Rosslyn, VA & Key Bridge From Washington Canoe Club

Networked
Client-Server Applications

Learning Objectives

- Demonstrate your ability to create networked client-server applications
- Utilize .NET Remoting to create client-server applications
- State the three things you must do to create a .NET Remoting application
- Create .NET Remoting applications with and without configuration files
- Create .NET Remoting applications using remote object interfaces
- Create client-server applications using the TcpListener and TcpClient classes
- Create a multithreaded TCP/IP server application
- Use a ParameterizedThreadStart delegate method to create a multithreaded client-server application
- Create a TCP/IP client application
- Send serialized objects between networked client-server applications
- Create a custom application protocol for use in a client-server application
- Use StreamReader and StreamWriter objects to send and receive data in a client-server application
- Use the NetworkStream to serialize/deserialize objects between client-server applications

Introduction

You're going to have a lot of fun in this chapter. It is here that you'll put into practical use many of the networking concepts discussed in the previous chapter.

You will start by learning how to build client-server applications using Microsoft's .NET remoting technology. .NET remoting makes it possible to utilize the services of an object hosted on a remote computer or locally across applications boundaries. I'll show you how to create .NET remoting applications using programmatic configuration and configuration files. I'll also show you how to call the services of a remote object via one of its implemented interfaces.

Once you've mastered the art of .NET remoting, you'll learn how to create client-server applications using the TcpListener, TcpClient, and other classes found in the System.Net namespace. I'll show you how to create a multi-threaded server application that can service requests from multiple clients. I'll also show you how to create a custom application protocol so your client-server applications can talk to each other.

If you are timid about the thought of building a networked application, don't be. I will show you what to do and how to compile your project files every step of the way. When you've finished this chapter, you will have the confidence to build your own networked client-server applications.

Also, what you learn here will be put to good use in *Chapter 20: Database Access & Multitiered Applications*.

Building Client-Server Applications With .NET Remoting

.NET remoting makes it possible to access the services of an object hosted locally but in a different application domain or hosted remotely on another computer located somewhere in network land. The .NET remoting infrastructure hides many of the nasty details normally associated with network programming, letting you focus on building value-added applications. You'll find .NET remoting to be an easy and powerful way to build client-server applications.

The Three Required Components of A .NET Remoting Application

All .NET remoting applications have three common components regardless of their complexity: a *remotable object*, a *server application* that hosts the remotable object and handles incoming service requests, and a *client application* that utilizes the services of the remotely-hosted object. Figure 19-1 illustrates these concepts.
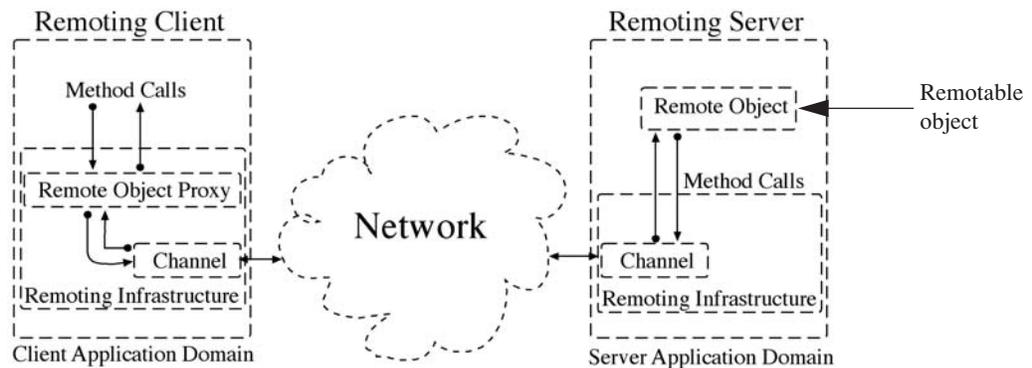

Figure 19-1: .NET Remoting Architecture

Referring to Figure 19-1 — the three components of a typical .NET remoting application include a remotable object, a server application that coordinates method calls to the remote object and makes its services available on a particular channel, and a client application that accesses the services of the remote object via the appropriate channel.

A remotable object is created from a class that inherits from System.MarshalByRefObject. This enables the object to be shared across application domains in .NET remoting applications. Note that the remotable object can be

 C# For Artists

simple or complex. In this section, I'll keep the remotable objects simple, but at the end of this section you'll see how a remote object can serve as a façade to a complex data-driven application.

The remoting server application hosts the remotable object and makes its services available via a channel. There are three primary channel types: *TcpChannel*, *HttpChannel*, and *IpcChannel*. The IpcChannel is used for *inter-process* communication between client and server applications hosted on the same machine.

The remoting client application accesses the services of the remote object via a *proxy* created by the .NET remoting infrastructure. The .NET remoting infrastructure is responsible for creating the proxy object, setting up and tearing down network communications between client and server applications, and marshaling and unmarshaling remote method calls and any returned objects between client and server. All this is done under the covers as you will soon see.

A Simple .NET Remoting Application

In this section I'll show you how to create a simple .NET remoting application. It all starts with the creation of a remotable class type you can use to create remotable objects. Example 19.1 gives the code for a class named TestClass.

19.1 TestClass.cs

```
1    using System;
2
3    public class TestClass : MarshalByRefObject{
4
5      private string _text;
6
7      public string Text {
8        get { return _text; }
9        set {
10         _text = value;
11         Console.WriteLine("Property changed --> " + _text);
12        }
13     }
14
15     public TestClass():this("This is the default text message!"){}
16
17     public TestClass(string s){
18       _text = s;
19     }
20   }
```

Referring to Example 19.1 — TestClass extends System.MarshalByRefObject. This tags objects of type TestClass as being remotable. TestClass has two constructors and one property named Text. Setting the Text property causes a short message to be written to the console. This is *not* normally a good thing to do in a property but in this case it will help to demonstrate some important remoting concepts.

Compile TestClass into a dynamically linked library (dll) by issuing the following compiler command at the command line:

<p align="center">csc /t:library TestClass.cs</p>

The reason you need to compile this into a dll is that you'll need to share this code with both the server and client applications.

Next, let's create the server application that will host an instance of TestClass. Example 19.2 gives the code for a class named RemotingServer.

19.2 RemotingServer.cs

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingServer {
7      public static void Main(){
8
9        try{
10       TcpChannel channel = new TcpChannel(8080);
11       ChannelServices.RegisterChannel(channel, false);
12       RemotingConfiguration.RegisterWellKnownServiceType(typeof(TestClass), "TestClass",
13                                                 WellKnownObjectMode.SingleCall);
14       Console.WriteLine("Listening for remote requests. Press any key to exit...");
15       Console.ReadLine();
16       } catch(ArgumentNullException ane){
17         Console.WriteLine("Channel argument was null!");
```

```
18          Console.WriteLine(ane);
19        } catch(RemotingException re){
20          Console.WriteLine("Channel has already been registered!");
21          Console.WriteLine(re);
22        } catch(Exception e){
23          Console.WriteLine(e);
24        }
25      } // end Main()
26    } // end class definition
```

Referring to Example 19.2 — notice first the list of namespaces you must rely upon. These include System.Runtime.Remoting, System.Runtime.Remoting.Channels, and System.Runtime.Remoting.Channels.Tcp. I am using the System.Runtime.Remoting.Channels.Tcp namespace because I'm going to make available the services of a TestClass object via a TcpChannel.

The first thing the server does is create the TcpChannel object to listen on port 8080. Be sure this port is not in use or the TcpChannel() constructor call will throw an exception.

Next, on line 11, the newly created channel is registered with the help of the ChannelServices.RegisterChannel() method. The second argument to the RegisterChannel() method specifies whether or not to enforce security on the channel. In this case I have opted not to enforce security by supplying a value of false.

The meat of the server comes on line 12 where an object of type TestClass is registered. The RemotingConfiguration.RegisterWellKnownServiceType() method takes three arguments: The first is the *type* of the object to be hosted, the second is a string indicating the *service name* by which the object can be accessed, and the third argument specifies whether calls to the remote object's methods are handled by a new instance of the object (*SingleCall*) or by one object that persists across multiple service requests (*Singleton*). In this example, I am using the SingleCall mode.

To compile this program I recommend putting the TestClass.dll in the same directory as the RemotingServer.cs class code and issuing the following compiler command:

```
csc /r:TestClass.dll RemotingServer.cs
```

The /r switch tells the compiler to include the indicated resource during compilation. When you've compiled the server give it a whirl. Figure 19-2 shows the RemotingServer running and waiting for an incoming connection.
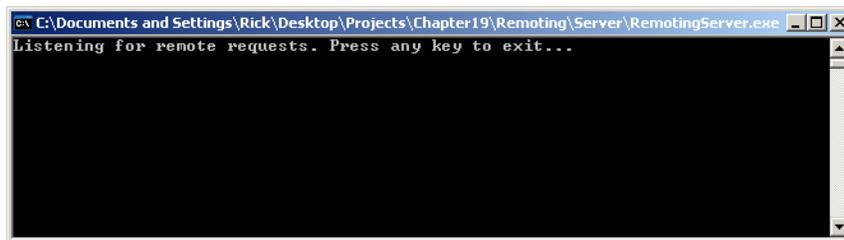

Figure 19-2: RemotingServer Waiting for Something to do

Finally, we need a client application that makes calls to the remote TestClass object hosted on the running RemotingServer application. Example 19.3 gives the code for the RemotingClient application.

19.3 RemotingClient.cs

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingClient {
7      public static void Main(){
8        try {
9        TcpChannel channel = new TcpChannel();
10       ChannelServices.RegisterChannel(channel, false);
11       TestClass test = (TestClass)Activator.GetObject(typeof(TestClass), "tcp://localhost:8080/TestClass" );
12       Console.WriteLine(test.Text);
13       test.Text = "This is a new string sent from the client application!";
14       Console.WriteLine(test.Text);
15        } catch(ArgumentNullException ane){
16          Console.WriteLine(ane);
17        } catch(RemotingException re){
18          Console.WriteLine(re);
19        } catch(Exception e){
20          Console.WriteLine(e);
21        }
22      }
23    }
```

Referring to Example 19.3 — on lines 9 and 10, a TcpChannel object is created and registered. The heavy lifting occurs on line 11 when an instance of the remote TestClass object is created with the help of the System.Activator.GetObject() method. This method takes two arguments: the *type* of object to create, and a string representing the *URL of the remote service*. In this example, as you'll recall from the RemotingServer code, the remote TestClass object is hosted on the server and is made available via a service named "TestClass" on port 8080. Thus, the URL given here must reflect that reality. Notice that the retrieved object must be cast to its proper type. Once this line of code executes, the client application can use the reference *test* as though the object it pointed to was on the local machine. All the network calls to the remote object are handled automatically by the .NET remoting infrastructure.

