```
// MyRemoteObject.cs

using System;

namespace Wrox.Samples
{
   public class MyRemoteObject : System.MarshalByRefObject
   {
      public MyRemoteObject()
      {
         Console.WriteLine("MyRemoteObject constructor called");
      }

      // serialized version

      public A GetA()
      {
         return new A(11);
      }

      // remote version

      public B GetB()
      {
         return new B(22);
      }
```

```
        public string Hello()
        {
            Console.WriteLine("Hello called");
            return "Hello, .NET Client!";
        }
    }
}
```

Marshal-By-Value

To create a class that will be serialized across the network, the class must be marked with the attribute `[Serializable]`. Objects of these classes don't have a remote identity, as they are marshaled to a stream before they're sent to the channel, and unmarshaled on the other side.

We add class `A` to the file `MyRemoteObject.cs`. This class has a field of type `int` that is also serializable. If a class is marked serializable, but has a reference to a class that is not serializable, the exception `SerializationException` will be thrown when marshaling this class. The class `A` defines the read-only property `Data` where the value of the field `data` is returned:

```
[Serializable]
public class A
{
   private int data;
   public A(int data)
   {
      Console.WriteLine("Constructor of serializable class A called");
      this.data = data;
   }
   public int Data
   {
      get
      {
         Console.WriteLine("A.Data called");
         return data;
      }
   }
}
```

Marshal-By-Reference

A class that should be called remotely is derived from `MarshalByRefObject`. We have used this class already for our remote object.

We add class `B` also to `MyRemoteObject.cs`. This class is similar to class `A` with the exception that it is not marked serializable; instead it derives from the class `MarshalByRefObject`. When we pass class `B` across the network, a proxy will be created.

Behind the scenes, `RemotingServices.Marshal()` is called to create an `ObjRef`, this object reference is sent across the channel, and with `RemotingServices.Unmarshal()` this object reference is used to create a proxy:

```
public class B : MarshalByRefObject
{
   private int data;
```

```
      public B(int data)
      {
         Console.WriteLine("Constructor of remotable class B called");
         this.data = data;
      }
      public int Data
      {
         get
         {
            Console.WriteLine("B.Data called");
            return data;
         }
      }
   }
```

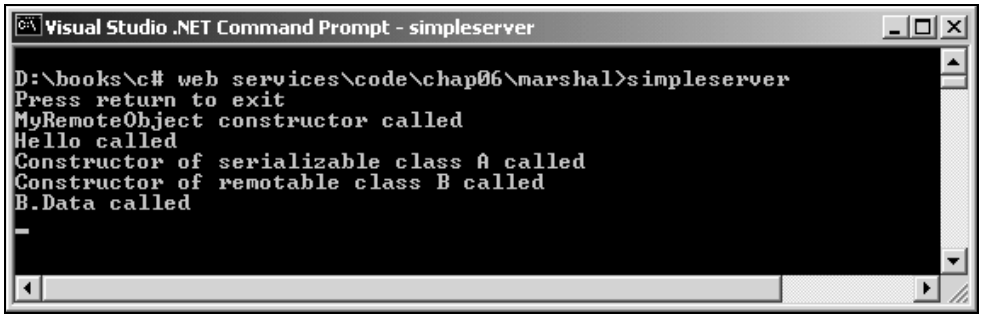
Client Example

The server is the same as we have created previously. It is entirely our choice whether we create the channel programmatically or with a configuration file, and whether we use a well-known or a client-activated object.

