C#

Socket Programming in C# − Part I
Contributed by Ashish Dhar
2003−07−24
[*Send Me Similar Content When Posted*]
[*Add Developer Shed Headlines To Your Site*]

DISCUSS NEWS SEND PRINT PDF

Article Index:

If you have dealt with sockets in the past, you may be interested in learning how it is done using C# technology. Read more ...The purpose of this article is to show you how you can do socket programming in C#. This article assumes some familiarity with the socket programming, though you need not to be expert in socket programming. There are several flavors to socket programming − like client side , server side , blocking or synchronous , non−blocking or asynchronous etc.

With all these flavors in mind , I have decided to break this subject into two parts. In the part 1 I will start with the client side blocking socket. Later on in the second part I will show you how to create server side and non−blocking.

Network programming in windows is possible with sockets. A socket is like a handle to a file. Socket programming resembles the file IO as does the Serial Communication. You can use sockets programming to have two applications communicate with each other. The application are typically on the different computers but they can be on same computer. For the two applications to talk to each either on the same or different computers using sockets one application is generally a server that keeps listening to the incoming requests and the other application acts as a client and makes the connection to the server application.

The server application can either accept or reject the connection. If the server accepts the connection, a dialog can begin with between the client and the server. Once the client is done with whatever it needs to do it can close the connection with the server. Connections are expensive in the sense that servers allow finite connections to occur. During the time client has an *active connection* it can send the data to the server and/or receive the data.

The complexity begins here. When either side (client or server) sends data the other side is supposed to read the data. But how will the other side know when data has arrived. There are two options − either the application needs to *poll* for the data at regular intervals or there needs to be some sort of mechanism that would enable application to get notifications and application can read the data at that time. Well , after all Windows is an event driven system and the notification system seems an obvious and best choice and it in fact is.

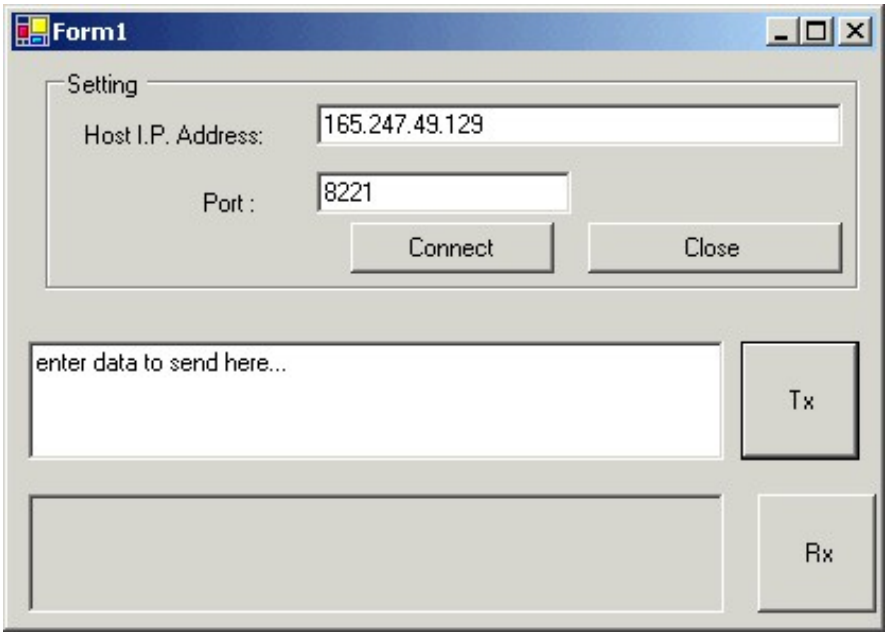
As I said the two applications that need to communicate with each other need to make a connection first. In order for the two application to make connections the two applications need to identify each other (or each other's computer). Computers on network have a unique identifier called I.P. address which is represented in *dot−notation* like 10.20.120.127 etc. Lets see how all this works in .NET.

System.Net.Sockets Namespace

Before we go any further, download the source code attached with this article. Extract the zip file to a folder say c:\Temp you will see following two folders :

- Server
- Client

In the Server folder there will be one EXE . And in the client there will be the source code in C# that is our client. There will be one file called SocketClient.sln which the solution file . If you double click that your VS.NET will be launched and you will see the project SocketClientProj in the solution. Under this project you will have SocketClientForm.cs file. Now build the code (by pressing Ctrl−Shift−B) and run the code you will see the following dialog box:



As you can see the dialog box has a field for Host IP address (which is the IP address of the machine on which you will run the Server Application (located under Server folder)). Also there is a field where you can specify port number at which the Server is listening. The server app I have provided here listens at port 8221. So I have specified port to be 8221.