In this example, on line 12, I write the value of the Text property to the console. I then attempt to change the Text property by setting it to a new string value.

To compile this program, you'll need to make a copy of the TestClass.dll, put it in the same folder where you have the RemotingClient.cs code and issue the following compiler command:

```
csc /r:TestClass.dll RemotingClient.cs
```

Figure 19-3 shows the results of running the RemotingClient application several times. Make sure to start the RemotingServer before running RemotingClient. Also, I have assumed you're testing this application on the local machine (localhost). If you have two computers on a network, change localhost to the appropriate IP address and recompile the programs before you start the server and run the client application.

Figure 19-3: Results of Running RemotingServer and RemotingClient with a SingleCall Mode Remote Object

SingleCall vs. Singleton

Referring to Figure 19-3 — notice that each time the RemotingClient application executes it gets the original, or default, remote object's Text property message, even though it sets the remote object's Text property to a new value. This is because all requests to the remote object are handled by a new object instance. (Note that the remote object's Text property is being changed by examining the RemotingServer's console output.) This behavior was set when I registered the remote object in the server code by using the WellKnownObjectMode.*SingleCall* mode.

If you want the remote object to persist and maintain state between client service requests, use the WellKnownObjectMode.*Singleton* mode. Example 19.4 gives the code for a slightly modified version of Remoting-Server that registers the TestClass object in Singleton mode.

19.4 RemotingServer.cs (Mod 1)

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingServer {
7      public static void Main(){
8
```

```
9        try{
10       TcpChannel channel = new TcpChannel(8080);
11       ChannelServices.RegisterChannel(channel, false);
12       RemotingConfiguration.RegisterWellKnownServiceType(typeof(TestClass), "TestClass",
13                                               WellKnownObjectMode.Singleton);
14       Console.WriteLine("Listening for remote requests. Press any key to exit...");
15       Console.ReadLine();
16       } catch(ArgumentNullException ane){
17         Console.WriteLine("Channel argument was null!");
18         Console.WriteLine(ane);
19       } catch(RemotingException re){
20         Console.WriteLine("Channel has already been registered!");
21         Console.WriteLine(re);
22       } catch(Exception e){
23         Console.WriteLine(e);
24       }
25
26    }
27  }
```

Referring to Example 19.4 — the only difference between this and the previous version of RemotingServer appears on line 13 where I've registered the TestClass object in the WellKnownObjectMode.Singleton mode. The TestClass and RemotingClient code remain unchanged. Recompile the RemotingServer class and restart the server. Figure 19-4 shows the results of running the RemotingClient application several times. Note the different behavior. The remote object's Text property persists across service requests.

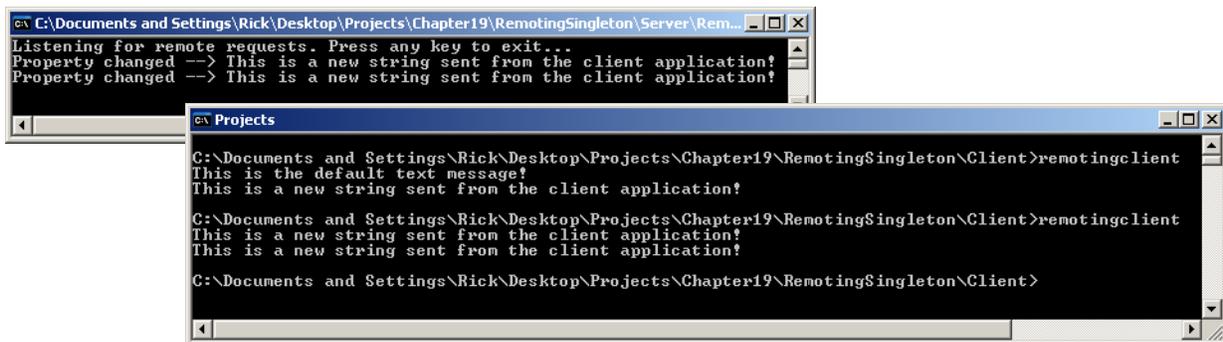Figure 19-4: Results of Hosting TestClass Remote Object in Singleton Mode

Accessing A Remote Object Via An Interface

In the previous remoting examples, the TestClass.dll file was deployed with both the client and server applications. This is required because you're accessing a class by name and the compiler must resolve that name by having access to the code where that name is defined.

As it turns out, because the .NET remoting infrastructure creates a proxy to the remote object, the proxy is, in effect, only an interface to the remote object. (The .NET remoting infrastructure builds the proxy from the TestClass definition contained within the TestClass.dll.) You can make internal changes to TestClass, recompile it, and deploy the dll on the server side, and the client will still run fine. If, however, you wanted to swap out the TestClass remote object with a remote object of a different type, you'd have to deploy the new dll to both the client and server applications and recompile them both. There is a better way — have TestClass implement an interface and deploy the interface to the remote client. That way, you can change the type of remote object anytime you need to, so long as it implements the interface expected by the remote client.

Let's see how this is done. I'm going to make a few changes to the code base. It all begins with the definition of an interface I'll call ITest, which is given in Example 19.5.

19.5 ITest.cs

```
1    using System;
2
3    public interface ITest {
4      string Text{
5        get;
6        set;
7      }
8    }
```

                   C# For Artists

Referring to Example 19.5 — the ITest interface is quite simple. It simply declares the read/write property named Text. Compile this interface into a dll using the following compiler command:

```
csc /t:library ITest.cs
```

Example 19.6 gives the code for the modified TestClass.

19.6 TestClass.cs (Mod 1)

```
1    using System;
2
3    public class TestClass : MarshalByRefObject, ITest {
4
5      private string _text;
6
7      public string Text {
8        get {  return _text; }
9        set {
10          _text = value;
11          Console.WriteLine("Property changed --> " + _text);
12          }
13      }
14
15      public TestClass():this("This is the new default text message!"){}
16
17      public TestClass(string s){
18        _text = s;
19      }
20   }
```

Referring to Example 19.6 — note now on line 3 that TestClass, in addition to extending MarshalByRefObject, implements the ITest interface. That's the only change to the TestClass code. Compile TestClass into a dll by using the following compiler command:

```
csc /t:library /r:ITest.dll TestClass.cs
```

Now, the RemotingServer code remains unchanged. The only thing you must do at this point is recompile RemotingServer and include a reference to both the TestClass.dll and ITest.dll files like so:

```
csc /r:TestClass.dll;ITest.dll RemotingServer.cs
```

The biggest changes are made to the RemotingClient application. Its code is given in Example 19.7.

19.7 RemotingClient.cs (Mod 1)

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingClient {
7      public static void Main(){
8        try {
9          TcpChannel channel = new TcpChannel();
10          ChannelServices.RegisterChannel(channel, false);
11          ITest test = (ITest)Activator.GetObject(typeof(ITest), "tcp://localhost:8080/TestClass" );
12          Console.WriteLine(test.Text);
13          test.Text = "This is a new string sent from the client application";
14          Console.WriteLine(test.Text);
15        } catch(ArgumentNullException ane){
16          Console.WriteLine("Channel argument was null!");
17          Console.WriteLine(ane);
18        } catch(RemotingException re){
19          Console.WriteLine("Channel has already been registered!");
20          Console.WriteLine(re);
21        } catch(Exception e){
22          Console.WriteLine(e);
23        }
24      }
25   }
```

Referring to Example 19.7 — note the changes made to line 11. The RemotingClient application is getting an instance of a remote object of type ITest. Also note that the URL to the remote object remains unchanged, and this is fine. What matters in the code is that you're referencing ITest, not TestClass. To compile this program you'll need to copy the ITest.dll file to the client directory (and delete the TestClass.dll) and use the following compiler command:

```
csc /r:ITest.dll RemotingClient.cs
```

