



The .NET Remoting Layer

Developers who are new to the .NET platform often assume that .NET is all about building Internet-centric applications (given that the term “.NET” often conjures the notion of “interNET” software). As you have already seen, however, this is simply not the case. In fact, the construction of web-centric programs is simply one tiny (but quite well-touted) aspect of the .NET platform. In this same vein of misinformation, many new .NET developers tend to assume that XML web services are the only way to interact with remote objects. Again, this is not true. Using the .NET remoting layer, you are able to build peer-to-peer distributed applications that have nothing to do with HTTP or XML (if you so choose).

The first goal of this chapter is to examine the low-level grunge used by the CLR to move information between application boundaries. Along the way, you will come to understand the numerous terms used when discussing .NET remoting, such as proxies, channels, marshaling by reference (as opposed to by value), server-activated (versus client-activated) objects, and so forth. After these background elements are covered, the remainder of the chapter offers numerous code examples that illustrate the process of building distributed systems using the .NET platform.

Defining .NET Remoting

As you recall from your reading in Chapter 13, an *application domain* (*AppDomain*) is a logical boundary for a .NET assembly, which is itself housed within a Win32 process. Understanding this concept is critical when discussing distributed computing under .NET, given that *remoting* is nothing more than the act of two objects communicating across application domains. The two application domains in question could be physically configured in any of the following manners:

- Two application domains in the same process (and thus on the same machine)
- Two application domains in separate processes on the same machine
- Two application domains in separate processes on different machines

Given these three possibilities, you can see that remoting does not necessarily need to involve two networked computers. In fact, each of the examples presented in this chapter can be successfully run on a single, stand-alone machine. Regardless of the distance between two objects, it is common to refer to each agent using the terms “client” and “server.” Simply put, the *client* is the entity that attempts to interact with remote objects. The *server* is the software agent that houses the remote objects.

The .NET Remoting Namespaces

Before we dive too deep into the details of the .NET remoting layer, we need to check out the functionality provided by the remoting-centric namespaces. The .NET base class libraries provide numerous namespaces that allow you to build distributed applications. The bulk of the types found within these namespaces are contained within `mscorlib.dll`, but the `System.Runtime.Remoting.dll` assembly does complement and extend the core namespaces. Table 18-1 briefly describes the role of the remoting-centric namespaces as of .NET 2.0.

Table 18-1. *.NET Remoting-centric Namespaces*

Namespace	Meaning in Life
<code>System.Runtime.Remoting</code>	This is the core namespace you must use when building any sort of distributed .NET application.
<code>System.Runtime.Remoting.Activation</code>	This relatively small namespace defines a handful of types that allow you to fine-tune the process of activating a remote object.
<code>System.Runtime.Remoting.Channels</code>	This namespace contains types that represent channels and channel sinks.
<code>System.Runtime.Remoting.Channels.Http</code>	This namespace contains types that use the HTTP protocol to transport messages and objects to and from remote locations.
<code>System.Runtime.Remoting.Channels.Ipc</code>	This namespace (which is new to .NET 2.0) contains types that leverage the Win32 interprocess communication (IPC) architecture. As you may know, IPC proves fast communications between AppDomains on the <i>same</i> physical machine.
<code>System.Runtime.Remoting.Channels.Tcp</code>	This namespace contains types that use the TCP protocol to transport messages and objects to and from remote locations.
<code>System.Runtime.Remoting.Contexts</code>	This namespace allows you to configure the details of an object's context.
<code>System.Runtime.Remoting.Lifetime</code>	This namespace contains types that manage the lifetime of remote objects.
<code>System.Runtime.Remoting.Messaging</code>	This namespace contains types used to create and transmit message objects.
<code>System.Runtime.Remoting.Metadata</code>	This namespace contains types that can be used to customize the generation and processing of SOAP formatting.
<code>System.Runtime.Remoting.Metadata.W3cXsd2001</code>	Closely related to the previous namespace, this namespace contains types that represent the XML Schema Definition (XSD) defined by the World Wide Web Consortium (W3C) in 2001.
<code>System.Runtime.Remoting.MetadataServices</code>	This namespace contains the types used by the <code>soapsuds.exe</code> command-line tool to convert .NET metadata to and from an XML schema for the remoting infrastructure.

Namespace	Meaning in Life
System.Runtime.Remoting.Proxies	This namespace contains types that provide functionality for proxy objects.
System.Runtime.Remoting.Services	This namespace defines a number of common base classes (and interfaces) that are typically only leveraged by other intrinsic remoting agents.

Understanding the .NET Remoting Framework

When clients and servers exchange information across application boundaries, the CLR makes use of several low-level primitives to ensure the entities in question are able to communicate with each other as transparently as possible. This means that as a .NET programmer, you are *not* required to provide reams and reams of grungy networking code to invoke a method on a remote object. Likewise, the server process is *not* required to manually pluck a network packet out of the queue and reformat the message into terms the remote object can understand. As you would hope, the CLR takes care of such details automatically using a default set of remoting primitives (although you are certainly able to get involved with the process if you so choose).

In a nutshell, the .NET remoting layer revolves around a careful orchestration that takes place between four key players:

- Proxies
- Messages
- Channels
- Formatters

Let's check out each entity in turn and see how their combined functionality facilitates remote method invocations.

Understanding Proxies and Messages

Clients and server objects do not communicate via a direct connection, but rather through the use of an intermediary termed a *proxy*. The role of a .NET proxy is to fool the client into believing it is communicating with the requested remote object in the *same application domain*. To facilitate this illusion, a proxy has the identical interface (i.e., members, properties, fields, and whatnot) as the remote type it represents. As far as the client is concerned, a given proxy *is* the remote object. Under the hood, however, the proxy is forwarding calls to the remote object.

Formally speaking, the proxy invoked directly by the client is termed the *transparent proxy*. This CLR autogenerated entity is in charge of ensuring that the client has provided the correct number of (and type of) parameters to invoke the remote method. Given this, you can regard the transparent proxy as a fixed interception layer that *cannot* be modified or extended programmatically.

Assuming the transparent proxy is able to verify the incoming arguments, this information is packaged up into another CLR-generated type termed the *message object*. By definition, all message objects implement the `System.Runtime.Remoting.Messaging.IMessage` interface:

```
public interface IMessage
{
    IDictionary Properties { get; }
}
```

As you can see, the `IMessage` interface defines a single property (named `Properties`) that provides access to a collection used to hold the client-supplied arguments. Once this message object has been populated by the CLR, it is then passed into a closely related type termed the *real proxy*.

The real proxy is the entity that actually passes the message object into the channel (described momentarily). Unlike the transparent proxy, the real proxy *can* be extended by the programmer and is represented by a base class type named (of course) `RealProxy`. Again, it is worth pointing out that the CLR will always generate a default implementation of the client-side real proxy, which will serve your needs most (if not all) of the time. Nevertheless, to gain some insight into the functionality provided by the abstract `RealProxy` base class, ponder the formal definition type:

```
public abstract class RealProxy : object
{
    public virtual ObjRef CreateObjRef(Type requestedType);
    public virtual bool Equals(object obj);
    public virtual IntPtr GetCOMIUnknown(bool fIsMarshaled);
    public virtual int GetHashCode();
    public virtual void GetObjectData(SerializationInfo info,
        StreamingContext context);
    public Type GetProxiedType();
    public static object GetStubData(RealProxy rp);
    public virtual object GetTransparentProxy();
    public Type GetType();
    public IConstructionReturnMessage InitializeServerObject(
        IConstructionCallMessage ctorMsg);
    public virtual IMessage Invoke(IMessage msg);
    public virtual void SetCOMIUnknown(IntPtr i);
    public static void SetStubData(RealProxy rp, object stubData);
    public virtual IntPtr SupportsInterface(ref Guid iid);
    public virtual string ToString();
}
```

Unless you are interested in building a custom implementation of the client-side real proxy, the only member of interest is `RealProxy.Invoke()`. Under the hood, the CLR-generated transparent proxy passes the formatted message object into the `RealProxy` type via its `Invoke()` method.

Understanding Channels

Once the proxies have validated and formatted the client-supplied arguments into a message object, this `IMessage`-compatible type is passed from the real proxy into a channel object. *Channels* are the entities in charge of transporting a message to the remote object and, if necessary, ensuring that any member return value is passed from the remote object back to the client. The .NET 2.0 base class libraries provide three channel implementations out of the box:

- TCP channel
- HTTP channel
- IPC channel

The *TCP channel* is represented by the `TcpChannel` class type and is used to pass messages using the TCP/IP network protocol. `TcpChannel` is helpful in that the formatted packets are quite lightweight, given that the messages are converted into a tight binary format using a related `BinaryFormatter` (yes, the same `BinaryFormatter` you saw in Chapter 17). Use of the `TcpChannel` type tends to result in faster remote access. The downside is that TCP channels are not firewall-friendly and may require the services of a system administrator to allow messages to pass across machine boundaries.

In contrast, the *HTTP channel* is represented by the `HttpChannel` class type, which converts message objects into a SOAP format using a related SOAP formatter. As you have seen, SOAP is XML-based and thus tends to result in beefier payloads than the payloads used by the `TcpChannel` type. Given this, using the `HttpChannel` can result in slightly slower remote access. On the plus side, HTTP is far more firewall-friendly, given that most firewalls allow textual packets to be passed over port 80.

Finally, as of .NET 2.0, we have access to the *IPC channel*, represented by the `IpChannel` type, which defines a communication channel for remoting using the IPC system of the Windows operating system. Because `IpChannel` bypasses traditional network communication to cross AppDomains, the `IpChannel` is much faster than the HTTP and TCP channels; however, it can be used only for communication between application domains *on the same physical computer*. Given this, you could never use `IpChannel` to build a distributed application that spans multiple physical computers. `IpChannel` can be an ideal option, however, when you wish to have two local programs share information in the fastest possible manner.

Regardless of which channel type you choose to use, understand that the `HttpChannel`, `TcpChannel`, and `IpChannel` types all implement the `IChannel`, `IChannelSender`, and `IChannelReceiver` interfaces. The `IChannel` interface (as you will see in just a bit) defines a small set of members that provide common functionality to all channel types. The role of `IChannelSender` is to define a common set of members for channels that are able to send information *to* a specific receiver. On the other hand, `IChannelReceiver` defines a set of members that allow a channel to receive information *from* a given sender.

To allow the client and server applications to register their channel of choice, you will make use of the `ChannelServices.RegisterChannel()` method, which takes a type implementing `IChannel`. Just to preview things to come, the following code snippet illustrates how a server-side application domain can register an HTTP channel on port 32469 (you'll see the client's role shortly):

```
// Create and register a server-side HttpChannel on port 32469.  
HttpChannel c = new HttpChannel(32469);  
ChannelServices.RegisterChannel(c);
```

Revisiting the Role of .NET Formatters

The final piece of the .NET remoting puzzle is the role of *formatter* objects. The `TcpChannel` and `HttpChannel` types both leverage an internal formatter, whose job it is to translate the message object into protocol-specific terms. As mentioned, the `TcpChannel` type makes use of the `BinaryFormatter` type, while the `HttpChannel` type uses the functionality provided by the `SoapFormatter` type. Given your work in the previous chapter, you should already have some insights as to how a given channel will format the incoming messages.

