3 DotNET Remoting

Lecture:	[Link]
Tutorial (Part 1):	[Link]

3.1 Introduction

.NET remoting is a technology which allows objects to be placed remotely across a network, where the object can be activated, and communicate with local objects using a communications channel. A formatter is then used to encode and decode the messages as they pass between the remote object and the application. The format of these message can either be:

- **Binary encoded**. This is used where **performance** is a critical factor.
- **XML encoded**. This is used when **interoperability** is important, and uses the standardized SOAP protocol.

A key element of .NET remoting is that objects can check the messages that are to be sent before they are sent to the channel, and the remote objects are activated in different ways, these are:

- **Client-activated objects**. These have a finite lease time, and once their lease has expired, they are deleted (using the garbage collector).
- **Server-activated objects**. These can either be defined with a *single call* or with a *singleton*. A single call accepts one request from a client, then performs the action, and is finally deleted (with the garbage collector). It is defined as stateless, as it does not hold onto parameters from previous calls. Singletons are **stateful** and can accept multiple calls, where they remember previous calls, and retain their information. They can thus communicate with multiple clients. Also the lifetime of singletons is controlled by lease-based lifetime.

3.2 Application boundaries

Microsoft Windows allows applications to run on virtual machines, where applications should not interfere with each other. Thus, a fault in one application should not affect another one. Each application thus has its own code and data area, which should not be accessed by other applications running on the machine. A process boundary is one which isolates a process from others which are running. It is necessary that each process can run in its own virtual space, and gain access to as much memory that they require. Other processes should not be able to interfere with this. If a process crashes, it should not affect other processes. .NET expands on this by apply this managed environment onto applications. This is because .NET uses the Common Language Runtime (CLR - Figure 3.1) which is a managed environment

for executing code, code access security, object management, debugging, cross-language integration, and profiling support. In standard Windows applications it is not possible to provide a degree of safety that an application does not step outwith its boundaries, as Windows applies process control boundary, rather than an application boundary. In .NET an application domain is created with the AppDomain class with the System **namespace**. Thus the application boundary ensures that:

- Each application has its own code, data and configuration settings.
- No other application can interfere with any other applications.

It is thought that application switching for processor time is more efficient than processor switching. Along with this, it is easier to monitor the operation of an application than it is to monitor a number of processes.

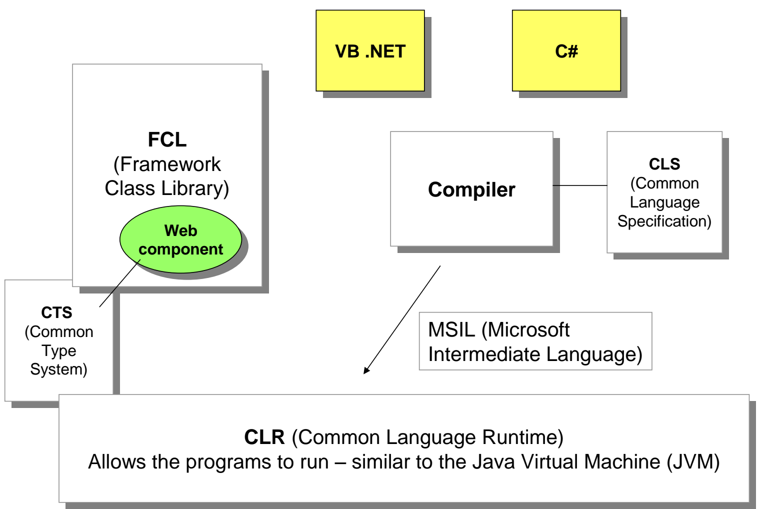

Figure 3.1: .NET Framework

3.3 Distributed Systems

Distributed systems allows resources to be distributed around a network, rather than tying them to a specific host. These resources could relate to hardware and software resources. This allows for less centralized approach, and typically improves the robustness of a system and also makes improved usage of network traffic. For example an organisation might use a centralized Web server for their document management. This has the advantage is that it is relatively easy to manage, but it becomes a centralized point of failure, also the network traffic is likely to be relatively large towards the server. Along with this the CPU usage it likely to be relatively large.

An improved system would be to distribute Web servers around the network, and for hosts to access them on a local level. This type of approach can be scaled down to an object-level, where objects can be distributed around a network, either outwith the application domain, or outwith the host. These objects could reside on

different types of operating systems or system types and thus allow applications to run over heterogeneous systems.

Another similar approach is to develop applications which is made of interconnected components. These components make it easier to design software, where tasks are split into elements. The distribution of components allows for the standardization of components, which could run remotely. For example a component which does a grammar checker could be standardized and when an application requires a grammar checker it calls it up remotely, and uses it.

In general distributed systems are generally more scaleable, more robust, and increase availability of services. The most common distributed protocols are RPC (Remote Procedure Calls), Microsoft Distributed Object Model (DCOM), Common Object Request Broker Architecture (CORBA) and Java Remote Invocation (RMI). These are typically applied to certain types of environments, such as DCOM on Windows-based systems and RPC in a UNIX environment. Some of these are legacy type systems which were designed to be simple, and have never really kept up-to-date with modern methods, especially related to security. The .NET framework hopes to produce a new standard for distributed applications.

The main namespaces used for .NET Remoting include:

- **System.Net**. This includes classes relating to the networking elements of the distributed system.
- **System.Runtime.Remoting**. This includes classes for the remote aspects of the .NET framework, such as mechanisms for remote communication between objects.
- **System.Web.Services**. This includes protocols relating to Web services, such as those relating to HTTP and SOAP. This is defined as the ASP.NET Web services framework.

3.4 Remote Objects

.NET remoting provides a simple mechanism to call remote objects, where a remote object is any object which is outside the application domain, and can thus also be local on the same machine. Within an application domain, the object is passed by **reference**, whereas primitive data types are passed by value. Obviously it is not possible to pass an object from a remote object by reference since the reference to the object is only valid on the side which contains the object (that is, they only have local significance). For an object to be passed over an application domain, they must be passed by value, and also **serialized** (that is, send over a single communications channel). This, thus, defines the structure of the object, and also its contents.

The process of communicating between objects and transferring them over an over application boundaries is known as **marshalling**. Objects are converted into remote objects when they derive from **MarshalByRefObject**. Then, when a client activates the remote object, it interfaces with a proxy for the remote object, as illustrated in Figure 3.2. The proxy object acts on behalf of the remote object, and, as previously mentioned, is created when the client actives a remote object. Its main

objective is to make sure that all the messages sent to the remote object are sent to the correct instance of the object.

