

54

.NET Remoting

WHAT'S IN THIS CHAPTER?

- An overview of .NET Remoting
- Contexts, which are used to group objects with similar execution requirements
- Implementing a simple remote object, client, and server
- The .NET Remoting architecture
- .NET Remoting configuration files
- Hosting .NET Remoting objects in ASP.NET
- Using Soapsuds to access the metadata of remote objects
- Calling .NET Remoting methods asynchronously
- Calling methods in the client with the help of events
- Using the `CallContext` to automatically pass data to the server

This chapter explores .NET Remoting. .NET Remoting can be used for accessing objects in another application domain (for example, on another server). It provides a faster format for communication between .NET applications on both the client and the server side.

In this chapter, you develop .NET Remoting objects, clients, and servers by using the HTTP, TCP, and IPC channels. First, you configure the client and server programmatically before you change the application to use configuration files instead, where only a few .NET Remoting methods are required. You also write small programs to use .NET Remoting asynchronously and for calling event handlers in the client application.

The .NET Remoting classes can be found in the namespace `System.Runtime.Remoting` and its subnamespaces. Many of these classes are in the core assembly `mscorlib`, and some needed only for cross-network communication are available in the assembly `System.Runtime.Remoting`.

WHY USE .NET REMOTING?

.NET Remoting is a technology for communication between different application domains. Using .NET Remoting for communication between application domains can happen inside the same process, between processes on a single system, or between processes on different systems.

Several different technologies can be used for communication with a client and a server application. You can program your application by using sockets, or you can use some helper classes from the `System.Net` namespace that make it easier to deal with protocols, IP addresses, and port numbers (see Chapter 24, “Networking,” for details). Using this technology, you always have to send data across the network. The data you send can be your own custom protocol where the packet is interpreted by the server, so that the server knows what methods should be invoked. You not only have to deal with the data that is sent but also have to create threads yourself.

Using ASP.NET Web services, you can send messages across the network. With ASP.NET Web services you get platform independence. You also get a loose coupling between client and server, which means that it’s easier to deal with versioning issues. See Chapter 55, “Web Services with ASP.NET,” for information on ASP.NET Web services.

With .NET Remoting you always have a tight coupling between client and server, because the same object types are shared. .NET Remoting brings CLR object functionality to methods that are invoked across different application domains.

The functionality of .NET Remoting can be described with the application types and protocols that are supported, and by looking at *CLR Object Remoting*.

CLR Object Remoting is an importing aspect of .NET Remoting. All the language constructs (such as constructors, delegates, interfaces, methods, properties, and fields) can be used with remote objects. .NET Remoting extends the CLR object functionality across the network. It deals with activation, distributed identities, lifetimes, and call contexts. This is a major difference from XML Web services. With XML Web services, the objects are abstracted and the client doesn’t need to know the object types of the server.

Today, the best choice for network communication is explained in Chapter 43, “Windows Communication Foundation.” WCF gives features of both ASP.NET Web services, such as its platform independence, as well as performance and flexibility as is offered by .NET Remoting for .NET to .NET communication. A place where .NET Remoting still has the advantage is in communication between application domains within a process. The MAF technology (`System.AddIn`) that is discussed in Chapter 50, “Managed Add-In Framework,” uses .NET Remoting behind the scenes. And of course there are a lot of existing .NET solutions based on .NET Remoting that wouldn’t take advantage of rewriting it to a newer technology.



Although SOAP is offered by .NET Remoting, don’t assume that this can be used for interoperability between different platforms. The SOAP Document style is not available with .NET Remoting. .NET Remoting is designed for .NET applications on both client and server sides. If you want to have interoperability, use Web services instead.

.NET Remoting is an extremely flexible architecture that can be used in *any application type over any transport*, using *any payload encoding*.

Using SOAP and HTTP is just one way to call remote objects. The transport channel is pluggable and can be replaced. With .NET 4 you get HTTP, TCP, and IPC channels represented by the classes `HttpChannel`, `TcpChannel`, and `IpChannel`, respectively. You can build transport channels to use UDP, IPX, SMTP, a shared memory mechanism, or message queuing — the choice is entirely yours.



The term pluggable is often used with .NET Remoting. Pluggable means that a specific part is designed so that it can be replaced by a custom implementation.

The payload is used to transport the parameters of a method call. This payload encoding can also be replaced. Microsoft delivers SOAP and binary encoding mechanisms. You can use either the SOAP formatter with the HTTP channel or HTTP with the binary formatter. Of course, both of these formatters can also be used with the TCP channel.



Although SOAP is used with .NET Remoting, you should be aware that only the SOAP RPC style is supported, whereas ASP.NET Web services supports both the DOC style (default) and the RPC style. The RPC style is obsolete in the new SOAP versions.

.NET Remoting not only enables you to use server functionality in every .NET application, but you can use this technology anywhere (regardless of whether you are building a console or a Windows application, a Windows service, or a COM+ component). .NET Remoting is also a good technology for peer-to-peer communication.

.NET REMOTING TERMS EXPLAINED

.NET Remoting can be used for accessing objects in another application domain. This technology can always be used whether the two objects live inside a single process, in separate processes, or on separate systems.

Remote assemblies can be configured to work locally in the application domain or as part of a remote application. If the assembly is part of the remote application, the client receives a proxy to talk to instead of the real object. The proxy is a representative of the remote object in the client process used by the client application to call methods. When the client calls a method in the proxy, the proxy sends a message into the channel that is passed on to the remote object.

.NET applications usually work within an application domain. An application domain can be seen as a subprocess within a process. Traditionally, processes were used as an isolation boundary. An application running in one process cannot access and destroy memory in another process. Cross-process communication is needed for applications to communicate with each other. With .NET, the application domain is the new safety boundary inside a process, because the MSIL code is type-safe and verifiable. As discussed in Chapter 18, different applications can run inside the same process but within different application domains. Objects inside the same application domain can interact directly. A proxy is needed to access objects in a different application domain.

The following list provides an overview of the key elements of this architecture:

- A *remote object* is an object that's running on the server. The client doesn't call methods on this object directly, but uses a proxy instead. With .NET it's easy to distinguish remote objects from local objects: any class that's derived from `MarshalByRefObject` never leaves its application domain. The client can call methods of the remote object via a proxy.
- A *channel* is used for communication between the client and the server. There are client and server parts of the channel. .NET Framework 4 offers channel types that communicate via TCP, HTTP, or IPC. You can also create a custom channel that communicates by using a different protocol.
- *Messages* are sent into the channel; they are created for communication between the client and the server. These messages hold the information about the remote object, the method name called, and all the arguments.
- The *formatter* defines how messages are transferred into the channel. .NET 4 has SOAP and binary formatters. The SOAP formatter can be used to communicate with Web services that are not based on the .NET Framework. Binary formatters are much faster and can be used efficiently in an intranet environment. Of course, you also have the option to create a custom formatter.
- A *formatter provider* is used to associate a formatter with a channel. By creating a channel, you can specify the formatter provider to be used, and this in turn defines the formatter that is used to transfer the data into the channel.
- The client calls methods on a proxy instead of the remote object. Two types of proxies exist: the *transparent proxy* and the *real proxy*. For the client, the transparent proxy looks like the remote object. On the transparent proxy, the client can call the methods implemented by the remote objects. In turn, the transparent proxy calls the `Invoke()` method on the real proxy. The `Invoke()` method uses the message sink to pass the message to the channel.

- A *message sink*, or *sink* for short, is an interceptor object. Interceptors are used on both the client and the server. A sink is associated with the channel. The real proxy uses the message sink to pass the message into the channel, so the sink can do some interception before the message goes into the channel. Depending on where the sink is used, it is known as an envoy sink, a server-context sink, an object-context sink, and so on.
- The client can use an *activator* to create a remote object on the server or to get a proxy of a server-activated object.
- `RemotingConfiguration` is a utility class to configure remote servers and clients. This class can be used either to read configuration files or to configure remote objects dynamically.
- `ChannelServices` is a utility class used to register channels and then to dispatch messages to them.

Figure 54-1 shows a conceptual picture of how these pieces fit together.

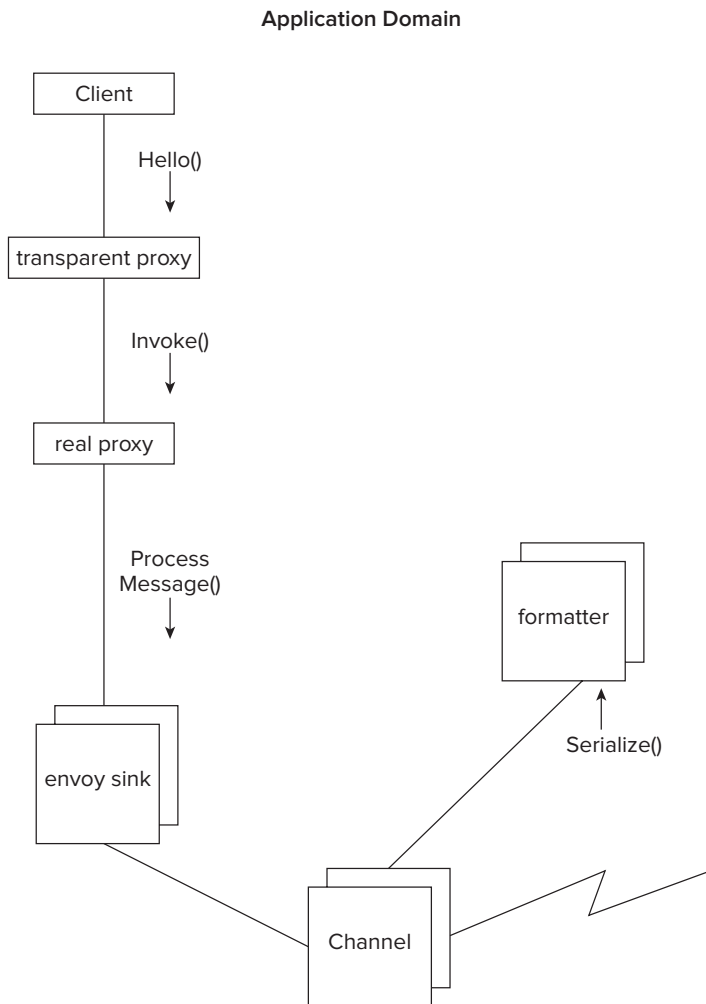


FIGURE 54-1

Client-Side Communication

When the client calls methods on a remote object, it actually calls them on a transparent proxy instead of the real object. The transparent proxy looks like the real object — it implements the public methods of the real object. The transparent proxy knows about the public methods of the real object by using the reflection mechanism to read the metadata from the assembly.

In turn, the transparent proxy calls the real proxy. The real proxy is responsible for sending the message to the channel. This proxy is pluggable: it can be replaced with a custom implementation. Such a custom implementation can be used to write a log, to use another way to find a channel, and so on. The default implementation of the real proxy locates the collection (or chain) of envoy sinks and passes the message to the first envoy sink. An envoy sink can intercept and change the message. Examples of such sinks are debugging sinks, security sinks, and synchronization sinks.

The last envoy sink sends the message into the channel. The formatter defines how the messages are sent over the wire. As previously stated, SOAP and binary formatters are available with .NET Framework 4. The formatter, however, is also pluggable. The channel is responsible for either connecting to a listening socket on the server or sending the formatted data. With a custom channel you can do something different; you just have to implement the code and to do what's necessary to transfer the data to the other side.

Server-Side Communication

Let's continue with the server side as shown in Figure 54-2:

- The channel receives the formatted messages from the client and uses the formatter to unmarshal the SOAP or binary data into messages. Then the channel calls server-context sinks.
- The server-context sinks are a chain of sinks, where the last sink in the chain continues the call to the chain of object-context sinks.
- The last object-context sink then calls the method in the remote object.