Once the formatted message has been generated, it is passed into the channel, where it will eventually reach its destination application domain, at which time the message is formatted from protocol-specific terms back to .NET-specific terms, at which point an entity termed the *dispatcher* invokes the correct method on the remote object.

All Together Now!

If your head is spinning from reading the previous sections, fear not! The transparent proxy, real proxy, message object, and dispatcher can typically be completely ignored, provided you are happy with the default remoting plumbing. To help solidify the sequence of events, ponder Figure 18-1, which illustrates the basic process of two objects communicating across distinct application domains.

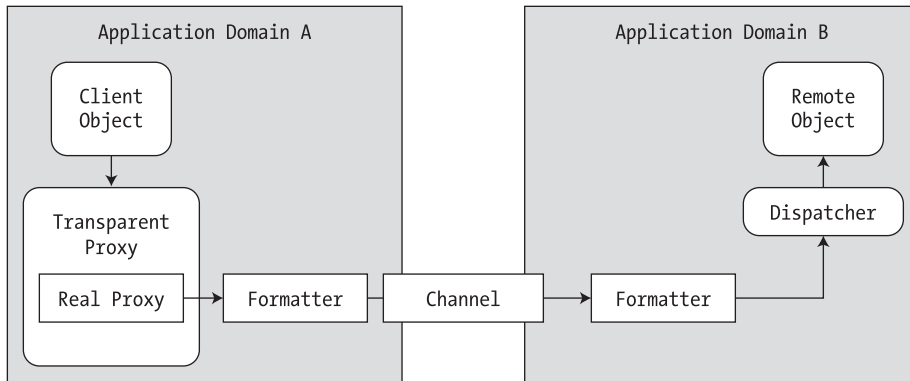


Figure 18-1. A high-level view of the default .NET remoting architecture

A Brief Word Regarding Extending the Default Plumbing

A key aspect of the .NET remoting layer is the fact that most of the default remoting layers can be extended or completely replaced at the whim of the developer. Thus, if you truly want (or possibly need) to build a custom message dispatcher, custom formatter, or custom real proxy, you are free to do so. You are also able to inject *additional* levels of indirection by plugging in custom types that stand between a given layer (e.g., a custom sink used to perform preprocessing or postprocessing of a given message). Now, to be sure, you may never need to retrofit the core .NET remoting layer in such ways. However, the fact remains that the .NET platform does provide the namespaces to allow you to do so.

Note This chapter does not address the topic of extending the default .NET remoting layer. If you wish to learn how to do so, check out *Advanced .NET Remoting* by Ingo Rammer (Apress, 2002).

Terms of the .NET Remoting Trade

Like any new paradigm, .NET remoting brings a number of TLAs (three-letter acronyms) into the mix. Thus, before you see your first code example, we do need to define a few terms used when describing the composition of a .NET remoting application. As you would guess, this terminology is used to describe a number of details regarding common questions that arise during the construction of a distributed application: How do we pass a type across application domain boundaries? When exactly is a remote type activated? How do we manage the lifetime of a remote object (and so forth)? Once you have an understanding of the related terminology, the act of building a distributed .NET application will be far less perplexing.

Object Marshaling Choices: MBR or MBV?

Under the .NET platform, you have two options regarding how a remote object is marshaled to the client. Simply put, *marshaling* describes how a remote object is passed between application domains. When you are designing a remotable object, you may choose to employ *marshal-by-reference* (MBR) or *marshal-by-value* (MBV) semantics. The distinction is as follows:

- *MBR objects*: The caller receives a proxy to the remote object.
- *MBV objects*: The caller receives a full copy of the object in its own application domain.

If you configure an MBR object type, the CLR ensures that the transparent and real proxies are created in the client's application domain, while the MBR object itself remains in the server's application domain. As the client invokes methods on the remote type, the .NET remoting plumbing (examined previously) takes over the show and will package, pass, and return information between application domain boundaries. To be sure, MBR objects have a number of traits above and beyond their physical location. As you will see, MBR objects have various configuration options regarding their activation options and lifetime management.

MBV objects, on the other hand, are *local copies* of remote objects (which leverage the .NET serialization protocol examined in Chapter 17). MBV objects have far fewer configuration settings, given that their lifetime is directly controlled by the client. Like any .NET object, once a client has released all references to an MBV type, it is a candidate for garbage collection. Given that MBV types are local copies of remote objects, as a client invokes members on the type, no network activity occurs during the process.

Now, understand that it will be quite common for a single server to provide access to numerous MBR and MBV types. As you may also suspect, MBR types tend to support methods that return various MBV types, which gives way to the familiar factory pattern (e.g., an object that creates and returns other related objects). The next question is, how do you configure your custom class types as MBR or MBV entities?

Configuring an MBV Object

The process of configuring an object as an MBV type is identical to the process of configuring an object for serialization. Simply annotate the type with the `[Serializable]` attribute:

```
[Serializable]
public class SportsCar
{...}
```

Configuring an MBR Object

MBR objects are not marked as such using a .NET attribute, but rather by deriving (directly or indirectly) from the `System.MarshalByRefObject` base class:

```
public class SportsCarFactory : MarshalByRefObject
{...}
```

Formally, the `MarshalByRefObject` type is defined as follows:

```
public abstract class MarshalByRefObject : object
{
    public virtual ObjRef CreateObjRef(Type requestedType);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual object GetLifetimeService();
    public Type GetType();
    public virtual object InitializeLifetimeService();
    public virtual string ToString();
}
```

Beyond the expected functionality provided by `System.Object`, Table 18-2 describes the role of the remaining members.

Table 18-2. *Key Members of System.MarshalByRefObject*

Member	Meaning in Life
CreateObjRef()	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object
GetLifetimeServices()	Retrieves the current lifetime service object that controls the lifetime policy for this instance
InitializeLifetimeServices()	Obtains a lifetime service object to control the lifetime policy for this instance

As you can tell, the gist of `MarshalByRefObject` is to define members that can be overridden to programmatically control the lifetime of the MBR object (more on lifetime management later in this chapter).

Note Just because you have configured a type as an MBV or MBR entity does not mean it is only usable within a remoting application, just that it *may* be used in a remoting application. For example, the `System.Windows.Forms.Form` type is a descendent of `MarshalByRefObject`; thus, if accessed remotely it is realized as an MBR type. If not, it is just another local object in the client's application domain.

Note As a corollary to the previous note, understand that if a .NET type is not serializable and does not include `MarshalByRefObject` in its inheritance chain, the type in question can only be activated and used in the originating application domain (meaning, the type is *context bound*; see Chapter 13 for more details).

Now that you understand the distinct traits of MBR and MBV types, let's check out some issues that are specific to MBR types (MBV types need not apply).

Activation Choices for MBR Types: WKO or CAO?

Another remoting-centric choice you face as a .NET programmer has to do with exactly *when* an MBR object is activated and *when* it should be a candidate for garbage collection on the server. This might seem like a strange choice to make, as you might naturally assume that MBR objects are created when the client requests them and die when the client is done with them. While it is true that the client is the entity in charge of instructing the remoting layer it wishes to communicate with a remote type, the server application domain may (or may not) create the type at the exact moment the client's code base requests it.

The reason for this seemingly strange behavior has to do with the optimization. Specifically, every MBR type may be configured to be activated using one of two techniques:

- As a well-known object (WKO)
- As a client-activated object (CAO)

Note A potential point of confusion is that fact that the acronym WKO is also called a *server-activated object (SAO)* in the .NET literature. In fact, you may see the SAO acronym in various .NET-centric articles and books. In keeping with the current terminology, I will use WKO throughout this chapter.

WKO objects are MBR types whose lifetimes are directly controlled by the server's application domain. The client-side application activates the remote type using a friendly, well-known string name (hence the term WKO). The server's application domain allocates WKO types when the client makes the first method call on the object (via the transparent proxy), *not* when the client's code base makes use of the new keyword or via the static `Activator.GetObject()` method, for example:

```
// Get a proxy to remote object. This line does NOT create the WKO type!
object remoteObj = Activator.GetObject( /* params seen later... */ );

// Invoke a method on remote WKO type. This WILL create the WKO object
// and invoke the ReturnMessage() method.
RemoteMessageObject simple = (RemoteMessageObject)remoteObj;
Console.WriteLine("Server says: {0}", simple.ReturnMessage());
```

The rationale for this behavior? This approach saves a network round-trip solely for the purpose of creating the object. As another interesting corollary, WKO types can *be created only via the type's default constructor*. This should make sense, given that the remote type's constructor is triggered only when the client makes the initial member invocation. Thus, the runtime has no other option than to invoke the type's default constructor.

Note Always remember: All WKO types must support a default constructor!

If you wish to allow the client to create a remote MBR object using a custom constructor, the server must configure the object as a CAO. CAO objects are entities whose lifetime is controlled by the client's application domain. When accessing a CAO type, a round-trip to the server occurs at the time the client makes use of the new keyword (using any of the type's constructors) or via the `Activator` type.

Stateful Configuration of WKO Types: Singleton or Single Call?

The final .NET design choice to consider with regard to MBR types has to do with how the server should handle multiple requests to a WKO type. CAO types need not apply, given that there is always a one-to-one correspondence between a client and a remote CAO type (because they are stateful).

Your first option is to configure a WKO type to function as a *singleton type*. The CLR will create a single instance of the remote type that will take requests from any number of clients, and it is a natural choice if you need to maintain stateful information among multiple remote callers. Given the fact that multiple clients could invoke the same method at the same time, the CLR places each client invocation on a new thread. It is *your* responsibility, however, to ensure that your objects are thread-safe using the same techniques described in Chapter 14.

In contrast, a *single call* object is a WKO type that exists only during the context of a single method invocation. Thus, if there are 20 clients making use of a WKO type configured with single call semantics, the server will create 20 distinct objects (one for each client), all of which are candidates for garbage collection directly after the method invocation. As you can guess, single call objects are far more scalable than singleton types, given that they are invariably stateless entities.

The server is the entity in charge of determining the stateful configuration of a given WKO type. Programmatically, these options are expressed via the `System.Runtime.Remoting.WellKnownObjectMode` enumeration:

```
public enum WellKnownObjectMode
{
    SingleCall,
    Singleton
}
```

Summarizing the Traits of MBR Object Types

As you have seen, configuring an MBV object is a no-brainer: Apply the [Serializable] attribute to allow copies of the type to be returned to the client's application domain. At this point, all interaction with the MBV type takes place in the client's locale. When the client is finished using the MBV type, it is a candidate for garbage collection, and all is well with the world.

With MBR types, however, you have a number of possible configuration choices. As you have seen, a given MBR type can be configured with regard to its time of activation, statefulness, and lifetime management. To summarize the array of possibilities, Table 18-3 documents how WKO and CAO types stack up against the traits you have just examined.

Table 18-3. Configuration Options for MBR Types

MBR Object Trait	WKO Behavior	CAO Behavior
Instantiation options	WKO types can only be activated using the default constructor of the type, which is triggered when the client makes the first method invocation.	CAO types can be activated using any constructor of the type. The remote object is created at the point the caller makes use of constructor semantics (or via the Activator type).
State management	WKO types can be configured as singleton or single call entities. Singleton types can service multiple clients and are therefore stateful. Single call types are alive only during a specific client-side invocation and are therefore stateless.	The lifetime of a CAO type is dictated by the caller; therefore, CAO types are stateful entities.
Lifetime management	Singleton WKO types make use of a lease-based management scheme (described later in this chapter). Single call WKO types are candidates for garbage collection after the current method invocation.	CAO types make use of a lease-based management scheme (described later in this chapter).