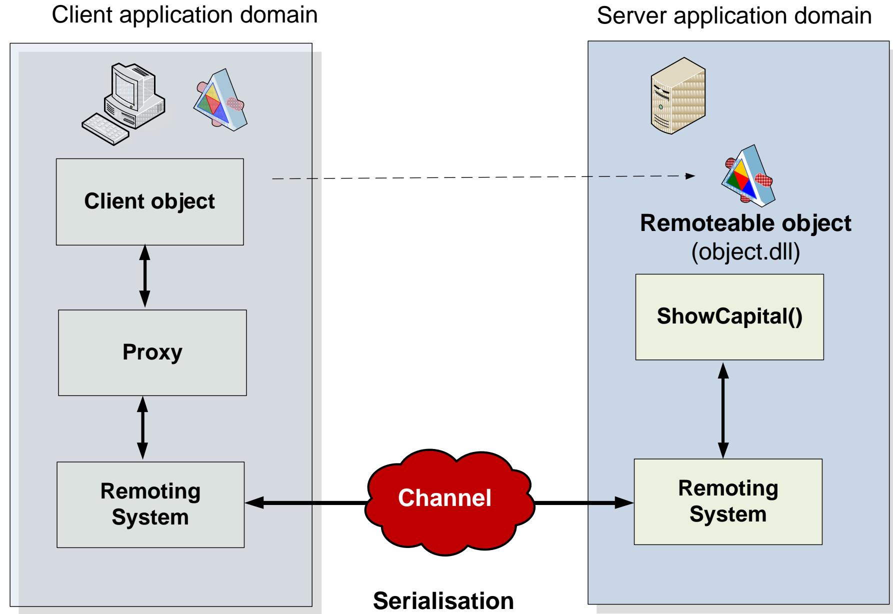

Figure 3.2: .NET remoting

On the remote machine, the remote object is initially registered into an application domain. The MarshalByRefObject is then used to encapsulate all the information required to locate and access the remote object, such as its class name, its class hierarchy, its interfaces and its communication channels. Activation occurs using the URL, and identifies the URI (Unique Reference Identifier) of the remote object.

A **MBR** (Marshal-by-reference) always resides on the server, and the methods are executed on the server, while the client communicates with the local proxy. The following shows an example of the ShowCapital class which derives from the MashalByRefObject, and contains a show() method.

```
public class ShowCapital : MarshalByRefObject
{
   public ShowCapital()
   {
   }
   public string show(string country)
   {
   }
}
```

Derive from MarshalByRefObject

The MarshalByValue (MBV) involves serializing values on the server, and sending them to the client. An MBV object is declared by a class with the Serializable attribute, such as:

```
[Serializable()]
public class NewMBVObject
{
}
```

3.4.1 Channels

Remote objects transfer messages between themselves using channels. These channels are thus used to send and receive objects, as well as sending information on the methods that require to be called. Each object must have at least one channel set up for this communication, and the channel must be registered before a remote object is called. Once the object is deleted, the channel which it uses is also deleted. Also the same channel cannot be used by different application domains on the same machine. These channels map onto TCP port numbers, and the application must be sure that it does not use one which is currently being used.

3.4.2 HTTP and TCP channels

Messages which are transferred by the SOAP protocol use an HTTP channel, whereas a TCP channel uses a binary format to serialize all message. In the HTTP channel, all the messages are converted into an XML format, and serialised (along with the required SOAP headers). The namespaces are:

System.Runtime.Remoting.Channels.HTTP for HTTP
System.Runtime.Remoting.Channels.TCP for TCP

For example to create a client connection to a server port of 1234:

```
TcpChannel channel = new TcpChannel(1234);
ChannelServices.RegisterChannel(channel);
```

The HTTPChannel can be used for a wide range of services which are hosted by a Web server (such as IIS). It has security built into it, but has an overhead of extra information (as it works at a higher level than TCP). The TCPChannel is more efficient in its operation, but does not have any security built into it.

3.4.3 Activation

A key element of the .NET remoting framework is that it supports the activation of remote objects as either a client or a server. Server activation is typically used when remote objects do not required to maintain their state between method calls, or where there are multiple clients who call methods on the same object instance where the object maintains its state between function calls. In a client-activated object, the client initiates the object and manages it for its lifetime.

Remote objects have to be initially registered with the remoting framework before the clients can use them. This is normally done when a hosting application starts up and then registers one or more channels and one or more remote objects. It then waits until it is terminated. When the hosting application is terminated, the objects and channels are deleted. For an object to be registered into the .NET remoting framework the following need to be set:

- **Assembly name**. This defines the assembly in which the class is contained in.
- **Type name**. This defines the data type of the remote object.
- **Object URI**. This is the indicator that clients use to locate the object.

- **Object mode**. This defines the server activation, such as SingleCall or Singleton.

Remote objects are registered using the RegisterWellKnownServiceType, by passing the required parameters into the method, such as:

```
RemotingConfiguration.RegisterWellKnownServiceType
   (typeof(newclass.ShowCapital),
   "ShowCapital1", WellKnownObjectMode.SingleCall);
```

which defines a **SingleCall** remote object, and the **ShowCapital** object, which is within the **newclass** namespace. **ShowCapital** is thus the name of the object, where the **ShowCapital1** is the object URI.

It is also possible to store the parameters in a configuration file then using Configure with the required configuration file, such as:

```
RemotingConfiguration.Configure("myconfig.config");
```

where the configuration file could be in the form of:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="Singleton" type="newclass.ShowCapital, newclass"
                  objectUri="ShowCapital1" />
         </service>
         <channels>
            <channel ref="tcp server" port="1234" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

This method has the advantage that it does not need to be compiled with the application, and can therefore be edited as required, and will be read in as required. Thus a change of parameters, such as a change of object name does not require a recompilation.

Once registered, the remote object will not instantiate itself, as this requires the client to initiate it, or call a method. This is achieved by a client which knows the URI of the remote object and by registering the channel it prefers using GetObject, such as:

```
TcpClientChannel channel = new TcpClientChannel();

ChannelServices.RegisterChannel(channel);

ShowCapital sh= (ShowCapital)
    Activator.GetObject(typeof(newclass.ShowCapital),
   "tcp://localhost:1234/ShowCapital1");
```

where:

- "tcp://localhost:1234/ShowCapital1". Specifies that the end point is ShowCapital using TCP port 1234.

Along with this, the compiler requires type information about the ShowCapital class when this client code is compiled. This can be defined with one of the following:

- With a reference to the assembly where the ShowCapital class is stored.
- By splitting the remote object into an **implementation** and **interface class** and then use the interface as a reference when compiling the client.
- Using SOAPSUDS tool to extract metadata directly from the endpoint. SOAPSUDS connects to the endpoint, and extracts the metadata, and generates an assembly or souce code that is then used in the client compilation.