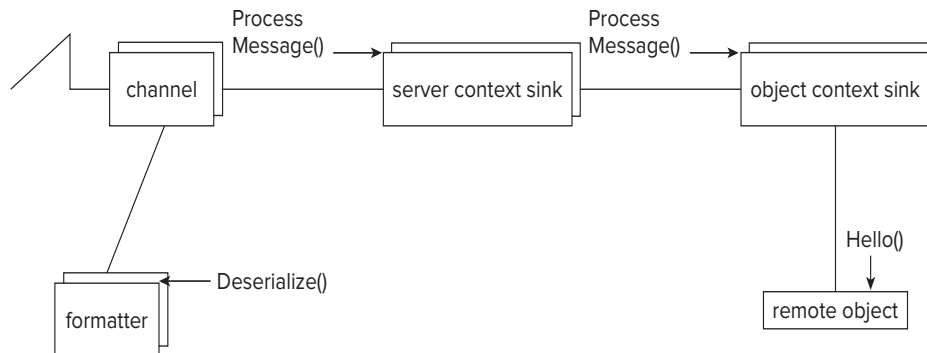


FIGURE 54-2

Note that the object-context sinks are confined to the object context, and the server-context sinks are confined to the server context. A single server-context sink can be used to access a number of object sinks.

A logging sink is one example where a sink can be useful. With a logging sink you can log messages sent and received. Examining if the server is available and dynamically changing to a different server can be done with a sink on the client system.



.NET Remoting is extremely customizable: you can replace the real proxy, add sink objects, or replace the formatter and channel. Of course, you can also use what's already provided.

Going through these layers, you may be wondering about the overhead, but there's not much overhead left if nothing is happening in there. If you add your own functionality, the overhead will depend on that.

CONTEXTS

Before you look at using .NET Remoting to build servers and clients that communicate across a network, this section turns to the cases where a channel is needed inside an application domain: calling objects across contexts.

If you've previously written COM+ components, you already know about COM+ contexts. Contexts in .NET are very similar. A context is a boundary containing a collection of objects. Likewise, with a COM+ context, the objects in such a collection require the same usage rules that are defined by the context attributes.

As you already know, a single process can have multiple application domains. An application domain is something like a subprocess with security boundaries. Application domains are discussed in Chapter 18.

An application domain can have different contexts. A context is used to group objects with similar execution requirements. Contexts are composed from a set of properties and are used for interception: when a *context-bound object* is accessed by a different context, an *interceptor* can do some work before the call reaches the object. This can be used for thread synchronization, transactions, and security management, for example.

A class derived from `MarshalByRefObject` is bound to the application domain. Outside the application domain a proxy is needed to access the object. A class derived from `ContextBoundObject`, which itself derives from `MarshalByRefObject`, is bound to a context. Outside the context, a proxy is needed to access the object.

Context-bound objects can have *context attributes*. A context-bound object without context attributes is created in the context of the creator. A context-bound object with context attributes is created in a new context or in the creator's context if the attributes are compatible.

To further understand contexts, you must be familiar with these terms:

- Creating an application domain creates the *default context* in this application domain. If a new object is instantiated that needs different context properties, a new context is created.
- *Context attributes* can be assigned to classes derived from `ContextBoundObject`. You can create a custom attribute class by implementing the interface `IContextAttribute`. The .NET Framework has one context attribute class in the namespace `System.Runtime.Remoting.Contexts: SynchronizationAttribute`.
- Context attributes define *context properties* that are needed for an object. A context property class implements the interface `IContextProperty`. Active properties contribute message sinks to the call chain. The class `ContextAttribute` implements both `IContextProperty` and `IContextAttribute`, and can be used as a base class for custom attributes.
- A *message sink* is an interceptor for a method call. With a message sink, method calls can be intercepted. Properties can contribute to message sinks.

Activation

A new context is created if an instance of a class that's created needs a context different from the calling context. The attribute classes that are associated with the target class are asked if all the properties of the current context are acceptable. If any of these properties are unacceptable, the runtime asks for all property classes associated with the attribute class and creates a new context. The runtime then asks the property classes for the sinks they want to install. A property class can implement one of the `IContributeXXSink` interfaces to contribute sink objects. Several of these interfaces are available to go with the variety of sinks.

Attributes and Properties

The properties of a context are defined with context attributes. A context attribute class is primarily an attribute — you can find out more about attributes in Chapter 14, “Reflection.” Context attribute classes must implement the interface `IContextAttribute`. A custom context attribute class can derive from the class `ContextAttribute`, because this class already has a default implementation of this interface.

The .NET Framework includes two context attribute classes: `System.Runtime.Remoting.Contexts.SynchronizationAttribute` and `System.Runtime.Remoting.Activation.UrlAttribute`. The `Synchronization` attribute defines synchronization requirements; it specifies the synchronization property that is needed by the object. With this attribute you can specify that multiple threads cannot access the object concurrently, but the thread accessing the object can change.

With the constructor of this attribute, you can set one of four values:

- `NOT_SUPPORTED` defines that the class should not be instantiated in a context where the synchronization is set.
- `REQUIRED` specifies that a synchronization context is required.
- `REQUIRES_NEW` always creates a new context.
- `SUPPORTED` means that it doesn't matter what context you get; the object can live in it.

Communication between Contexts

How does the communication between contexts happen? The client uses a proxy instead of the real object. The proxy creates a message that is transferred to a channel, and sinks can intercept. Does this sound familiar? It ought to. The same mechanism is used for communication across different application domains or different systems. A TCP or HTTP channel is not required for the communication across contexts, but a channel is used here too. `CrossContextChannel` can use the same virtual memory in both the client and server sides of the channel, and formatters are not required for crossing contexts.

REMOTE OBJECTS, CLIENTS, AND SERVERS

Before stepping into the details of the .NET Remoting architecture, this section looks briefly at a remote object and a very small, simple server and client application that uses this remote object. After that the required steps and options are discussed in more detail.

Figure 54-3 shows the major .NET Remoting classes in the client and server application. The remote object that will be implemented is called `Hello`. `HelloServer` is the main class of the application on the server, and `HelloClient` is used for the client.

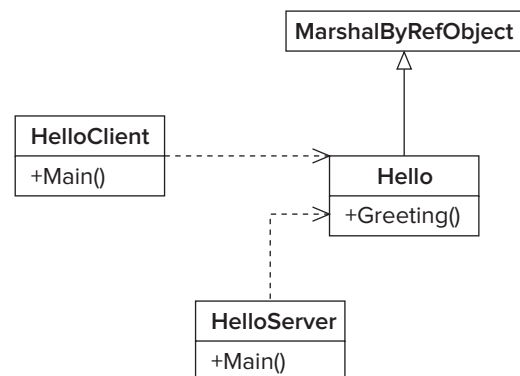


FIGURE 54-3

Remote Objects

Remote objects are required for distributed computing. An object that should be called remotely from a different system must be derived from `System.MarshalByRefObject`. `MarshalByRefObject` objects are *confined to the application domain* in which they were created. This means that they are never passed across application domains; instead, a proxy object is used to access the remote object from another application domain. The other application domain can live inside the same process, in another process, or on another system.

A remote object has *distributed identity*. Because of this, a reference to the object can be passed to other clients, and they will still access the same object. The proxy knows about the identity of the remote object.

The `MarshalByRefObject` class has, in addition to the inherited methods from the `Object` class, methods to initialize and to get the lifetime services. The lifetime services define how long the remote object lives. Lifetime services and leasing features are dealt with later in this chapter.

To see .NET Remoting in action, create a Class Library for the remote object. The class `Hello` derives from `System.MarshalByRefObject`. In the constructor a message is written to the console that provides information about the object's lifetime. In addition, add the method `Greeting()` that will be called from the client.

To distinguish easily between the assembly and the class in the following sections, give them different names in the arguments of the method calls used. The name of the assembly is `RemoteHello`, and the class is named `Hello`:



```
using System;

namespace Wrox.ProCSharp.Remoting
{
    public class Hello : System.MarshalByRefObject
    {
        public Hello()
        {
            Console.WriteLine("Constructor called");
        }

        public string Greeting(string name)
        {
            Console.WriteLine("Greeting called");
            return "Hello, " + name;
        }
    }
}
```

code snippet RemoteHello/RemoteHello/Hello.cs

A Simple Server Application

For the server, create a new C# console application called `HelloServer`. To use the `TcpServerChannel` class, you have to reference the `System.Runtime.Remoting` assembly. It's also required that you reference the `RemoteHello` assembly that was created earlier in the "Remote Objects" section.

In the `Main()` method, an object of type `System.Runtime.Remoting.Channels.Tcp.TcpServerChannel` is created with the port number 8086. This channel is registered with the `System.Runtime.Remoting.Channels.ChannelServices` class to make it available for remote objects. The remote object type is registered by calling the method `RemotingConfiguration.RegisterWellKnownServiceType()`.

In the example, the URI type of the remote object class used by the client and a mode are specified. The mode `WellKnownObjectMode.SingleCall` means that a new instance is created for every method call; in the sample application no state is held in the remote object.



.NET Remoting allows creating stateless and stateful remote objects. In the first example, well-known single-call objects that don't hold state are used. The other object type is called client-activated. Client-activated objects hold state. Later in this chapter, when looking at the object activation sequence, you learn more details about these differences and how these object types can be used.

After registration of the remote object, it is necessary to keep the server running until a key is pressed:



Available for
download on
Wrox.com

```
using System;
using System.Collections.Generic;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace Wrox.ProCSharp.Remoting
{
    class Program
    {
        static void Main()
        {
            var channel = new TcpServerChannel(8086);
            ChannelServices.RegisterChannel(channel, true);
            RemotingConfiguration.RegisterWellKnownServiceType(
                typeof(Hello), "Hi", WellKnownObjectMode.SingleCall);
            Console.WriteLine("Press return to exit");
            Console.ReadLine();
        }
    }
}
```

code snippet RemoteHello /HelloServer/Program.cs

A Simple Client Application

The client is again a C# console application: `HelloClient`. With this project, you also have to reference the `System.Runtime.Remoting` assembly so that the `TcpClientChannel` class can be used. In addition, you have to reference the `RemoteHello` assembly. Although the object will be created on the remote server, the assembly is needed on the client for the proxy to read the type information during runtime.

In the client program, create a `TcpClientChannel` object that's registered in `ChannelServices`. For the `TcpChannel`, you can use the default constructor, so a free port is selected. Next, the `Activator` class is used to return a proxy to the remote object. The proxy is of type `System.Runtime.Remoting.Proxies.__TransparentProxy`. This object looks like the real object because it offers the same methods. The transparent proxy uses the real proxy to send messages to the channel:



Available for
download on
Wrox.com

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

namespace Wrox.ProCSharp.Remoting
{
    class Program
    {
```

```

static void Main()
{
    Console.WriteLine("Press return after the server is started");
    Console.ReadLine();

    ChannelServices.RegisterChannel(new TcpClientChannel(), true);
    Hello obj = (Hello)Activator.GetObject(
        typeof(Hello), "tcp://localhost:8086/Hi");

    if (obj == null)
    {
        Console.WriteLine("could not locate server");
        return;
    }
    for (int i=0; i< 5; i++)
    {
        Console.WriteLine(obj.Greeting("Stephanie"));
    }
}
}

```

code snippet RemoteHello/HelloClient/Program.cs



A proxy is an object used by the client application in place of the remote object. Proxies that are used in Chapter 55 have a similar functionality to the proxies in this chapter. The implementation of proxies for Web services and proxies for .NET Remoting is very different.

Now you can start the server and then the client. Within the client console, the text `Hello, Stephanie` appears five times. With your server console window you can see the output shown. With every method call a new instance gets created because the `WellKnownObjectMode.SingleCall` activation mode was selected:

```

Press return to exit
Constructor called
Greeting called
Constructor called
Greeting called
Constructor called
Greeting called
Constructor called
Greeting called
Constructor called
Greeting called

```

.NET REMOTING ARCHITECTURE

Now that you've seen a simple client and server application in action, let's take a look at the .NET Remoting architecture before we dive into the details. Based on the previously created program, you look at the architecture and the mechanisms for extensibility.

This section explores the following topics:

- The functionality of a channel and how a channel can be configured
- Formatters and how they are used
- The utility classes `ChannelServices` and `RemotingConfiguration`

- Different ways to activate remote objects, and how stateless and stateful objects can be used with .NET Remoting
- Functionality of message sinks
- How to pass objects by value and by reference
- Lifetime management of stateful objects with .NET Remoting leasing mechanisms

Channels

A channel is used to communicate between a .NET client and a server. .NET Framework 4 ships with channel classes that communicate using TCP, HTTP, or IPC. You can create custom channels for other protocols.

The HTTP channel uses the HTTP protocol for communication. Because firewalls usually have port 80 open so clients can access web servers, .NET Remoting Web services can listen to port 80 so they can easily be used by these clients.

It's also possible to use the TCP channel on the Internet, but here the firewalls must be configured so that clients can access a specified port that's used by the TCP channel. The TCP channel can be used to communicate more efficiently in an intranet environment compared to the HTTP channel.

The IPC channel is best for communication on a single system across different processes. It uses the Windows interprocess communication mechanism and thus it is faster than the other channels.

Performing a method call on the remote object causes the client channel object to send a message to the remote channel object.

Both the server and the client application must create a channel. This code shows how a `TcpServerChannel` can be created on the server side:

```
using System.Runtime.Remoting.Channels.Tcp;
...
TcpServerChannel channel = new TcpServerChannel(8086);
```

The port on which the TCP socket is listening is specified in the constructor argument. The server channel must specify a well-known port, and the client must use this port when accessing the server. To create a `TcpClientChannel` on the client, however, it isn't necessary to specify a well-known port. The default constructor of `TcpClientChannel` chooses an available port, which is passed to the server at connection time so that the server can return data to the client.

Creating a new channel instance immediately switches the socket to the listening state, which can be verified by typing `netstat -a` at the command line.



A very useful tool for testing to see what data is sent across the network is `tcpTrace`. `tcpTrace` can be downloaded from www.pocketsoap.com/tcptrace.

The HTTP channels can be used similarly to the TCP channels. You can specify the port where the server can create the listening socket.

A server can listen to multiple channels. This code creates and registers HTTP, TCP, and IPC channels:

```
var tcpChannel = new TcpServerChannel(8086);
var httpChannel = new HttpServerChannel(8085);
var ipcChannel = new IpServerChannel("myIPCPort");

// register the channels
ChannelServices.RegisterChannel(tcpChannel, true);
ChannelServices.RegisterChannel(httpChannel, false);
ChannelServices.RegisterChannel(ipcChannel, true);
```

A channel class must implement the `IChannel` interface. The `IChannel` interface has these two properties:

- `ChannelName` is a read-only property that returns the name of the channel. The name of the channel depends on the type; for example, the HTTP channel is named HTTP.
- `ChannelPriority` is a read-only property. More than one channel can be used for communication between a client and a server. The priority defines the order of the channel. On the client, the channel with the higher priority is chosen first to connect to the server. The bigger the priority value, the higher the priority. The default value is 1, but negative values are allowed to create lower priorities.

Additional interfaces are implemented depending on whether the channel is a client channel or a server channel. The server versions of the channels implement the `IChannelReceiver` interface; the client versions implement the `IChannelSender` interface.

The `HttpChannel`, `TcpChannel`, and `IPCChannel` classes can be used for both the client and the server. They implement `IChannelSender` and `IChannelReceiver`. These interfaces derive from `IChannel`.

The client-side `IChannelSender` has, in addition to `IChannel`, a single method called `CreateMessageSink()`, which returns an object that implements `IMessageSink`. The `IMessageSink` interface can be used for putting synchronous as well as asynchronous messages into the channel. With the server-side interface `IChannelReceiver`, the channel can be put into listening mode using `StartListening()`, and stopped again with `StopListening()`. The property `ChannelData` can be used to access the received data.

You can get information about the configuration of the channels using properties of the channel classes.

For both channels, the properties `ChannelName`, `ChannelPriority`, and `ChannelData` are offered.

The `ChannelData` property can be used to get information about the URIs that are stored in the `ChannelDataStore` class. With the `HttpServerChannel` there's also a `Scheme` property. The following code shows a helper method, `ShowChannelProperties()`, in the file that displays this information:

```
static void ShowChannelProperties(IChannelReceiver channel)
{
    Console.WriteLine("Name: {0}", channel.ChannelName);
    Console.WriteLine("Priority: {0}", channel.ChannelPriority);
    if (channel is HttpServerChannel)
    {
        HttpServerChannel httpChannel = channel as HttpServerChannel;
        Console.WriteLine("Scheme: {0}", httpChannel.ChannelScheme);
    }
    if (channel is TcpServerChannel)
    {
        TcpServerChannel tcpChannel = channel as TcpServerChannel;
        Console.WriteLine("Is secured: {0}", tcpChannel.IsSecured);
    }

    ChannelDataStore data = (ChannelDataStore)channel.ChannelData;
    if (data != null)
    {
        foreach (string uri in data.ChannelUris)
        {
            Console.WriteLine("URI: {0}", uri);
        }
    }
    Console.WriteLine();
}
```

Setting Channel Properties

You can set all the properties of a channel in a list using the constructor `HttpServerChannel(IDictionary, IServerChannelSinkProvider)`. The generic `Dictionary` class implements `IDictionary`, so you can set

the Name, Priority, and Port property with the help of this class. To use the Dictionary class, you have to import the System.Collections.Generic namespace.

With the constructor of the class HttpServerChannel, you can pass an object that implements the interface IServerChannelSinkProvider in addition to the IDictionary parameter. In the example, a BinaryServerFormatterSinkProvider is set instead of the SoapServerFormatterSinkProvider, which is the default of the HttpServerChannel. The default implementation of the BinaryServerFormatterSinkProvider class associates a BinaryServerFormatterSink class with the channel that uses a BinaryFormatter object to convert the data for the transfer:

```
var properties = new Dictionary<string, string>();
properties["name"] = "HTTP Channel with a Binary Formatter";
properties["priority"] = "15";
properties["port"] = "8085";
var sinkProvider = new BinaryServerFormatterSinkProvider();
var httpChannel = new HttpServerChannel(properties, sinkProvider);
```

Depending on the channel types, different properties can be specified. Both the TCP and the HTTP channels support the name and priority channel property used in the example. These channels also support other properties such as bindTo, which specifies an IP address for binding that can be used if the computer has multiple IP addresses configured. rejectRemoteRequests is supported by the TCP server channel to allow client connections only from the local computer.

Pluggability of a Channel

A custom channel can be created to send the messages by using a transport protocol other than HTTP, TCP, or IPC, or the existing channels can be extended to offer more functionality:

- The sending part must implement the IChannelSender interface. The most important part is the CreateMessageSink() method, with which the client sends a URL, and here a connection to the server can be instantiated. A message sink must be created, which is then used by the proxy to send messages to the channel.
- The receiving part must implement the IChannelReceiver interface. You have to start the listening in the ChannelData get property. Then you can wait in a separate thread to receive data from the client. After unmarshaling the message, you can use ChannelServices.SyncDispatchMessage() to dispatch the message to the object.

Formatters

The .NET Framework delivers two formatter classes:

- System.Runtime.Serialization.Formatters.Binary.BinaryFormatter
- System.Runtime.Serialization.Formatters.Soap.SoapFormatter

Formatters are associated with channels through formatter sink objects and formatter sink providers.

Both of these formatter classes implement the interface System.Runtime.Remoting.Messaging.IRemotingFormatter, which defines the methods Serialize() and Deserialize() to transfer the data to and from the channel.

The formatter is also pluggable. When you're writing a custom formatter class, an instance must be associated with the channel you want to use. This is done by using a formatter sink and a formatter sink provider. The formatter sink provider, for example, SoapServerFormatterSinkProvider, can be passed as an argument when creating a channel as shown earlier. A formatter sink provider implements the interface IServerChannelSinkProvider for the server and IClientChannelSinkProvider for the client. Both of these interfaces define a CreateSink() method where a formatter sink must be returned. The SoapServerFormatterSinkProvider returns an instance of the class SoapServerFormatterSink.

On the client side, the `SoapClientFormatterSink` class uses the `SyncProcessMessage()` and `AsyncProcessMessage()` methods of the `SoapFormatter` class to serialize the message. The `SoapServerFormatterSink` class deserializes the message, again using the `SoapFormatter` class.

All these sink and provider classes can be extended and replaced with custom implementations.

ChannelServices and RemotingConfiguration

The `ChannelServices` utility class is used to register channels into the .NET Remoting runtime. With this class, you can also access all registered channels. This is extremely useful if configuration files are used to configure the channel, because here the channel is created implicitly, as you see later.

A channel is registered using the static method `ChannelServices.RegisterChannel()`. The first parameter of this method requires the channel. Setting the second parameter to `true` verifies if security is available with the channel. The `HttpChannel` class doesn't have security support, so the checking is set to `false` for this channel.

Here you can see the server code to register the HTTP, TCP, and IPC channels:

```
var tcpChannel = new TcpChannel(8086);
var httpChannel = new HttpChannel(8085);
var ipcChannel = new IpcChannel("myIPCPort");
ChannelServices.RegisterChannel(tcpChannel, true);
ChannelServices.RegisterChannel(httpChannel, false);
ChannelServices.RegisterChannel(ipcChannel, true);
```

The `ChannelServices` utility class can now be used to dispatch synchronous and asynchronous messages, and to unregister specific channels. The `RegisteredChannels` property returns an `IChannel` array of all the channels you registered. You can also use the `GetChannel()` method to get to a specific channel by its name. With the help of `ChannelServices` you can write a custom administration utility that manages your channels. Here is a small example to show how the server channel can be stopped from listening to incoming requests:

```
HttpServerChannel channel =
    (HttpServerChannel)ChannelServices.GetChannel("http");
channel.StopListening(null);
```

The `RemotingConfiguration` class is another .NET Remoting utility class. On the server side it's used to register remote object types for server-activated objects, and to marshal remote objects to a marshaled object reference class `ObjRef`. `ObjRef` is a serializable representation of an object that's sent over the wire. On the client side, `RemotingServices` is used to unmarshal a remote object to create a proxy from the object reference.

Server for Well-Known Objects

Here is the server-side code to register a well-known remote object type to the `RemotingServices`:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(Hello),                // Type
    "Hi",                        // URI
    WellKnownObjectMode.SingleCall); // Mode
```

The first argument of `RegisterWellKnownServiceType()`, `typeof(Hello)`, specifies the type of the remote object. The second argument, "Hi", is the uniform resource identifier of the remote object, used by the client to access the remote object. The last argument is the mode of the remote object. The mode can be a value of the `WellKnownObjectMode` enumeration: `SingleCall` or `Singleton`.

- `SingleCall` means that the object holds no state. With every call to the remote object a new instance is created. A single-call object is created from the server with the `RemotingConfiguration.RegisterWellKnownServiceType()` method and a `WellKnownObjectMode.SingleCall` argument. This is very efficient on the server, because it means that you needn't hold any resources for maybe thousands of clients.
- With a `singleton` the object is shared for all clients of the server; typically, such object types can be used if you want to share some data between all clients. This shouldn't be a problem for read-only data, but with read-write data you have to be aware of locking issues and scalability. A singleton object is created by the server with the `RemotingConfiguration.RegisterWellKnownServiceType()` method and a `WellKnownObjectMode.Singleton` argument. You have to pay attention to the locking of resources held by the singleton object; you have to make sure that data can't be corrupted when clients are accessing the singleton object concurrently, but you also have to check that the locking is done efficiently enough to reach the required scalability.

Server for Client-Activated Objects

If a remote object should hold state for a specific client, you can use client-activated objects. The “Object Activation” section looks at how to call server-activated or client-activated objects on the client side. On the server side, client-activated objects must be registered in a different way from server-activated objects.

Instead of calling `RemotingConfiguration.RegisterWellKnownServiceType()`, you have to call `RemotingConfiguration.RegisterActivatedServiceType()`. With this method, only the type is specified, not the URI. The reason for this is that for client-activated objects, the clients can instantiate different object types with the same URI. The URI for all client-activated objects must be defined using `RemotingConfiguration.ApplicationName`:

```
RemotingConfiguration.ApplicationName = "HelloServer";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Hello));
```

Object Activation

Clients can use and create a remote `Activator` class. You can get a proxy to a server-activated or well-known remote object by using the `GetObject()` method. The `CreateInstance()` method returns a proxy to a client-activated remote object.

