Implement the following interface running under Microsoft .NET or Rotor:

```
namespace ReverseAPI
{
   public interface Reverser
   {
      string reverse(string arg);
   }
}
```

Use a TCP channel listening on port 8421 in order to perform this task.

The "dedicated" language of Microsoft .NET is called C# to show some progress compared to C++. Even though it lacks the most important feature of a programming language, the templates, it fits quite well in the .NET framework and allows a quick and easy development of distributed systems.

One of .NET's core features is Remoting which encapsulates a connectivity platform more or less comparable to Corba CCM. Together with the Introspection API it enables one to abdicate the use of an additional interface description language like IDL, CEDL or MIDL.

I decided to compile the interface (not its implementation !) to a binary file, i.e. a DLL. That file has to be present both at the client and the server side. By distributing a binary file, I do not have to rely on a source code based solution invoking a compiler and gain much flexibility as well as protection of intellectual properties.

In conclusion, the system looks like this:

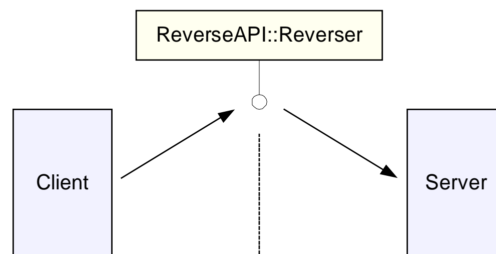

Figure 1: Client-Server Scenario

The server has access to the `RerverseAPI::Reverser` interface and to its actual implementation (I called it `ReverserImplementation`). On the other hand, the client can access the interface but not its implementation. Indeed, he does not need to know how the concrete implementation is programmed.

Reversing a string turns out to be a matter of just four lines:

```
string strResult = "";

// iterate through the string and build a reverse copy
foreach (char letter in arg)
   strResult = letter + strResult;

return strResult;
```

The main task of the server is to register its interface implementation in the .NET framework. To do so, the program obtains a new `TcpServerChannel` at port 8421. Then, the method `RemotingConfiguration.RegisterWellKnownServiceType` adds the implementation to the name service. I decided to use a `SingleCall` because the service does not have a long activation time.

```
class Server
{
   static void Main()
   {
      // reserve port 8421
      ChannelServices.RegisterChannel(new TcpServerChannel(8421));
```

```
        // register the implementation (clients access it through the interface !)
        // using the URI "ReverserServer" and SingleCall activation
        RemotingConfiguration.RegisterWellKnownServiceType
           (typeof(ReverserImplementation), "ReverserServer",
            WellKnownObjectMode.SingleCall);

        // and wait for incoming requests ...
        Console.WriteLine("Server ready ... press Enter to terminate !");
        Console.ReadLine();
     }
  }
```

The client registers a channel, too. This time, a client channel is invoked with no explicit port number. A bit tricky is to obtain a remote object just from an interface because we cannot make use of the standard mechanism that depends on the `new` keyword since it is impossible to apply `new` to an interface. As a workaround, all I have to provide to the static method `Activator.GetObject` is an interface – exactly what I wanted to do. The URI has to be preceded by `tcp://` in order to indicate that the TCP based binary protocol is utilized.

```
      // request an available TCP based channel
      ChannelServices.RegisterChannel(new TcpClientChannel());

      // create a proxy for the remote object
      Reverser proxy = (Reverser)Activator.GetObject(typeof(Reverser),
                         "tcp://localhost:8421/ReverserServer");
```

Now that the proxy came to life, the client can call any operation he likes to. Due to the simple structure of the given `Reverser` interface, the client is unfortunately restricted to `reverse`.

```
      // actually invoke the proxy
      Console.WriteLine("Reversing the string \"Microsoft .NET\": \"" +
                         proxy.reverse("Microsoft .NET") + "\"");
```

There are a few conditions I did not mention: in a distributed system the network connection may fail, the server may be unable to provide an object the fits to the proxy and so on. In the code you find at the end of this paper I catch these exceptions (`RemoteException`) and verify that the proxy points to a valid object (not `null`).

It was interesting how simple and easy a distributed system can be written using the astonishing flexible .NET framework and C#. With the help of Microsoft's Visual Studio .NET the whole project was completed in less than an hour. The same program written for Corba – assignment 3 – required a far high effort of time and the resulting program was a bit longer and syntactically complex. Not to forget that the C#/.NET solution runs out-of-the-box if the framework is installed.

Reverser.cs

```
// ///////////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 7, Stephan Brumme, 702544
//
// Exactly as provided by Mr. von Löwis !
//

namespace ReverseAPI
{
   public interface Reverser
   {
      string reverse(string arg);
   }
}
```

ReverserImplementation.cs

```
// ///////////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 7, Stephan Brumme, 702544
//
// Implements the Reverse interface
//

namespace ReverseAPI
{
   /// <summary>
   /// Class that actually implements the Reverser interface
   /// </summary>
   public class ReverserImplementation : System.MarshalByRefObject, Reverser
   {
      public string reverse(string arg)
      {
         string strResult = "";

         // iterate through the string and build a reverse copy
         foreach (char letter in arg)
            strResult = letter + strResult;

         return strResult;
      }
   }
}
```

Server.cs

```
// ///////////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 7, Stephan Brumme, 702544
//
// Microsoft .NET server that exports a "reverse" operation
//

// for console output
using System;

// remoting
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

// reverse operation
using ReverseAPI;


namespace Aufgabe7
{
    /// <summary>
    /// The server class registers itself at port 8421
    /// </summary>
    class Server
    {
        static void Main()
        {
            // reserve port 8421
            ChannelServices.RegisterChannel(new TcpServerChannel(8421));

            // register the implementation (clients access it through the interface !)
            // using the URI "ReverserServer" and SingleCall activation
            RemotingConfiguration.RegisterWellKnownServiceType
               (typeof(ReverserImplementation), "ReverserServer",
                WellKnownObjectMode.SingleCall);

            // and wait for incoming requests ...
            Console.WriteLine("Server ready ... press Enter to terminate !");
            Console.ReadLine();
        }
    }
}
```

Client.cs

```
// ////////////////////////////////////////////////////////
// Lecture on the CORBA Component Model, summer term 2003
// Assignment 7, Stephan Brumme, 702544
//
// Microsoft .NET client that calls a remote "reverse" operation
//

// for console output
using System;

// remoting
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

// reverse operation
using ReverseAPI;


namespace Aufgabe7
{
   /// <summary>
   /// Encapsulates the Main method
   /// </summary>
   class Client
   {
      static void Main()
      {
         // catch any RemoteException
         try
         {
            // request an available TCP based channel
            ChannelServices.RegisterChannel(new TcpClientChannel());

            // create a proxy for the remote object
            Reverser proxy = (Reverser)Activator.GetObject(typeof(Reverser),
                                "tcp://localhost:8421/ReverserServer");

            // failed ?
            if (proxy == null)
            {
               Console.WriteLine("Invalid reference.");
               return;
            }

            // actually invoke the proxy
            Console.WriteLine("Reversing the string \"Microsoft .NET\": \"" +
                               proxy.reverse("Microsoft .NET") + "\"");
         }
         catch (RemotingException)
         {
            // something went wrongs
            Console.WriteLine("Server not found.");
         }
      }
   }
}
```