In the client we can now call the new methods of the remote object class. Let us change the `Main()` method in the file `SimpleClient.cs` to call the newly created remote methods:

```
      static void Main(string[] args)
      {
         RemotingConfiguration.Configure("SimpleClient.exe.config");
         MyRemoteObject obj = new MyRemoteObject();
         obj.Hello();
         A a = obj.GetA();
         int a1 = a.Data;
         B b = obj.GetB();
         int b1 = b.Data;
      }
```

The remote object class, the server, and the client can be compiled as we saw earlier. When we start the server and the client, we get the console output of the server. `B` is a remote class, so `B.Data` is called on the server as we can see here:



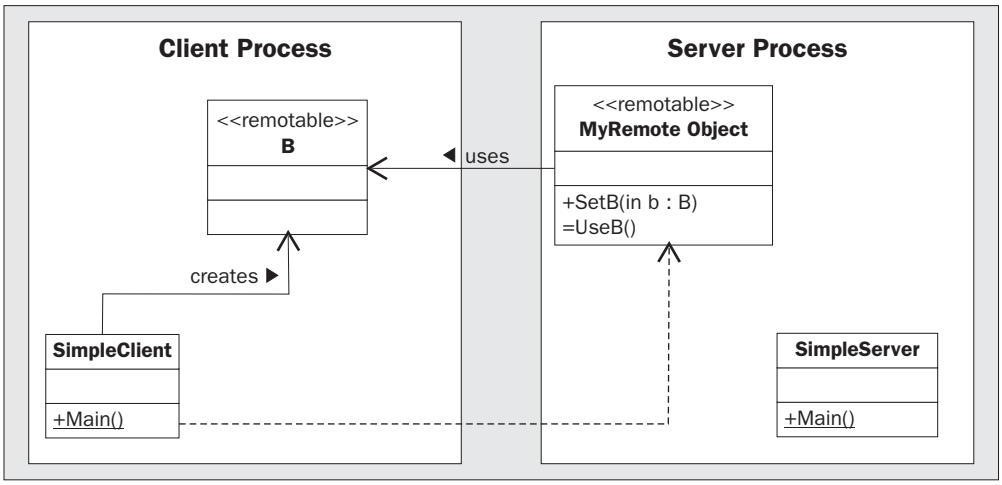
In the client console window we can see the `Console.WriteLine()` output of the `A` class because this class was serialized to the client.

Passing objects of classes that are neither marked `[Serializable]` nor derived from `MarshalByRefObject` will generate the exception System.Runtime.Serialization.SerializationException: The type typename is not marked as serializable.

Sending MBR Objects to the Server

Sending marshal-by-value objects from the client to the server is very similar to sending them in the other direction, but what about sending a remotable object to the server? How does the server come back to the client?

Instead of showing the derivation of `MarshalByRefObject` in this picture, I've introduced a stereotype `<<remotable>>` to make it easier to read. We can see `<<remotable>>` as indicatng inheritance from `MarshalByRefObject`. The MBR class `B` will be instantiated from the client, and passed with the method `SetB()` to the remote object `MyRemoteObject`. The remote object will use the object that was instantiated on the client across the network:



Let's add two methods to the remote object class in the file `MyRemoteObject.cs`. With the method `SetB()` the client can pass a remotable object of type `B` to the server. The method `SetB()` accepts an object of class `B`. In the last section we created class `B` so that it derives from `MarshalByRefObject`, so the object will not be copied but a proxy will be created instead. The method `SetB()` stores the reference to the proxy of `B` in the field `b`. The method `UseB()` uses the field `b` to call the property `Data` and passes the value returned to the client.

```
public class MyRemoteObject : System.MarshalByRefObject
{
   public MyRemoteObject()
   {
      Console.WriteLine("MyRemoteObject Constructor called");
   }
   private B b;
   public void SetB(B b)
   {
      Console.WriteLine("SetB called");
      this.b = b;
   }
   public int UseB()
   {
      Console.WriteLine("UseB called");
      return b.Data;
   }

// ...
```

One important thing to note here is that the class `MyRemoteObject` now has state. Using this class as a well-known object would result in `b` being set to `null` with every method call, so `UseB()` wouldn't work. This remote object class can only be used in client-activated object configurations.