After specifying these parameters we need to connect to the server. So pressing Connect will connect to the server and to close the connection press Close. To send some data to the server type some data in the field near the button name Tx and if you press Rx the application will block unless there is some data to read.
With this info lets now try to check the code behind this:

Socket programming in .NET is made possible by Socket class present inside the System.Net.Sockets namespace.
Socket class has several method and properties and a constructor.
The first step is to create an object of this class.
Since there is only one constructor we have no choice but to use it.

Here is how to create the socket:

```
m_socListener = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.IP);
```

The first parameter is the address family which we will use interNetwork − other options include Banyan NetBios etc.
AddressFamily is an enum defined in Sockets namespace.
Next we need to specify socket type: and we would use reliable two way connection−based sockets (stream) instead of un−reliable Connectionless sockets (datagrams) . So we obviously specify stream as the socket type

and finally we are using TCP/IP so we would specify protocol type as Tcp.

Once we have created a Socket we need to make a connection to the server since we are using connection−based communication.
To connect to the remote computer we need to know the IP Address and port at which to connect.
In .NET there is a class under System.Net namespace called IPEndPoint which represents a network computer as an IP address and a port number.

The IPEndPoint has two constructors − one that takes a IP Address and Port number and one that takes long and port number. Since we have computer IP address we would use the former

public IPEndPoint(System.Net.IPAddress address, int port);

As you can see the first parameter takes a IPAddress object. If you examine the IPAddress class you will see that it has a static method called Parse that returns IPAddress given a string (of dot notation) and second parameter will be the port number. Once we have endpoint ready we can use Connect method of Socket class to connect to the end point (remote server computer).
Here is the code:

```
System.Net.IPAddress ipAdd = System.Net.IPAddress.Parse("10.10.101.200");
System.Net.IPEndPoint remoteEP = new IPEndPoint (iAdd,8221);
m_socClient.Connect (remoteEP);
```

These three lines of code will make a connection to the remote host running on computer with IP 10.10.101.200 and listening at port 8221. If the Server is running and started (listening), the connection will succeed. If however the server is not running an exception called SocketException will be thrown. If you catch the exception and check the Message property of the exception in this case you see following text:

"No connection could be made because the target machine actively refused it."

Similarly if you already have made a connection and the server somehow dies , you will get following exception if you try to send data.

"An existing connection was forcibly closed by the remote host"

Assuming that the connection is made, you can send data to other side using the Send method of the Socket class. Send method has several overloads. All of them take a byte array . For example if you want to send "Hello There" to host you can use following call:

```
try
{
String szData = "Hello There";
byte[] byData = System.Text.Encoding.ASCII.GetBytes(szData);
m_socClient.Send(byData);
}
catch (SocketException se)
{
MessageBox.Show ( se.Message );
}
```

Note that the Send method is blocking. What it means the call will block till the data has been sent or an exception has been thrown. There is an non−blocking version of the send which we will discuss in the next part of this article.
Similar to Send there is a Receive method on the Socket class. You can receive data using following call:

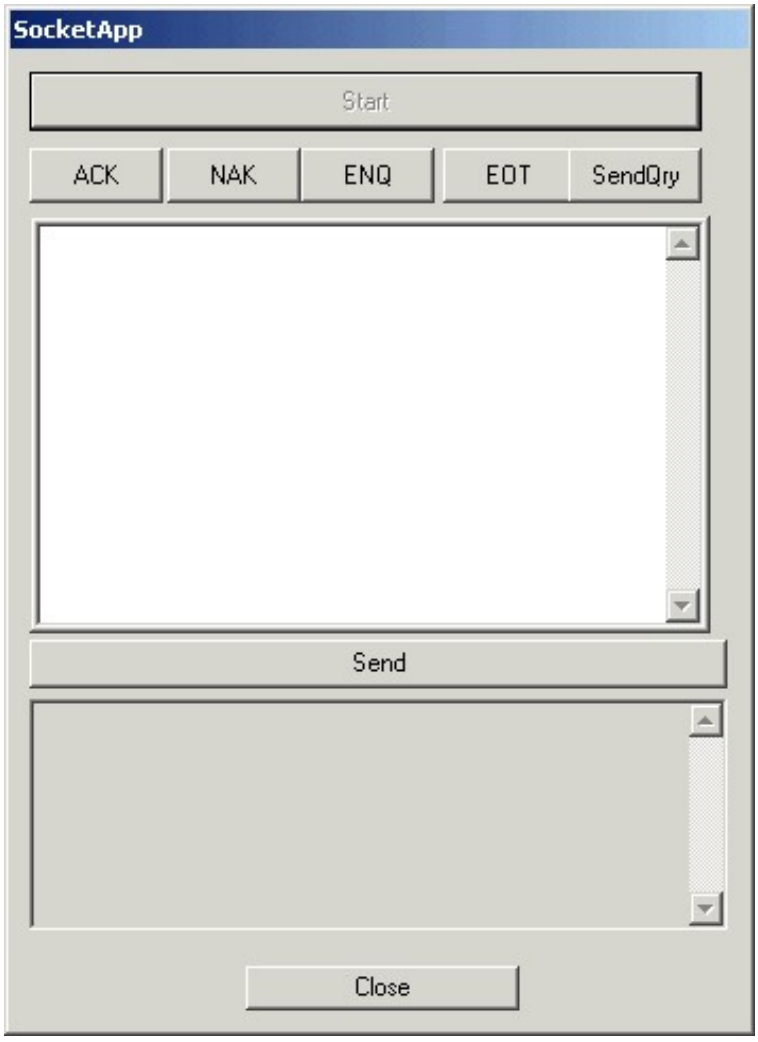
```
byte [] buffer = new byte[1024];
int iRx = m_socClient.Receive (buffer);
```

The Receive method again is blocking. It means that if there is no data available the call will block until some

data arrives or an exception is thrown.

Non−blocking version of Receive method is more useful than the non−blocking version of Send because if we opt for block Receive , we are effectively doing polling. There is no events about data arrival. This model does not work well for serious applications. But all that is the subject of our next part of this article. For now we will settle with the blocking version.
In order to use the source code and application here you would need to run the Server first:

Here is the way Server looks like:



When you launch the Server, click Start to start listening. The Server listens at port 8221. So make sure you specify the port number 8221 in the port field of our client application. And in the IPAddress field of Client App enter the IP Address of the machine on which the Server is running. If you send some data to server from the client by pressing Tx button, you will see that data in the grayed out edit box.

C#

Socket Programming in C# − Part II
Contributed by Ashish Dhar
2003−07−25
[*Send Me Similar Content When Posted*]
[*Add Developer Shed Headlines To Your Site*]

Article Index:

This is the second part of Ashish's two part series about handling sockets in the C# language. Read this article to learn how to use sockets with the .Net framework.This is the second part of the previous article about the socket programming. In the earlier article we created a client but that client used to make blocking IO calls (Receive) to read data at regular intervals (via clicking the Rx button). But as I said in my earlier article, that model does not work very well in a real world application. Also since Windows is an events−based system, the application (client) should get notifications of some kind whenever the data is received so that client can read it rather than client continuously polling for data.

Well that is possible with a little effort. If you read the first part of this article, you already know that the Socket class in the Systems.Net.Sockets namespace has several methods like Receive and Send which are blocking calls. Besides there are also functions like BeginReceive , BeginSend etc. These are meant for asynchronous IO . For example , there are at least two problems with the blocking Receive:

1. When you call Receive function the call blocks if no data is present, the call blocks till some data arrives.
2. Even if there is data when you made the receive call , you don't know when to call next time. You need to do polling which is not an efficient way.

Although you can argue that one can overcome these *shortcomings* by multithreading meaning that one can spawn a new thread and let that thread do the polling and notifies the main thread of the data. Well this concept will work well. But even if you create a new thread it would require your main thread to share the CPU time with this new thread. Windows operating system (Windows NT /2000 /XP) provide what is called Completion Port IO model for doing overlapped (asynchronous) IO.

The details of IO Completion port are beyond the scope of the current discussion, but to make it simple you can think of IO Completion Ports as the most efficient mechanism for doing asynchronous IO in Windows that is provided by the Operating system. Completion Port model can be applied to any kind of IO including the file read /write and serial communication.

The .NET asynchronous socket programming helper class's Socket provides the similar model.