Now, start up the server and run the RemotingClient application several times. You'll see that outwardly it behaves like the previous version of the program as is shown in Figure 19-5. Inwardly, however, you've built yourself a .NET remoting application that can more flexibly respond to object changes on the back-end. And we're going to take this flexibility one step further in the following section when I show you how to configure both the server and client applications with configuration files.
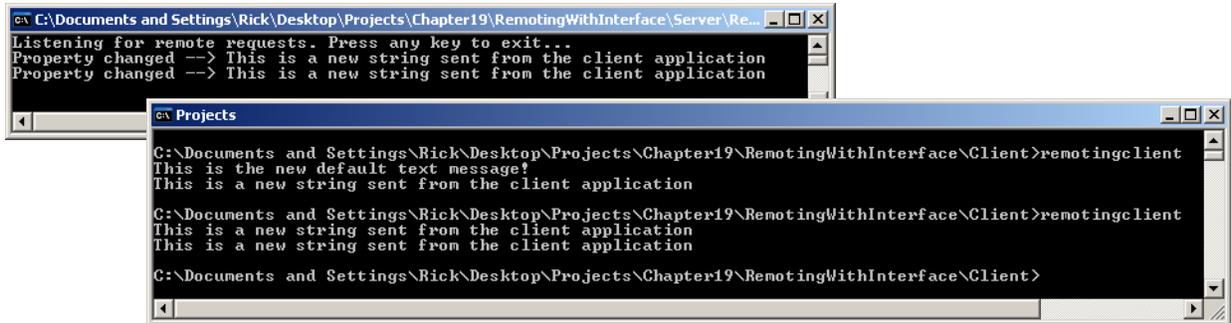
Figure 19-5: Results of Accessing a Remote Object via an Interface

Using Configuration Files

Both remote client and server applications can be configured via configuration files. A configuration file contains information about channels, remote object types, service names, etc. Using configuration files simplifies code and increases application flexibility by eliminating the need to recompile code when you want to make simple changes to certain application properties.

The use of configuration files requires changes to both the client and server remoting application code. The code for ITest and TestClass remains unchanged. Example 19.8 gives the code for the RemotingServer application modified to use a configuration file.

19.8 RemotingServer.cs (Mod 2)

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingServer {
7      public static void Main(){
8        try {
9          RemotingConfiguration.Configure("server.config", false);
10         Console.WriteLine("Listening for remote requests. Press any key to exit...");
11         Console.ReadLine();
12       } catch(Exception e){
13          Console.WriteLine(e);
14       }
15     }
16   }
```

Referring to Example 19.8 — the RemotingConfiguration.Configure() method specifies a configuration file named server.config. The content of the server.config configuration file is listed in Example 19.9.

19.9 server.config

```
1    <configuration>
2      <system.runtime.remoting>
3        <application>
4          <service>
5           <wellknown mode="Singleton" type="TestClass, TestClass" objectUri="TestClass" />
6          </service>
7          <channels>
8           <channel ref="tcp" port="8080" />
9          </channels>
10       </application>
11     </system.runtime.remoting>
12   </configuration>
```

Referring to Example 19.9 — the server.config file contains XML tags that represent remoting configuration settings. (A full description of the remoting configuration file schema can be found on the MSDN website.) Configuration settings for one or more services hosted on one or more channels appear within the <application></application> tags. In this example the TestClass remote object type is hosted on a TcpChannel on port 8080 with a service URI named TestClass. Note on line 5 the type attribute is set to "TestClass, TestClass". The first TestClass refers to the type name; the second TestClass refers to the application domain where TestClass can be found. (The typename and application domain are the same in this case.)

Use the following compiler command to compile the RemotingServer code:

```
csc /r:TestClass.dll;ITest.dll RemotingServer.cs
```

To run the RemotingServer application, make sure the server.config file is located in the same directory. (Otherwise, provide an absolute path name to the server.config file when you specify it in the call to the Configure() method.)

I made a few changes to the RemotingClient application as well. The modified code appears in Example 19.10.

19.10 RemotingClient.cs

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingClient {
7      public static void Main(){
8       try {
9         RemotingConfiguration.Configure("client.config", false);
10        WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
11        ITest test = (ITest)Activator.GetObject(typeof(ITest), client_types[ 0].ObjectUrl );
12        Console.WriteLine(test.Text);
13        test.Text = "This is a new string sent from the client application";
14        Console.WriteLine(test.Text);
15       } catch(Exception e){
16         Console.WriteLine(e);
17       }
18     }
19   }
```

Referring to Example 19.10 — the RemotingConfiguration.Configure() method is used to load a configuration file named client.config. Now, for maximum flexibility, on line 10, I have used the GetRegisteredWellKnowClientTypes() method to retrieve an array of registered client types. (In this example there is only one, as you'll see when you examine the client.config file.) I then use the client_types array to access the URL for the first registered client type, which in this case refers to the TestClass remote object service hosted on localhost port 8080. By doing this, and using the Activator.GetObject() method, I can change the client to access different remote objects without recompiling, provided those remote objects implement the ITest interface.

Example 19.11 lists the content of the client.config file.

19.11 client.config

```
1    <configuration>
2      <system.runtime.remoting>
3        <application>
4          <client>
5            <wellknown type="ITest, ITest" url="tcp://localhost:8080/TestClass" />
6          </client>
7        </application>
8      </system.runtime.remoting>
9    </configuration>
```

Use the following compiler command to compile the RemotingClient application:

<div align="center">csc /r:ITest.dll RemotingClient.cs</div>

To run the RemotingClient application, ensure the client.config file is in the same directory as the application. Figure 19-6 shows the results of running the RemotingServer and RemotingClient applications having been configured with configuration files.
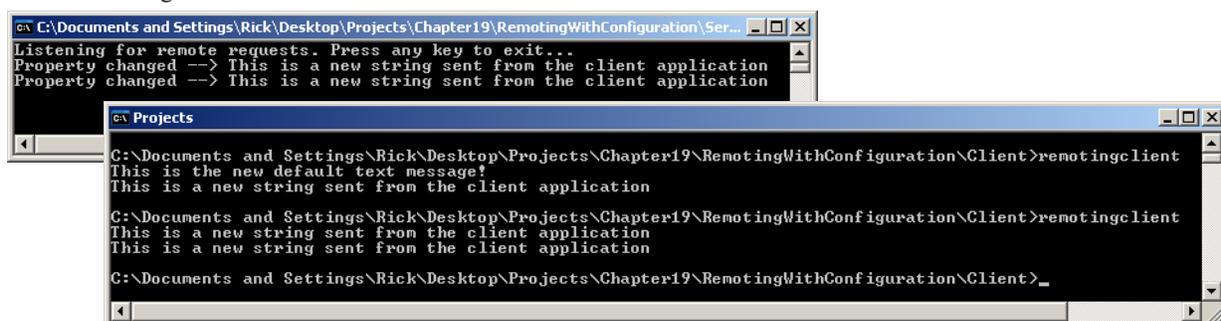

Figure 19-6: Results of Running RemotingServer and RemotingClient with Configuration Files

Passing Objects Between Client And Server

Remote object method calls can take parameters and return object's just like ordinary objects. A user-defined type intended for transmission across a network must be tagged with the Serializable attribute. Let's see how this is done. The following extended example shows how a remoting client can access a remoting server to get a list of Person objects. To keep things relatively simple, the remote object creates and populates a collection of Person objects. (These Person objects could easily be retrieved from a database, which you'll see done in the following chapter!)

Example 19.12 gives the code for the Person class, which is used in this application.

19.12 Person.cs

```
1    using System;
2
3    [ Serializable]
4    public class Person {
5
6      //enumeration
7      public enum Sex {MALE, FEMALE};
8
9      // private instance fields
10     private String   _firstName;
11     private String   _middleName;
12     private String   _lastName;
13     private Sex      _gender;
14     private DateTime _birthday;
15
16
17     //private default constructor
18     private Person(){}
19
20     public Person(String firstName, String middleName, String lastName,
21                   Sex gender, DateTime birthday){
22       FirstName = firstName;
23       MiddleName = middleName;
24       LastName = lastName;
25       Gender = gender;
26       BirthDay = birthday;
27     }
28
29     // public properties
30     public String FirstName {
31       get { return _firstName; }
32       set { _firstName = value; }
33     }
34
35     public String MiddleName {
36       get { return _middleName; }
37       set { _middleName = value; }
38     }
39
40     public String LastName {
41       get { return _lastName; }
42       set { _lastName = value; }
43     }
44
45     public Sex Gender {
46       get { return _gender; }
47       set { _gender = value; }
48     }
49
50     public DateTime BirthDay {
51       get { return _birthday; }
52       set { _birthday = value; }
53     }
54
55     public int Age {
56       get {
57        int years = DateTime.Now.Year - _birthday.Year;
58         int adjustment = 0;
59        if((DateTime.Now.Month <= _birthday.Month) && (DateTime.Now.Day < _birthday.Day)){
60          adjustment = 1;
61        }
62        return years - adjustment;
63       }
64      }
65
66     public String FullName {
67       get { return FirstName + " " + MiddleName + " " + LastName; }
68     }
```

```
69
70     public String FullNameAndAge {
71       get { return FullName + " " + Age; }
72     }
73
74     public override String ToString(){
75       return FullName + " is a " + Gender + " who is " + Age + " years old.";
76     }
77
78   } // end Person class
```

Referring to Example 19.12 — the Person class has been tagged as being serializable by the addition on line 3 of the Serializable attribute. Compile this class into a dll using the following compiler command:

<p align="center"><code>csc /t:library Person.cs</code></p>

Example 19.13 gives the code for the ISurrealistServer interface.

<p align="right">19.13 ISurrealistServer.cs</p>

```
1   using System;
2   using System.Collections.Generic;
3
4   public interface ISurrealistServer {
5     List<Person> GetSurrealists();
6   }
```

Referring to Example 19.13 — the ISurrealistServer interface declares one method named GetSurrealists(), which returns a list of Person objects. Compile this code into a dll by using the following compiler command:

<p align="center"><code>csc /t:library /r:Person.dll ISurrealistServer.cs</code></p>

Example 19.14 gives the code for the SurrealistServer class.