The client can be changed to call the remote object. In the client code, we create a new instance of the class `B` and initialize it with the value `44`. While `obj` is instantiated remotely with the `new` operator, `b` is created locally because the class `B` is not listed in the client configuration file, but the class `MyRemoteObject` is. Accessing the `Data` property merely performs a local call:

```
// SimpleClient.cs

class SimpleClient
{
   static void Main(string[] args)
   {
      RemotingConfiguration.Configure("SimpleClient.exe.config");
      MyRemoteObject obj = new MyRemoteObject(333);
      obj.Hello();
      B b = new B(44);

      // This is a local call!

      int b1 = b.Data;
```

When calling `SetB()` with `MyRemoteObject`, we pass a reference to b to the server. Because `B` is derived from `MarshalByRefObject` a reference can be passed. Next we call the `UseB()` method where this reference is used within the server. The remote object will try to access the object that is running in the client process:

```
      // Pass a reference to the object running in the client
      // to the server
```

```
        obj.SetB(b);

        // Use the object in the client from the server

        b1 = obj.UseB();
```

To run this application we have to use a bi-directional channel. Because the client application now has to act both as a client to call methods in the remote object, and as a server where the instance b is invoked from the server, we must use the `TcpChannel` that can do both. If we use the wrong channel, the result will be a `RemotingException` with the following error message:

> The remoting proxy has no channel sinks which means either the server has no registered server channels that are listening, or this application has no suitable client channel to talk to the server.

Be sure to specify a reference to the channel `tcp` in both the client and server configuration files as we did earlier instead of `tcp client` and `tcp server`.

Running the server and client application, we get this output in the client:

The first time we see the output of the `B.Data` is when we access this property locally. The second output we see is when a method of the object running in the client is invoked from the server by calling `obj.UseB()` in the client application.

Tracking Services

A great feature for analyzing a running application is Tracking Services. This service can be used to observe the marshaling process of MBR objects. Indeed it can be very simply activated: we have only to create a tracking handler and register this handler with the utility class `TrackingServices`.

Tracking Handler

A tracking handler implements the interface `ITrackingHandler`. This interface defines three methods that are called by the remoting infrastructure when marshaling and unmarshaling occurs:

ITrackingHandler methods	Description
`MarshaledObject`	This method is called when an MBR object is marshaled
`UnmarshaledObject`	This method is called when an MBR object is unmarshaled
`DisconnectedObject`	`DisconnectedObject()` is called when an MBR object is disconnected, for example when the lease time of an client-activated object expires

The class `MyTrackingHandler` is implemented in the assembly `MyTrackingHandler` that is available both for the client and the server to make this tracking available both for clients and servers. This class implements the interface `ITrackingHandler`, so we have to implement the methods `MarshaledObject()`, `UnmarshaledObject()`, and `DisconnectedObject()`. We use the URI property of the `ObjRef` that is passed to output the path to the object; in addition, we output the type of the marshaled object. With the first argument, it would also be possible to access the properties of the object to marshal:

```
// MyTrackingHandler.cs
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Services;

namespace Wrox.Samples
{
   public class MyTrackingHandler: ITrackingHandler
   {
      public void MarshaledObject(object obj, ObjRef or)
      {
         Console.WriteLine();
         Console.WriteLine("The object " + or.URI + " is marshaled");
         Console.WriteLine("Type: " + or.TypeInfo.TypeName);
      }
      public void UnmarshaledObject(object obj, ObjRef or)
      {
         Console.WriteLine();
         Console.WriteLine("The object " + or.URI + " is unmarshaled");
         Console.WriteLine("Type: " + or.TypeInfo.TypeName);
      }
      public void DisconnectedObject(object obj)
      {
         Console.WriteLine(obj.ToString() + " is disconnected");
      }
   }
}
```

Register Tracking Handler

Both in the server and in the client application, in the files `SimpleClient.cs` and `SimpleServer.cs`, we can create a new instance of the `MyTrackingHandler` class and register it with the tracking service:

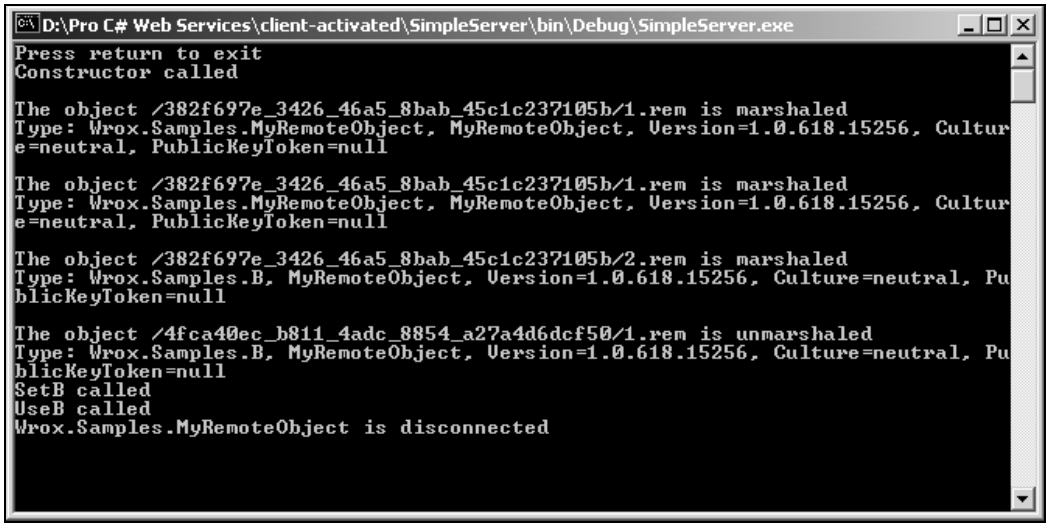
```
static void Main(string[] args)
{
   RemotingConfiguration.Configure("SimpleServer.exe.config");
   TrackingServices.RegisterTrackingHandler(new MyTrackingHandler());
```

Running the Program

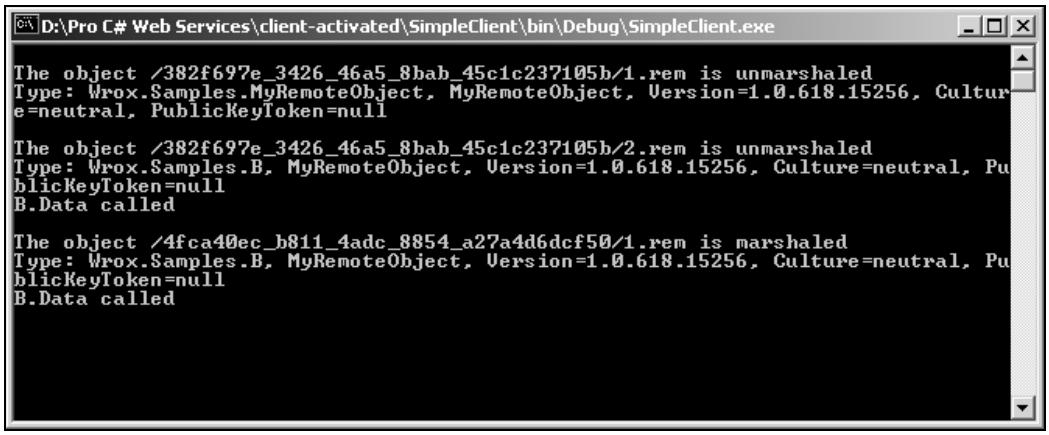
We run the application with a client that will:

1. Create the remote object `MyRemoteObject`
2. Call the method `obj.GetB()` to return the MBR object of class `B` to the client
3. Call the method `obj.SetB()` to pass the MBR object of class `B` to the server

We can see the output in the following screenshots. The first picture shows the server output where `MyRemoteObject` and `B` are marshaled, and with the call to `SetB()`, `B` is unmarshaled. The last line of the output shows that the lease time of the remote object expired, so it was disconnected:



In the client output `MyRemoteObject` and `B` are unmarshaled before the call to `SetB()`, where `B` is marshaled:



Asynchronous Remoting

Calling remote methods across the network can take some time. We can call the methods asynchronously, which means that we start the methods, and during the time the method call is running on the server we can do some other tasks on the client. .NET Remoting can be used asynchronously in the same way as local methods.