Another method is

```
RemotingConfiguration.RegisterWellKnownClientType(
   typeof(ShowCapital),
   "tcp://localhost:1234/ShowCapital");
```

None of these calls actually initiates the remote object, as only a proxy is created which will be used to contact the object. The connection is only made when a method is called on the remote object. Once this happens, the remote framework extracts the URI, and initiates the required object, and forwards the required method to the object. If it is a SingleCall, the object is destroyed after the method has completed.

3.5 Applying .NET Remoting

The following sections show a remote objects are initiated and then how they can be used to communicate between a client and a server.

3.5.1 Creating a remotable class

The following steps shows how a remote class is created (Figure 3.3). These are:

1. Create a new blank solution (named NetRemoting1).
2. Add a new class library (for example, named newclass). Select Add New Project, then select Class Library, as illustrated in Figure 3.3.
3. Add **System.RunTime.Remoting.dll** as a Reference (if required).
4. Change the name of the class file to a new name (for example, ShowCapital.cs).
5. Add the following code:

C# Code 3.1:

```
using System;
using System.Data;

namespace newclass
{
    public class ShowCapital : MarshalByRefObject
    {
        public ShowCapital()
```

```
            {
            }
            public string show(string country)
            {
                if (country.ToLower() == "england") return ("London");
                else if (country.ToLower() == "scotland") return ("Edinburgh");
                else return ("Not known");
            }
        }
    }
```

and the results are shown in Figure 3.5. This produces a class named ShowCapital, which has a method of show(). This class, once built, produces a remotable class in the form of a DLL file, which is placed in the bin\Debug folder. In this case it will create a DLL named newclass.dll, as this is the name of the namespace.

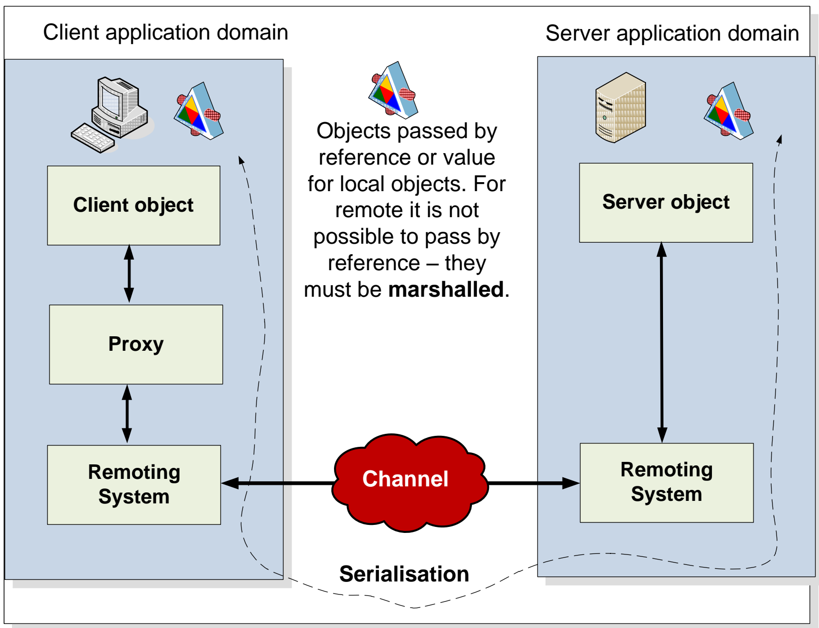

Figure 3.3: Remotable class

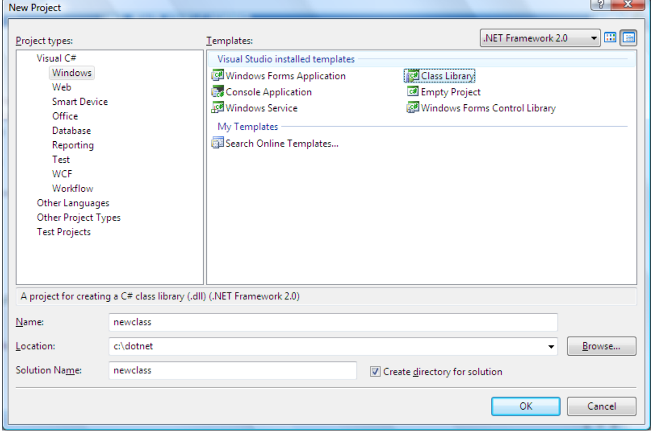

Figure 3.4: Adding a new class

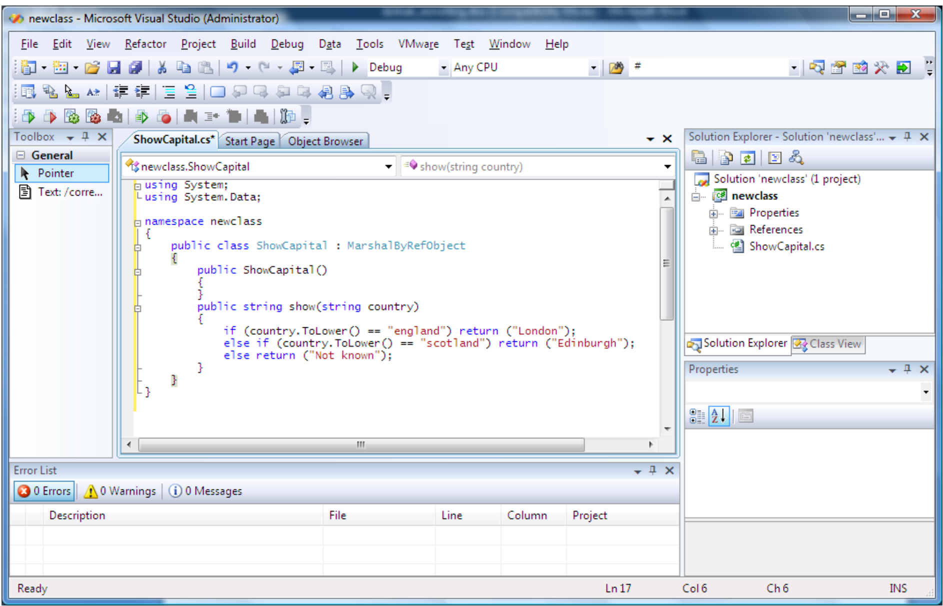

Figure 3.5: Completed example

3.5.2 Create a server-activated object

This section shows how an application can be created which activates the remotable object. As long as the application runs, the remotable object can be called by a client, and invoked.