<p align="right">19.14 SurrealistServer.cs</p>

```
1   using System;
2   using System.Collections.Generic;
3
4   public class SurrealistServer : MarshalByRefObject, ISurrealistServer {
5
6     private List<Person> surrealists  = null;
7
8     public SurrealistServer(){
9       this.InitializeSurrealists();
10    }
11
12    public List<Person> GetSurrealists(){
13      Console.WriteLine("Request for surrealists received!");
14      return surrealists;
15    }
16
17    private void InitializeSurrealists(){
18      surrealists = new List<Person>();
19      Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
20       Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
21       Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
22       Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
23       Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
24       Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
25                                                  new DateTime(1887, 07, 28));
26
27      surrealists.Add(p1);
28      surrealists.Add(p2);
29      surrealists.Add(p3);
30      surrealists.Add(p4);
31      surrealists.Add(p5);
32      surrealists.Add(p6);
33    }
34
35  }
```

Referring to Example 19.14 — the SurrealistServer class extends MarshalByRefObject and implements the ISurrealistServer interface. It declares a private field named surrealists. It actually creates the list object and populates it with six Person objects in the body of the InitializeSurrealists() method. The GetSurrealists() method simply returns the list object. So, over the network, the entire list of Person objects is returned to the client application when it calls this method.

Next, let's make some changes to the server.config file, as are shown in Example 19.15

<p align="right">19.15 server.config</p>

```
1   <configuration>
2     <system.runtime.remoting>
3       <application>
4         <service>
5           <wellknown mode="Singleton" type="SurrealistServer, SurrealistServer"
```

```
6                                              objectUri="SurrealistServer" />
7          </service>
8          <channels>
9            <channel ref="tcp" port="8080" />
10         </channels>
11       </application>
12     </system.runtime.remoting>
13   </configuration>
```

Referring to Example 19.15 — the changes made to the server.config file reflect the new name of the remote object class and the name of the service by which it can be accessed. These changes appear in lines 5 and 6.

Finally, the code for RemotingServer remains unchanged from the last example, but I repeat it here for continuity in Example 19.16.

19.16 RemotingServer.cs

```
1    using System;
2    using System.Runtime.Remoting;
3    using System.Runtime.Remoting.Channels;
4    using System.Runtime.Remoting.Channels.Tcp;
5
6    public class RemotingServer {
7      public static void Main(){
8        try {
9          RemotingConfiguration.Configure("server.config", false);
10         Console.WriteLine("Listening for remote requests. Press any key to exit...");
11         Console.ReadLine();
12       } catch(Exception e){
13         Console.WriteLine(e);
14       }
15     }
16   }
```

To compile this application, make sure the Person.dll, ISurrealistServer.dll, and SurrealistServer.dll files are in the same directory and use the following compiler command:

```
csc /r:Person.dll;ISurrealistServer.dll;SurrealistServer.dll RemotingServer.cs
```

When you have finished compiling the RemotingServer application you can start the server. It's time now to write the code for the RemotingClient application. The RemotingClient code is given in Example 19.17.

19.17 RemotingClient.cs

```
1    using System;
2    using System.Collections.Generic;
3    using System.Runtime.Remoting;
4    using System.Runtime.Remoting.Channels;
5    using System.Runtime.Remoting.Channels.Tcp;
6
7    public class RemotingClient {
8      public static void Main(){
9        try {
10         RemotingConfiguration.Configure("client.config", false);
11         WellKnownClientTypeEntry[] client_types = RemotingConfiguration.GetRegisteredWellKnownClientTypes();
12         ISurrealistServer surrealist_server =
13             (ISurrealistServer)Activator.GetObject(typeof(ISurrealistServer), client_types[ 0 ].ObjectUrl );
14
15         List<Person> surrealists = surrealist_server.GetSurrealists();
16         foreach(Person p in surrealists){
17           Console.WriteLine(p);
18         }
19       } catch(Exception e){
20         Console.WriteLine(e);
21       }
22     }
23   }
```

Referring to Example 19.17 — the RemotingClient gets its configuration from the client.config file, which is given in the next example. It then gets a reference to an ISurrealistServer object and calls its GetSurrealists() method. It then iterates over the list of Person objects in the body of the `foreach` statement on line 16 and writes each object's ToString() data to the console.

Example 19.18 gives the contents of the client.config file.

19.18 client.config

```
1    <configuration>
2      <system.runtime.remoting>
3        <application>
4          <client>
5            <wellknown type="ISurrealistServer, ISurrealistServer"
6                       url="tcp://localhost:8080/SurrealistServer" />
7          </client>
8        </application>
9      </system.runtime.remoting>
```

```
10    </configuration>
```

Referring to Example 19.18 — the changes to the client.config file appear on lines 5 and 6 and reflect the name of the remote object type and the service where it can be found.

To compile the RemotingClient application, make copies of the Person.dll and ISurrealistServer.dll, place them in the client code directory, and use the following compiler command:

```
csc /r:Person.dll;ISurrealistServer.dll RemotingClient.cs
```

Figure 19-7 shows the results of running the RemotingServer and RemotingClient applications.
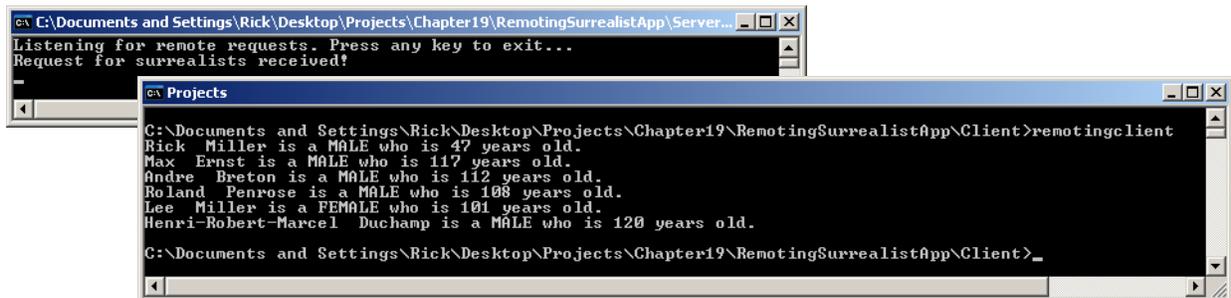
Figure 19-7: Results of Sending a Collection of Person Objects to a Remoting Client

Quick Review

All .NET remoting applications have three common components, regardless of their complexity: a *remotable object*, a *server application* that hosts the remotable object and handles incoming service requests, and a *client application* that utilizes the services of the remotely-hosted object.

A remotable object is created from a class that inherits from System.MarshalByRefObject. This enables the object to be shared across application domains in .NET remoting applications. The remotable object can be simple or complex.

The remoting server application hosts the remotable object and makes its services available via a *channel*. There are three primary channel types: *TcpChannel*, *HttpChannel*, and *IpcChannel*. The IpcChannel is used for *inter-process communication* between client and server applications hosted on the same machine.

The remoting client application accesses the services of the remote object via a *proxy* created automatically by the .NET remoting infrastructure. Once a remoting client application creates a reference to a remote object, it uses the services of the remote object, via the remote object's proxy, as if the remote object were a local object. The underlying complexities associated with calling the remote object's methods or properties are handled automatically by the .NET remoting infrastructure. Remote objects accessed in this manner must extend MarshalByRefObject and any other interfaces as required.

Remote objects can be hosted in *SingleCall* or *Singleton* mode. In SingleCall mode, a new remote object is used to respond to each client service request. In Singleton mode, remote objects persist and maintain state across multiple client service requests.

Remoting client applications can access the services of remote objects via one or more of the remote object's interfaces. This makes changing the implementation of the remote object easier, as long as the new object implements one of the interfaces expected by the client application.

For maximum deployment flexibility, place .NET remoting application deployment data in *configuration files*.

Complex objects sent between remoting client and server applications must be tagged as being serializable by using the *Serializable* attribute.

Client-Server Applications With TcpListener And TcpClient

In this section you'll get a little more down in the weeds with network programming by using the *TcpListener* and *TcpClient* classes to create *client-server applications*. Unlike .NET remoting, you'll need to know how to handle the details of establishing a network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

TCP/IP Client-Server Overview

The steps required to write a TCP/IP client-server application using the System.Net.TcpListener and System.Net.TcpClient classes are highlighted in the following illustrations.
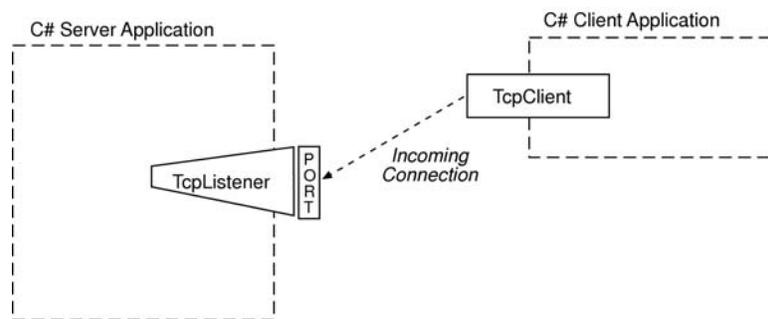
Figure 19-8: Server Application Listens on a Host and Port for Incoming TcpClient Connections

Referring to Figure 19-8 — a server application uses a TcpListener object to listen for incoming TcpClient connections on a particular IP address (or multiple IP addresses) and port number. The client application uses a TcpClient object to connect to a particular machine, given its IP address or DNS name (*i.e.*, www.warrenworks.com) which is then mapped to an IP address, and specified port number.
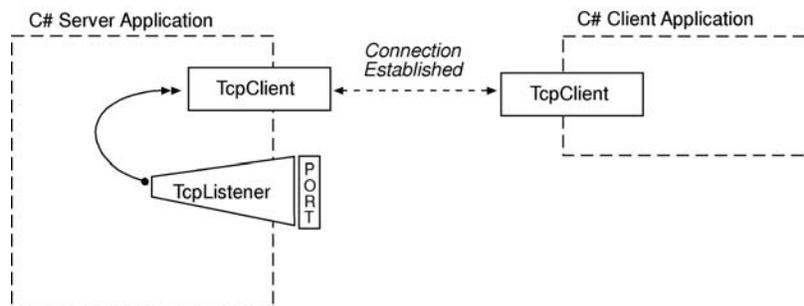
Figure 19-9: TcpListener Accepts Incoming TcpClient Connection

Referring to Figure 19-9 — when the TcpListener object detects an incoming TcpClient connection, it "accepts" the connection, which results in the creation of a server-side TCPclient object. Client-server communication takes place between the server-side and client-side TcpClient objects, as is shown in Figure 19-10.