Calling Local Methods Asynchronously

One way to do this is to create a thread that makes the remote call while another thread answers users' requests or does something else. It is a lot easier, however, to do it with the built-in support provided in the .NET Framework in the form of **delegates**. With delegates, threads are created automatically, and we don't have to do it ourselves.

First, let us look at an asynchronous example without calling remote methods.

In the sample class `AsyncLocal` we can see the `LongCall()` method, which takes some time to process. We want to call this method asynchronously:

```
using System;
using System.Threading;
namespace Wrox.Samples
{
   class AsyncLocal
   {
      public void LongCall()
      {
         Console.WriteLine("LongCall started");
         Thread.Sleep(5000);
         Console.WriteLine("LongCall finished");
      }
```

To make this method asynchronous we have to declare a **delegate**. The delegate must have the same signature and return type as the method that should be called asynchronously. The method `LongCall()` accepts no arguments and has a `void` return type, and the same is true for the defined delegate `LongCallDelegate`. With the keyword `delegate` the C# compiler creates a class that is derived either from `Delegate` or from `MulticastDelegate`, depending whether there is a return type or not. We can see this generated class by opening the generated assembly with the `ildasm` utility:

```
      private delegate void LongCallDelegate();
      public AsyncLocal()
      {
      }
```

In the `Main()` method we can now create a new instance of the class `LongCallDelegate`, and pass `obj.LongCall`. The constructor of the delegate class requires the target as an argument. We can look at the delegate as a type-safe, object-oriented function pointer. The object itself is passed with the method name of the function to the constructor:

```
      static void Main(string[] args)
      {
         AsyncLocal obj = new AsyncLocal();
         LongCallDelegate d = new LongCallDelegate(obj.LongCall);
```

Now we can start the method by calling `BeginInvoke()` on the delegate. Visual Studio .NET does not display this method with the IntelliSense feature because it is not available at the time of programming this method call; instead, it will be created as soon as we compile the project. The compiler creates three methods for the delegate class: `Invoke()`, `BeginInvoke()`, and `EndInvoke()`. `Invoke()` can be used to call the method synchronously, whereas `BeginInvoke()` and `EndInvoke()` are used to call the method asynchronously.

`BeginInvoke()` has two arguments in addition to the arguments that are declared with the delegate. With the first argument, an `AsyncCallback` delegate can be passed to this method. With the asynchronous callback, we can define a method with an `IAsyncResult` argument that will be called when the method is finished. In this sample, we pass `null` with both arguments, and put the returned reference to `IAsyncResult` into the variable `ar`:

```
            IAsyncResult ar = d.BeginInvoke(null, null);
```

The method call is started asynchronously. We can now doing something else in the main thread at the same time:

```
            Thread.Sleep(1000);
            Console.WriteLine("Main running concurrently");
```

As soon as we want to be sure that the asynchronous method completed (maybe we need some result from this method), the reference to the `IAsyncResult` interface can be used. With the `IAsyncResult` interface, we can check if the method is finished by checking the `IsCompleted` property. We can wait until the method completes by using the `AsyncWaitHandle` property and call `WaitOne()`. `WaitOne()` is a blocking call that waits until the method that belongs to the `IAsyncResult` is finished:

```
            ar.AsyncWaitHandle.WaitOne();
            if (ar.IsCompleted)
               d.EndInvoke(ar);
            Console.WriteLine("Main finished");
         }
      }
   }
```

Calling Remote Methods Asynchronously

The same programming model used for local calls can also be used for remote calls. We will extend the remoting example by using some arguments and return values with a remote method.