Instead of using the `Activator` class, the `new` operator can also be used to activate remote objects. To make this possible, the remote object must also be configured within the client using the `RemotingConfiguration` class.

Application URL

In all activation scenarios, you have to specify a URL to the remote object. This URL is the same one you'd use when browsing with a web browser. The first part specifies the protocol followed by the server name or IP address, the port number, and a Uniform Resource Identifier (URI) that was specified when registering the remote object on the server in this form:

protocol://server:port/URI

In the code samples, three URL examples are used continuously in the code. With the URL, the protocol is specified with `http`, `tcp`, or `ipc`. With the HTTP and TCP channels the server name is `localhost` and the port numbers are `8085` and `8086`. With the IPC channel there's no need to define a hostname, because IPC is possible only on a single system. With all protocols the URI is `Hi`, as follows:

```
http://localhost:8085/Hi
tcp://localhost:8086/Hi
ipc://myIPCPort/Hi
```

Activating Well-Known Objects

In the previous simple client example in the “A Simple Client Application” section, well-known objects have been activated. Here, you take a more detailed look at the activation sequence:

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

Hello obj = (Hello)Activator.GetObject(typeof(Hello),
                                     "tcp://localhost:8086/Hi");
```

`GetObject()` is a static method of the class `System.Activator` that calls `RemotingServices.Connect()` to return a proxy object to the remote object. The first argument of this method specifies the type of the remote object. The proxy implements all public and protected methods and properties, so that the client can call these methods as it would do on the real object. The second argument is the URL to the remote object. Here, the string `tcp://localhost:8086/Hi` is used. `tcp` defines the protocol that is used, the hostname, and the port number is `localhost:8086`, and finally `Hi` is the URI of the object that is specified on the server with the method `RemotingConfiguration.RegisterWellKnownServiceType()`.

Instead of using `Activator.GetObject()`, you can also use `RemotingServices.Connect()` directly:

```
Hello obj = (Hello)RemotingServices.Connect(typeof(Hello),
                                             "tcp://localhost:8086/Hi");
```

If you prefer to use the `new` operator to activate well-known remote objects, the remote object can be registered on the client using `RemotingConfiguration.RegisterWellKnownClientType()`. The arguments are similar: the type of the remote object and the URI. `new` doesn't really create a new remote object, it returns a proxy similar to `Activator.GetObject()` instead. If the remote object is registered with a flag, `WellKnownObjectMode.SingleCall`, the rule always stays the same — the remote object is created with every method call:

```
RemotingConfiguration.RegisterWellKnownClientType(typeof(Hello),
                                                  "tcp://localhost:8086/Hi");
Hello obj = new Hello();
```

Activating Client-Activated Objects

Remote objects can hold state for a client. `Activator.CreateInstance()` creates a client-activated remote object. Using the `Activator.GetObject()` method, the remote object is created on a method call, and is destroyed when the method is finished. The object doesn't hold state on the server. The situation is different with `Activator.CreateInstance()`. With the static `CreateInstance()` method an activation sequence is started to create the remote object. This object lives until the lease time is expired and a garbage collection occurs. The leasing mechanism is discussed later in the “Lifetime Management” section.

Some of the overloaded `Activator.CreateInstance()` methods can only be used to create local objects. To create remote objects a method is needed where it's possible to pass activation attributes. One of these overloaded methods is used in the example. This method accepts two string parameters and an array of objects. The first parameter is the name of the assembly, and the second is the type of the remote object class. With the third parameter it is possible to pass arguments to the constructor of the remote object class. The channel and the object name are specified in the object array with the help of a `UrlAttribute`. To use the `UrlAttribute` class, the namespace `System.Runtime.Remoting.Activation` must be imported:

```
object[] attrs = {new UrlAttribute("tcp://localhost:8086/HelloServer") };
ObjectHandle handle = Activator.CreateInstance(
    "RemoteHello", "Wrox.ProCSharp.Remoting.Hello", attrs);
if (handle == null)
{
    Console.WriteLine("could not locate server");
    return;
}

Hello obj = (Hello)handle.Unwrap();
Console.WriteLine(obj.Greeting("Christian"));
```


Of course, for client-activated objects, it's again possible to use the `new` operator instead of the `Activator` class, but then you have to register the client-activated object using `RemotingConfiguration.RegisterActivatedClientType()`. In the architecture of client-activated objects, the `new` operator not only returns a proxy but also creates the remote object:

```
RemotingConfiguration.RegisterActivatedClientType(typeof(Hello),
                                                "tcp://localhost:8086/HelloServer");
Hello obj = new Hello();
```

Proxy Objects

The `Activator.GetObject()` and `Activator.CreateInstance()` methods return a proxy to the client. Actually, two proxies are used: the transparent proxy and the real proxy. The transparent proxy looks like the remote object — it implements all public methods of the remote object. These methods just call the `Invoke()` method of the `RealProxy`, where a message containing the method to call is passed. The real proxy sends the message to the channel with the help of message sinks.

With `RemotingServices.IsTransparentProxy()`, you can check if your object is really a transparent proxy. You can also get to the real proxy using `RemotingServices.GetRealProxy()`. Using the Visual Studio debugger, it's now easy to see all the properties of the real proxy:

```
ChannelServices.RegisterChannel(new TCPChannel());
Hello obj = (Hello)Activator.GetObject(typeof(Hello),
                                       "tcp://localhost:8086/Hi");

if (obj == null)
{
    Console.WriteLine("could not locate server");
    return;
}
if (RemotingServices.IsTransparentProxy(obj))
{
    Console.WriteLine("Using a transparent proxy");
    RealProxy proxy = RemotingServices.GetRealProxy(obj);

    // proxy.Invoke(message);
}
```

Pluggability of a Proxy

The real proxy can be replaced with a custom proxy. A custom proxy can extend the base class `System.Runtime.Remoting.Proxies.RealProxy`. The type of the remote object is received in the constructor of the custom proxy. Calling the constructor of the `RealProxy` creates a transparent proxy in addition to the real proxy. In the constructor, the registered channels can be accessed with the `ChannelServices` class to create a message sink, `ISharedChannelSink.CreateMessageSink()`. Besides implementing the constructor, a custom channel has to override the `Invoke()` method. In `Invoke()` a message is received that can be analyzed and sent to the message sink.

Messages

The proxy sends a message into the channel. On the server side, a method call can be made after analyzing the message. This section takes a closer look at the messages.

The .NET Framework has some message classes for method calls, responses, return messages, and so on. What all the message classes have in common is that they implement the `IMessage` interface. This interface has a single property: `Properties`. This property represents a dictionary with the `IDictionary` interface, which packages the URI to the object, `MethodName`, `MethodSignature`, `TypeName`, `Args`, and `CallContext`.

Figure 54-4 shows the hierarchy of the message classes and interfaces. The message that is sent to the real proxy is an object of type `MethodCall`. With the interfaces `IMethodCallMessage` and `IMethodMessage`, you can have easier access to the properties of the message than through the `IMessage` interface. Instead of

using the `IDictionary` interface, you have direct access to the method name, the URI, the arguments, and so on. The real proxy returns a `ReturnMessage` to the transparent proxy.

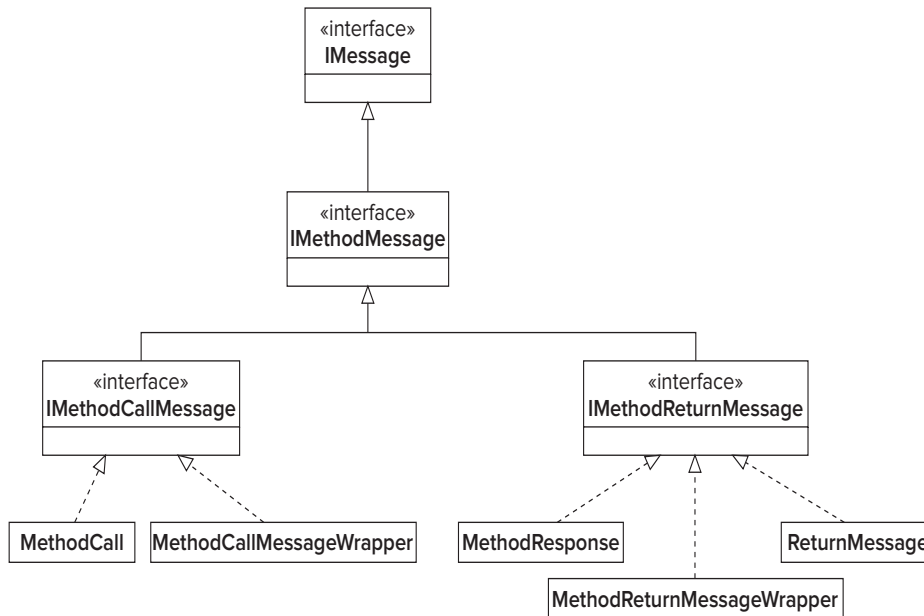


FIGURE 54-4

Message Sinks

The `Activator.GetObject()` method calls `RemotingServices.Connect()` to connect to a well-known object. In the `Connect()` method, an `Unmarshal()` happens where not only the proxy, but also envoy sinks are created. The proxy uses a chain of envoy sinks to pass the message to the channel. All the sinks are interceptors that can change the messages and perform some additional actions such as creating a lock, writing an event, performing security checks, and so on.

All message sinks implement the interface `IMessageSink`. This interface defines one property and two methods:

- The property `NextSink` is used by a sink to get to the next sink and pass the message along.
- For synchronous messages, `SyncProcessMessage()` is invoked by a previous sink or by the remoting infrastructure. It has an `IMessage` parameter to send a message and to return a message.
- For asynchronous messages, `AsyncProcessMessage()` is invoked by a previous sink in the chain, or by the remoting infrastructure. `AsyncProcessMessage()` has two parameters: a message and a message sink that receives the reply.

The following sections look at the three different message sinks available for use.

Envoy Sink

You can get to the chain of envoy sinks using the `IEnvoyInfo` interface. The marshaled object reference `ObjRef` has the `EnvoyInfo` property, which returns the `IEnvoyInfo` interface. The envoy list is created from the server context, so the server can inject functionality into the client. Envoys can collect identity information about the client and pass that information to the server.

Server Context Sink

When the message is received on the server side of the channel, it is passed to the server-context sinks. The last of the server-context sinks routes the message to the object sink chain.

Object Sink

The object sink is associated with a particular object. If the object class defines particular context attributes, context sinks are created for the object.

Passing Objects in Remote Methods

The parameter types of remote method calls aren't limited only to basic data types but can also be classes that you define yourself. For remoting, three types of classes must be differentiated:

Marshal-by-value classes — These classes are serialized through the channel. Classes that should be marshaled must be marked with the `Serializable` attribute. Objects of these classes don't have a remote identity, because the complete object is marshaled through the channel, and the object that is serialized to the client is independent of the server object (or the other way around). Marshal-by-value classes are also called *unbound classes* because they don't have data that depends on the application domain. Serialization is discussed in detail in Chapter 29, "Manipulating Files and the Registry."

Marshal-by-reference classes — These classes have a remote identity. The objects are not passed across the wire, but a proxy is returned instead. A class marshaled by reference must derive from `MarshalByRefObject`. `MarshalByRefObject`s are known as *application domain-bound objects*. A specialized version of `MarshalByRefObject` is `ContextBoundObject`: the abstract class `ContextBoundObject` is derived from `MarshalByRefObject`. If a class is derived from `ContextBoundObject`, a proxy is needed even in the same application domain when context boundaries are crossed. Such objects are called *context-bound objects*, and they are only valid in the creation context.

- **Nonremotable classes** — These classes are not serializable and don't derive from `MarshalByRefObject`. Classes of these types cannot be used as parameters in a remote object's public methods. These classes are bound to the application domain where they are created. Nonremotable classes should be used if the class has a data member that is only valid in the application domain, such as a Win32 file handle.

To see marshaling in action, change the remote object to send an object to the client: the class `MySerialized` will be sent marshal-by-value. In the method a message is written to the console so that you can verify whether the call was made on the client or on the server. In addition, the `Hello` class is extended to return a `MySerialized` instance:



Available for
download on
Wrox.com