1. Add a New Project, and select Console Application, and name it (for example newclass2, as shown in Figure 3.6).
2. Next add the references to the first project (the DLL file), and to the System.Runtime.Remoting (Figure 3.7).
3. Next the following code can be added:

C# Code 3.2:

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

namespace newclass2
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            TcpServerChannel channel = new TcpServerChannel(1234);

            ChannelServices.RegisterChannel(channel,false);

            RemotingConfiguration.RegisterWellKnownServiceType
                (typeof(newclass.ShowCapital), "ShowCapital",
                WellKnownObjectMode.SingleCall);
            Console.WriteLine("Starting...");
            Console.ReadLine();
        }
    }
}
```

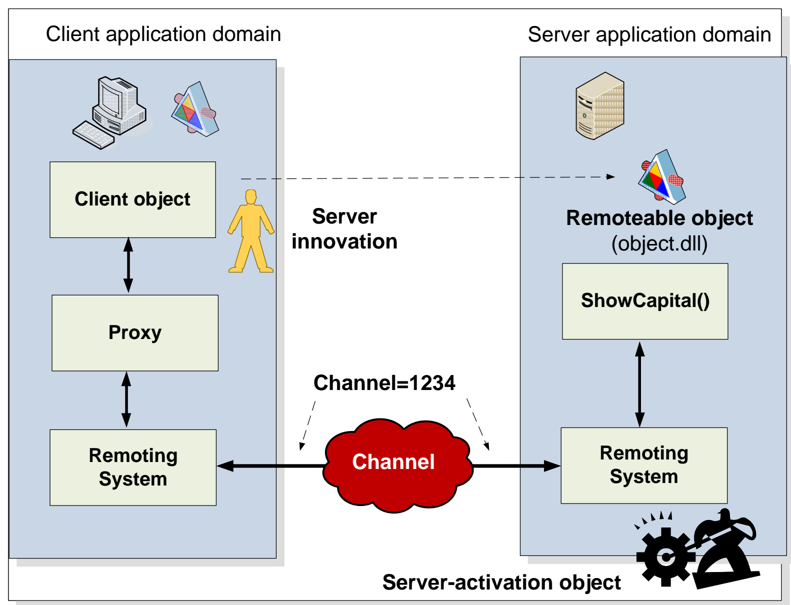

Figure 3.6: Server-activation object (SAO)

For this:

```
RemotingConfiguration.RegisterWellKnownServiceType
   (typeof(newclass.ShowCapital), "ShowCapital",  WellKnownObjectMode.SingleCall);
```

defines a **SingleCall** activation mode (which means it will run once and then be deleted), on the **ShowCapital** object, which is within the **newclass** namespace. **ShowCapital** is thus the name of the object, where the **ShowCapital1** is the object URI. This refers to a DLL file named after the namespace (in this case **newclass.dll**). The channel used is 1234. Once the server-activated component has been started it will wait for the client to connect to it. As it is a SingleCall it will call the remote object once, and then it will be deleted. Thus every time it is called, it will not remember the previous state. If the Singleton option is used the remote object will stay active and will thus store its state.

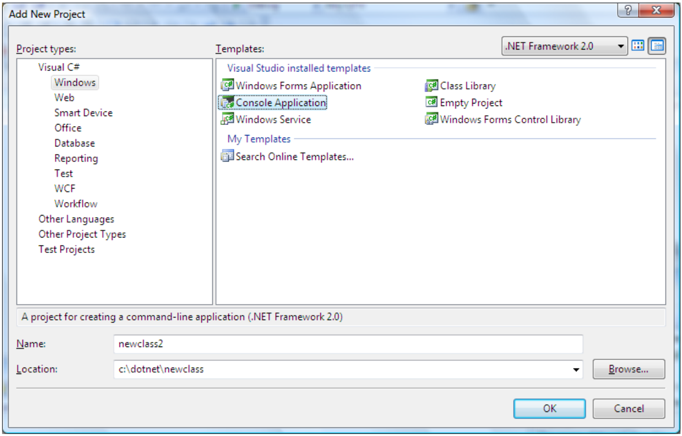

Figure 3.7: .NET remoting

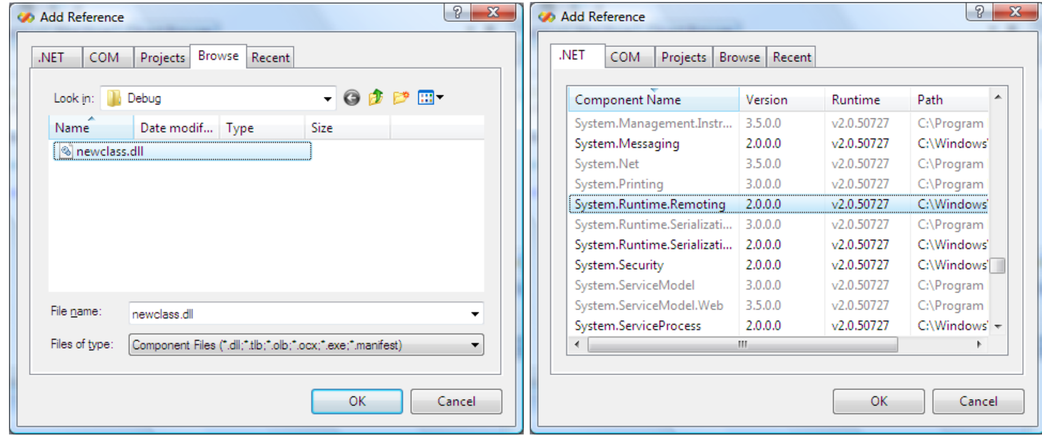

Figure 3.8: .NET remoting referencing

3.5.3 Instantiating and invoking a server-activated object

This section shows how an application can invoke a server-activated object. This runs on the client and calls the server to invoke the remotable class. The main steps are:

1. Add a New Project to the solution, using a Windows Application (Figure 3.9).
2. Next a reference is added for a the System.Runtime.Remoting assembly (Figure 3.10) and the remotable class (newclass), as shown in Figure 3.11.

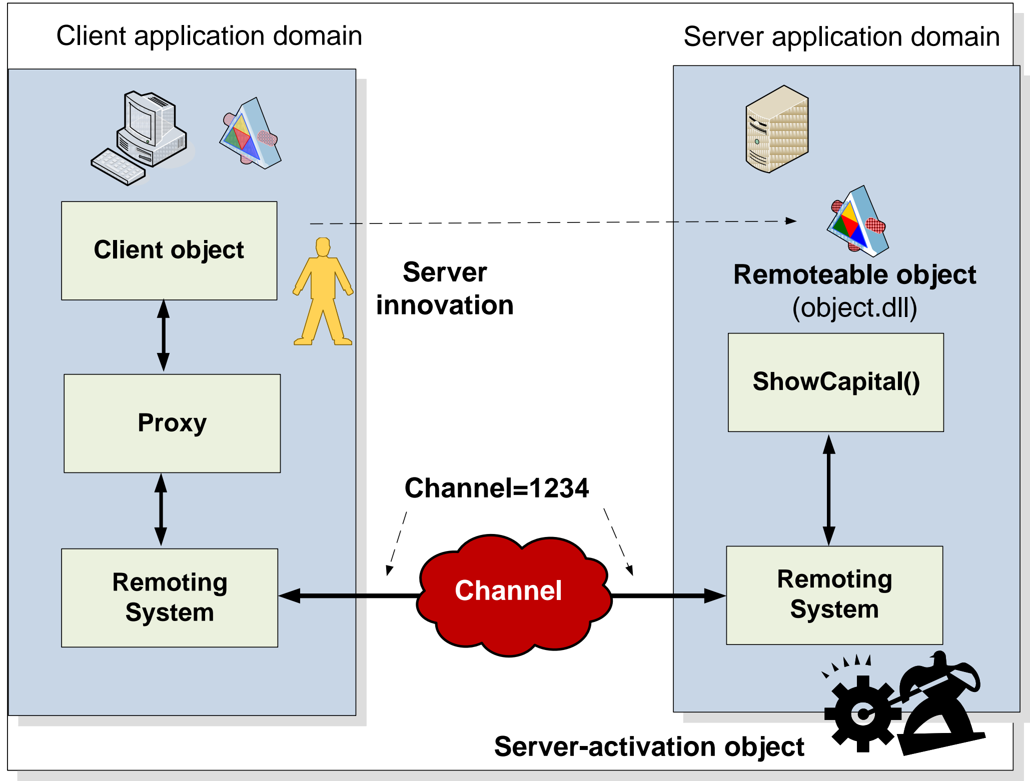

Figure 3.9: Server invocation

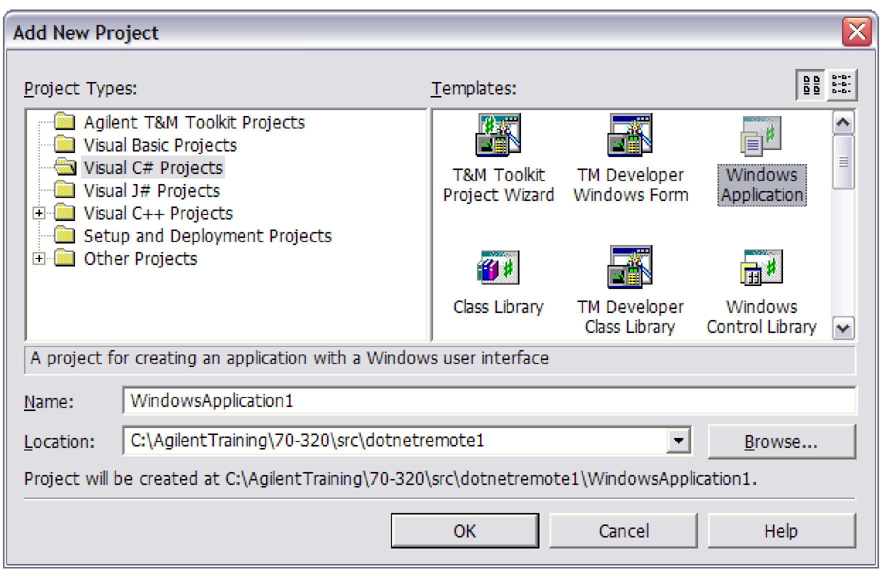

Figure 3.10: .NET remoting

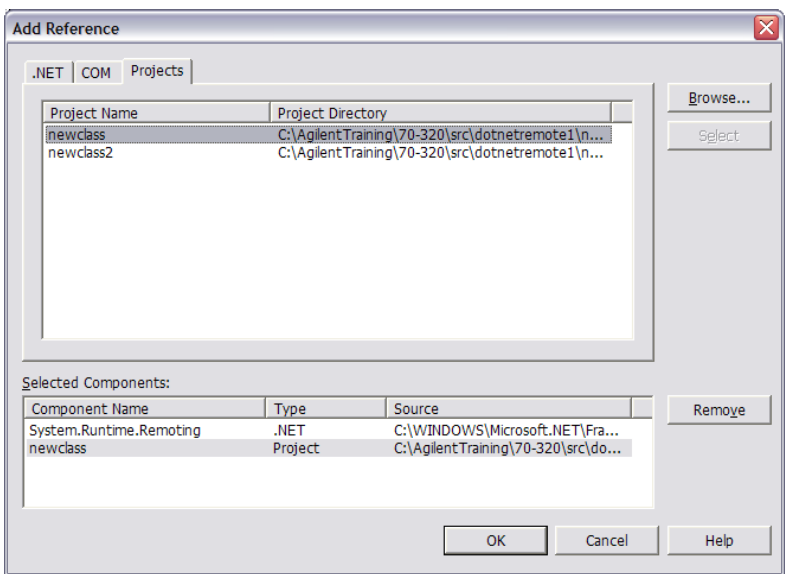

Figure 3.11: .NET remoting

3. Add the following code:

C# Code 3.3:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;

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

namespace WindowsFormsApplication1
```

```
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        ShowCapital sh;

        private void button1_Click(object sender, System.EventArgs e)
        {
            string country, cap;

            country = textBox1.Text;
            cap = sh.show(country);
            textBox2.Text = cap;
        }

        private void Form1_Load(object sender, System.EventArgs e)
        {
            TcpClientChannel channel = new TcpClientChannel();

            ChannelServices.RegisterChannel(channel,false);

            RemotingConfiguration.RegisterWellKnownClientType(
                typeof(ShowCapital),"tcp://localhost:1234/ShowCapital");
            sh = new ShowCapital();
        }

    }
}
```

This then connects to server (in this case, with localhost, which has the IP address of 127.0.0.1), using TCP port of 1234.

3.5.4 Creating a control executable

It is possible to create an orderly start for the project, by selecting the properties of the solution, and then selecting Multiple Startup Project. This can then be used to create an orderly innovation of the programs, as the server must be started before the client. Obviously the remotable class will not be started-up, thus its action is None, as illustrated in Figure 3.12. A sample run is given in Figure 3.13.