Basic Deployment of a .NET Remoting Project

Enough acronyms! At this point you are almost ready to build your first .NET remoting application. Before you do, however, I need to discuss one final detail: deployment. When you are building a .NET remoting application, you are almost certain to end up with three (yes, three, not two) distinct .NET assemblies that will constitute the entirety of your remote application. I am sure you can already account for the first two assemblies:

- *The client:* This assembly is the entity that is interested in obtaining access to a remote object (such as a Windows Forms or console application).
- *The server:* This assembly is the entity that receives channel requests from the remote client and hosts the remote objects.

So then, where does the third assembly fit in? In many cases, the server application is typically a host to a third assembly that defines and implements the remote objects. For convenience, I'll call

this assembly the *general assembly*. This decoupling of the assembly containing the remote objects and server host is quite important, in that both the client and the server assemblies typically set a reference to the general assembly to obtain the metadata definitions of the remotable types.

In the simplest case, the general assembly is placed into the application directory of the client and server. The only possible drawback to this approach is the fact that the client has a reference to an assembly that contains CIL code that is never used (which may be a problem if you wish to ensure that the end user cannot view proprietary code). Specifically, the only reason the client requires a reference to the general assembly is to obtain the metadata descriptions of the remotable types. You can overcome this glitch in several ways, for example:

- Construct your remote objects to make use of interface-based programming techniques. Given this, the client is able to set a reference to a .NET binary that contains nothing but interface definitions.
- Make use of the `soapsuds.exe` command-line application. Using this tool, you are able to generate an assembly that contains nothing but metadata descriptions of the remote types.
- Manually build an assembly that contains nothing but metadata descriptions of the remote types.

To keep things simple over the course of this chapter, you will build and deploy general assemblies that contain the required metadata as well as the CIL implementation.

Note If you wish to examine how to implement general assemblies using each of these alternatives, check out *Distributed .NET Programming in C#* by Tom Barnaby (Apress, 2002).

Building Your First Distributed Application

There is nothing more satisfying than building a distributed application using a new platform. To illustrate how quickly you're able to get up and running with the .NET remoting layer, let's build a simple example. As mentioned, the entirety of this example consists of three .NET assemblies:

- A general assembly named `SimpleRemotingAsm.dll`
- A client assembly named `SimpleRemoteObjectClient.exe`
- A server assembly named `SimpleRemoteObjectServer.exe`

Building the General Assembly

First, let's create the general assembly, `SimpleRemotingAsm.dll`, which will be referenced by both the server and client applications. `SimpleRemotingAsm.dll` defines a single MBR type named `RemoteMessageObject`, which supports two public members. The `DisplayMessage()` method prints a client-supplied message on the server's console window, while `ReturnMessage()` returns a message to the client. Here is the complete code of this new C# class library:

```
namespace SimpleRemotingAsm
{
    // This is a type that will be
    // marshaled by reference (MBR) if accessed remotely.
    public class RemoteMessageObject: MarshalByRefObject
    {
        public RemoteMessageObject()
        { Console.WriteLine("Constructing RemoteMessageObject!"); }
    }
}
```

```

        // This method takes an input string
        // from the caller.
        public void DisplayMessage(string msg)
        { Console.WriteLine("Message is: {0}", msg); }

        // This method returns a value to the caller.
        public string ReturnMessage()
        { return "Hello from the server!"; }
    }
}

```

The major point of interest is the fact that the type derives from the `System.MarshalByRefObject` base class, which ensures that the derived class will be accessible via a client-side proxy. Also note the custom default constructor that will print out a message when an instance of the type comes to life. That's it. Go ahead and build your new `SimpleRemotingAsm.dll` assembly.

Building the Server Assembly

Recall that server assemblies are essentially hosts for general assemblies that contain the remotable objects. Create a console program named `SimpleRemoteObjectServer`. The role of this assembly is to open a channel for the incoming requests and register `RemoteMessageObject` as a WKO. To begin, reference the `System.Runtime.Remoting.dll` and `SimpleRemotingAsm.dll` assemblies, and update `Main()` as follows:

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using SimpleRemotingAsm;

namespace SimpleRemoteObjectServer
{
    class SimpleObjServer
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** SimpleRemoteObjectServer started! *****");
            Console.WriteLine("Hit enter to end.");

            // Register a new HttpChannel
            HttpChannel c = new HttpChannel(32469);
            ChannelServices.RegisterChannel(c);

            // Register a WKO type, using singleton activation.
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(SimpleRemotingAsm.RemoteMessageObject),
                "RemoteMsgObj.soap",
                WellKnownObjectMode.Singleton);
            Console.ReadLine();
        }
    }
}

```

`Main()` begins by creating a new `HttpChannel` type using an arbitrary port ID. This port is opened on registering the channel via the static `ChannelServices.RegisterChannel()` method. Once the channel has been registered, the remote server assembly is now equipped to process incoming messages via port number 32469.

Note The number you assign to a port is typically up to you (or your system administrator). Do be aware, however, that port IDs below 1024 are reserved for system use.

Next, to register the `SimpleRemotingAsm.RemoteMessageObject` type as a WKO requires the use of the `RemotingConfiguration.RegisterWellKnownServiceType()` method. The first argument to this method is the type information of the type to be registered. The second parameter to `RegisterWellKnownServiceType()` is a simple string (of your choosing) that will be used to identify the object across application domain boundaries. Here, you are informing the CLR that this object is to be realized by the client using the name `RemoteMsgObj.soap`.

The final parameter is a member of the `WellKnownObjectMode` enumeration, which you have specified as `WellKnownObjectMode.Singleton`. Recall that singleton WKO types ensure that a single instance of the `RemoteMessageObject` will service all incoming requests. Build your server assembly and let's move on to the client-side code.

Building the SimpleRemoteObjectClient.exe Assembly

Now that you have a listener that is hosting your remotable object, the final step is to build an assembly that will request access to its services. Again, let's use a simple console application. Set a reference to `System.Runtime.Remoting.dll` and `SimpleRemotingAsm.dll`. Implement `Main()` as follows:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using SimpleRemotingAsm;

namespace SimpleRemoteObjectClient
{
    class SimpleObjClient
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** SimpleRemoteObjectClient started! *****");
            Console.WriteLine("Hit enter to end.");

            // Create a new HttpChannel.
            HttpChannel c = new HttpChannel();
            ChannelServices.RegisterChannel(c);

            // Get a proxy to remote WKO type.
            object remoteObj = Activator.GetObject(
                typeof(SimpleRemotingAsm.RemoteMessageObject),
                "http://localhost:32469/RemoteMsgObj.soap");

            // Now use the remote object.
            RemoteMessageObject simple = (RemoteMessageObject)remoteObj;
            simple.DisplayMessage("Hello from the client!");
            Console.WriteLine("Server says: {0}", simple.ReturnMessage());
            Console.ReadLine();
        }
    }
}
```

A few notes about this client application. First, notice that the client is also required to register an HTTP channel, but the client does not specify a port ID, as the end point is specified by the client-supplied activation URL. Given that the client is interacting with a registered WKO type, you are limited to triggering the type's default constructor. To do so, make use of the `Activator.GetObject()` method, specifying two parameters. The first is the type information that describes the remote object you are interested in interacting with. Read that last sentence again. Given that the `Activator.GetObject()` method requires the object's metadata description, it should make more sense as to why the client is *also* required to reference the general assembly! Again, at the end of the chapter you'll examine various ways to clean up this aspect of your client-side assembly.

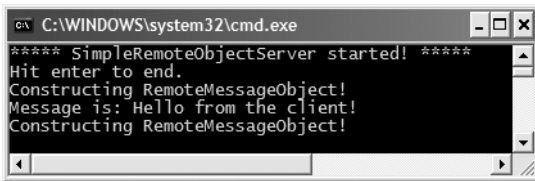
The second parameter to `Activator.GetObject()` is termed the *activation URL*. Activation URLs that describe a WKO type can be generalized into the following format:

```
ProtocolScheme://ComputerName:Port/ObjectUri
```

Finally, note that the `Activator.GetObject()` method returns a generic `System.Object` type, and thus you must make use of an explicit cast to gain access to the members of the `RemoteMessageObject`.

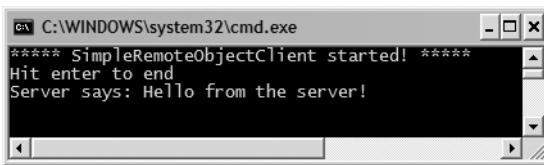
Testing the Remoting Application

To test your application, begin by launching the server application, which will open an HTTP channel and register `RemoteMessageObject` for remote for access. Next, launch an instance of the client application. If all is well, your server window should appear as shown in Figure 18-2, while the client application displays what you see in Figure 18-3.



```
C:\WINDOWS\system32\cmd.exe
***** SimpleRemoteObjectServer started! *****
Hit enter to end.
Constructing RemoteMessageObject!
Message is: Hello from the client!
Constructing RemoteMessageObject!
```

Figure 18-2. The server's output



```
C:\WINDOWS\system32\cmd.exe
***** SimpleRemoteObjectClient started! *****
Hit enter to end
Server says: Hello from the server!
```

Figure 18-3. The client's output

Understanding the ChannelServices Type

As you have seen, when a server application wishes to advertise the existence of a remote type, it makes use of the `System.Runtime.Remoting.Channels.ChannelServices` type. `ChannelServices` provides a small set of static methods that aid in the process of remoting channel registration, resolution, and URL discovery. Table 18-4 documents some of the core members.

Table 18-4. *Select Members of the ChannelServices Type*

Member	Meaning in Life
RegisteredChannels	This property gets or sets a list of currently registered channels, each of which is represented by the IChannel interface.
DispatchMessage()	This method dispatches incoming remote calls.
GetChannel()	This method returns a registered channel with the specified name.
GetUrlsForObject()	This method returns an array of all the URLs that can be used to reach the specified object.
RegisterChannel()	This method registers a channel with the channel services.
UnregisterChannel()	This method unregisters a particular channel from the registered channels list.

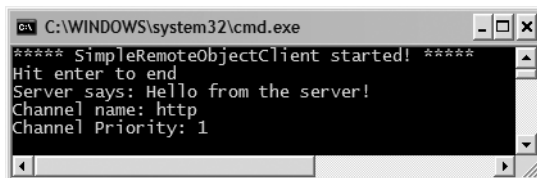
In addition to the aptly named `RegisterChannel()` and `UnregisterChannel()` methods, `ChannelServices` defines the `RegisteredChannels` property. This member returns an array of `IChannel` interfaces, each representing a handle to each channel registered in a given application domain. The definition of the `IChannel` interface is quite straightforward:

```
public interface IChannel
{
    string ChannelName { get; }
    int ChannelPriority { get; }
    string Parse(string url, ref String objectURI);
}
```

As you can see, each channel is given a friendly string name as well as a priority level. To illustrate, if you were to update the `Main()` method of the `SimpleRemoteObjectClient` application with the following logic:

```
// List all registered channels.
IChannel[] channelObjs = ChannelServices.RegisteredChannels;
foreach(IChannel i in channelObjs)
{
    Console.WriteLine("Channel name: {0}", i.ChannelName);
    Console.WriteLine("Channel Priority: {0}", i.ChannelPriority);
}
```

you would find the client-side console now looks like Figure 18-4.