```
using System;

namespace Wrox.ProCSharp.Remoting
{
    [Serializable]
    public class MySerialized
    {
        public MySerialized(int val)
        {
            a = val;
        }
        public void Foo()
        {
            Console.WriteLine("MySerialized.Foo called");
        }
        public int A
    }
}
```

```

        {
            get
            {
                Console.WriteLine("MySerialized.A called");
                return a;
            }
            set
            {
                a = value;
            }
        }
        protected int a;
    }

    public class Hello : MarshalByRefObject
    {
        public Hello()
        {
            Console.WriteLine("Constructor called");
        }
        public string Greeting(string name)
        {
            Console.WriteLine("Greeting called");
            return "Hello, " + name;
        }
        public MySerialized GetMySerialized()
        {
            return new MySerialized(4711);
        }
    }
}

```

code snippet PassingObjects/RemoteHello/Hello.cs

The client application also needs to be changed to see the effects when using marshaled-by-value and marshaled-by-reference objects. Invoke the methods `GetMySerialized()` and `GetMyRemote()` to retrieve the new objects. Also make use of the method `RemotingServices.IsTransparentProxy()` to check whether or not the returned object is a proxy:



```

ChannelServices.RegisterChannel(new TcpClientChannel(), false);
Hello obj = (Hello)Activator.GetObject(typeof(Hello),
    "tcp://localhost:8086/Hi");
if (obj == null)
{
    Console.WriteLine("could not locate server");
    return;
}
MySerialized ser = obj.GetMySerialized();
if (!RemotingServices.IsTransparentProxy(ser))
{
    Console.WriteLine("ser is not a transparent proxy");
}
ser.Foo();

```

code snippet PassingObjects /HelloClient/Program.cs

In the client console window, you can see that the `ser` object is called on the client. This object is not a transparent proxy because it's serialized to the client:

Press return after the server is started

```

ser is not a transparent proxy
MySerialized.Foo called

```

Security and Serialized Objects

One important difference with .NET Remoting and ASP.NET Web services is how objects are marshaled. With ASP.NET Web services, only the public fields and properties are transferred across the wire. .NET Remoting uses a different serialization mechanism to serialize all data, including all private data. Malicious clients could use the serialization and deserialization phases to harm the application.

To take this problem into account, two automatic deserialization levels are defined when passing objects across .NET Remoting boundaries: low and full.

By default, low-level deserialization is used. With low-level deserialization it is not possible to pass `ObjRef` objects and objects that implement the `ISponsor` interface. To make this possible, you can change the deserialization level to full. You can do this programmatically by creating a formatter sink provider, and assign the property `TypeFilterLevel`. For the binary formatter, the provider class is `BinaryServerFormatterSinkProvider`, whereas for the SOAP formatter the provider class is `SoapServerFormatterSinkProvider`.

The following code shows how you can create a TCP channel with full serialization support:



```
var serverProvider = new BinaryServerFormatterSinkProvider();
serverProvider.TypeFilterLevel = TypeFilterLevel.Full;

var clientProvider = new BinaryClientFormatterSinkProvider();

var properties = new Dictionary<string, string>();
properties["port"] = 6789;

var channel = new TcpChannel(properties, clientProvider, serverProvider);
```

code snippet PassingObjects/HelloServer/Program.cs

At first, a `BinaryServerFormatterSinkProvider` is created where the property `TypeFilterLevel` is set to `TypeFilterLevel.Full`. The enumeration `TypeFilterLevel` is defined in the namespace `System.Runtime.Serialization.Formatters`, so you have to declare this namespace. For the client side of the channel, a `BinaryClientFormatterSinkProvider` is created. Both the client-side and the server-side formatter sink provider instances are passed to the constructor of the `TcpChannel`, as well as the `IDictionary` properties that define the attributes of the channel.

Directional Attributes

Remote objects are never transferred over the wire, whereas value types and serializable classes are transferred. Sometimes the data should be sent in one direction only. This can be especially important when the data is transferred over the network. For example, if you want to send data in a collection to the server for the server to perform some calculation on this data and return a simple value to the client, it would not be very efficient to send the collection back to the client. With COM it was possible to declare the directional attributes `[in]`, `[out]`, and `[in, out]` to the arguments if the data should be sent to the server, to the client, or in both directions.

C# has similar attributes as part of the language: `ref` and `out` method parameters. The `ref` and `out` method parameters can be used for value types and for reference types that are serializable. Using the `ref` parameter, the argument is marshaled in both directions, and with the `out` parameter the data is sent from the server to the client. With no parameters the data is sent to the server.



You can read more about the `out` and `ref` keywords in Chapter 3, “Objects and Types.”

Lifetime Management

How do a client and a server detect if the other side is not available anymore, and what are the problems you might get into?

For a client the answer can be simple. As soon as the client does a call to a method on the remote object, you get an exception of type `System.Runtime.Remoting.RemotingException`. You just have to handle this exception and do what's necessary; for example, perform a retry, write to a log, inform the user, and so on.

What about the server? When does the server detect if the client is not around anymore, meaning that the server can go ahead and clean up any resources it's holding for the client? You could wait until the next method call from the client — but maybe it will never arrive. In the COM realm, the DCOM protocol used a ping mechanism. The client sent a ping to the server with the information about the object referenced. A client can have hundreds of objects referenced on the server, so the information in the ping can be very large. To make this mechanism more efficient, DCOM didn't send all the information about all objects, just what is different from the previous ping.

This ping mechanism was efficient on a LAN, but it is not suitable for scalable solutions — imagine thousands of clients sending ping information to the server! .NET Remoting has a much more scalable solution for lifetime management: the *Leasing Distributed Garbage Collector (LDGC)*.

This lifetime management is only active for *client-activated objects* and well-known singleton objects. Single-call objects can be destroyed after every method call because they don't hold state. Client-activated objects do state and you should be aware of the resources used. A lease is created for client-activated objects that are referenced outside the application domain. A lease has a lease time, and when the lease time reaches zero, the lease expires, the remote object is disconnected, and finally, it is garbage collected.

Lease Renewals

If the client calls a method on the object when the lease has expired, an exception is thrown. If you have a client where the remote object could be needed for more than 300 seconds (the default value for lease times), you have three ways to renew a lease:

- **Implicit renewal** — This renewal of the lease is automatically done when the client calls a method on the remote object. If the current lease time is less than the `RenewOnCallTime` value, the lease is set to `RenewOnCallTime`.
- **Explicit renewal** — With this renewal, the client can specify the new lease time. This is done with the `Renew()` method of the `ILease` interface. You can get to the `ILease` interface by calling the `GetLifetimeService()` method of the transparent proxy.
- **Sponsoring renewal** — In the case of this renewal, the client can create a sponsor that implements the `ISponsor` interface and registers the sponsor in the leasing services using the `Register()` method of the `ILease` interface. The sponsor defines the lease extension time. When a lease expires, the sponsor is asked for an extension of the lease. The sponsoring mechanism can be used if you want long-lived remote objects on the server.

Leasing Configuration Values

The values that can be configured are the following:

- `LeaseTime` defines the time until a lease expires.
- `RenewOnCallTime` is the time the lease is set on a method call if the current lease time has a lower value.
- If a sponsor is not available within the `SponsorshipTimeout`, the remoting infrastructure looks for the next sponsor. If there are no more sponsors, the lease expires.
- The `LeaseManagerPollTime` defines the time interval at which the lease manager checks for expired objects.

The default values are listed in the following table.

LEASE CONFIGURATION	DEFAULT VALUE (SECONDS)
LeaseTime	300
RenewOnCallTime	120
SponsorshipTimeout	120
LeaseManagerPollTime	10

Classes Used for Lifetime Management

The `ClientSponsor` class implements the `ISponsor` interface. It can be used on the client side for lease extension. With the `ILease` interface you can get all information about the lease, all the lease properties, and the current lease time and state. The state is specified with the `LeaseState` enumeration. With the `LifetimeServices` utility class, you can get and set the properties for the lease of all remote objects in the application domain.

Getting Lease Information Example

In this small code example, the lease information is accessed by calling the `GetLifetimeService()` method of the transparent proxy. For the `ILease` interface, you have to declare the namespace `System.Runtime.Remoting.Lifetime`. For the class `UrlAttribute` you have to import the namespace `System.Runtime.Remoting.Activation`.

The leasing mechanism can only be used with stateful (client-activated and singleton) objects. Single-call objects are instantiated with every method call anyway, so the leasing mechanism doesn't apply.

To offer client-activated objects with the server, you can change the remoting configuration to a call to `RegisterActivatedServiceType()`:

```
RemotingConfiguration.ApplicationName = "Hello";
RemotingConfiguration.RegisterActivatedServiceType(typeof(Hello));
```

In the client application, the instantiation of the remote object must be changed, too. Instead of using the method `Activator.GetObject()`, `Activator.CreateInstance()` is used to invoke client-activated objects:

```
ChannelServices.RegisterChannel(new TcpClientChannel(), true);

object[] attrs = {new UrlAttribute("tcp://localhost:8086/Hi") };
Hello obj = (Hello)Activator.CreateInstance(typeof(Hello), null, attrs);
```

To show the leasing time, you can use the `ILease` interface returned by calling `GetLifetimeService()` from the proxy object:

```
ILease lease = (ILease)obj.GetLifetimeService();
if (lease != null)
{
    Console.WriteLine("Lease Configuration:");
    Console.WriteLine("InitialLeaseTime: {0}", lease.InitialLeaseTime);
    Console.WriteLine("RenewOnCallTime: {0}", lease.RenewOnCallTime);
    Console.WriteLine("SponsorshipTimeout: {0}", lease.SponsorshipTimeout);
    Console.WriteLine(lease.CurrentLeaseTime);
}
```

Changing Default Lease Configurations

The server itself can change the default lease configuration for all remote objects of the server using the `System.Runtime.Remoting.Lifetime.LifetimeServices` utility class:

```
LifetimeServices.LeaseTime = TimeSpan.FromMinutes(10);
LifetimeServices.RenewOnCallTime = TimeSpan.FromMinutes(2);
```

If you want different default lifetimes depending on the type of the remote object, you can change the lease configuration of the remote object by overriding the `InitializeLifetimeService()` method of the base class `MarshalByRefObject`:

```
public class Hello : System.MarshalByRefObject
{
    public override Object InitializeLifetimeService()
    {
        ILease lease = (ILease)base.InitializeLifetimeService();
        lease.InitialLeaseTime = TimeSpan.FromMinutes(10);
        lease.RenewOnCallTime = TimeSpan.FromSeconds(40);
        return lease;
    }
}
```

The lifetime services configuration can also be done by using a configuration file that is discussed next.

CONFIGURATION FILES

Instead of writing the channel and object configuration in the source code, you can use configuration files. This way the channel can be reconfigured, additional channels can be added, and so on, without changing the source code. Like all the other configuration files on the .NET platform, XML is used. The same application and configuration files that you read about in Chapter 18 and in Chapter 21, “Security,” are used here, too. For .NET Remoting, there are some more XML elements and attributes to configure the channel and the remote objects. The difference with the remoting configuration file is that this configuration needn’t be in the application configuration file itself; the file can have any name. To make the build process easier, in this chapter, the remoting configuration is written inside the application configuration files named with a `.config` file extension after the filename of the executable.

The code download (from www.wrox.com) contains the following example configuration files in the root directory of the client and the server examples: `clientactivated.config` and `wellknown.config`. With the client example, you will also find the file `wellknownhttp.config` that specifies an HTTP channel to a well-known remote object. To use these configurations, the files must be renamed to the name that is used with the parameter of the `RemotingConfiguration.Configure()` method and placed in the directory containing the executable file.

Here is just one example of how such a configuration file might look:



```
<configuration>
  <system.runtime.remoting>
    <application name="Hello">
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="Hi" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

code snippet ConfigurationFiles/HelloServer/app.config

`<configuration>` is the XML root element for all .NET configuration files. All the remoting configurations can be found in the subelement `<system.runtime.remoting>`. `<application>` is a subelement of `<system.runtime.remoting>`.