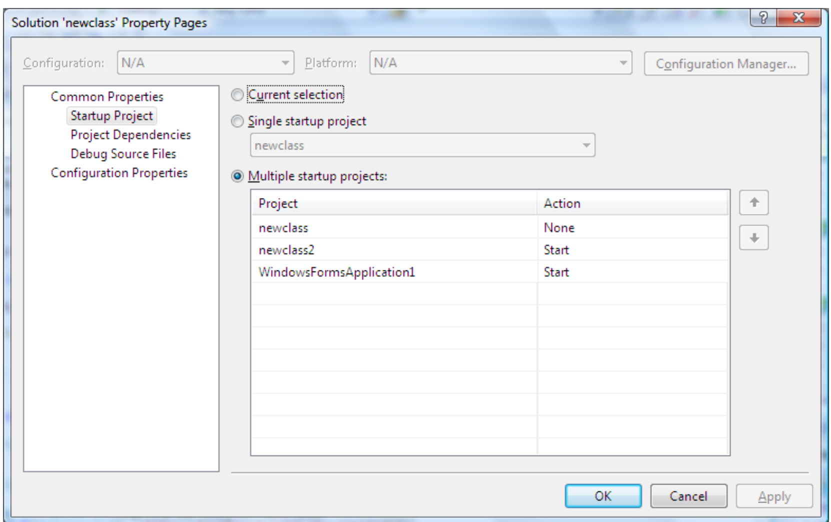

Figure 3.12: .NET remoting

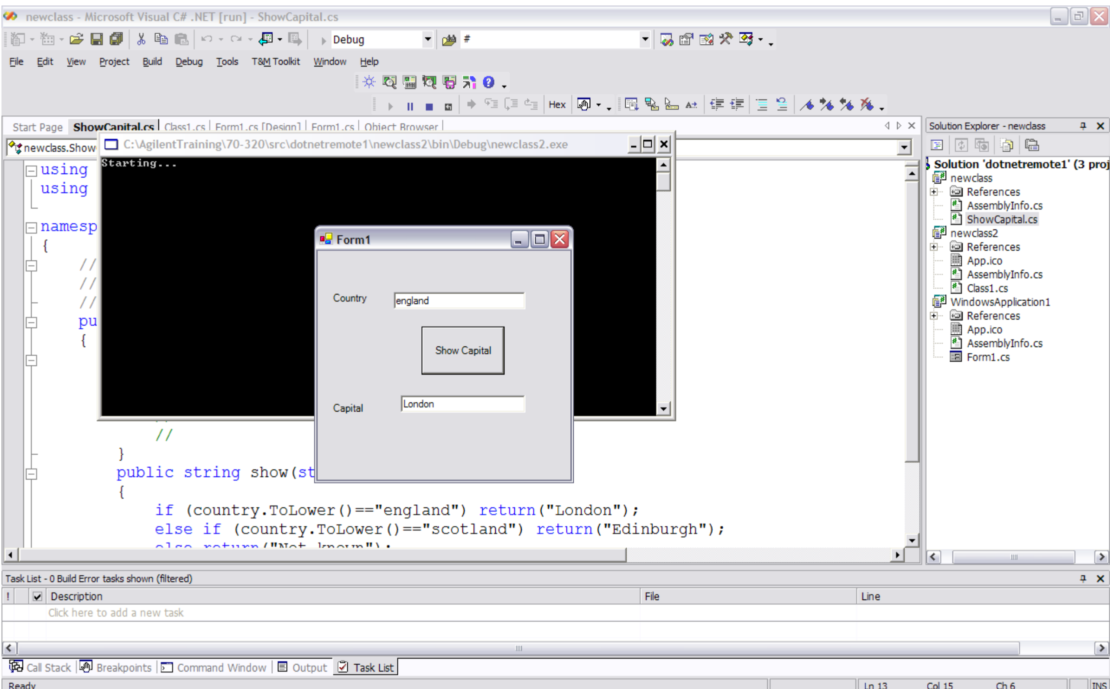

Figure 3.13: .NET remoting

3.6 Tutorial 1

Tutorial (Part 1): [Link]

Q3.1 Implement the code in C# Code 3.1, C# Code 3.2 and C# Code 3.3, and prove that the application works. Next, complete the following:

Tutorial (Part 2): [Link]

(a) Add a debug message within C# Code 3.1 so that it displays the country which is being searched for.

Tutorial (Part 3): [Link]

(b) Add a counter within C# Code 3.1 so that the remoteable object returns a counter to show the number of times it has been called (as illustrated in Figure 3.14). Show thus, for the SingleCall that the counter will always show the same value.

(c) Next modify C# Code 3.2, so that is uses the Singleton method, and show that the counter will increment for every access.

Tutorial (Part 4): [Link]

(d) Modify the application so that it uses HTTP rather than TCP.

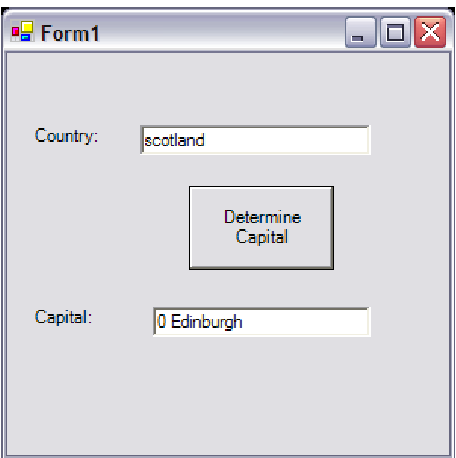

Figure 3.14: Example

Q3.2 Contact a neighbour in the lab, and ask them to place your server component (dll and exe) from Tutorial Q3.1 onto their machine. Ask them to run

the server component. Next run the Windows program to contact their server component, and prove that it works.
Note: Make sure you use the IP address of your neighbour's PC.

Q3.3 Modify the server C# Code 3.2 so that it listens on port 80 (or any other reserved ports), and, if you have a WWW server running on your machine, prove that the error message is:

```
An unhandled exception of type 'System.Net.Sockets.SocketException' occurred
in system.runtime.remoting.dll

Additional information: Only one usage of each socket address (proto-
col/network address/port) is normally permitted
```

Q3.3 From the Command Prompt, run netstat -a, and determine the TCP server ports which are open. Next run netstat -a -v, and observe the output. What does the 0.0.0.0 address identify?

Q3.4 Create a program which uses .NET remoting to implement the following:

(a) Returns the IP address for a specified domain name. An example is given in Figure 3.15.

Note, to get the IP addresses the following can be used:

```
if (strHostName=="") strHostName = Dns.GetHostName();
IPHostEntry ipEntry = Dns.GetHostByName(strHostName);
IPAddress [] addr = ipEntry.AddressList;
```

Where the first IP address can be return (addr[0]).

(b) Created another method which returns the hostname for a given IP address.

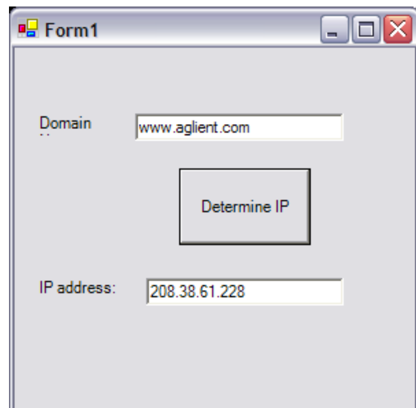

Figure 3.15: Example

3.6.1 Client-activated object

The lifetime of server-activated objects (**SAO**) is directly controlled by the **server**, whereas the lifetime of client-activated objects (**CAO**) is controlled by the calling application program, just as if it were local to the client.