Both the TcpListener and TcpClient objects provide *wrappers* around *socket* objects. You could use socket objects directly to conduct client-server communication, but doing so is beyond the scope of this book. To learn more about socket programming check out the excellent book *TCP/IP Sockets In C#: Practical Guide for Programers* by David B. Makofske, et. al., ISBN-13: 978-0-12-466051-9.
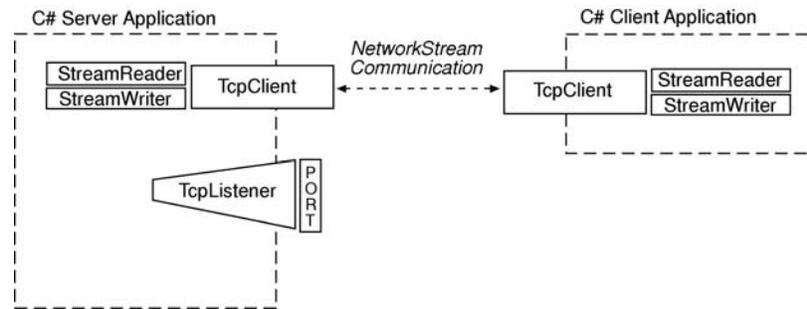
Figure 19-10: TcpClients Communicate via a NetworkStream using StreamReader and StreamWriter Objects

A Simple Client-Server Application

OK, let's put some of what you just learned in the previous section into practical use. The following examples implement a simple client-server application using the TcpListener and TcpClient classes. The application consists of two parts: an EchoServer, which listens for incoming TcpClient connections, and an EchoClient which connects to an EchoServer. When a connection between the EchoServer and EchoClient is established, messages sent from the client to the server are written to the server console and then sent back to the client for display on the client console. Example 19.19 gives the code for the EchoServer application.

19.19 EchoServer.cs

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5
6    public class EchoServer {
7      public static void Main(){
8        TcpListener listener = null;
9        try {
10         listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
11         listener.Start();
12         Console.WriteLine("EchoServer started...");
13         while(true){
14           Console.WriteLine("Waiting for incoming client connections...");
15           TcpClient client = listener.AcceptTcpClient();
16           Console.WriteLine("Accepted new client connection...");
17           StreamReader reader = new StreamReader(client.GetStream());
18           StreamWriter writer = new StreamWriter(client.GetStream());
19           String s = String.Empty;
20           while(!(s = reader.ReadLine()).Equals("Exit")){
21             Console.WriteLine("From client -> " + s);
22             writer.WriteLine("From server -> " + s);
23             writer.Flush();
24           }
25           reader.Close();
26           writer.Close();
27           client.Close();
28         }
29       } catch(Exception e){
30         Console.WriteLine(e);
31       } finally{
32         if(listener != null){
33           listener.Stop();
34         }
35       }
36     } // end Main()
37   } // end class definition
```

Referring to Example 19.19 — notice first the list of namespaces required for this particular application. It includes System.IO, System.Net, and System.Net.Sockets. The EchoServer application starts by creating an instance of TcpListener, which listens on the local machine IP address of 127.0.0.1 port 8080. (Make sure the port you choose is not in use.) The listener is then started on line 11 by a call to its Start() method. Incoming client connections are processed in the body of the while loop, which begins on line 13. On line 15, the listener.AcceptTcpClient() method blocks at that point until it detects an incoming TcpClient connection, at which time it unblocks and returns an instance of TcpClient and assigns it to the client reference. The term *block* refers to a *blocking I/O operation*. The

EchoServer application effectively stops everything until the AcceptTcpClient() method returns, at which time processing continues.

When the listener detects the incoming TcpClient connection, the application prints a short message stating so to the console, and then, on lines 17 and 18, it creates StreamReader and StreamWriter objects using the client.GetStream() method. The server uses these StreamReader and StreamWriter objects to communicate with the client. On line 19, the application creates a string variable named *s* and uses it to store incoming client strings. The body of the `while` loop, which begins on line 20, processes client-server communication by reading the incoming client string, printing it to the server's console, and then sending it back to the client via the writer.WriteLine() method. Note on line 23 the writer.Flush() method must be called to actually send the string on its way. The `while` loop repeats until the incoming string equals "Exit", at which time the EchoServer returns to listening for new incoming TcpClient connections.

Example 19.20 gives the code for the EchoClient application.

19.20 EchoClient.cs

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5
6    public class EchoClient {
7      public static void Main(){
8        try {
9          TcpClient client = new TcpClient("127.0.0.1", 8080);
10         StreamReader reader = new StreamReader(client.GetStream());
11         StreamWriter writer = new StreamWriter(client.GetStream());
12         String s = String.Empty;
13         while(!s.Equals("Exit")){
14          Console.Write("Enter a string to send to the server: ");
15          s = Console.ReadLine();
16          Console.WriteLine();
17          writer.WriteLine(s);
18          writer.Flush();
19          String server_string = reader.ReadLine();
20          Console.WriteLine(server_string);
21         }
22         reader.Close();
23         writer.Close();
24         client.Close();
25        } catch(Exception e){
26           Console.WriteLine(e);
27        }
28      } // end Main()
29    } // end class definition
```

Referring to Example 19.20 — the EchoClient application creates a TcpClient object that connects to the IP address 127.0.0.1 port 8080. If all goes well, lines 10 and 11 execute and the application creates the StreamReader and StreamWriter objects, which it uses to communicate with the server. On line 12, a string variable named *s* is created and used to send data to the server and to control the processing of the `while` loop, which starts on the following line. On line 15, the Console.ReadLine() method reads a line of text from the console and assigns it to s. On lines 17 and 18, it sends the string s to the server with calls to writer.WriteLine() and writer.Flush(). It then immediately reads the server's response with a call to the reader.ReadLine() method, which assigns the incoming string to the string variable named server_string and then prints the value of server_string to the console. The EchoServer application repeats this processing loop until the user enters the string "Exit" at the console.

To run this application, compile the EchoServer.cs and EchoClient.cs files, start the EchoServer, then run the EchoClient application. Figure 19-11 shows the results of running these applications.

Building A Multithreaded Server

While the previous example served well to illustrate basic client-server principles, the server application in its current form is only able to communicate with one client at a time. In this section, I'll show you how to make a few modifications to EchoServer that will enable it to serve multiple clients simultaneously. A server application that can process multiple simultaneous client connections is referred to as a *multithreaded* server.

The following general steps are required to turn EchoServer into a MultiThreadedEchoServer:

      Step 1: Create a separate client processing method that handles network stream communication and
            other applicable processing between server and client applications.

Figure 19-11: Results of Running the EchoClient and EchoServer Applications

Step 2: For each incoming client connection, spawn a separate thread, passing to it the name of the client processing method.

That's it! Let's see how these modifications look in the code. Example 19.21 gives the code for the MultiThread-edEchoServer class.