**Figure 18-4.** *Enumerating client-side channels*

Understanding the RemotingConfiguration Type

Another key remoting-centric type is `RemotingConfiguration`, which as its name suggests is used to configure various aspects of a remoting application. Currently, you have seen this type in use on the server side (via the call to the `RegisterWellKnownServiceType()` method). Table 18-5 lists additional static members of interest, some of which you'll see in action over the remainder of this chapter.

Table 18-5. *Members of the RemotingConfiguration Type*

Member	Meaning in Life
<code>ApplicationId</code>	Gets the ID of the currently executing application
<code>ApplicationName</code>	Gets or sets the name of a remoting application
<code>ProcessId</code>	Gets the ID of the currently executing process
<code>Configure()</code>	Reads the configuration file and configures the remoting infrastructure
<code>GetRegisteredActivatedClientTypes()</code>	Retrieves an array of object types registered on the client as types that will be activated remotely
<code>GetRegisteredActivatedServiceTypes()</code>	Retrieves an array of object types registered on the service end that can be activated on request from a client
<code>GetRegisteredWellKnownClientTypes()</code>	Retrieves an array of object types registered on the client end as well-known types
<code>GetRegisteredWellKnownServiceTypes()</code>	Retrieves an array of object types registered on the service end as well-known types
<code>IsWellKnownClientType()</code>	Checks whether the specified object type is registered as a well-known client type
<code>RegisterActivatedClientType()</code>	Registers an object on the client end as a type that can be activated on the server
<code>RegisterWellKnownClientType()</code>	Registers an object on the client end as a well-known type (single call or singleton)
<code>RegisterWellKnownServiceType()</code>	Registers an object on the service end as a well-known type (single call or singleton)

Recall that the .NET remoting layer distinguishes between two types of MBR objects: WKO (server-activated) and CAO (client-activated). Furthermore, WKO types can be configured to make use of singleton or single call activations. Using the functionality of the `RemotingConfiguration` type, you are able to dynamically obtain such information at runtime. For example, if you update the `Main()` method of your `SimpleRemoteObjectServer` application with the following:

```
static void Main(string[] args)
{
    ...
    // Set a friendly name for this server app.
    RemotingConfiguration.ApplicationName = "First server app!";
    Console.WriteLine("App Name: {0}",
        RemotingConfiguration.ApplicationName);

    // Get an array of WellKnownServiceTypeEntry types
    // that represent all the registered WKOs.
    WellKnownServiceTypeEntry[] WKOs =
        RemotingConfiguration.GetRegisteredWellKnownServiceTypes();
    // Now print their statistics.
```

```

foreach(WellKnownServiceTypeEntry wko in WKOs)
{
    Console.WriteLine("Asm name containing WKO: {0}", wko.AssemblyName);
    Console.WriteLine("URL to WKO: {0}", wko.ObjectUri);
    Console.WriteLine("Type of WKO: {0}", wko.ObjectType);
    Console.WriteLine("Mode of WKO: {0}", wko.Mode);
}
}

```

you would find a list of all WKO types registered by this server application domain. As you iterate over the array of `WellKnownServiceTypeEntry` types, you are able to print out various points of interest regarding each WKO. Given that your server's application registered only a single type (`SimpleRemotingAsm.RemoteMessageObject`), you'll receive the output shown in Figure 18-5.

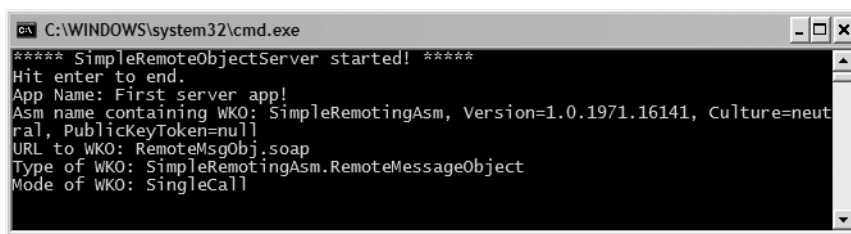


Figure 18-5. *Server-side statistics*

The other major method of the `RemotingConfiguration` type is `Configure()`. As you'll see in just a bit, this static member allows the client- and server-side application domains to make use of remotings configuration files.

Revisiting the Activation Mode of WKO Types

Recall that WKO types can be configured to function under singleton or single call activation. Currently, your server application has registered your WKO to employ singleton activation semantics:

```

// Singletons can service multiple clients.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "RemoteMsgObj.soap",
    WellKnownObjectMode.Singleton);

```

Again, singleton WKOs are capable of receiving requests from multiple clients. Thus, singleton objects maintain a one-to-many relationship between themselves and the remote clients. To test this behavior for yourself, run the server application (if it is not currently running) and launch three separate client applications. If you look at the output for the server, you will find a single call to the `RemoteMessageObject`'s default constructor.

Now to test the behavior of single call objects, modify the server to register the WKO to support single call activation:

```

// Single call types maintain a 1-to-1 relationship
// between client and WKO.
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "RemoteMsgObj.soap",
    WellKnownObjectMode.SingleCall);

```

Once you have recompiled and run the server application, again launch three clients. This time you can see that a new `RemoteMessageObject` is created for each client request. As you might be able to gather, if you wish to share stateful data between multiple remote clients, singleton activation provides one possible alternative, as all clients are communicating with a single instance of the remote object.

Source Code The `SimpleRemotingAsm`, `SimpleRemoteObjectServer`, and `SimpleRemoteObjectClient` projects are located under the Chapter 18 directory.

Deploying the Server to a Remote Machine

At this point, you have just crossed an application and process boundary on a single machine. If you're connected to an additional machine, let's extend this example to allow the client to interact with the `RemoteMessageObject` type across a machine boundary. To do so, follow these steps:

1. On your server machine, create and share a folder to hold your server-side assemblies.
2. Copy the `SimpleRemoteObjectServer.exe` and `SimpleRemotingAsm.dll` assemblies to this server-side share point.
3. Open your `SimpleRemoteObjectClient` project workspace and retrofit the activation URL to specify the name of the remote machine, for example:

```
// Get a proxy to remote object.  
object remoteObj = Activator.GetObject(  
    typeof(SimpleRemotingAsm.RemoteMessageObject),  
    "http://YourRemoteBoxName:32469/RemoteMsgObj.soap");
```

4. Execute the `SimpleRemoteObjectServer.exe` application on the server machine.
5. Execute the `SimpleRemoteObjectClient.exe` application on the client machine.
6. Sit back and grin.

Note Activation URLs may specify a machine's IP address in place of its friendly name.

Leveraging the TCP Channel

Currently, your remote object is accessible via the HTTP network protocol. As mentioned, this protocol is quite firewall-friendly, but the resulting SOAP packets are a bit on the bloated side (given the nature of XML data representation). To lighten the payload, you can update the client and server assemblies to make use of the TCP channel, and therefore make use of the `BinaryFormatter` type behind the scenes. Here are the relevant updates to the server assembly:

Note When you are defining an object to be URI-accessible via a TCP endpoint, it is common (but not required) to make use of the `*.rem` (i.e., remote) extension.

```
// Server adjustments!  
using System.Runtime.Remoting.Channels.Tcp;  
...  
static void Main(string[] args)
```

```

{
...
    // Create a new TcpChannel
    TcpChannel c = new TcpChannel(32469);
    ChannelServices.RegisterChannel(c);

    // Register a 'well-known' object in single call mode.
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(SimpleRemotingAsm.RemoteMessageObject),
        "RemoteMsgObj.rem",
        WellKnownObjectMode.SingleCall);
    Console.ReadLine();
}

```

Notice that you are now registering a `System.Runtime.Remoting.Channels.Tcp.TcpChannel` type to the .NET remoting layer. Also note that the object URI has been altered to support a more generic name (`RemoteMsgObj.rem`) rather than the SOAP-centric `*.soap` extension. The client-side updates are equally as simple:

```

// Client adjustments!
using System.Runtime.Remoting.Channels.Tcp;
...
static void Main(string[] args)
{
...
    // Create a new TcpChannel
    TcpChannel c = new TcpChannel();
    ChannelServices.RegisterChannel(c);
    // Get a proxy to remote object.
    object remoteObj = Activator.GetObject(
        typeof(SimpleRemotingAsm.RemoteMessageObject),
        "tcp://localhost:32469/RemoteMsgObj.rem");

    // Use object.
    RemoteMessageObject simple = (RemoteMessageObject)remoteObj;
    simple.DisplayMessage("Hello from the client!");
    Console.WriteLine("Server says: {0}", simple.ReturnMessage());
    Console.ReadLine();
}

```

The only point to be aware of here is that the client's activation URL now must specify the `tcp://` channel qualifier rather than `http://`. Beyond that, the bulk of the code base is identical to the previous `HttpChannel` logic.

Source Code The `TCPSimpleRemoteObjectServer` and `TCPSimpleRemoteObjectClient` projects are located under the Chapter 18 directory (both projects use the `SimpleRemotingAsm.dll` created previously).

A Brief Word Regarding the `IpcChannel`

Before moving on to an examination of remoting configuration files, recall that .NET 2.0 also provides the `IpcChannel` type, which provides the fastest possible manner in which two applications *on the same machine* can exchange information. Given that this chapter is geared toward covering distributed programs that involve two or more computers, interested readers should look up `IpcChannel` in the .NET Framework 2.0 SDK documentation (as you might guess, the code is just about identical to working with `HttpChannel` and `TcpChannel`).

Remoting Configuration Files

At this point you have successfully built a distributed application using the .NET remoting layer. One issue you may have noticed in these first examples is the fact that the client and the server applications have a good deal of hard-coded logic within their respective binaries. For example, the server specifies a fixed port ID, fixed activation mode, and fixed channel type. The client, on the other hand, hard-codes the name of the remote object it is attempting to interact with.

As you might agree, it is wishful thinking to assume that initial design notes remain unchanged once an application is deployed. Ideally, details such as port ID and object activation mode (and what-not) could be altered on the fly without needing to recompile and redistribute the client or server code bases. Under the .NET remoting scheme, all the aforementioned issues can be circumvented using the remoting configuration file.

As you will recall from Chapter 11, *.config can be used to provide hints to the CLR regarding the loading of externally referenced assemblies. The same *.config files can be used to inform the CLR of a number of remoting-related details, on both the client side and the server side.