First, we extend the remote object class `MyRemoteObject` with the long-running method `LongTimeAdd()`. Nothing else changes with the server:

```
// MyRemoteObject.cs

// ...

   public int LongTimeAdd(int val1, int val2, int ms)
   {
      Thread.Sleep(ms);
      return val1 + val2;
   }
```

In the client program, we have to declare a delegate with the same parameters as the `LongTimeAdd()` method:

```
// SimpleClient.cs

// ...
```

```
class SimpleClient
{
   private delegate int LongTimeAddDelegate(int val1, int val2,
           int ms);
```

In the `Main()` method after creating the remote object we create a instance of the delegate and pass the method `LongTimeAdd` to the constructor:

```
static void Main(string[] args)
{
   RemotingConfiguration.Configure("SimpleClient.exe.config");
   MyRemoteObject obj = new MyRemoteObject();
   LongTimeAddDelegate d = new LongTimeAddDelegate(obj.LongTimeAdd);
```

The remote method can now be started asynchronously with the `BeginInvoke()` method of the delegate class. Instead of two arguments `BeginInvoke()` now has five arguments – the first three arguments are the ones that are passed to the remote method. Here we pass the values `3` and `4` to add with a sleep time of `100` ms. The method `BeginInvoke()` will return immediately; we can do some other stuff in the main thread:

```
   IAsyncResult ar = d.BeginInvoke(3, 4, 100, null, null);
   Console.WriteLine("LongTimeAdd started");
```

To get the result that is returned with the method `LongTimeAdd()` we have to wait until the method is completed. This is done using `ar.AsyncWaitHandle.WaitOne()`. Calling the delegate method `EndInvoke()` we get the result that is returned from `LongTimeAdd()`:

```
   ar.AsyncWaitHandle.WaitOne();
   if (ar.IsCompleted)
   {
      Console.WriteLine("LongTimeAdd finished");
      int result = d.EndInvoke(ar);
      Console.WriteLine("result: " + result);
   }
```

Callbacks with Delegates

.NET Remoting supports two types of callbacks: with remote objects that are passed to the server, and with delegates. Using a delegate we can pass a `AsyncCallback` delegate when calling the method `BeginInvoke()` to have a callback when the asynchronous remote method completes. We have to define a method that has the same signature and return type that the delegate `AsyncCallback` defines. The delegate `AsyncCallback` is defined with this signature in the assembly `mscorlib`:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

We implement the method `LongTimeAddCallback` that has the same signature and return type. The implementation of this method is similar to how we handled the asynchronous method in the last code sample. It doesn't matter if the method is implemented as a class method or an instance method. We only need access to the delegate that is used to start the remote method. To make the delegate available to the static method we'll now declare `d` as static member variable:

```
private delegate int LongTimeAddDelegate(int val1, int val2, int ms);
private static LongTimeAddDelegate d;
public static void LongTimeAddCallback(IAsyncResult ar)
{
   if (ar.IsCompleted)
   {
      Console.WriteLine("LongTimeAdd finished");
      int result = d.EndInvoke(ar);
      Console.WriteLine("result: " + result);
   }
}
```

In the `Main()` method, we have to create a new instance of the `AsyncCallback` delegate and pass a reference to the function that should be called asynchronously. With the `BeginInvoke()` method we pass the instance of the `AsyncCallback` so that the method `LongTimeAddCallback()` will be invoked when the remote method `LongTimeAdd()` completes:

```
static void Main(string[] args)
{
   RemotingConfiguration.Configure("SimpleClient.exe.config");
   MyRemoteObject obj = new MyRemoteObject();
   d = new LongTimeAddDelegate(obj.LongTimeAdd);
   AsyncCallback cb =
                 new AsyncCallback(SimpleClient.LongTimeAddCallback);
   IAsyncResult ar = d.BeginInvoke(3, 4, 3000, cb, null);
   Console.WriteLine("LongTimeAdd started");
```

One Way

If a remote method doesn't return a result to the client we can call the method in a **fire-and-forget** style: we call the method asynchronously, but there is no requirement to wait for the method to complete. With .NET Remoting such a method is called **OneWay**: data is sent to the server, but none is returned.