The following list describes the main elements and attributes of the parts within `<system.runtime.remoting>`:

- With the `<application>` element, you can specify the name of the application using the attribute name. On the server side, this is the name of the server, and on the client side it's the name of the client application. As an example for a server configuration, `<application name="Hello">` defines the remote application name Hello, which is used as part of the URL by the client to access the remote object.
- On the server, the element `<service>` is used to specify a collection of remote objects. It can have `<wellknown>` and `<activated>` subelements that specify the type of the remote object as well-known or client-activated.
- The client part of the `<service>` element is `<client>`. As with the `<service>` element, it can have `<wellknown>` and `<activated>` subelements to specify the type of the remote object. Unlike the `<service>` counterpart, `<client>` has a `url` attribute to specify the URL to the remote object.
- `<wellknown>` is an element that's used on the server and the client to specify well-known remote objects. The server part could look like this:

```
<wellknown mode="SingleCall"
  type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
  objectURI="Hi" />
```

- Although the mode attribute `SingleCall` or `Singleton` can be specified, the type is the type of the remote class, including the namespace `Wrox.ProCSharp.Remoting.Hello`. It is followed by the assembly name `RemoteHello`. `objectURI` is the name of the remote object that's registered in the channel.
- On the client, the `type` attribute is the same as it is for the server version. `mode` and `objectURI` are not needed, but the `url` attribute is used to define the path to the remote object instead: protocol, hostname, port number, application name, and the object URI:

```
<wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
  url="tcp://localhost:6791/Hello/Hi" />
```

- The `<activated>` element is used for client-activated objects. With the `type` attribute the type and the assembly must be defined for both the client and the server application:

```
<activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
```

- To specify the channel, the `<channel>` element is used. It's a subelement of `<channels>` so that a collection of channels can be configured for a single application. Its use is similar for clients and servers. With the XML attribute `ref`, you reference a preconfigured channel name. For the server channel, you have to set the port number with the XML attribute `port`. The XML attribute `displayName` is used to specify a name for the channel that is used from the .NET Framework Configuration tool, as discussed later in this chapter:

```
<channels>
  <channel ref="tcp" port="6791" displayName="TCP Channel" />
  <channel ref="http" port="6792" displayName="HTTP Channel" />
  <channel ref="ipc" portName="myIPCPort" displayName="IPC Channel"
  />
</channels>
```



Predefined channels have the names `tcp`, `http`, and `ipc`, which define the classes `TcpChannel`, `HttpChannel`, and `IpcChannel`.

Server Configuration for Well-Known Objects

This example file, `Wellknown_Server.config`, has the value `Hello` for the name property. In the following configuration file, the TCP channel is set to listen on port 6791 and the HTTP channel is set to listen on

port 6792. The IPC channel is configured with the port name `myIPCPort`. The remote object class is `Wrox.ProCSharp.Remoting.Hello` in the assembly `RemoteHello`, the object is called `Hi` in the channel, and the object mode is `SingleCall`:



```
<configuration>
  <system.runtime.remoting>
    <application name="Hello">
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="Hi" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

code snippet ConfigurationFiles/HelloServer/Wellknown.config

Client Configuration for Well-Known Objects

For well-known objects, you have to specify the assembly and the channel in the client configuration file `Wellknown_Client.config`. The types for the remote object are in the `RemoteHello` assembly, `Hi` is the name of the object in the channel, and the URI for the remote type `Wrox.ProCSharp.Remoting.Hello` is `ipc://myIPCPort/Hello/Hi`. In the client, an IPC channel is used as well, but no port is specified, so a free port is selected. The channel that is selected with the client must correspond to the URL:



```
<configuration>
  <system.runtime.remoting>
    <application name="Client">
      <client displayName="Hello client for well-known objects">
        <wellknown type = "Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          url="ipc://myIPCPort/Hello/Hi" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

code snippet ConfigurationFiles/HelloClient/Wellknown_Client.config

After a small change in the configuration file, you're using the HTTP channel (as you can see in `WellknownHttp_Client.config`):



```
<client>
  <wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
    url="http://localhost:6792/Hello/Hi" />
</client>
<channels>
  <channel ref="http" displayName="HTTP Channel (HelloClient)" />
</channels>
```

code snippet ConfigurationFiles/HelloClient/WellknownHttp_Client.config

Server Configuration for Client-Activated Objects

By changing only the configuration file (located in `ClientActivated_Server.config`), you can change the server configuration from server-activated to client-activated objects. Here the `<activated>` subelement of the `<service>` element is specified. With the `<activated>` element for the server configuration, only the `type` attribute must be specified. The `name` attribute of the `application` element defines the URI:



```
<configuration>
  <system.runtime.remoting>
    <application name="HelloServer">
      <service>
        <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
      </service>
    <channels>
      <channel ref="http" port="6788"
        displayName="HTTP Channel (HelloServer)" />
      <channel ref="tcp" port="6789"
        displayName="TCP Channel (HelloServer)" />
      <channel ref="ipc" portName="myIPCPort"
        displayName="IPC Channel (HelloServer)" />
    </channels>
  </application>
</system.runtime.remoting>
</configuration>
```

code snippet ConfigurationFiles/HelloServer/ClientActivated.config

Client Configuration for Client-Activated Objects

The `ClientActivated_Client.config` file defines the client-activated remote object using the `url` attribute of the `<client>` element and the `type` attribute of the `<activated>` element:



```
<configuration>
  <system.runtime.remoting>
    <application>
      <client url="http://localhost:6788/HelloServer"
        displayName="Hello client for client-activated objects">
        <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
      </client>
    <channels>
      <channel ref="http" displayName="HTTP Channel (HelloClient)" />
      <channel ref="tcp" displayName="TCP Channel (HelloClient)" />
      <channel ref="ipc" displayName="IPC Channel (HelloClient)" />
    </channels>
  </application>
</system.runtime.remoting>
</configuration>
```

code snippet ConfigurationFiles/HelloClient/ClientActivated_Client.config

Server Code Using Configuration Files

In the server code, you have to configure remoting using the static method `Configure()` from the `RemotingConfiguration` class. Here all channels that are defined in the configuration file are created and configured with the .NET Remoting runtime. Maybe you also want to know about the channel

configurations from the server application — that's why the static methods `ShowActivatedServiceTypes()` and `ShowWellKnownServiceTypes()` were created; they are called after loading and starting the remoting configuration:



Available for
download on
Wrox.com

```
public static void Main()
{
    RemotingConfiguration.Configure("HelloServer.exe.config", false);
    Console.WriteLine("Application: {0}", RemotingConfiguration.ApplicationName);
    ShowActivatedServiceTypes();
    ShowWellKnownServiceTypes();
    System.Console.WriteLine("press return to exit");
    System.Console.ReadLine();
}
```

code snippet ConfigurationFiles/HelloServer/Program.cs

These two functions show configuration information of well-known and client-activated types:

```
public static void ShowWellKnownServiceTypes()
{
    WellKnownServiceTypeEntry[] entries =
        RemotingConfiguration.GetRegisteredWellKnownServiceTypes();
    foreach (var entry in entries)
    {
        Console.WriteLine("Assembly: {0}", entry.AssemblyName);
        Console.WriteLine("Mode: {0}", entry.Mode);
        Console.WriteLine("URI: {0}", entry.ObjectUri);
        Console.WriteLine("Type: {0}", entry.TypeName);
    }
}

public static void ShowActivatedServiceTypes()
{
    ActivatedServiceTypeEntry[] entries =
        RemotingConfiguration.GetRegisteredActivatedServiceTypes();
    foreach (var entry in entries)
    {
        Console.WriteLine("Assembly: {0}", entry.AssemblyName);
        Console.WriteLine("Type: {0}", entry.TypeName);
    }
}
```

Client Code Using Configuration Files

In the client code, it is only necessary to configure the remoting services using the configuration file `client.exe.config`. After that, you can use the new operator to create new instances of the remote class `Hello`, no matter whether you work with server-activated or client-activated remote objects. However, there's a small difference — with client-activated objects, it's now possible to use *nondefault constructors* with the new operator. This isn't possible for server-activated objects. Single-call objects can have no state because they are destroyed with every call; singleton objects are created just once. Calling nondefault constructors is only possible with client-activated objects because this is the only type whereby invoking the new operator the remote object is instantiated.

In the `Main()` method of the file `HelloClient.cs`, you can now change the remoting code to use the configuration file with `RemotingConfiguration.Configure()` and create the remote object with the new operator:



Available for
download on
Wrox.com

```
RemotingConfiguration.Configure("HelloClient.exe.config", false);
Hello obj = new Hello();
if (obj == null)
{
    Console.WriteLine("could not locate server");
}
```

```

    return;
}
for (int i=0; i < 5; i++)
{
    Console.WriteLine(obj.Greeting("Stephanie"));
}

```

code snippet ConfigurationFiles/HelloClient/Program.cs

Delayed Loading of Client Channels

With .NET Remoting, three channels are configured by default that can be used automatically if the client doesn't configure a channel:

```

<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="http client" displayName="http client (delay loaded)"
        delayLoadAsClientChannel="true" />
      <channel ref="tcp client" displayName="tcp client (delay loaded)"
        delayLoadAsClientChannel="true" />
      <channel ref="ipc client" displayName="ipc client (delay loaded)"
        delayLoadAsClientChannel="true" />
    </channels>
  </application>
</system.runtime.remoting>

```

The XML attribute `delayLoadAsClientChannel` with a value `true` specifies that the channel should be used from a client that doesn't configure a channel. The runtime tries to connect to the server using the delay-loaded channels. So, it is not necessary to configure a channel in the client configuration file, and a client configuration file for the well-known object you have used earlier can look as simple as this:

```

<configuration>
  <system.runtime.remoting>
    <application name="Client">
      <client url="tcp://localhost:6791/Hello">
        <wellknown type = "Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          url="tcp://localhost:6791/Hello/Hi" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>

```

Debugging Configuration

If you have a misconfigured server configuration file (for example, by specifying the wrong name of the remote assembly), the error is not detected when the server starts. Instead, the error is detected when the client instantiates a remote object and invokes a method. The client gets the exception that the remote object assembly cannot be found. By specifying the configuration `<debug loadTypes="true" />`, the remote object is loaded and instantiated on the server when the server starts up. This way you get an error on the server if the configuration file is misconfigured:

```

<configuration>
  <system.runtime.remoting>
    <application name="Hello">
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="Hi" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>

```

```

        <channel ref="tcp" port="6791"
            displayName="TCP Channel (HelloServer)" />
    </channels>
</application>
<debug loadTypes="true" />
</system.runtime.remoting>
</configuration>

```

Lifetime Services in Configuration Files

Leasing configuration for remote servers can also be done with the application configuration files.

The `<lifetime>` element has the attributes `leaseTime`, `sponsorshipTimeOut`, `renewOnCallTime`, and `pollTime`, as shown in this example:

```

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

```

Using configuration files, it is possible to change the remoting configuration by editing files instead of working with source code. You can easily change the channel to use HTTP instead of TCP, change a port, the name of the channel, and so on. With the addition of a single line, the server can listen to two channels instead of one.

Formatter Providers

Earlier in this chapter we discussed where properties of the formatter provider need to be changed to support marshaling of all objects across the network. Instead of doing this programmatically, as was done earlier, you can configure the properties of a formatter provider in a configuration file.

The following server configuration file is changed within the `<channel>` element, insofar as `<serverProviders>` and `<clientProviders>` are defined as child elements. With the `<serverProviders>`, the built-in providers `wsdl`, `soap`, and `binary` are referenced, and with the `soap` and `binary` providers the property `typeFilterLevel` is set to `Full`:

```

<configuration>
  <system.runtime.remoting>
    <application name="HelloServer">
      <service>
        <activated type="Wrox.ProCSharp.Remoting.Hello, RemoteHello" />
      </service>
      <channels>
        <channel ref="tcp" port="6789" displayName="TCP Channel (HelloServer)">
          <serverProviders>
            <provider ref="wsdl" />
            <provider ref="soap" typeFilterLevel="Full" />
            <provider ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <provider ref="binary" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>

```

HOSTING SERVERS IN ASP.NET

Up to this point, all the sample servers have been running in self-hosted .NET servers. A self-hosted server must be launched manually. A .NET Remoting server can also be started in a lot of other application types. In a Windows Service, the server can be automatically started at boot time, and in addition the process can run with the credentials of the system account. For more details on Windows Services, see Chapter 25, “Windows Services.”