19.21 MultiThreadedEchoServer.cs

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5    using System.Threading;
6
7    public class MultiThreadedEchoServer {
8
9      private static void ProcessClientRequests(Object argument){
10       TcpClient client = (TcpClient)argument;
11       try {
12         StreamReader reader = new StreamReader(client.GetStream());
13         StreamWriter writer = new StreamWriter(client.GetStream());
14         String s = String.Empty;
15         while(!(s = reader.ReadLine()).Equals("Exit")){
16           Console.WriteLine("From client -> " + s);
17           writer.WriteLine("From server -> " + s);
18           writer.Flush();
19         }
20         reader.Close();
21         writer.Close();
22         client.Close();
23         Console.WriteLine("Closing client connection!");
24       } catch(IOException){
25         Console.WriteLine("Problem with client communication. Exiting thread.");
26       } finally{
27         if(client != null){
28           client.Close();
29         }
30       }
31     }
32
33     public static void Main(){
34       TcpListener listener = null;
35       try {
36         listener = new TcpListener(IPAddress.Parse("127.0.0.1"), 8080);
37         listener.Start();
38         Console.WriteLine("MultiThreadedEchoServer started...");
39         while(true){
40           Console.WriteLine("Waiting for incoming client connections...");
41           TcpClient client = listener.AcceptTcpClient();
42           Console.WriteLine("Accepted new client connection...");
43           Thread t = new Thread(ProcessClientRequests);
44           t.Start(client);
45         }
46       } catch(Exception e){
47         Console.WriteLine(e);
48       } finally{
49         if(listener != null){
50           listener.Stop();
51         }
```

```
52          }
53     } // end Main()
54  } // end class definition
```

Referring to Example 19.21 — first, a new namespace, System.Threading, has been added to list of using directives to gain access to the Thread class. I created a new method on line 9 named ProcessClientRequests(). Note that this method takes one argument of type Object, which is cast immediately to a TcpClient object. The reason this cast is necessary is because the ProcessClientRequests() method has the signature of a ParameterizedThreadStart delegate, which specifies one argument of type Object. You'll see how this method is actually used in the body of the Main() method.

I copied the bulk of the ProcessClientRequests() method from the previous version of EchoClient starting with the creation of the StreamReader and StreamWriter objects. It includes the whole of the second, or inner, `while` loop. I enclosed the method's code within its own `try/catch/finally` block because once the separate thread begins execution, it must handle any exceptions it generates.

In the body of the Main() method, the TcpListener object is created as before on line 36 and is started on line 37. The `while` loop beginning on line 39 repeats forever waiting for incoming client connections. When it detects an incoming client connection, the AcceptTcpClient() method returns a reference to a new TcpClient object and processing continues with the creation of a new Thread object on line 43. The name of the method this thread will execute, ProcessClientRequests, is passed to the Thread constructor. An alternative call to the Thread constructor could look like this:

```
Thread t = new Thread(new ParameterizedThreadStart(ProcessClientRequests));
```

In this example, the ParameterizedThreadStart delegate object is explicitly created and passed to the Thread constructor. The new thread is started with a call to t.Start() on line 44, passing to it the reference to the TcpClient object named *client*. When line 44 completes execution, the `while` loop continues and the server returns to listening for incoming client connections.

You now have a multithreaded server application! The code for EchoClient, given in the previous section, remains unchanged.

Figure 19-12 shows the results of running the MultiThreadedServer and connecting to it from two EchoClient applications.
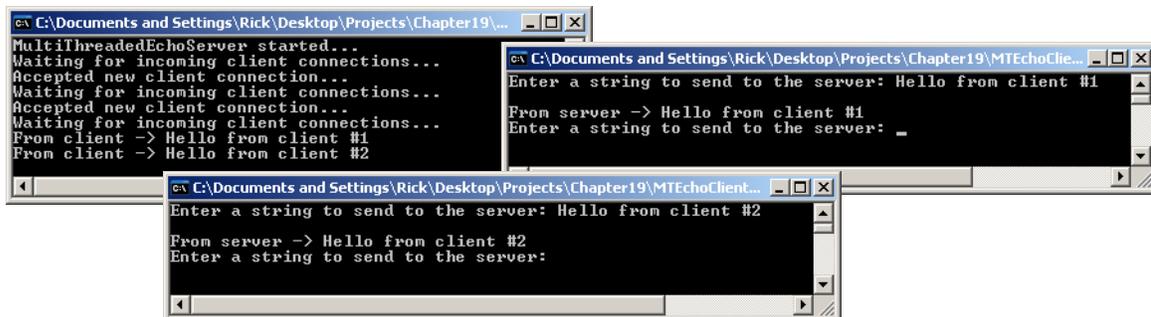

Figure 19-12: Two Clients Connected to MultiThreadedClientServer

Listening On Multiple IP Addresses

The MultiThreadedServer has some nice functionality, but in its current form the TcpListener binds to only one server IP address, which in the previous examples has been the local loopback adapter 127.0.0.1. It would be nice to listen on several IP addresses simultaneously. The only change necessary to do this is to use "IPAddress.Any" when creating the TcpListener object. Example 19.22 gives the code for the MultiIPEchoServer class.

19.22 MultiIPEchoServer.cs

```
1   using System;
2   using System.Drawing;
3   using System.IO;
4   using System.Net;
5   using System.Net.Sockets;
6   using System.Net.NetworkInformation;
7   using System.Threading;
8
9   public class MultiIPEchoServer {
10
```

                   C# For Artists

```
11     private static void ProcessClientRequests(Object argument){
12       TcpClient client = (TcpClient)argument;
13       try {
14         StreamReader reader = new StreamReader(client.GetStream());
15         StreamWriter writer = new StreamWriter(client.GetStream());
16         String s = String.Empty;
17         while(!(s = reader.ReadLine()).Equals("Exit")){
18           Console.WriteLine("From client -> " + s);
19           writer.WriteLine("From server -> " + s);
20           writer.Flush();
21         }
22         reader.Close();
23         writer.Close();
24         client.Close();
25         Console.WriteLine("Client connection closed!");
26       } catch(IOException){
27         Console.WriteLine("Problem with client communication. Exiting thread.");
28       } finally{
29          if(client != null){
30            client.Close();
31          }
32       }
33     }
34
35     private static void ShowServerNetworkConfig(){
36       Console.ForegroundColor = ConsoleColor.Yellow;
37       NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
38       foreach(NetworkInterface adapter in adapters){
39         Console.WriteLine(adapter.Description);
40         Console.WriteLine("\tAdapter Name: " + adapter.Name);
41         Console.WriteLine("\tMAC Address: " + adapter.GetPhysicalAddress());
42         IPInterfaceProperties ip_properties = adapter.GetIPProperties();
43         UnicastIPAddressInformationCollection addresses = ip_properties.UnicastAddresses;
44         foreach(UnicastIPAddressInformation address in addresses){
45           Console.WriteLine("\tIP Address: " + address.Address);
46         }
47       }
48       Console.ForegroundColor = ConsoleColor.White;
49     }
50
51     public static void Main(){
52       TcpListener listener = null;
53       try {
54         ShowServerNetworkConfig();
55         listener = new TcpListener(IPAddress.Any, 8080);
56         listener.Start();
57         Console.WriteLine("MultiIPEchoServer started...");
58         while(true){
59           Console.WriteLine("Waiting for incoming client connections...");
60           TcpClient client = listener.AcceptTcpClient();
61           Console.WriteLine("Accepted new client connection...");
62           Thread t = new Thread(ProcessClientRequests);
63           t.Start(client);
64         }
65       } catch(Exception e){
66          Console.WriteLine(e);
67       } finally{
68          if(listener != null){
69            listener.Stop();
70          }
71        }
72     } // end Main()
73   } // end class definition
```

Referring to Example 19.22 — I have added another method to the server code named ShowServerNetworkConfig() which begins on line 35. I've also added another namespace, System.Net.NetworkInformation, to the list of `using` directives.

Referring to the ShowServerNetworkConfig() method — the first thing it does is set Console.ForegroundColor to Color.Yellow. This makes the network information stand out from the ordinary client-server interaction messages. Next, on line 37, the NetworkInterface.GetAllNetworkInterfaces() method is called. This returns an array of NetworkInterface objects. The `foreach` statement starting on line 38 iterates over the array of NetworkInterface objects and prints out various properties about each one including the interface's *Description*, *Name*, and *Physical* or *MAC* addresses. On line 42, I create an IPInterfaceProperties object with the help of the adapter.GetIPProperties() method and use it to get a collection of UnicastIPAddressInformation objects for each adapter. The `foreach` loop starting on line 44 iterates over the collection of UnicastIPAddressInformation objects and prints each IP address to the console. Finally, the method concludes by resetting the Console.ForegroundColor to Color.White.

The only changes to the Main() method include the addition of line 54, where I make a call to the ShowServerNetworkConfig() method, and on line 55 where I bind the TcpListener object to all available machine IP addresses by using IPAddress.Any.

The EchoClient has been changed to connect to an IP address given in the form of a command-line argument when the program executes. The code for the modified EchoClient class is given in Example 19.23.

19.23 EchoClient.cs (Mod 1)

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5
6    public class EchoClient {
7      public static void Main(String[] args){
8       IPAddress ip_address = IPAddress.Parse("127.0.0.1"); //default
9       int port = 8080;
10      try{
11        if(args.Length >= 1){
12          ip_address = IPAddress.Parse(args[ 0]);
13        }
14      } catch(FormatException){
15        Console.WriteLine("Invalid IP address entered. Using default IP of: " + ip_address.ToString());
16      }
17      try {
18       Console.WriteLine("Attempting to connect to server at IP address: {0}  port: {1}",
19                                                 ip_address.ToString(), port);
20       TcpClient client = new TcpClient(ip_address.ToString(), port);
21       Console.WriteLine("Connection successful!");
22       StreamReader reader = new StreamReader(client.GetStream());
23       StreamWriter writer = new StreamWriter(client.GetStream());
24       String s = String.Empty;
25       while(!s.Equals("Exit")){
26        Console.Write("Enter a string to send to the server: ");
27        s = Console.ReadLine();
28        Console.WriteLine();
29        writer.WriteLine(s);
30        writer.Flush();
31        if(!s.Equals("Exit")){
32          String server_string = reader.ReadLine();
33          Console.WriteLine(server_string);
34        }
35       }
36       reader.Close();
37       writer.Close();
38       client.Close();
39      } catch(Exception e){
40        Console.WriteLine(e);
41      }
42     } // end Main()
43   } // end class definition
```

Referring to Example 19.23 — the EchoClient class now checks the argument array for the presence of a valid IP address. If the given IP address is malformed, the IPAddress.Parse() method throws an exception and assigns the default IP address value of 127.0.0.1 to the ip_address field. On line 20, the ip_address field is used in the TcpClient constructor call where it is converted into a string. The remainder of the code remains unchanged from the previous example.