A one way method must be declared with the `OneWay` attribute in the remote object class. The attribute class `OneWayAttribute` is part of the namespace `System.Runtime.Remoting.Messaging`. We add the `TakeAWhile()` method to the `RemoteObjectClass`:

```
[OneWay]
public void TakeAWhile(int ms)
{
   Console.WriteLine("TakeAWhile started");
   Thread.Sleep(ms);
   Console.WriteLine("TakeAWhile completed");
}
```

In the client the one way method `TakeAWhile()` can be called like a synchronous method, but it will be completed asynchronously. The method call will return immediately:

```
MyRemoteObject obj = new MyRemoteObject();
obj.TakeAWhile(5000);
```

If a synchronous method of a remote object throws an exception this exception will be propagated to the client. This is different with `OneWay` methods:

> **If a `OneWay` method throws an exception this exception will not be propagated to the client.**

Call Contexts

Client-activated objects can hold state. For this kind of object we must hold state on the server. Well-known objects are stateless. Every time we call a method a new object gets instantiated on the server. We have to pass all the data that we need on the server with every method call, but it is not necessary to pass the data with method arguments. If, for example, a user `ID` is needed with every method call to the remote object, the **call context** can be used.

The call context flows with the logical thread and is passed with every remote method call. A logical thread is started from the calling thread and flows through all method calls that are started from the calling thread, and passes through different contexts, different application domains, and different processes.

The `CallContext` class is a utility class in the namespace `System.Runtime.Remoting.Messaging`. We can add data to the context with `CallContext.SetData()`, and reading the data can be done with `CallContext.GetData()`.

> **We cannot pass basic data types with the context, but only objects of classes that implement the interface `ILogicalThreadAffinative`.**

A class that can be passed with the call context must implement the interface `ILogicalThreadAffinative`. This interface doesn't define a single method; it is just a marker to the .NET Remoting runtime. A logical thread can spawn multiple physical threads as such a call can cross process boundaries, but it is bound to a flow of method calls. Implementing the interface `ILogicalThread Affinative` in a class means that objects of this class can flow with the logical thread. Additionally, the class is marked with the `Serializable` attribute, so that the object can be marshaled into the channel. The class `UserName` defines a `LastName` and `FirstName` property that will flow with every method call:

```
[Serializable]
public class UserName : ILogicalThreadAffinative
{
   public UserName()
   {
   }
   private string lastName;
   public string LastName
   {
      get
      {
         return lastName;
      }
      set
      {
         lastName = value;
      }
   }
   private string firstName;
   public string FirstName
   {
      get
      {
         return firstName;
      }
```

```
         set
         {
            firstName = value;
         }
      }
   }
```

In the client program we can create a `UserName` object that will be passed with the remote method `Hello()`. `CallContext.SetData()` assigns the `UserName` object to the call context:

```
         RemotingConfiguration.Configure("SimpleClient.exe.config");
         MyRemoteObject obj = new MyRemoteObject();
         UserName user = new UserName();
         user.FirstName = "Christian";
         user.LastName = "Nagel";
         CallContext.SetData("User", user);
         obj.Hello();
```

In the remote object we can read the call context by calling `CallContext.GetData()`. For the context name "`User`" a `UserName` object was passed, so we can cast the return value of `GetData()` to `UserName`:

```
      public string Hello()
      {
         Console.WriteLine("Hello called");
         UserName user = (UserName)CallContext.GetData("User");
         if (user != null)
         {
            Console.WriteLine("context passed from {0} {1}",
                    user.FirstName, user.LastName);
         }
         return "Hello, Client!";
      }
```

Summary

In this chapter we have covered the .NET Remoting architecture. We looked into the parts of a client / server application using:

- The remote object that derives from `MarshalByRefObject`
- A channel that is used for the communication between client and server (we used the TCP channels)
- The proxy that is used by the client to create messages that are marshaled into the channel

.NET Remoting applications can make use of application configuration files where we can specify channels and remote objects, or all of this can be done programmatically.

.NET Remoting supports both stateful and stateless solutions with client-activated and well-known objects. Stateful objects have a leasing mechanism that is scalable to solutions with thousand of clients. We saw how to pass call contexts to the server.

In the next chapter we will look in more detail at the different channels and formatters that come with the framework, and consider some application scenarios that will illustrate how this technology can be used.