When you build a remoting *.config file, the <system.runtime.remoting> element is used to hold various remoting-centric details. Do be aware that if you're building an application that already has a *.config file that specifies assembly resolution details, you're free to add remoting elements within the same file. Thus, a single *.config file that contains remoting and binding information would look something like this:

```
<configuration>
  <system.runtime.remoting>
    <!-- configure client/server remoting settings here -->
  </system.runtime.remoting>
  <runtime>
    <!-- binding assembly settings here -->
  </runtime>
</configuration>
```

If your configuration file has no need to specify assembly binding logic, you can omit the <runtime> element and make use of the following skeleton *.config file:

```
<configuration>
  <system.runtime.remoting>
    <!-- configure client/server remoting settings here -->
  </system.runtime.remoting>
</configuration>
```

Building Server-Side *.config Files

Server-side configuration files allow you to declare the objects that are to be reached via remote invocations as well as channel and port information. Basically, using the <service>, <wellknown>, and <channels> elements, you are able to replace the following server-side logic:

```
// Hard-coded HTTP server logic.
HttpChannel c = new HttpChannel(32469);
ChannelServices.RegisterChannel(c);
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(SimpleRemotingAsm.RemoteMessageObject),
    "RemoteMsgObj.soap",
    WellKnownObjectMode.Singleton);
```

with the following *.config file:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown
          mode="Singleton"
          type="SimpleRemotingAsm.RemoteMessageObject, SimpleRemotingAsm"
          objectUri="RemoteMsgObj.soap"/>
        </service>
      <channels>
        <channel ref="http" port="32469"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Notice that much of the relevant server-side remoting information is wrapped within the scope of the `<service>` (not *server*) element. The child `<wellknown>` element makes use of three attributes (*mode*, *type*, and *objectUri*) to specify the well-known object to register with the .NET remoting layer. The child `<channels>` element contains any number of `<channel>` elements that allow you to define the type of channel (in this case, HTTP) to open on the server. TCP channels would simply make use of the *tcp* string token in place of *http*.

As the `SimpleRemoteObjectServer.exe.config` file contains all the necessary information, the server-side `Main()` method cleans up considerably. All you are required to do is make a single call to `RemotingConfiguration.Configure()` and specify the name of your configuration file.

```

static void Main(string[] args)
{
    // Register a 'well-known' object using a *.config file.
    RemotingConfiguration.Configure("SimpleRemoteObjectServer.exe.config");
    Console.WriteLine("Server started! Hit enter to end");
    Console.ReadLine();
}

```

Building Client-Side *.config Files

Clients are also able to leverage remoting-centric **.config* files. Unlike a server-side configuration file, client-side configuration files make use of the `<client>` element to identify the name of the well-known object the caller wishes to interact with. In addition to providing the ability to dynamically change the remoting information without the need to recompile the code base, client-side **.config* files allow you to create the proxy type directly using the C# `new` keyword, rather than the `Activator.GetObject()` method. Thus, if you have the following client-side **.config* file:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "SimpleRemoteObjectClient">
        <wellknown
          type="SimpleRemotingAsm.RemoteMessageObject, SimpleRemotingAsm"
          url="http://localhost:32469/RemoteMsgObj.soap"/>
        </client>
      <channels>
        <channel ref="http"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

you are able to update the client's `Main()` method as follows:

```
static void Main(string[] args)
{
    RemotingConfiguration.Configure("SimpleRemoteObjectClient.exe.config");
    // Using *.config file, the client is able to directly 'new' the type.
    RemoteMessageObject simple = new RemoteMessageObject();
    simple.DisplayMessage("Hello from the client!");
    Console.WriteLine("Server says: {0}", simple.ReturnMessage());
    Console.WriteLine("Client started! Hit enter to end");
    Console.ReadLine();
}
```

Of course, when you run the application, the output is identical. If the client wishes to make use of the TCP channel, the `url` property of the `<wellknown>` element and `<channel>` ref property must make use of the `tcp` token in place of `http`.

Source Code The `SimpleRemoteObjectServerWithConfig` and `SimpleRemoteObjectClientWithConfig` projects are located under the Chapter 18 subdirectory (both of which make use of the `SimpleRemotingAsm.dll` created previously).

Working with MBV Objects

Our first remoting applications allowed client-side access to a single WKO type. Recall that WKO types are (by definition) MBR types, and therefore client access takes place via an intervening proxy. In contrast, MBV types are local copies of a server-side object, which are typically returned from a public member of an MBR type. Although you already know how to configure an MBV type (mark a class with the `[Serializable]` attribute), you have not yet seen an example of MBV types in action (beyond passing string data between the two parties). To illustrate the interplay of MBR and MBV types, let's see another example involving three assemblies:

- The general assembly named `CarGeneralAsm.dll`
- The client assembly named `CarProviderClient.exe`
- The server assembly named `CarProviderServer.exe`

As you might assume, the code behind the client and server applications is more or less identical to the previous example, especially since these applications will again make use of `*.config` files. Nevertheless, let's step through the process of building each assembly one at a time.

Building the General Assembly

During our examination of object serialization in Chapter 17, you created a type named `JamesBondCar` (in addition to the dependent `Radio` and `Car` classes). The `CarGeneralAsm.dll` code library will reuse these types, so begin by using the Project ► Add Existing Item menu command and include these `*.cs` files into this new Class Library project (the automatically provided `Class1.cs` file can be deleted). Given that each of these types has already been marked with the `[Serializable]` attribute, they are ready to be marshaled by value to a remote client.

All you need now is an MBR type that provides access to the `JamesBondCar` type. To make things a bit more interesting, however, your MBR object (`CarProvider`) will maintain a generic `List<>` of `JamesBondCar` types. `CarProvider` will also define two members that allow the caller to obtain a specific `JamesBondCar` as well as receive the entire `List<>` of types. Here is the complete code for the new class type:

```

namespace CarGeneralAsm
{
    // This type is an MBR object that provides
    // access to related MBV types.
    public class CarProvider : MarshalByRefObject
    {
        private List<JamesBondCar> theJBCars =
            new List<JamesBondCar>();

        // Add some cars to the list.
        public CarProvider()
        {
            Console.WriteLine("Car provider created");
            theJBCars.Add(new JamesBondCar("QMobile", 140, true, true));
            theJBCars.Add(new JamesBondCar("Flyer", 140, true, false));
            theJBCars.Add(new JamesBondCar("Swimmer", 140, false, true));
            theJBCars.Add(new JamesBondCar("BasicJBC", 140, false, false));
        }
        // Get all the JamesBondCars.
        public List<JamesBondCar> GetAllAutos()
        { return theJBCars; }
        // Get one JamesBondCar.
        public JamesBondCar GetJBCByIndex(int i)
        { return (JamesBondCar)theJBCars[i]; }
    }
}

```

Notice that the `GetAllAutos()` method returns the internal `List<>` type. The obvious question is how this member of the `System.Collections.Generic` namespace is marshaled back to the caller. If you look up this type using the .NET Framework 2.0 SDK documentation, you will find that `List<>` has been decorated with the `[Serializable]` attribute:

```

[SerializableAttribute()]
public class List<T> : IList, ICollection, IEnumerable

```

Therefore, the entire contents of the `List<>` type will be marshaled by value to the caller (provided the contained types are also serializable)! This brings up a very good point regarding .NET remoting and members of the base class libraries. In addition to the custom MBV and MBR types you may create yourself, understand that any type in the base class libraries that is decorated with the `[Serializable]` attribute is able to function as an MBV type in the .NET remoting architecture. Likewise, any type that derives (directly or indirectly) from `MarshalByRefObject` will function as an MBR type.

Note Be aware that the `SoapFormatter` does not support serialization of generic types. If you build methods that receive or return generic types (such as the `List<>`), you must make use of the `BinaryFormatter` and the `TcpChannel` object.

Building the Server Assembly

The server host assembly (`CarProviderServer.exe`) has the following logic within `Main()`:

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using CarGeneralAsm;

```

```

namespace CarProviderServer
{
    class CarServer
    {
        static void Main(string[] args)
        {
            RemotingConfiguration.Configure("CarProviderServer.exe.config");
            Console.WriteLine("Car server started! Hit enter to end");
            Console.ReadLine();
        }
    }
}

```

The related *.config file is just about identical to the server-side *.config file you created in the previous example. The only point of interest is to define an object URI value that makes sense for the CarProvider type:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
          type="CarGeneralAsm.CarProvider, CarGeneralAsm"
          objectUri="carprovider.rem" />
      </service>
      <channels>
        <channel ref="tcp" port="32469" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Building the Client Assembly

Last but not least, we have the client application that will make use of the MBR CarProvider type in order to obtain discrete JamesBondCars types as well as the List<> type. Once you obtain a type from the CarProvider, you'll send it into the UseCar() helper function from processing:

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using CarGeneralAsm;
using System.Collections.Generic;

namespace CarProviderClient
{
    class CarClient
    {
        private static void UseCar(JamesBondCar c)
        {
            Console.WriteLine("-> Name: {0}", c.PetName);
            Console.WriteLine("-> Max speed: {0}", c.MaxSpeed);
            Console.WriteLine("-> Seaworthy? : {0}", c.isSeaworthy);
            Console.WriteLine("-> Flight worthy? : {0}", c.isFlightWorthy);
            Console.WriteLine();
        }
    }
}

```

```

static void Main(string[] args)
{
    RemotingConfiguration.Configure("CarProviderClient.exe.config");
    // Make the car provider.
    CarProvider cp = new CarProvider();
    // Get first JBC.
    JamesBondCar qCar = cp.GetJBCByIndex(0);
    // Get all JBCs.
    List<JamesBondCar> allJBCs = cp.GetAllAutos();
    // Use first car.
    UseCar(qCar);
    // Use all cars in List<>.
    foreach (JamesBondCar j in allJBCs)
        UseCar(j);
    Console.WriteLine("Client started! Hit enter to end");
    Console.ReadLine();
}
}
}

```

The client side *.config file is also what you would expect. Simply update the activation URL:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "CarClient">
        <wellknown
          type="CarGeneralAsm.CarProvider, CarGeneralAsm"
          url="tcp://localhost:32469/carprovider.rem"/>
        </client>
      <channels>
        <channel ref="tcp"/>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Now, run your server and client applications (in that order, of course) and observe the output. Your client-side console window will whirl through the JamesBondCars and print out the statistics of each type. Recall that as you interact with the List<> and JamesBondCar types, you are operating on their members within the client's application domain, as they have both been marked with the [Serializable] attribute.

To prove that point, update the UseCar() helper function to call the TurnOnRadio() method on the incoming JamesBondCar. Now, run the server and client applications once again. Notice that the message box appears on the client machine! Had the Car, Radio, and JamesBondCar types been configured as MBR types, the server would be the machine displaying the message box prompts. If you wish to verify this, derive each type from MarshalByRefObject and recompile all three assemblies (to ensure Visual Studio 2005 copies the latest CarGeneralAsm.dll into the client's and server's application directory). When you run the application once again, the message boxes appear on the remote machine.

Source Code The CarGeneralAsm, CarProviderServer, and CarProviderClient projects are located under the Chapter 18 subdirectory.

Understanding Client-Activated Objects

All of these current remoting examples have made use of WKO. Recall that WKOs have the following characteristics:

- WKOs can be configured either as singleton or single call.
- WKOs can only be activated using the type's default constructor.
- WKOs are instantiated on the server on the first client-side member invocation.

CAO types on the other hand, can be instantiated using any constructor on the type and are created at the point the client makes use of the C# `new` keyword or `Activator` type. Furthermore, the lifetime of CAO types is monitored by the .NET leasing mechanism. Do be aware that when you configure a CAO type, the .NET remoting layer will generate a specific CAO remote object to service each client. Again, the big distinction is the fact that CAOs are always alive (and therefore stateful) beyond a single method invocation.