To run these versions of the client and server applications, compile the code and start the MultiIPEchoServer, then run the modified EchoClient application. The results of these changes can be seen in Figure 19-13. Note how each client connects to the server via a different IP address.

Sending Objects Between Client And Server

In the previous examples, I've limited the exchange between client and server application to strings. In this section I'll show you how to serialize a complex object on the server side and send it to the client for deserialization. Remember that in the case of .NET remoting applications, the hard work of serializing complex objects is done for you by the remoting framework. Not here, no, no, no. If you want to serialize a complex object and send it across the network you'll need to get your hands dirty.

The following two examples implement a SurrealistEchoServer. The application actually consists of three classes: SurrealistEchoServer.cs, SurrealistDB.cs, and Person.cs, which is not repeated here. Example 19.24 gives the code for the SurrealistEchoServer class.

                   C# For Artists

Figure 19-13: Results of Running MultiIPEchoServer and EchoClient (Mod 1) Applications

19.24 SurrealistEchoServer.cs

```
1     using System;
2     using System.Drawing;
3     using System.IO;
4     using System.Net;
5     using System.Net.Sockets;
6     using System.Net.NetworkInformation;
7     using System.Threading;
8     using System.Runtime.Serialization;
9     using System.Runtime.Serialization.Formatters.Binary;
10
11    public class SurrealistEchoServer {
12
13      private static void ProcessClientRequests(Object argument){
14        TcpClient client = (TcpClient)argument;
15        try {
16          StreamReader reader = new StreamReader(client.GetStream());
17          StreamWriter writer = new StreamWriter(client.GetStream());
18          String s = String.Empty;
19          while(!(s = reader.ReadLine()).Equals("Exit")){
20            switch(s){
21              case "GetSurrealists" : {
22                Console.WriteLine("From client -> " + s);
23                SerializeSurrealists(client.GetStream());
24                client.GetStream().Flush();
25                break;
26              }
27              default: {
28                Console.WriteLine("From client -> " + s);
29                writer.WriteLine("From server -> " + s);
30                writer.Flush();
31                break;
32              }
33            } // end switch
34          } // end while
35          reader.Close();
```

```
36            writer.Close();
37            client.Close();
38            Console.WriteLine("Client connection closed!");
39         } catch(IOException){
40            Console.WriteLine("Problem with client communication. Exiting thread.");
41         } catch(NullReferenceException){
42            Console.WriteLine("Incoming string was null! Client may have terminated prematurly.");
43         } catch(Exception e){
44            Console.WriteLine("Unknown exception occured.");
45            Console.WriteLine(e);
46         } finally{
47            if(client != null){
48               client.Close();
49            }
50         }
51      } // end ProcessClientRequests()
52
53      private static void SerializeSurrealists(NetworkStream stream){
54         SurrealistDB db = new SurrealistDB();
55         BinaryFormatter bf = new BinaryFormatter();
56         bf.Serialize(stream, db.GetSurrealists());
57      } // end SerializeSurrealists()
58
59      private static void ShowServerNetworkConfig(){
60         Console.ForegroundColor = ConsoleColor.Yellow;
61         NetworkInterface[] adapters = NetworkInterface.GetAllNetworkInterfaces();
62         foreach(NetworkInterface adapter in adapters){
63            Console.WriteLine(adapter.Description);
64            Console.WriteLine("\tAdapter Name: " + adapter.Name);
65            Console.WriteLine("\tMAC Address: " + adapter.GetPhysicalAddress());
66            IPInterfaceProperties ip_properties = adapter.GetIPProperties();
67            UnicastIPAddressInformationCollection addresses = ip_properties.UnicastAddresses;
68            foreach(UnicastIPAddressInformation address in addresses){
69               Console.WriteLine("\tIP Address: " + address.Address);
70            }
71         }
72         Console.ForegroundColor = ConsoleColor.White;
73      } // end ShowServerNetworkConfig()
74
75      public static void Main(){
76         TcpListener listener = null;
77         try {
78            ShowServerNetworkConfig();
79            listener = new TcpListener(IPAddress.Any, 8080);
80            listener.Start();
81            Console.WriteLine("SurrealistEchoServer started...");
82            while(true){
83               Console.WriteLine("Waiting for incoming client connections...");
84               TcpClient client = listener.AcceptTcpClient();
85               Console.WriteLine("Accepted new client connection...");
86               Thread t = new Thread(ProcessClientRequests);
87               t.Start(client);
88            }
89         } catch(Exception e){
90            Console.WriteLine(e);
91         } finally{
92            if(listener != null){
93               listener.Stop();
94            }
95         }
96      } // end Main()
97   } // end class definition
```

Referring to Example 19.24 — first, note the addition of several namespaces required to perform object serialization. These include System.Runtime.Serialization and System.Runtime.Serialization.Formatters.Binary.

The SurrealistEchoServer class's ProcessClientRequests() method has been slightly modified. It echoes client strings as before, but if the client string equals "GetSurrealists", it returns to the client a serialized collection of Person objects. It does this with a call to its SerializeSurrealists() method, which begins on line 53.

The SerializeSurrealists() method takes a NetworkStream object as an argument. On line 54, it creates an instance of SurrealistsDB followed by the creation of a BinaryFormatter object on the next line. The BinaryFormatter serializes the List<Person> object returned by the db.GetSurrealists() method into the stream. When the SerializeSurrealists() method returns, the network stream is flushed to send the collection of People objects on their way to the client.

Example 19.25 gives the code for the SurrealistDB class.

```
1    using System;
2    using System.Collections.Generic;
3
4    public class SurrealistDB {
5
6      private List<Person> surrealists = null;
7
8      public SurrealistDB(){
9        this.InitializeSurrealists();
10     }
11
12     public List<Person> GetSurrealists(){
13       return surrealists;
14     }
15
16     private void InitializeSurrealists(){
17       surrealists = new List<Person>();
18       Person p1 = new Person("Rick", "", "Miller", Person.Sex.MALE, new DateTime(1961, 02, 04));
19        Person p2 = new Person("Max", "", "Ernst", Person.Sex.MALE, new DateTime(1891, 04, 02));
20        Person p3 = new Person("Andre", "", "Breton", Person.Sex.MALE, new DateTime(1896, 02, 19));
21        Person p4 = new Person("Roland", "", "Penrose", Person.Sex.MALE, new DateTime(1900, 10, 14));
22        Person p5 = new Person("Lee", "", "Miller", Person.Sex.FEMALE, new DateTime(1907, 04, 23));
23        Person p6 = new Person("Henri-Robert-Marcel", "", "Duchamp", Person.Sex.MALE,
24                                                      new DateTime(1887, 07, 28));
25
26       surrealists.Add(p1);
27       surrealists.Add(p2);
28       surrealists.Add(p3);
29       surrealists.Add(p4);
30       surrealists.Add(p5);
31       surrealists.Add(p6);
32     }
33   } // end class definition
```

Referring to Example 19.25 — The SurrealistDB class initializes a list of People objects and provides a GetSurrealists() method which returns the populated list. (**Note:** This class could easily have been written to connect to a data base to fetch the required information. You'll see how that's done in Chapter 20.)

The EchoClient class must be modified to accept and deserialize the incoming list of People objects. Example 19.26 gives the code for the modified EchoClient class.

```
1    using System;
2    using System.IO;
3    using System.Net;
4    using System.Net.Sockets;
5    using System.Runtime.Serialization;
6    using System.Runtime.Serialization.Formatters.Binary;
7    using System.Collections.Generic;
8
9    public class EchoClient {
10
11
12     static List<Person> DeserializeSurrealists(NetworkStream stream){
13       BinaryFormatter bf = new BinaryFormatter();
14       return (List<Person>)bf.Deserialize(stream);
15     }
16
17     static void WriteSurrealistDataToConsole(List<Person> surrealists){
18       foreach(Person p in surrealists){
19         Console.WriteLine(p);
20       }
21     }
22
23     public static void Main(String[] args){
24      IPAddress ip_address = IPAddress.Parse("127.0.0.1"); //default
25      int port = 8080;
26      try{
27        if(args.Length >= 1){
28          ip_address = IPAddress.Parse(args[0]);
29        }
30      } catch(FormatException){
31        Console.WriteLine("Invalid IP address entered. Using default IP of: " + ip_address.ToString());
32      }
33      try {
34       Console.WriteLine("Attempting to connect to server at IP address: {0} port: {1}",
35                                                    ip_address.ToString(), port);
36       TcpClient client = new TcpClient(ip_address.ToString(), port);
37       Console.WriteLine("Connection successful!");
38       StreamReader reader = new StreamReader(client.GetStream());
```

```
39        StreamWriter writer = new StreamWriter(client.GetStream());
40        String s = String.Empty;
41        while(!s.Equals("Exit")){
42         Console.Write("Enter \"GetSurrealists\" to retrieve list from server: ");
43         s = Console.ReadLine();
44         Console.WriteLine();
45         switch(s){
46           case "GetSurrealists" : {
47              writer.WriteLine(s);
48              writer.Flush();
49              WriteSurrealistDataToConsole(DeserializeSurrealists(client.GetStream()));
50              Console.WriteLine();
51              break;
52           }
53           case "Exit" : {
54              writer.WriteLine(s);
55              writer.Flush();
56              break;
57           }
58           default: {
59              writer.WriteLine(s);
60              writer.Flush();
61              String server_string = reader.ReadLine();
62              Console.WriteLine(server_string);
63              Console.WriteLine();
64              break;
65           }
66         }
67        }
68        reader.Close();
69        writer.Close();
70        client.Close();
71        } catch(Exception e){
72           Console.WriteLine(e);
73        }
74     } // end Main()
75   } // end class definition
```

Referring to Example 19.26 — the modified EchoClient application sends strings to the server as before. When the string it sends equals "GetSurrealists" the server returns a serialized list of People objects. (*i.e.*, List<People>) The client must then deserialize the object and cast it to its expected type, which it does with the DeserializeSurrealists() method. Once the list of People objects is deserialized, the EchoClient application calls the WriteSurrealistDataToConsole() method. All this action takes place on line 49!

