In this laboratory project you will build a number of Client/Server applications using C# and the .NET framework. The first will be a simple console application with one-way text transfer. Next, you will modify this application to provide for two-way communication.

One of the simplest types of machine-to-machine communication is the client-to-server with a static IP address. In the first part of this laboratory will create a simple client and a server as console applications. The server will be able to accept a connection from any client, but the client must know the IP address of the server. The source code for the client is provided below.

```csharp
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace Client1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            // TODO: Add code to start application here
            string sendStr="";
            UdpClient theClient= new UdpClient("server IP here",9050);

            while (!sendStr.Trim().ToUpper().Equals("END"))
            {
                // Console.Write("# ");
                sendStr=Console.ReadLine();
                byte[] myData= new byte[1024];
                myData=Encoding.ASCII.GetBytes(sendStr);
                theClient.Send(myData,myData.Length);
            }
            theClient.Close();
        }
    }
}
```

This client/server application will use the UDP (User Datagram Protocol). Since this protocol provides only limited error correction it is more applicable to message broadcasting rather than file transfer. You should compare this protocol with TCP.
In this application, a message typed into the console will be transferred to the designated server as an ASCII encoded text string. The server for this application is very similar in form to the client. As shown in the source code below the server listened for and establishes a communication socket with ANY client.

```csharp
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

namespace Server1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            // TODO: Add code to start application here
            //
            string rcvData="";
            IPEndPoint IPEP= new IPEndPoint(IPAddress.Any,9050);
            UdpClient theSock= new UdpClient(IPEP);
            IPEndPoint fromClient;
            while (!rcvData.Trim().ToUpper().Equals("END"))
            {
                byte[] myData= new byte[1024];
                fromClient= new IPEndPoint(IPAddress.Any,0);
                myData= theSock.Receive(ref fromClient);
                rcvData=Encoding.ASCII.GetString(myData);
                Console.WriteLine(fromClient.ToString() + " " + rcvData);
            }
            theSock.Close();
        }
    }
}
```

For this application we have chosen port 9050 somewhat arbitrarily. Note that when the server receives the text string "end" (in upper- or lowercase) it closes the socket to end the communication. Build, run and test this simple client/server application in the laboratory or on your own computer.

**Client/Server Application**

Now we will build a simple client/server console application for two-way communication.

**Step 1:** Create a new project called ClientServer_1 (console application). Rename the source code from Class1.cs to Chat1.cs (be sure to rename Class1 in the program as well).

**Step 2:** Create a private static Thread called (clientThread) in the Chat1 class.

```csharp
private static System.Threading.Thread clientThread;
```
Step 3: Write a method to create threads. This method will be called from `Main()` as shown below. The `CreateThreads()` method will be used to start the client and server threads. Let’s build the `RunClientThread()` first.

```csharp
public static void Main(string[] args)
{
    CreateThreads();
}

private static void CreateThreads()
{
    clientThread = new System.Threading.Thread(new
    System.Threading.ThreadStart(RunClientThread));
    clientThread.Start();
}

private static void RunClientThread()
{
    // RunThread code goes here
}
```

We can avoid typing the namespace paths by adding a `using System.Threading` command to our code, as shown,

```csharp
using System;
using System.Threading;
:
:
private static void CreateThreads()
{
    clientThread = new Thread(new ThreadStart(RunClientThread));
    clientThread.Start();
}
```

Step 4: Now we can expand the operations of the `RunClientThread()` method. First the method will create an empty string to hold the text typed into the console window. Next you will create a new `UdpClient`. You must know the IP address of the server that will be receiving the text message. Again we select port 9050.

```csharp
private static void RunClientThread()
{
    string sendStr="";
    UdpClient theClient= new UdpClient("server IP here",9050);
    while (!sendStr.Trim().ToUpper().Equals("END"))
    {
        sendStr=Console.ReadLine();
        byte[] myData= new byte[1024];
        myData=Encoding.ASCII.GetBytes(sendStr);
        theClient.Send(myData,myData.Length);
    }
    theClient.Close();
}
```

The `Console.ReadLine()` is a function that reads a line of text from the console window and assigns it to the specified string variable (`sendStr` in this case). Communication between machines will be with `bytes` rather than ASCII text so we create a byte array of length 1024 to hold the converted message. `GetBytes` converts the text in `sendStr` into binary (`bytes`), `GetBytes` is a method of the property `ASCII` from the `Encoding` class of the `System.Text` namespace as shown in the diagram below:
Step 5: Now create a `RunServerThread()` method in a manner analogous to the `RunClientThread()`. Just as with the `clientThread` you will need to create a private static `Thread` variable in the `Chat1` class and an instance of the `serverThread` in the `CreateThreads()` method.

```csharp
private static void RunServerThread()
{
    string rcvData="";
    IPEndPoint IPEP = new IPEndPoint(IPAddress.Any,9050);
    UdpClient theSock = new UdpClient(IPEP);
    IPEndPoint fromClient;

    while (!rcvData.Trim().ToUpper().Equals("END"))
    {
        byte[] myData = new byte[1024];
        fromClient = new IPEndPoint(IPAddress.Any,0);
        myData = theSock.Receive(ref fromClient);
        rcvData = Encoding.ASCII.GetString(myData);
        Console.WriteLine(fromClient.ToString() + " " + rcvData);
    }
    theSock.Close();
}
```

In the server we must specify the port in the `IPEndPoint` class (we use port 9050 as before). We also create a `UdpClient` socket called `theSock`.

Step 6: Finally create a new project `ClientServer_2`. The source code will be identical except for IP address specified in the `RunClientThread()` (assuming that the communicating applications are running on different machines). You should have `ClientServer_1` send to port 9050 and receive from port 9051 and vice versa for `ClientServer_2`. This avoids a possible exception and prevents echoing of the text in the console window of the sending application when both applications are running on the same machine.

Build and test a pair of `ClientServer` applications. Submit the annotated source code.