There’s special support for .NET Remoting servers for ASP.NET. ASP.NET can be used for the automatic startup of remote servers. ASP.NET-hosted Remoting uses a different file for configuration than EXE-hosted applications, but it has the same syntax.

To use the infrastructure from the Internet Information Services (IIS) and ASP.NET, you have to create a class that derives from `System.MarshalByRefObject` and has a default constructor. The code used earlier for the server to create and register the channel is no longer necessary; that’s done by the ASP.NET runtime. You have to create only a virtual directory on the Web server that maps a directory into which you put the configuration file `Web.config`. The assembly of the remote class must reside in the `bin` subdirectory.

To configure a virtual directory on the web server, you can use the IIS MMC. Selecting the Default Web Site and opening the Action menu creates a new Virtual Directory.

The configuration file `Web.config` on the web server must be put in the home directory of the virtual web site. With the default IIS configuration, the channel that will be used listens to port 80:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="SingleCall"
          type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          objectUri="HelloService.soap" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```



If the remoting object is hosted within IIS, the name of the remote object must end either with `.soap` or `.rem`, depending on the type of formatter that is used (SOAP or the binary).

The client can now connect to the remote object using the following configuration file. The URL that must be specified for the remote object here is the web server `localhost`, followed by the web application name `RemoteHello` (specified when creating the virtual web site), and the URI of the remote object `HelloService.soap` that is defined in the file `Web.config`. It’s not necessary to specify the port number 80, because that’s the default port for the HTTP protocol. Not specifying a `<channels>` section means that the delay-loaded HTTP channel from the configuration file `machine.config` is used:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client url="http://localhost/RemoteHello">
        <wellknown type="Wrox.ProCSharp.Remoting.Hello, RemoteHello"
          url="http://localhost/RemoteHello/HelloService.soap" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```



Hosting remote objects in ASP.NET only supports well-known objects—client-activated objects are not possible.

CLASSES, INTERFACES, AND SOAPSUDS

In the .NET Remoting examples you have seen so far, you have always copied the assembly of the remote object not only to the server but also to the client application. This way the MSIL code of the remote object is on both the client and the server system, although in the client application only the metadata is needed. However, copying the remoting object assembly means that it's not possible for the client and the server to be programmed independently. A much better way to use just the metadata is to use interfaces or the `Soapsuds.exe` utility instead.

Interfaces

You get a cleaner separation of the client and server code by using interfaces. An interface simply defines the methods without implementation. This way the contract (the interface) is separated from the implementation, and just the contract is needed on the client system. Here are the necessary steps for using an interface:

1. Define an interface that will be placed in a separate assembly.
2. Implement the interface in the remote object class. To do this, the assembly of the interface must be referenced.
3. On the server side no more changes are required. The server can be programmed and configured in the usual way.
4. On the client side, reference the assembly of the interface instead of the assembly of the remote class.
5. The client can now use the interface of the remote object rather than the remote object class. The object can be created using the `Activator` class as it was done earlier. You can't use the `new` operator in this way, because the interface itself cannot be instantiated.

The interface defines the contract between the client and server. The two applications can now be developed independently of each other. If you also stick to the old COM rules about interfaces (that interfaces should never be changed), you will not have any versioning problems.

Soapsuds

You can also use the `Soapsuds` utility to get the metadata from an assembly if an HTTP channel and the SOAP formatter are used. `Soapsuds` can convert assemblies to XML schemas and XML schemas to wrapper classes — it also works in the other direction.

The following command converts the type `Hello` from the assembly `RemoteHello` to the assembly `HelloWrapper`, where a transparent proxy is generated that calls the remote object:

```
soapsuds -types:Wrox.ProCSharp.Remoting.Hello,RemoteHello -oa:HelloWrapper.dll
```

With `Soapsuds` you can also get the type information directly from a running server, if the HTTP channel and the SOAP formatter are used:

```
soapsuds -url:http://localhost:6792/hello/hi?wsdl -oa:HelloWrapper.dll
```

In the client, you can now reference the generated assembly instead of the original one. Some of the `Soapsuds` options are listed in the following table.

OPTION	DESCRIPTION
-url	Retrieve schema from the specified URL
-proxyurl	If a proxy server is required to access the server, specify the proxy with this option
-types	Specify a type and assembly to read the schema information from it
-is	Input schema file
-ia	Input assembly file
-os	Output schema file
-oa	Output assembly file

ASYNCHRONOUS REMOTING

If server methods take some time to complete and the client needs to do some different work at the same time, it isn't necessary to start a separate thread to make the remote call. By making an asynchronous call, you cause the method to start but return immediately to the client. Asynchronous calls can be made on a remote object, as they are made on a local object with the help of a delegate. With methods that don't return a value, you can also use the *OneWay* attribute.

Using Delegates with .NET Remoting

To make an asynchronous method, you can use a delegate with the same argument and return value as the *Greeting()* method of the remote object. The generic *Func<T>* and *Action<T>* delegates are very helpful with this. The *Greeting()* method requires a string parameter that returns a string type, *Func<string, string>*. This is the delegate definition to invoke this method. You start the *Greeting()* call using the *BeginInvoke()* method of the delegate. The second argument of *BeginInvoke()* is an *AsyncCallback* instance that defines the method *HelloClient.Callback()*. It is called when the remote method is finished. In the *Callback()* method, the remote call is finished using *EndInvoke()*:

```
using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
    public class HelloClient
    {
        private static string greeting;

        public static void Main()
        {
            RemotingConfiguration.Configure("HelloClient.exe.config");
            Hello obj = new Hello();
            if (obj == null)
            {
                Console.WriteLine("could not locate server");
                return;
            }
            Func<string, string> d = new Func<string, string>(obj.Greeting);
            IAsyncResult ar = d.BeginInvoke("Stephanie", null, null);

            // do some work and then wait
            ar.AsyncWaitHandle.WaitOne();
            string greeting = null;
            if (ar.IsCompleted)
```

```

        {
            greeting = d.EndInvoke(ar);
        }

        Console.WriteLine(greeting);
    }
}

```

You can find more information about delegates in Chapter 8, “Delegates, Lambdas, and Events,” and about using delegates asynchronously in Chapter 20, “Threads, Tasks, and Synchronization.”

OneWay Attribute

A method that has a `void` return and only input parameters can be marked with the `OneWay` attribute. The `OneWay` attribute (defined within the namespace `System.Runtime.Remoting.Messaging`) makes a method automatically asynchronous, regardless of how the client calls it. Adding the method `TakeAWhile()` to your remote object class `RemoteHello` creates a fire-and-forget method. If the client calls it by the proxy, the proxy immediately returns to the client. On the server, the method finishes some time later:

```

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

```

SECURITY WITH .NET REMOTING

.NET Remoting has supported security since .NET 2.0. .NET Remoting offers confidentiality of the data that is transferred across the wire as well as authentication of the user.

Security can be configured with every channel. TCP, HTTP, and IPC channels support security configuration. With the server configuration you have to define the minimum security requirement of the communication, while the client configuration defines the capabilities regarding security. If the client defines a security configuration that is lower than the minimum requirements defined by the server, the communication fails.

Let’s start with the server configuration. The following extract of the configuration file shows how the security can be defined for the server.

With the XML attribute `protectionLevel`, you can specify whether the server requires encrypted data to be sent across the network. The possible values that can be set with the `protectionLevel` attribute are `None`, `Sign`, and `EncryptAndSign`. With the value `Sign` a signature is created that is used to verify whether the data that is sent didn’t change on its way across the network. However, it is still possible to read the data with a sniffer that reads all the data on the network. With the value `EncryptAndSign`, the data that is sent is also encrypted to make it impossible that systems not officially participating with the message transfer can read the data.

The attribute `impersonate` can be set to `true` or `false`. If `impersonate` is set to `true`, the server can impersonate the user of the client to access resources in its name:

```

<channels>
  <channel ref="tcp" port="9001"
    secure="true"
    protectionLevel="EncryptAndSign"
    impersonate="false" />
</channels>

```

With the client configuration, the capabilities of the network communication are defined. If the capabilities don't fulfill the server requirements, the communication fails. You can see `protectionLevel` and `TokenImpersonationLevel` configured in the following sample configuration.

`protectionLevel` can be set to the same options as you have seen with the server configuration file.

The `TokenImpersonationLevel` can be set to `Anonymous`, `Identification`, `Impersonation`, and `Delegation`. With the `Anonymous` value, the server cannot identify the user of the client. If the value is set to `Identification`, the server can find out the user identity of the server. If the server is configured with `impersonate` set to `true`, the communication fails if the client is only configured with `Anonymous` or `Identification`. Setting `TokenImpersonationLevel` to `Impersonation` allows the server to impersonate the client. Setting `TokenImpersonationLevel` to `Delegation` allows the server to impersonate the client not only to access resources local on the server but also to use the user identification to access resources on other servers. `Delegation` is only possible if Kerberos is used for logging on the user, as is possible with Active Directory:

```
<channels>
  <channel ref="tcp"
    secure="true"
    protectionLevel="EncryptAndSign"
    TokenImpersonationLevel="Impersonate" />
</channels>
```

If you create the channel programmatically instead of using configuration files, you can define the security setting programmatically as well. A collection that contains all security settings can be passed to the constructor of the channel as shown. The collection class must implement the interface `IDictionary`, for example the generic `Dictionary` class or the `Hashtable` class:

```
var dict = new Dictionary<string, string>();
dict.Add("secure", "true");

var clientChannel = new TcpClientChannel(dict, null);
```

You can read more about security in Chapter 21.

REMOTING AND EVENTS

Not only can the client invoke methods on the remote object across the network but the server can do the same — invoking methods in the client. For this, a mechanism that you already know from the basic language features is used: *delegates and events*.

In principle, the architecture is simple. The server has a remotable object that the client can call, and the client has a remotable object that the server can call:

- The remote object in the server must declare an external function (a delegate) with the signature of the method that the client will implement in a handler.
- The arguments that are passed with the handler function to the client must be marshalable, so all the data sent to the client must be serializable.
- The remote object must also declare an instance of the delegate function modified with the `event` keyword; the client will use this to register a handler.
- The client must create a sink object with a handler method that has the same signature as the delegate defined, and it has to register the sink object with the event in the remote object.

Take a look at an example. To see all the parts of event handling with .NET Remoting, create five classes: `Server`, `Client`, `RemoteObject`, `EventSink`, and `StatusEventArgs`. The dependencies of these classes are shown in Figure 54-5.

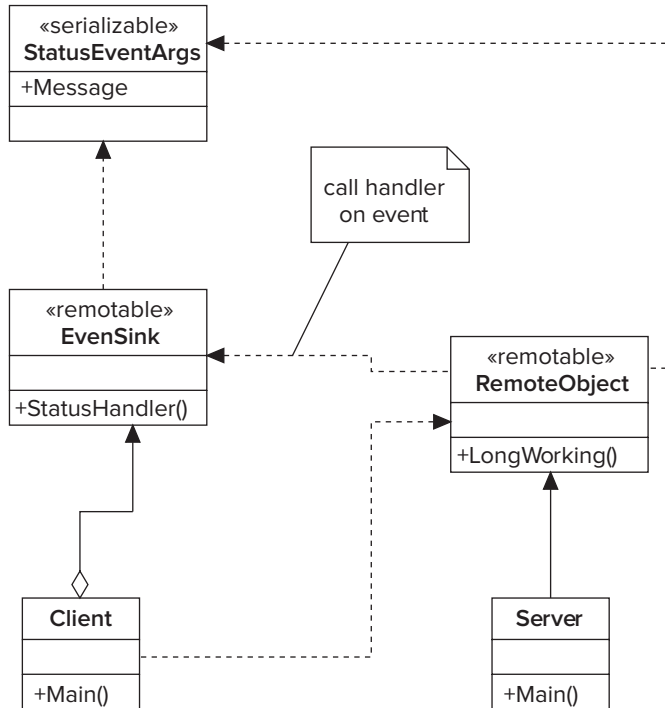


FIGURE 54-5

The `Server` class is a remoting server such as the one you are already familiar with. The `Server` class will create a channel based on information from a configuration file and register the remote object that's implemented in the `RemoteObject` class in the remoting runtime. The remote object declares the arguments of a delegate and fires events in the registered handler functions. The argument that's passed to the handler function is of type `StatusEventArgs`. The class `StatusEventArgs` must be serializable so that it can be marshaled to the client.