To run these applications, copy the Person.dll into both client and server directories, then change to the server directory and compile the server application using the following compiler commands.

First compile the SurrealistDB class into a dll:

<div align="center">csc /t:library /r:Person.dll SurrealistDB.cs</div>

Then compile the server itself:

<div align="center">csc /r:Person.dll;SurrealistDB.dll SurrealistEchoServer.cs</div>

Change to the client directory and compile the EchoClient class like so:

<div align="center">csc /r:Person.dll EchoClient.cs</div>

Finally, start the SurrealistEchoServer application and then run the EchoClient application. Figure 19-14 gives the results of fetching some surrealists from the server.

Quick Review

When building client-server applications using the *TcpListener* and *TcpClient* classes, you'll need to know how to handle the details of establishing the network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

The TcpListener and TcpClient classes provide *wrappers* around *Socket* objects. You can access Socket objects directly if you need more control over client-server network communication.

The general steps required to create a client-server application using TcpListener and TcpClient include the following: 1) create a TcpListener object that listens for incoming TcpClient connections on a specified IP address and port number, 2) on the client side, create a TcpClient object that connects to a particular server identified by an IP address and a particular port number, 3) when the listener detects an incoming TcpClient connection its AcceptTcpClient() method returns (unblocks) and creates a server-side TcpClient object, 4) use the TcpClient's GetStream()

Figure 19-14: Results of Running SurrealistEchoServer and EchoClient (Mod 2)

method to get a reference to the NetworkStream object, 5) use the NetworkStream object to create StreamReader and StreamWriter objects. Remember to follow a call to StreamWriter.Write() with a call to StreamWriter.Flush().

Multithreaded servers can service multiple simultaneous client connections. Use the Thread class to create a separate client processing thread. This frees up the server to listen for new incoming client connections.

Objects passed between client-server applications must be tagged Serializable. Use a BinaryFormatter to serialize an object to the NetworkStream. Call NetworkStream.Flush() to send the object on its way.

SUMMARY

All .NET remoting applications have three common components, regardless of their complexity: a *remotable object*, a *server application* that hosts the remotable object and handles incoming service requests, and a *client application* that utilizes the services of the remotely-hosted object.

A remotable object is created from a class that inherits from System.MarshalByRefObject. This enables the object to be shared across application domains in .NET remoting applications. Note that the remotable object can be simple or complex.

The remoting server application hosts the remotable object and makes its services available via a *channel*. There are three primary channel types: *TcpChannel*, *HttpChannel*, and *IpcChannel*. The IpcChannel is used for *inter-process* communication between client and server applications hosted on the same machine.

The remoting client application accesses the services of the remote object via a *proxy* created automatically by the .NET remoting infrastructure. Once a remoting client application creates a reference to a remote object, it uses the services of the remote object, via the remote object's proxy, as if the remote object were a local object. The underlying complexities associated with calling the remote object's methods or properties are handled automatically by the

.NET remoting infrastructure. Remote objects accessed in this manner must extend MarshalByRefObject and any other interfaces as required.

Remote objects can be hosted in *SingleCall* or *Singleton* mode. In SingleCall mode, a new remote object is used to respond to each client service request. In Singleton mode, remote objects persist and maintain state across multiple client service requests.

Remoting client applications can access the services of remote objects via one or more of the remote object's interfaces. This makes changing the implementation of the remote object easier, as long as the new object implements one of the interfaces expected by the client application.

For maximum deployment flexibility, place .NET remoting application deployment data in *configuration files*.

Complex objects sent between remoting client and server applications must be tagged as being serializable by using the *Serializable* attribute.

When building client-server applications using the *TcpListener* and *TcpClient* classes, you'll need to know how to handle the details of establishing the network connection between client and server applications, how to send data between the client and server so they can perform useful work, and how to use threads to enable a server to handle multiple client requests simultaneously.

The TcpListener and TcpClient classes provide *wrappers* around *Socket* objects. You can access Socket objects directly if you need more control over client-server network communication.

The general steps required to create a client-server application using TcpListener and TcpClient include the following: 1) create a TcpListener object that listens for incoming TcpClient connections on a specified IP address and port number, 2) on the client side, create a TcpClient object that connects to a particular server identified by an IP address and a particular port number, 3) when the listener detects an incoming TcpClient connection, its AcceptTcpClient() method returns (unblocks) and creates a server-side TcpClient object, 4) use the TcpClient's GetStream() method to get a reference to the NetworkStream object, 5) use the NetworkStream object to create StreamReader and StreamWriter objects. Remember to follow a call to StreamWriter.Write() with a call to StreamWriter.Flush().

Multithreaded servers can service multiple simultaneous client connections. Use the Thread class to create a separate client processing thread. This frees up the server to listen for new incoming client connections.

Objects passed between client-server applications must be tagged *Serializable*. Use a BinaryFormatter to serialize an object to the NetworkStream. Call NetworkStream.Flush() to send the object on its way.

Skill-Building Exercises

1. **API Drill:** Visit the System.Runtime.Remoting, System.Runtime.Remoting.Channels, and System.Runtime.Remoting.Channels.Tcp namespaces and list each class, structure, interface, and enumeration. Write a brief description of its purpose. Browse each entry's members including its methods and properties.

2. **Programming Drill:** Compile and execute all the exercises in this chapter.

3. **Code Drill:** Trace the execution of all the exercises in this chapter.

4. **Programming Drill:** Modify this chapter's .NET remoting examples to use the HttpChannel.

5. **API Drill:** Explore the System.Net and System.Net.Sockets namespaces and list each class, structure, interface, and enumeration. Write a brief description of its purpose. Browse each entry's members including its methods and properties.

6. **Programming Drill:** Modify this chapter's client-server examples to use UDP vs. TCP.

7. **Programming Drill:** Modify this chapter's client-server examples to better handle the possibility of a network outage between client and server applications. For example, modify the EchoClient to gracefully recover if it tries to send something to a server but experiences a long network delay. (**Hint:** Explore the TcpClient class's properties section on MSDN.)

8. **Extra Reading:** Procure the book *TCP/IP SOCKETS IN C#: Practical Guide for Programmers* and read it from front to back!

9. **Programming Drill:** Is a .NET remoting server multithreading capable? To answer this question, modify the last RemotingClient example given in Example 19.17 so that it repeatedly sets the Text property on the remote object and then goes to sleep (*i.e.*, Thread.Sleep()) for 3 seconds. Start the RemotingServer application and then start two or more RemotingClient applications and see what happens.

10. **Deployment Drill:** Deploy and test this chapter's example applications on different machines. That is, start the server on one machine and run the client application on another machine on the same network. (**Note:** Only the client applications capable of connecting to IP addresses other than 127.0.0.1 will work in this scenario.)

Suggested Projects

1. **Network Robot Rat:** Write a client-server version of the robot rat application. Create a server application that displays robot rat images in a GUI representation of the floor. Each incoming client connection should have their very own image of robot rat displayed and moved on the floor. Use the client application to control the movements of the robot rat. In the client application show an image of the floor, which gives the position of that client's robot rat. Alternatively, give the client application a spy capability (enabled by the server) that lets it see the positions of all the other connected clients's robot rats.

2. **Network Employee Management Application:** Write a client-server application that lets users remotely access and manipulate a file containing employee information. Use the client application to view a list of employees, add a new employee, edit an existing employee, and create new employees. Use the Employee example code given in Chapter 11. Give the client application a graphical user interface.

3. **Chat Program:** Write a program that lets multiple users connect and chat. The server should request the user name from new client connections. The client application should be able to see a list of connected users. Have the server echo user messages to all connected clients.

4. **Email Client:** Study the members of the System.Net.Mail namespace. Write a client program that lets you connect to your email provider, download and read your messages, and create and send new messages.

Self-Test Questions

1. What three things do all .NET remoting applications have in common?

2. What class must be extended to create a remotable object?

3. What attribute must you tag a class with before you can transmit objects of its type between .NET remoting applications?

4. List and briefly describe the purpose of the three primary remoting channel types.

5. What is the primary benefit derived from accessing a remote object via an interface?

6. What's the difference between a remote object deployed in the *SingleCall* mode vs. the *Singleton* mode?

7. Describe the roles of the TcpListener and TcpClient classes in a typical client-server application.

8. What happens when you call the TcpListener.AcceptTcpClient() method?

9. Describe in general terms what you need to do to create a multithreaded server application.

10. Why must you follow a call to NetworkStream.Write() with a call to NetworkStream.Flush()?

References

David B. Makofske, et. al. *TCP/IP SOCKETS IN C#: Practical Guide for Programmers*. Morgan Kaufmann Publishers, 2004, ISBN-13: 978-0-12-466051-9, ISBN-10:0-12-466051-7

Microsoft Developer Network (MSDN) *.NET Framework 3.0 and3.5 Reference Documentation* [www.msdn.com]

Notes