To illustrate the construction, hosting, and consumption of CAO types, let's retrofit the previous automobile-centric general assembly. Assume that your MBR `CarProvider` class has defined an additional constructor that allows the client to pass in an array of `JamesBondCar` types that will be used to populate the generic `List<>`:

```
public class CarProvider : MarshalByRefObject
{
    private List<JamesBondCar> theJBCars
        = new List<JamesBondCar>();

    public CarProvider(JamesBondCar[] theCars)
    {
        Console.WriteLine("Car provider created with custom ctor");
        theJBCars.AddRange(theCars);
    }
    ...
}
```

To allow the caller to activate the `CarProvider` using your new constructor syntax, you need to build a server application that registers `CarProvider` as a CAO type rather than a WKO type. This may be done programmatically (à la the `RemotingConfiguration.RegisterActivatedServiceType()` method) or using a server-side `*.config` file. If you wish to hard-code the name of the CAO object within the host server's code base, all you need to do is pass in the type information of the type(s) (after creating and registering a channel) as follows:

```
// Hard-code the fact that CarProvider is a CAO type.
RemotingConfiguration.RegisterActivatedServiceType(
    typeof(CAOCarGeneralAsm.CarProvider));
```

If you would rather leverage the `*.config` file, replace the `<wellknown>` element with the `<activated>` element as follows:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <activated type = "CAOCarGeneralAsm.CarProvider,
          CAOCarGeneralAsm"/>
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

```

    </application>
  </system.runtime.remoting>
</configuration>

```

Finally, you need to update the client application, not only by way of the *.config file (or programmatically in the code base) to request access to the remote CAO, but also to indeed trigger the custom constructor of the CarProvider type. Here are the relevant updates to the client-side Main() method:

```

static void Main(string[] args)
{
    // Read updated *.config file.
    RemotingConfiguration.Configure("CA0CarProviderClient.exe.config");
    // Create array of types to pass to provider.
    JamesBondCar[] cars =
    {
        new JamesBondCar("Viper", 100, true, false),
        new JamesBondCar("Shaken", 100, false, true),
        new JamesBondCar("Stirred", 100, true, true)
    };
    // Now trigger the custom ctor.
    CarProvider cp = new CarProvider(cars);
    ...
}

```

The updated client-side *.config file also makes use of the <activated> element, as opposed to <wellknown>. In addition, the <client> element now requires the url property to define the location of the registered CAO. Recall that when the server registered the CarProvider as a WKO, the client specified such information within the <wellknown> element.

```

<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "CarClient"
        url = "tcp://localhost:32469">
        <activated type = "CA0CarGeneralAsm.CarProvider, CA0CarGeneralAsm" />
      </client>
    <channels>
      <channel ref="tcp"/>
    </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

If you would rather hard-code the client's request to the CAO type, you can make use of the RegistrationServices.RegisterActivatedClientType() method as follows:

```

static void Main(string[] args)
{
    // Use hard-coded values.
    RemotingConfiguration.RegisterActivatedClientType(
        typeof(CA0CarGeneralAsm.CarProvider),
        "tcp://localhost:32469");
    ...
}

```

If you now execute the updated server and client assemblies, you will be pleased to find that you are able to pass your custom array of JamesBondCar types to the remote CarProvider via the overloaded constructor.

Source Code The `CAOCarGeneralAsm`, `CAOCarProviderServer`, and `CAOCarProviderClient` projects are located under the Chapter 18 subdirectory.

The Lease-Based Lifetime of CAO/WKO-Singleton Objects

As you have seen, WKO types configured with single call activation are alive only for the duration of the current method call. Given this fact, WKO single call types are stateless entities. As soon as the current invocation has completed, the WKO single call type is a candidate for garbage collection.

On the other hand, CAO types and WKO types that have been configured to use singleton activation are both, by their nature, stateful entities. Given these two object configuration settings, the question that must be asked is, how does the server process know when to destroy these MBR objects? Clearly, it would be a huge problem if the server machine garbage-collected MBR objects that were currently in use by a remote client. If the server machine waits too long to release its set of MBR types, this may place undo stress on the system, especially if the MBR object(s) in question maintain valuable system resources (database connections, unmanaged types, and whatnot).

The lifetime of a CAO or WKO-singleton MBR type is governed by a “lease time” that is tightly integrated with the .NET garbage collector. If the lease time of a CAO or WKO-singleton MBR type expires, the object is ready to be garbage-collected on the next collection cycle. Like any .NET type, if the remote object has overridden `System.Object.Finalize()` (via the C# destructor syntax), the .NET runtime will indeed trigger the finalization logic.

The Default Leasing Behavior

CAO and WKO-singleton MBR types have what is known as a *default lease*, which lasts for five minutes. If the runtime detects five minutes of inactivity have passed for a CAO or WKO-singleton MBR type, the assumption is that the client is no longer making use of the object and therefore the remote object may be garbage-collected. However, when the default lease expires, this does not imply that the object is *immediately* marked for garbage collection. In reality, there are many ways to influence the behavior of the default lease.

First and foremost, anytime the remote client invokes a member of the remote CAO or WKO-singleton MBR type, the lease is renewed back to its five-minute limit. In addition to the automatic client-invocation-centric renew policy, the .NET runtime provides three additional alternatives:

- *.config files can be authored that override the default lease settings for remote objects.
- Server-side lease sponsors can be used to act on behalf of a remote object whose lease time has expired.
- Client-side lease sponsors can be used to act on behalf of a remote object whose lease time has expired.

We will check out these options over the next several sections, but for the time being let's examine the default lease settings of a remote type. Recall that the `MarshalByRefObject` base class defines a member named `GetLifetimeService()`. This method returns a reference to an internally implemented object that supports the `System.Runtime.Remoting.Lifetime.ILease` interface. As you would guess, the `ILease` interface can be used to interact with the leasing behavior of a given CAO or WKO-singleton type. Here is the formal definition:

```
public interface ILease
{
    TimeSpan CurrentLeaseTime { get; }
```

```

LeaseState CurrentState { get; }
TimeSpan InitialLeaseTime { get; set; }
TimeSpan RenewOnCallTime { get; set; }
TimeSpan SponsorshipTimeout { get; set; }
void Register(System.Runtime.Remoting.Lifetime.ISponsor obj);
void Register(System.Runtime.Remoting.Lifetime.ISponsor obj,
    TimeSpan renewalTime);
TimeSpan Renew(TimeSpan renewalTime);
void Unregister(System.Runtime.Remoting.Lifetime.ISponsor obj);
}

```

The `ILease` interface not only allows you to obtain information regarding the current lease (via `CurrentLeaseTime`, `CurrentState`, and `InitialLeaseTime`), but also provides the ability to build lease “sponsors” (more details on this later). Table 18-6 documents role of each `ILease` member.

Table 18-6. *Members of the `ILease` Interface*

Member	Meaning in Life
<code>CurrentLeaseTime</code>	Gets the amount of time remaining before the object deactivates, if it does not receive further method invocations.
<code>CurrentState</code>	Gets the current state of the lease, represented by the <code>LeaseState</code> enumeration.
<code>InitialLeaseTime</code>	Gets or sets the initial amount of time for a given lease. The initial lease time of an object is the amount of time following the initial activation before the lease expires if no other method calls occur.
<code>RenewOnCallTime</code>	Gets or sets the amount of time by which a call to the remote object increases the <code>CurrentLeaseTime</code> .
<code>SponsorshipTimeout</code>	Gets or sets the amount of time to wait for a sponsor to return with a lease renewal time.
<code>Register()</code>	Overloaded. Registers a sponsor for the lease.
<code>Renew()</code>	Renews a lease for the specified time.
<code>Unregister()</code>	Removes a sponsor from the sponsor list.

To illustrate the characteristics of the default lease of a CAO or WKO-singleton remote object, assume that your current `CAOCarGeneralAsm` project has defined a new internal class named `LeaseInfo`. `LeaseInfo` supports a static member named `LeaseStats()`, which dumps select statistics regarding the current lease for the `CarProvider` type to the server-side console window (be sure to specify a `using` directive for the `System.Runtime.Remoting.Lifetime` namespace to inform the compiler where the `ILease` type is defined):

```

internal class LeaseInfo
{
    public static void LeaseStats(ILease itfLease)
    {
        Console.WriteLine("***** Lease Stats *****");
        Console.WriteLine("Lease state: {0}", itfLease.CurrentState);
        Console.WriteLine("Initial lease time: {0}:{1}",
            itfLease.InitialLeaseTime.Minutes,
            itfLease.InitialLeaseTime.Seconds);
        Console.WriteLine("Current lease time: {0}:{1}",
            itfLease.CurrentLeaseTime.Minutes,
            itfLease.CurrentLeaseTime.Seconds);
        Console.WriteLine("Renew on call time: {0}:{1}",
            itfLease.RenewOnCallTime.Minutes,

```

```

        itfLease.RenewOnCallTime.Seconds);
        Console.WriteLine();
    }
}

```

Now that you have this helper type in place, assume `LeaseInfo.LeaseStats()` is called within the `GetJBCByIndex()` and `GetAllAutos()` methods of the `CarProvider` type. Once you recompile the server and client assemblies (again, simply to ensure Visual Studio 2005 copies the latest and greatest version of the `CarGeneralAsm.dll` to the client and server application directories), run the application once again. Your server's console window should now look something like Figure 18-6.

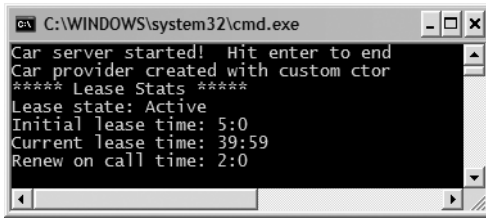


Figure 18-6. *The default lease information for CarProvider*

Altering the Default Lease Characteristics

Obviously, the default lease characteristics of a CAO/WKO-singleton type may not be appropriate for each and every CAO or WKO-singleton remote object. If you wish to alter these default settings, you have two approaches:

- You can adjust the default lease settings using a server-side `*.config` file.
- You can programmatically alter the settings of a type's default lease by overriding members of the `MarshalByRefObject` base class.

While each of these options will indeed alter the default lease settings, there is a key difference. When you make use of a server-side `*.config` file, the lease settings affect *all* objects hosted by the server process. In contrast, when you override select members of the `MarshalByRefObject` type, you are able to change lease settings on an object-by-object basis.

To illustrate changing the default lease settings via a remoting `*.config` file, assume you have updated the server-side XML data with the following additional `<lifetime>` element:

```

<configuration>
  <system.runtime.remoting>
    <application>
      <lifetime leaseTime = "15M" renewOnCallTime = "5M"/>
      <service>
        <activated type = "CarGeneralAsm.CarProvider, CarGeneralAsm"/>
      </service>
      <channels>
        <channel ref="tcp" port="32469" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

Notice how the `leaseTime` and `renewOnCallTime` properties have been marked with the `M` suffix, which as you might guess stands for the number of minutes to set for each lease-centric unit of time. If you wish, your `<lifetime>` element may also suffix the numerical values with `MS` (milliseconds), `S` (seconds), `H` (hours), or even `D` (days).

Now recall that when you update the server's `*.config` file, you have effectively changed the leasing characteristics for each CAO/WKO-singleton object hosted by the server. As an alternative, you may choose to programmatically override the `InitializeLifetime()` method in a specific remote type:

```
public class CarProvider : MarshalByRefObject
{
    public override object InitializeLifetimeService()
    {
        // Obtain the current lease info.
        ILease itfLeaseInfo =
            (ILease) base.InitializeLifetimeService();
        // Adjust settings.
        itfLeaseInfo.InitialLeaseTime = TimeSpan.FromMinutes(50);
        itfLeaseInfo.RenewOnCallTime = TimeSpan.FromMinutes(10);
        return itfLeaseInfo;
    }
    ...
}
```

Here, the `CarProvider` has altered its `InitialLeaseTime` value to 50 minutes and its `RenewOnCallTime` value to 10. Again, the benefit of overriding `InitializeLifetimeServices()` is the fact that you can configure each remote type individually.