The `Client` class represents the client application. This class creates an instance of the `EventSink` class and registers the `StatusHandler()` method of this class as a handler for the delegate in the remote object. `EventSink` must be remotable like the `RemoteObject` class, because this class will also be called across the network.

Remote Object

The remote object class is implemented in the file `RemoteObject.cs`. The remote object class must be derived from `MarshalByRefObject`, as you already know from the previous examples. To enable the client to register an event handler that can be called from within the remote object, you have to declare an external function with the `delegate` keyword. Declare the delegate `StatusEvent()` with two arguments: the `sender` (so the client knows about the object that fired the event) and a variable of type `StatusEventArgs`. Into the argument class you can put all the additional information that you want to send to the client.

The method that will be implemented in the client has some strict requirements. It can only have input parameters — return types, ref, and out parameters are not allowed — and the argument types must be either [Serializable] or remotable (derived from MarshalByRefObject). These requirements are fulfilled by the parameters that are defined with the EventHandler<StatusEventArgs> delegate.

Within the RemoteObject class, declare an event name Status of type EventHandler<StatusEventArgs>.

The client must add an event handler to the Status event to receive status information from the remote object:



```
public class RemoteObject : MarshalByRefObject
{
    public RemoteObject()
    {
        Console.WriteLine("RemoteObject constructor called");
    }
    public event EventHandler<StatusEventArgs> Status;
```

code snippet RemotingAndEvents/RemoteObject/RemoteObject.cs

The LongWorking() method checks whether an event handler is registered before the event is fired by calling Status(this, e). To verify that the event is fired asynchronously, fire an event at the start of the method before doing the Thread.Sleep() and after the sleep:

```
public void LongWorking(int ms)
{
    Console.WriteLine("RemoteObject: LongWorking() Started");
    StatusEventArgs e = new StatusEventArgs(
        "Message for Client: LongWorking() Started");

    // fire event
    if (Status != null)
    {
        Console.WriteLine("RemoteObject: Firing Starting Event");
        Status(this, e);
    }
    System.Threading.Thread.Sleep(ms);
    e.Message = "Message for Client: LongWorking() Ending";
    // fire ending event
    if (Status != null)
    {
        Console.WriteLine("RemoteObject: Firing Ending Event");
        Status(this, e);
    }
    Console.WriteLine("RemoteObject: LongWorking() Ending");
}
```

Event Arguments

As you've seen in the RemoteObject class, the class StatusEventArgs is used as an argument for the delegate. With the Serializable attribute an instance of this class can be transferred from the server to the client. Here is a simple property of type string to send a message to the client:



```
[Serializable]
public class StatusEventArgs : EventArgs
{
    public StatusEventArgs(string message)
    {
        this.Message = message;
    }
    public string Message { get; set; }
}
```

code snippet RemotingAndEvents/RemoteObject/RemoteObject.cs

Server

The server is implemented within a console application. With `RemotingConfiguration.Configure()`, the configuration file is read and thus the channel and remote objects are set up. With the `Console.ReadLine()` the server waits until the user stops the application:



```
using System;
using System.Runtime.Remoting;
namespace Wrox.ProCSharp.Remoting
{
    class Server
    {
        static void Main()
        {
            RemotingConfiguration.Configure("Server.exe.config", true);
            Console.WriteLine("press return to exit");
            Console.ReadLine();
        }
    }
}
```

code snippet RemotingAndEvents/RemoteServer/Program.cs

Server Configuration File

The server configuration file, `Server.exe.config`, is also created as already discussed. There is just one important point: the remote object must keep state for the client because the client at first registers the event handler and calls the remote method afterward. You cannot use single-call objects with events, so the `RemoteObject` class is configured as a client-activated type. Also, to support delegates, you have to enable full serialization by specifying the `typeFilterLevel` attribute with the `<provider>` element:



```
<configuration>
  <system.runtime.remoting>
    <application name="CallbackSample">
      <service>
        <activated type="Wrox.ProCSharp.Remoting.RemoteObject,
                      RemoteObject" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

code snippet RemotingAndEvents/RemoteServer/app.config

Event Sink

An event sink library is required for use by the client and is invoked by the server. The event sink implements the handler `StatusHandler()` that's defined with the delegate. As previously noted, the method

can only have input parameters and only a void return. The `EventSink` class must also inherit from the class `MarshalByRefObject` to make it remotable, because it will be called remotely from the server:



```
using System;
using System.Runtime.Remoting.Messaging;

namespace Wrox.ProCSharp.Remoting
{
    public class EventSink : MarshalByRefObject
    {
        public EventSink()
        {
        }
        public void StatusHandler(object sender, StatusEventArgs e)
        {
            Console.WriteLine("EventSink: Event occurred: " + e.Message);
        }
    }
}
```

code snippet RemotingAndEvents/EventSink/EventSink.cs

Client

The client reads the client configuration file with the `RemotingConfiguration` class, which is not different from the clients that have been discussed so far. The client creates an instance of the remotable sink class `EventSink` locally. The method that should be called from the remote object on the server is passed to the remote object:



```
using System;
using System.Runtime.Remoting;

namespace Wrox.ProCSharp.Remoting
{
    class Client
    {
        static void Main()
        {
            RemotingConfiguration.Configure("Client.exe.config", true);
            Console.WriteLine("wait for server...");
            Console.ReadLine();
        }
    }
}
```

code snippet RemotingAndEvents/RemoteClient/Program.cs

The differences start here. You have to create an instance of the remotable sink class `EventSink` locally. Because this class will not be configured with the `<client>` element, it's instantiated locally. Next, the remote object class `RemoteObject` is instantiated. This class is configured in the `<client>` element, so it's instantiated on the remote server:

```
var sink = new EventSink();
var obj = new RemoteObject();
if (!RemotingServices.IsTransparentProxy(obj))
{
    Console.WriteLine("check your remoting configuration");
    return;
}
```

Now you can register the handler method of the `EventSink` object in the remote object. `StatusEvent` is the name of the delegate that was defined in the server. The `StatusHandler()` method has the same arguments as defined in the `StatusEvent`.

By calling the `LongWorking()` method, the server will call back into the method `StatusHandler()` at the beginning and at the end of the method:

```
// register client sink in server - subscribe to event
obj.Status += sink.StatusHandle;
obj.LongWorking(5000);
```

Now you are no longer interested in receiving events from the server, so you are unsubscribing from the event. The next time you call `LongWorking()`, no events will be received:

```
// unsubscribe from event
obj.Status -= sink.StatusHandler;
obj.LongWorking(5000);
Console.WriteLine("press return to exit");
Console.ReadLine();
    }
}
```

Client Configuration File

The configuration file for the client, `client.exe.config`, is nearly the same configuration file for client-activated objects that we've already seen. The difference is in defining a port number for the channel. Because the server must reach the client with a known port, you have to define the port number for the channel as an attribute of the `<channel>` element. It isn't necessary to define a `<service>` section for your `EventSink` class, because this class will be instantiated from the client with the new operator locally. The server does not access this object by its name; it will receive a marshaled reference to the instance instead:



```
<configuration>
  <system.runtime.remoting>
    <application name="Client">
      <client url="tcp://localhost:6791/CallbackSample">
        <activated type="Wrox.ProCSharp.Remoting.RemoteObject,
          RemoteObject" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

code snippet RemotingAndEvents/RemoteClient/app.config

Running Programs

Here is the resulting output of the server. The constructor of the remote object is called once because you have a client-activated object. Next, you can see that the call to `LongWorking()` has started and an event is fired to the client. The next start of the `LongWorking()` method doesn't fire events, because the client has already unregistered its interest in the event:

```
press return to exit
RemoteObject constructor called
RemoteObject: LongWorking() Started
RemoteObject: Firing Starting Event
RemoteObject: Firing Ending Event
```



```
RemoteObject: LongWorking() Ending
RemoteObject: LongWorking() Started
RemoteObject: LongWorking() Ending
```

With the client output you can see the events that made it across the network:

```
wait for server...
```

```
EventSink: Event occurred: Message for Client: LongWorking() Started
EventSink: Event occurred: Message for Client: LongWorking() Ending
```

CALL CONTEXTS

Client-activated objects can hold state for a specific client. With client-activated objects the server allocates resources for every client. With server-activated `SingleCall` objects, a new instance is created for every instance call, and no resources are held on the server; these objects can't hold state for a client. For state management, you can keep state on the client side; the state information is sent with every method call to the server. To implement such a state management, it is not necessary to change all method signatures to include an additional parameter that passes the state to the server; this is automatically done by the *call context*.

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

You can assign data to the call context using `CallContext.SetData()`. The class of the object that's used as data for the `SetData()` method must implement the interface `ILogicalThreadAffinative`. You can get this data again in the same logical thread (but possibly a different physical thread) using `CallContext.GetData()`.

For the data of the call context, create a new C# Class Library with the class `CallContextData`. This class will be used to pass some data from the client to the server with every method call. The class that's passed with the call context must implement the `System.Runtime.Remoting.Messaging.ILogicalThreadAffinative` interface. This interface doesn't have a method; it's just a markup for the runtime to define that instances of this class should flow with a logical thread. The `CallContextData` class must also be marked with the `Serializable` attribute so it can be transferred through the channel:

```
using System;
using System.Runtime.Remoting.Messaging;
namespace Wrox.ProCSharp.Remoting
{
    [Serializable]
    public class CallContextData : ILogicalThreadAffinative
    {
        public CallContextData()
        {
        }
        public string Data { get; set; }
    }
}
```

In the remote object class `Hello`, change the `Greeting()` method to access the call context. To use the `CallContextData` class, you have to reference the previously created assembly `CallContextData` in the file `CallContextData.dll`. To work with the `CallContext` class, the namespace `System.Runtime.Remoting.Messaging` must be opened. The variable `cookie` holds the data that is sent from the client to the server. The name "cookie" is chosen because the context works similarly to a browser-based cookie, where the client automatically sends data to the web server:

```
public string Greeting(string name)
{
    Console.WriteLine("Greeting started");
    CallContextData cookie = (CallContextData)CallContext.GetData("mycookie");
```

```

        if (cookie != null)
        {
            Console.WriteLine("Cookie: " + cookie.Data);
        }
        Console.WriteLine("Greeting finished");
        return "Hello, " + name;
    }
}

```

In the client code, the call context information is set by calling `CallContext.SetData()`. With this method an instance of the class `CallContextData` is assigned to be passed to the server. Now every time the method `Greeting()` is called in the `for` loop, the context data is automatically passed to the server:

```

CallContextData cookie = new CallContextData();
cookie.Data = "information for the server";
CallContext.SetData("mycookie", cookie);
for (int i=0; i < 5; i++)
{
    Console.WriteLine(obj.Greeting("Christian"));
}

```

You can use such a call context to send information about the user, the name of the client system, or simply a unique identifier used on the server side to get some state information from a database.

SUMMARY

In this chapter, you've seen that .NET Remoting facilitates the task of invoking methods across the network. A remote object has to inherit from `MarshalByRefObject`. In the server application, only a single method is needed to load the configuration file so that the channels and remote objects are both set up and running. Within the client, you load the configuration file and can use the `new` operator to instantiate the remote object.

You also used .NET Remoting without the help of configuration files. On the server, you simply created a channel and registered a remote object. On the client, you created a channel and used the remote object.

Furthermore, you learned that the .NET Remoting architecture is flexible and can be extended. All parts of this technology such as channels, proxies, formatters, message sinks, and so on are pluggable and can be replaced with custom implementations.

You used HTTP, TCP, and IPC channels for communication across the network, and SOAP and binary formatters to format the parameters before sending them.

You learned about the stateless and stateful object types that are used by well-known and client-activated objects. With client-activated objects you have seen how the leasing mechanism is used to specify the lifetime of remote objects.

You have also seen that .NET Remoting is very well-integrated in other parts of the .NET Framework, such as calling asynchronous methods, performing callbacks using the `delegate` and `event` keywords, and so on.