An SAO the remotable object is defined with:

```
RemotingConfiguration.RegisterWellKnownServiceType
   (typeof(newclass.ShowCapital), "ShowCapital",
   WellKnownObjectMode.SingleCall);
```

A CAO the remotable object is defined with:

```
RemotingConfiguration.RegisterActivatedServiceType
   (typeof(newclass.ShowCapital));
```

When initiating and invoking a SAO:

```
RemotingConfiguration.RegisterWellKnownClientType(
   typeof(ShowCapital), "tcp://localhost:1234/ShowCapital");
```

When initiating and invoking a CAO:

```
RemoteConfiguration.RegisterActivedClientType
   (typeof(newclass.ShowCapital),"tcp://localhost:1234");
```

3.6.2 Tutorial 2

Q3.5 Modify Tutorial Q3.1 so that it uses CAO instead of SAO.

Q3.6 Modify Tutorial Q3.2 so that it uses CAO instead of SAO.

Q3.7 Implement a remote object which has the following methods:

- Square(x). Which determines the square of a number.
- SquareRoot(x). Which determines the square root of a number.
- Power(x,n). Which determines the power of a number (x) raised to another number (n).

3.7 Configuration Files for remoting

In the previous examples the channel and the definitions for the remote object have been defined within the program (programmatic configuration). Thus it is not possible to change them once the application has been compiled. An enhanced method which allows the channel and remote object definition to be defined as an external configuration is to use an XML file to define the parameters used in .NET remoting (declarative configuration). This is achieved either at a machine-level with the machine.config file (which can be found in the config folder), or at an application level with an application configuration file. In an ASP.NET application the name of the configuration file is web.config. The general format is:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime> </lifetime>
         <service>
            <wellknown/> <activated/>
         </service>
         <client>
            <wellknown/> <activated/>
         </client>
         <channels> </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

It includes the full name of the application, with a .config extension. Thus an application named myapp.exe will have an associated configuration file of myapp.exe.config.

3.7.1 Register server-activated object with configuration file

RemotingConfiguration.Configure is used to register a server-activated object, such as:

```
static void Main(string[] args)
{
   RemotingConfiguration.Configure("..//..//dotnetremote1.exe.config");

   Console.WriteLine("Starting...");
   Console.ReadLine();
```

The file dotnetexample1.exe.config:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="Singleton" type="newclass.ShowCapital, newclass"
                 objectUri="ShowCapital1" />
         </service>
         <channels>
            <channel ref="tcp server" port="1234" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

This achieves the same as:

```
RemotingConfiguration.RegisterWellKnownServiceType
   (typeof(newclass.ShowCapital),
   "ShowCapital1", WellKnownObjectMode.Singleton);
```

which defines a **Singleton** remote activation, and the **ShowCapital** object, which is within the **newclass** namespace. **ShowCapital** is thus the name of the object, where the **ShowCapital1** is the object URI.

3.7.2 Register client-side configuration

The client-side configuration is similar, but uses the <client> element. An example of the configuration file is:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown type="newclass.ShowCapital, newclass"
            url="tcp://localhost:1234/ShowCapital" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

and for the call:

```
private void Form1_Load(object sender, System.EventArgs e)
{
   RemotingConfiguration.Configure("app2.config");
   sh= new ShowCapital();

}
```

Figure 3.16 shows an example of the output.

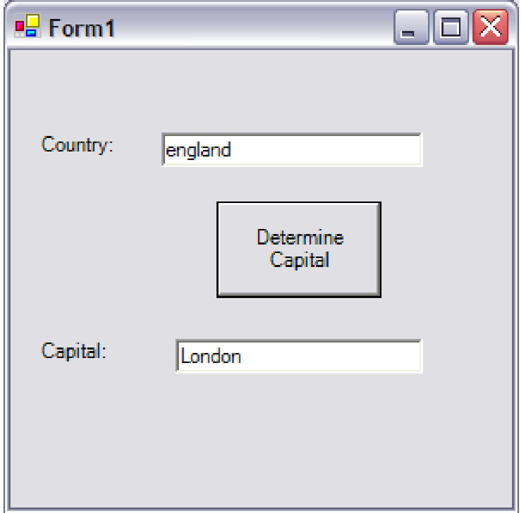

Figure 3.16: .NET remoting

3.7.3 Tutorial 3

Q3.8 Modify C# Code 3.1, C# Code 3.2 and C# Code 3.3, so that they use an XML configuration file for their remoting parameters.

(a) Set one port to 1234, and the other to 9999. Prove that the program will create an exception.
(b) Next, change both ports to 9999, and prove that the system works.

Q3.9 Run the program, and from the command prompt, run **netstat**, and determine that a TCP connection has been made. Outline its details:

Q3.10 Set up the client and server to communicate on port 5555 (using the configuration file). Next, contact a neighbour in the lab, and ask them to place your server component (dll and exe) onto their machine. Ask them to run the server component, and run the windows program to contact their component, and prove that it works.

Note: Make sure you use the IP address of your neighbours PC.

Next, ask your neighbour to change the port to 8888 on the server side, and change the port on your machine. Prove that the system can still communicate.

3.8 Interface assemblies to compile remote clients

An interface is an alternative method of creating an abstract class, and it does not contain any implementation of the methods, at all. It provides a base class for all the derived classes. This is useful in hiding the code from other developers, as, in the previous examples, the assembly to the remotable class has been included in the clients project. In the following an interface named IDCapital is created, which has a method of show():

C# Code 3.4:

```
using System;

namespace IDCapital
{
   public interface IDCap
   {
      string show(string str);
   }
}
```

This is the interface assembly. Next we can define the remotable object which implements the interface assembly:

C# Code 3.5:

```
using System;
using System.Data;
using IDCapital;

namespace newclass
{
   public class ShowCapital : MarshalByRefObject, IDCapital.IDCap
   {

      public string show(string country)
      {
         if (country.ToLower()=="england") return("London");
         else if (country.ToLower()=="scotland") return("Edinburgh");
         else return("Not known");
      }
   }
}
```

After which a remoting server can be created to register the remotable object that implements an interface assembly:

C# Code 3.6:

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

namespace newclass2
{
   class Class1
   {
      [STAThread]
```

```
      static void Main(string[] args)
      {
         TcpServerChannel channel = new TcpServerChannel(1234);

         ChannelServices.RegisterChannel(channel);

         RemotingConfiguration.RegisterWellKnownServiceType
            (typeof(newclass.ShowCapital), "ShowCapital",
            WellKnownObjectMode.SingleCall);
         Console.WriteLine("Starting...");
         Console.ReadLine();
      }
   }
}
```

Finally a remote client is created which invokes the remotable object that implements the interface assembly:

C# Code 3.7:

```
. . .
      IDCap sh;

      private void button1_Click(object sender, System.EventArgs e)
      {
         string country,cap;

         country=textBox1.Text;
         cap=sh.show(country);
         textBox2.Text= cap;
      }
      private void Form1_Load(object sender, System.EventArgs e)
      {
         TcpClientChannel channel = new TcpClientChannel();

         ChannelServices.RegisterChannel(channel);

         sh= (IDCapital.IDCap) Activator.GetObject(typeof(IDCap),
            "tcp://localhost:1234/ShowCapital");

      }
```

The references are added as defined in Figure 3.17.

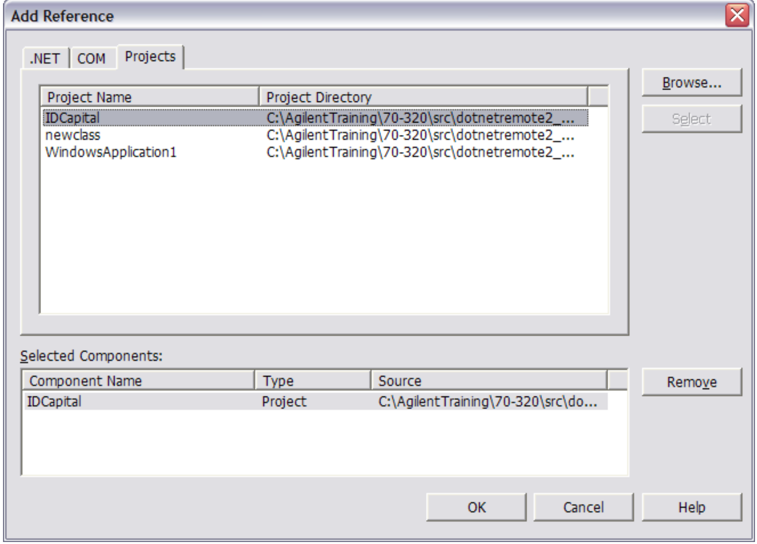

Figure 3.17: .NET remoting

3.8.1 Soapsuds Tool for Interface Assembly Generation

Another method which can be used to generate an interface assembly rather than the implementation assembly is to use the Soapsuds tool. For this the tool requires the URL of the remotable object, of which Soapsuds automatically generates the interface assemblies. An example, based on the previous sections is:

```
soapsuds -url:http://localhost:1234/ShowCapital?wsdl -oa:newclass.dll -nowp
```

which uses the required URL, and generates a DLL named newclass.dll. For example:

```
> soapsuds -url:http://localhost:1234/ShowCapital?wsdl -oa:newclass.dll -nowp
```

and listing the directory gives:

```
>dir
15/01/2005  21:20              1,078 App.ico
15/01/2005  21:20              2,426 AssemblyInfo.cs
16/01/2005  18:11    <DIR>           bin
16/01/2005  17:55                732 Class1.cs
16/01/2005  18:38              3,584 newclass.dll
16/01/2005  17:56              4,737 newclass2.csproj
16/01/2005  18:27              1,803 newclass2.csproj.user
16/01/2005  17:16    <DIR>           obj
```

The references are then added as Figure 3.18 and Figure 3.19.

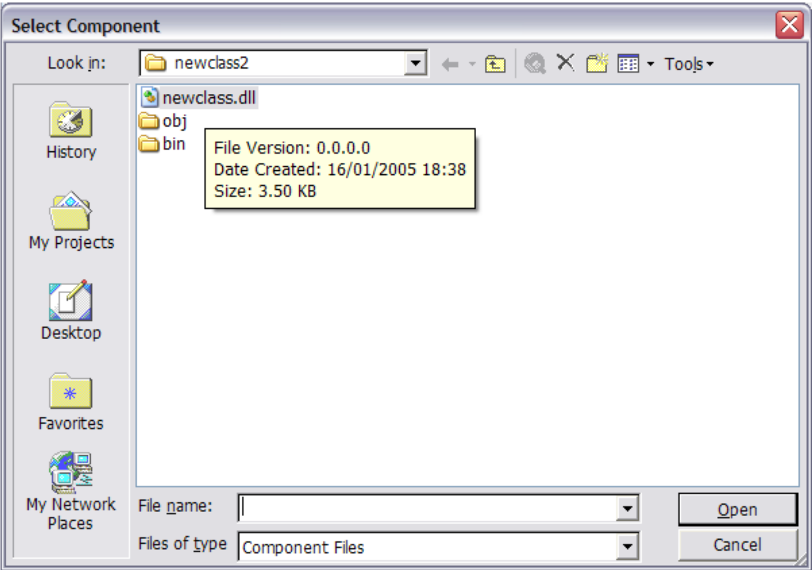

Figure 3.18: DLL

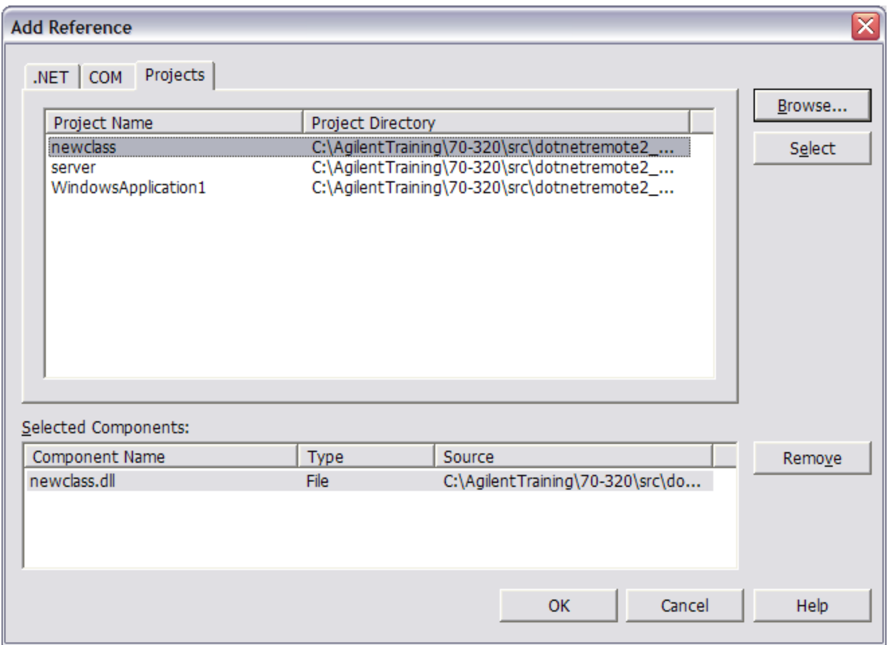

Figure 3.19: References

3.8.2 Tutorial 4

Q3.11 Modify C# Code 3.1, C# Code 3.2 and C# Code 3.3, so it uses an interface assembly.