Finally, on an odd note, if you wish to disable lease-based lifetime for a given CAO/WKO-singleton object type, you may override `InitializeLifetimeServices()` and simply return null. If you do so, you have basically configured an MBR type that will *never* die as long as the hosting server application is alive and kicking.

Server-Side Lease Adjustment

As you have just seen, when an MBR type overrides `InitializeLifetimeServices()`, it is able to change its default leasing behavior at the time of activation. However, for the sake of argument, what if a remote type desires to change its current lease *after* its activation cycle? For example, assume the `CarProvider` has a new method whose implementation requires a lengthy operation (such as connecting to a remote database and reading a large set of records). Before beginning the task, you may programmatically adjust your lease such that if you have less than one minute, you renew the lease time to ten minutes. To do so, you can make use of the inherited `MarshalByRefObject.GetLifetimeService()` and `ILease.Renew()` methods as follows:

```
// Server-side lease adjustment.
// Assume this new method is of the CarProvider type.
public void DoLengthyOperation()
{
    ILease itfLeaseInfo = (ILease)this.GetLifetimeService();
    if(itfLeaseInfo.CurrentLeaseTime.TotalMinutes < 1.0)
        itfLeaseInfo.Renew(TimeSpan.FromMinutes(10));

    // Do lengthy task...
}
```

Client-Side Lease Adjustment

On an additional `ILease`-related note, it is possible for the client's application domain to adjust the current lease properties for a CAO/WKO-singleton type it is communicating with across the wire. To do so, the client makes use of the static `RemotingServices.GetLifetimeService()` method. As a parameter to this member, the client passes in the reference to the remote type as follows:

```
// Client-side lease adjustment.
CarProvider cp = new CarProvider(cars);
ILease itfLeaseInfo = (ILease)RemotingServices.GetLifetimeService(cp);
if(itfLeaseInfo.CurrentLeaseTime.TotalMinutes < 10.0)
    itfLeaseInfo.Renew(TimeSpan.FromMinutes(1000));
```

This approach can be helpful if the client's application domain is about to enter a lengthy process on the same thread of execution that is using the remote type. For example, if a single-threaded application is about to print out a 100-page document, the chances are quite good that a remote CAO/WKO-singleton type may time out during the process. The other (more elegant) solution, of course, is to spawn a secondary thread of execution, but I think you get the general idea.

Server-Side (and Client-Side) Lease Sponsorship

The final topic regarding the lease-based lifetime of a CAO/WKO-singleton object to consider is the notion of *lease sponsorship*. As you have just seen, every CAO/WKO-singleton entity has a default lease, which may be altered in a number of ways on both the server side as well as the client side. Now, regardless of the type's lease configuration, eventually an MBR object's time will be up. At this point, the runtime will garbage-collect the entity . . . well, almost.

The truth of the matter is that before an expired type is truly marked for garbage collection, the runtime will check to see if the MBR object in question has any registered lease sponsors. Simply put, a sponsor is a type that implements the `ISponsor` interface, which is defined as follows:

```
public interface System.Runtime.Remoting.Lifetime.ISponsor
{
    TimeSpan Renewal(ILease lease);
}
```

If the runtime detects that an MBR object has a sponsor, it will *not* garbage-collect the type, but rather call the `Renewal()` method of the sponsor object to (once again) add time to the current lease. On the other hand, if the MBR has no sponsor, the object's time is truly up.

Assuming that you have created a custom class that implements `ISponsor`, and thus implements `Renewal()` to return a specific unit of time (via the `TimeSpan` type), the next question is how exactly to associate the type to a given remote object. Again, this operation may be performed by either the server's application domain or the client's application domain.

To do so, the interested party obtains an `ILease` reference (via the inherited `GetLifetimeService()` method on the server or using the static `RemotingServices.GetLifetimeService()` method on the client) and calls `Register()`:

```
// Server-side sponsor registration.
CarSponsor mySponsor = new CarSponsor();
ILease itfLeaseInfo = (ILease)this.GetLifetimeService();
itfLeaseInfo.Register(mySponsor);

// Client-side sponsor registration.
CarSponsor mySponsor = new CarSponsor();
CarProvider cp = new CarProvider(cars);
ILease itfLeaseInfo = (ILease)RemotingServices.GetLifetimeService(cp);
itfLeaseInfo.Register(mySponsor);
```

In either case, if a client or server wishes to revoke sponsorship, it may do so using the `IRelease.Unregister()` method, for example:

```
// Remove the sponsor for a given object.  
itfLeaseInfo.Unregister(mySponsor);
```

Note Client-side sponsored objects, in addition to implementing `ISponsor`, must also derive from `MarshalByRefObject`, given that the client must pass the sponsor to the remote application domain!

So, as you can see, the lifetime management of stateful MBR types is a bit more complex than a standard garbage collection. On the plus side, you have a *ton* of control regarding when a remote type is destined to meet its maker. However, as you may have gathered, there is the chance that a remote type may be removed from memory without the client's knowledge. Should a client attempt to invoke members on a type that has already been removed from memory, the runtime will throw a `System.Runtime.Remoting.RemotingException`, at which point the client may create a brand-new instance of the remote type or simply take an alternative course of action.

Source Code The `CAOCarGeneralAsmLease`, `CAOCarProviderServerLease`, and `CAOCarProviderClientLease` projects are located under the Chapter 18 subdirectory.

Alternative Hosts for Remote Objects

Over the course of this chapter, you have constructed numerous console-based server hosts, which provide access to some set of remote objects. If you have a background in the classic Distributed Component Object Model (DCOM), this step may have seemed a bit odd. In the world of DCOM, it was not unusual to build a single server-side COM server that contained the remote objects and was also in charge of receiving incoming requests from some remote client. This single *.exe DCOM application would quietly load in the background without presenting a looming command window.

When you are building a .NET server assembly, the chances are quite good that the remote machine does not need to display any sort of UI. Rather, all you really wish to do is build a server-side entity that opens the correct channel(s) and registers the remote object(s) for client-side access. Moreover, when you build a simple console host, you (or someone) is required to manually run the server-side *.exe assembly, due to the fact that .NET remoting will not automatically run a server-side *.exe when called by a remote client.

Given these two issues, the question then becomes, how can you build an invisible listener that loads automatically? .NET programmers have two major choices at their disposal when they wish to build a transparent host for various remote objects:

- Build a .NET Windows service application to host the remote objects.
- Allow IIS to host the remote objects.

Hosting Remote Objects Using a Windows Service

Perhaps the ideal host for remote objects is a Windows service, given that it

- Can be configured to load automatically on system startup
- Runs as an invisible background process
- Can be run under specific user accounts

As luck would have it, building a custom Windows service using the .NET platform is extremely simple when contrasted to the raw Win32 API. To illustrate, let's create a Windows Service project named *CarWinService* (see Figure 18-7) that will be in charge of hosting the remote types contained within the *CarGeneralAsm.dll*.

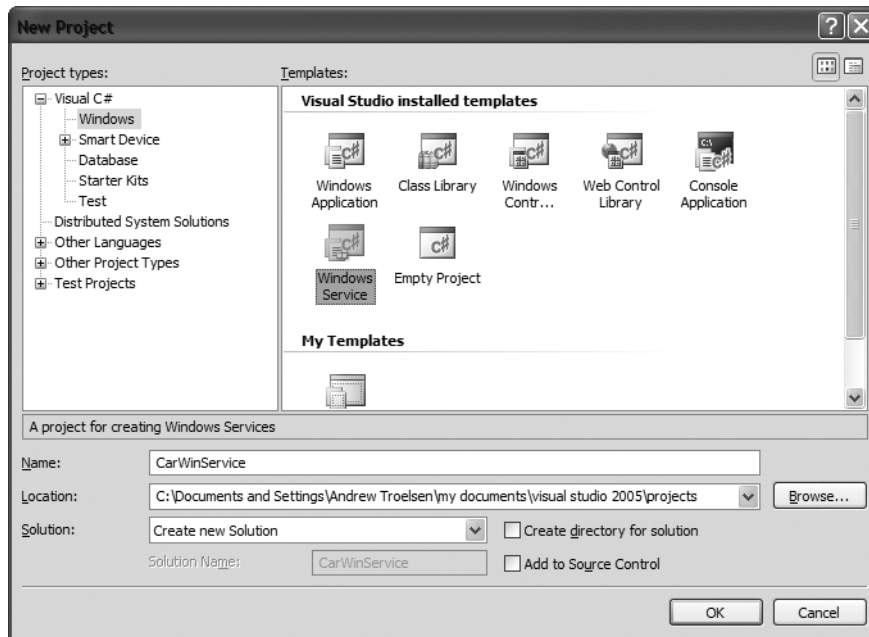


Figure 18-7. *Creating a new Windows Service project workspace*

Visual Studio 2005 responds by generating a partial class (named *Service1* by default), which derives from *System.ServiceProcess.ServiceBase*, and another class (*Program*), which implements the service's *Main()* method. Given that *Service1* is a rather nondescript name for your custom service, the first order of business is to change the values of the (*Name*) and *ServiceName* properties to *CarService* using the IDE's Properties window. The distinction between these two settings is that the (*Name*) value is used to define the name used to refer to your type in the code base, while the *ServiceName* property marks the name to display to Windows service-centric configuration tools.

Before moving on, be sure you set a reference to the *CarGeneralAsm.dll* and *System.Remoting.dll* assemblies, and specify the following additional using directives to the file containing the *CarService* class definition:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels;
using System.Diagnostics;
```

Implementing the *Main()* Method

The *Main()* method of the *Program* class is in charge of running each service defined in the project by passing an array of *ServiceBase* types into the static *Service.Run()* method. Given that you have renamed your custom service from *Service1* to *CarService*, you should find the following class definition (comments deleted for clarity):

```
static class Program
{
    static void Main()
    {
        ServiceBase[] ServicesToRun;
        ServicesToRun = new ServiceBase[] { new CarService() };
        ServiceBase.Run(ServicesToRun);
    }
}
```

Implementing CarService.OnStart()

You can likely already assume what sort of logic should happen when your custom service is started on a given machine. Recall that the role of CarService is to perform the same tasks as your custom console-based service. Thus, if you wish to register CarService as a WKO-singleton type that is available via HTTP, you could add the following code to the OnStart() method (as you would hope, you may make use of the RemotingConfiguration type to load up a server-side remoting *.config file, rather than hard-coding your implementation, when hosting remote objects using a Windows service):

```
protected override void OnStart(string[] args)
{
    // Create a new HttpChannel.
    HttpChannel c = new HttpChannel(32469);
    ChannelServices.RegisterChannel(c);
    // Register as single call WKO.
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(CarGeneralAsm.CarProvider),
        "CarProvider.soap",
        WellKnownObjectMode.SingleCall);

    // Log successful startup.
    EventLog.WriteEntry("CarWinService",
        "CarWinService started successfully!",
        EventLogEntryType.Information);
}
```

Note that once the type has been registered, you log a custom message to the Windows event log (via the System.Diagnostics.EventLog type) to document that the host machine successfully started your service.

Implementing OnStop()

Technically speaking, the CarService does not demand any sort of shutdown logic. For illustrative purposes, let's post another event to the EventLog to log the termination of the custom Windows service:

```
protected override void OnStop()
{
    EventLog.WriteEntry("CarWinService",
        "CarWinService stopped",
        EventLogEntryType.Information);
}
```

Now that the service is complete, the next task is to install this service on the remote machine.

Adding a Service Installer

Before you can install your service on a given machine, you need to add an additional type into your current CarWinService project. Specifically, any Windows service (written using .NET or the Win32 API) requires a number of registry entries to be made to allow the OS to interact with the service itself. Rather than making these entries manually, you can simply add an Installer type to a Windows service project, which will configure your ServiceBase-derived type correctly when installed on the target machine.

To add an installer for the CarService, open the design-time service editor (by double-clicking the CarService.cs file from Solution Explorer), right-click anywhere within the designer, and select Add Installer (see Figure 18-8).

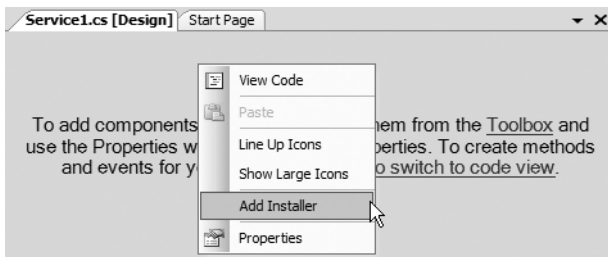


Figure 18-8. Including an installer for the custom Windows service

This selection will add a new component that derives from the System.Configuration.Install.Installer base class. On your designer will be two components. The serviceInstaller1 type represents a specific service installer for a specific service in your project. If you select this icon and view the Properties window, you will find that the ServiceName property has been set to the CarService class type.

The second component (serviceProcessInstaller1) allows you to establish the identity under which the installed service will execute. By default, the Account property is set to User. Using the Properties window of Visual Studio 2005, change this value to **LocalService** (see Figure 18-9).

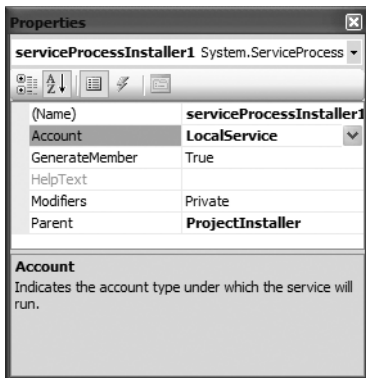


Figure 18-9. Establishing the identity of the CarService

That's it! Now compile your project.

Installing the CarWinService

Installing CarService.exe on a given machine (local or remote) requires two steps:

1. Move the compiled service assembly (and any necessary external assemblies; CarGeneralAsm.dll in this example) to the remote machine.
2. Run the installutil.exe command-line tool, specifying your service as an argument.

Assuming step 1 is complete, open a Visual Studio 2005 command window, navigate to the location of the CarWinService.exe assembly, and issue the following command (note that this same tool can be used to uninstall a service as well):

```
installutil carwinservice.exe
```

Once this Windows service has been properly installed, you are now able to start and configure it using the Services applet, which is located under the Administrative Tools folder of your system's Control Panel. Once you have located your CarService (see Figure 18-10), click the Start link to load and run the binary.

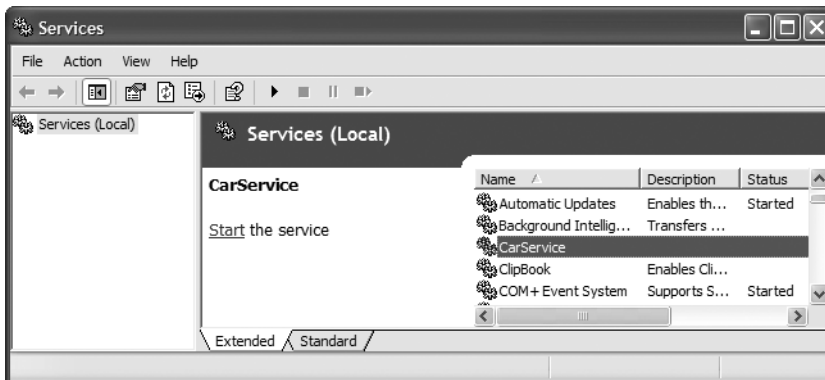


Figure 18-10. *The Windows Services applet*

Source Code The CarWinService project is located under the Chapter 18 subdirectory.

Hosting Remote Objects Using IIS

Hosting a remote assembly under IIS is even simpler than building a Windows service, as IIS is preprogrammed to allow incoming HTTP requests via port 80. Now, given the fact that IIS is a *web* server, it should stand to reason that IIS is only able to host remote objects using the `HttpChannel` type (unlike a Windows service, which can also leverage the `TcpChannel` type). Assuming this is not perceived as a limitation, follow these steps to leverage the remoting support of IIS:

1. On your hard drive, create a new folder to hold your `CarGeneralAsm.dll`. Within this folder, create a subdirectory named `\Bin`. Now, copy the `CarGeneralAsm.dll` to this subdirectory (e.g., `C:\IISCarService\Bin`).
2. Open the Internet Information Services applet on the host machine (located under the Administrative Tools folder in your system's Control Panel).
3. Right-click the Default Web Site node and select **New ► Virtual Directory**.
4. Create a virtual directory that maps to the root folder you just created (`C:\IISCarService`). The remaining default settings presented by the New Virtual Directory Wizard are fine.
5. Finally, create a new configuration file named `web.config` to control how this virtual directory should register the remote type (see the following code). Make sure this file is saved under the root folder (in this example, `C:\IISCarService`).

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
          type="CarGeneralAsm.CarProvider, CarGeneralAsm"
          objectUri="carprovider.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

Now that your `CarGeneralAsm.dll` has been configured to be reachable via HTTP requests under IIS, you can update your client-side `*.config` file as follows (using the name of your IIS host, of course):

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client displayName = "CarClient">
        <wellknown
          type="CarGeneralAsm.CarProvider, CarGeneralAsm"
          url="http://NameTheRemoteIISHost/IISCarHost/carprovider.soap"/>
        </client>
      </application>
    </system.runtime.remoting>
  </configuration>
```

At this point, you are able to run your client application as before.

Asynchronous Remoting

To wrap things up, let's examine how to invoke members of a remote type asynchronously. In Chapter 14, you were first introduced to the topic of asynchronous method invocations using delegate types. As you would expect, if a client assembly wishes to call a remote object asynchronously,

the first step is to define a custom delegate to represent the remote method in question. At this point, the caller can make use of any of the techniques seen in Chapter 14 to invoke and receive the method return value.

By way of a simple illustration, create a new console application (AsyncWKOCarProviderClient) and set a reference to the first iteration of the CarGeneralAsm.dll assembly. Now, update the Program class as so:

```
class Program
{
    // The delegate for the GetAllAutos() method.
    internal delegate List<JamesBondCar> GetAllAutosDelegate();

    static void Main(string[] args)
    {
        Console.WriteLine("Client started! Hit enter to end");
        RemotingConfiguration.Configure
            ("AsyncWKOCarProviderClient.exe.config");
        // Make the car provider.
        CarProvider cp = new CarProvider();

        // Make the delegate.
        GetAllAutosDelegate getCarsDel =
            new GetAllAutosDelegate(cp.GetAllAutos);
        // Call GetAllAutos() asynchronously.
        IAsyncResult ar = getCarsDel.BeginInvoke(null, null);

        // Simulate client-side activity.
        while(!ar.IsCompleted)
        { Console.WriteLine("Client working..."); }

        // All done! Get return value from delegate.
        List<JamesBondCar> allJBCs = getCarsDel.EndInvoke(ar);

        // Use all cars in List.
        foreach(JamesBondCar j in allJBCs)
            UseCar(j);
        Console.ReadLine();
    }
}
```

Notice how the client application first declares a delegate that matches the signature of the GetAllAutos() method of the remote CarProvider type. When the delegate is created, you pass in the name of the method to call (GetAllAutos), as always. Next, you trigger the BeginInvoke() method, cache the resulting IAsyncResult interface, and simulate some work on the client side (recall that the IAsyncResult.IsCompleted property allows you to monitor if the associated method has completed processing). Finally, once the client's work has completed, you obtain the List<> returned from the CarProvider.GetAllAutos() method by invoking the EndInvoke() member, and pass each JamesBondCar into a static helper function named UseCar():

```
public static void UseCar(JamesBondCar j)
{
    Console.WriteLine("Can car fly? {0}", j.isFlightWorthy);
    Console.WriteLine("Can car swim? {0}", j.isSeaWorthy);
}
```

Again, the beauty of the .NET delegate type is the fact that the logic used to invoke remote methods asynchronously is identical to the process of local method invocations.

Source Code The AsyncWKOCarProviderClient project is located under the Chapter 18 subdirectory.

The Role of the [OneWay] Attribute

Imagine that your `CarProvider` has a new method named `AddCar()`, which takes a `JamesBondCar` input parameter and returns nothing. The key point is that it returns *nothing*. As you might assume given the name of the `System.Runtime.Remoting.Messaging.OneWayAttribute` class, the .NET remoting layer passes the call to the remote one-way method, but does *not* bother to set up the infrastructure used to return a given value (hence the name *one-way*). Here is the update:

```
// Home of the [OneWay] attribute.
using System.Runtime.Remoting.Messaging;
...
namespace CarGeneralAsm
{
    public class CarProvider : MarshalByRefObject
    {
        ...
        // The client can 'fire and forget' when calling this method.
        [OneWay]
        public void AddCar(JamesBondCar newJBC)
        { theJBCars.Add(newJBC); }
    }
}
```

Callers would invoke this method directly as always:

```
// Make the car provider.
CarProvider cp = new CarProvider();
// Add a new car.
cp.AddCar(new JamesBondCar("Zippy", 200, false, false));
```

From the client's point of view, the call to `AddCar()` is completely asynchronous, as the CLR will ensure that a background thread is used to remotely trigger the method. Given that `AddCar()` has been decorated with the `[OneWay]` attribute, the client is unable to obtain any return value from the call. Because `AddCar()` returns `void`, this is not an issue.

In addition to this restriction, also be aware that if you have a `[OneWay]` method that defines output or reference parameters (via the `out` or `ref` keyword), the caller will *not* be able to obtain the callee's modification(s). Furthermore, if the `[OneWay]` method happens to throw an exception (of any type), the caller is completely oblivious of this fact. In a nutshell, remote objects can mark select methods as `[OneWay]` to allow the caller to employ a fire-and-forget mentality.

Summary

In this chapter, you examined how to configure distinct .NET assemblies to share types between application boundaries. As you have seen, a remote object may be configured as an MBV or MBR type. This choice ultimately controls how a remote type is realized in the client's application domain (a copy or transparent proxy).

If you have configured a type to function as an MBR entity, you are suddenly faced with a number of related choices (WKO versus CAO, single call versus singleton, and so forth), each of which was addressed during this chapter. As well, you examined the process of tracking the lifetime of a remote object via the use of leases and lease sponsorship. Finally, you revisited of the role of the .NET delegate type to understand how to asynchronously invoke a remote method (which, as luck would have it, is identical to the process of asynchronously invoking a local type).