BeginReceive

.NET framework's Socket class provides BeginReceive method to receive data asynchronously i.e., in an non−blocking manner The BeginReceive method has following signature:

public **IAsyncResult BeginReceive**(byte[] buffer, int offset, int size, SocketFlags socketFlags, AsyncCallback callback, object state);

The way BeginReceive function works is that you pass the function a buffer , a callback function (delegate) which will be called whenever data arrives.

The last parameter, object, to the BeginReceive can be any class derived from object (even null) .
When the callback function is called it means that the BeginReceive function completed which means that the data has arrived.
The callback function needs to have the following signature:

void AsyncCallback(IAsyncResult ar);

As you can see the callback returns void and is passed in one parameter , **IAsyncResult** interface , which contains the status of the asynchronous receive operation.

The IAsyncResult interface has several properties. The first parameter − AsyncState − is an object which is same as the last parameter that you passed to BeginReceive(). The second property is AsyncWaitHandle which we will discuss in a moment. The third property indicates whether the receive was really asynchronous or it finished synchronously. The important thing to follow here is that it not necessary for an asynchronous function to always finish asynchronously − it can complete immediately if the data is already present. Next parameter is IsComplete which indicates whether the operation has completed or not.

If you look at the signature of the BeginReceive again you will note that the function also returns ***IAsyncResult***. This is interesting. Just now I said that I will talk about the second peoperty of the IAsyncResult in a moment. Now is that moment. The second parameter is called **AsyncWaitHandle.**

The AsyncWaitHandle is of type **WaitHandle**, a class defined in the System.Threading namespace. WaitHandle class encapsulates a Handle (which is a pointer to int or handle) and provides a way to wait for that handle to become signaled. The class has several static methods like WaitOne (which is similar to WaitForSingleObject) WaitAll (similar to WaitForMultipleObjects with waitAll true) , WaitAny etc. Also there are overloads of these functions available with timeouts.

Coming back to our discussion of IAsyncResult interface, the handle in AsyncWaitHandle (WaitHandle) is signalled when the receive operation completes. So if we wait on that handle infinitely we will be able to know when the receive completed. This means if we pass that WaitHandle to a different thread, the different thread can wait on that handle and can notify us of the fact that the data has arrived and so that we can read the data. So you must be wondering if we use this mechanism why would we use callback function. We won't. Thats right. If we choose to use this mechanism of the WaitHandle then the callback function parameter to the BeginReceive can be null as shown here:

```
//m_asynResult is declared of type IAsyncResult and assumming that m_socClient has made a connection.
m_asynResult = m_socClient.BeginReceive(m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None,null,null);
if ( m_asynResult.AsyncWaitHandle.WaitOne () )
{
int iRx = 0 ;
iRx = m_socClient.EndReceive (m_asynResult);
char[] chars = new char[iRx + 1];
System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
int charLen = d.GetChars(m_DataBuffer, 0, iRx, chars, 0);
System.String szData = new System.String(chars);
txtDataRx.Text = txtDataRx.Text + szData;
}
```

Even though this mechanism will work fine using multiple threads, we will for now stick to our callback mechanism where the system notifies us of the completion of asynchronous operation which is Receive in this

case .
Lets say we made the call to BeginReceive and after some time the data arrived and our callback function got called.Now question is where's the data? The data is now available in the buffer that you passed as the first parameter while making call to BeginReceive() method . In the following example the data will be available in m_DataBuffer :

BeginReceive(**m_DataBuffer**,0,m_DataBuffer.Length,SocketFlags.None, pfnCallBack,null);

But before you access the buffer you need to call EndReceive() function on the socket. The EndReceive will return the number of bytes received . Its not legal to access the buffer before calling EndReceive.
To put it all together look at the following simple code:

```
byte[] m_DataBuffer = new byte [10];
IAsyncResult m_asynResult;
public AsyncCallback pfnCallBack ;
public Socket m_socClient;
// create the socket...
public void OnConnect()
{
m_socClient = new Socket (AddressFamily.InterNetwork,SocketType.Stream ,ProtocolType.Tcp );
// get the remote IP address...
IPAddress ip = IPAddress.Parse ("10.10.120.122");
int iPortNo = 8221;
//create the end point
IPEndPoint ipEnd = new IPEndPoint (ip.Address,iPortNo);
//connect to the remote host...
m_socClient.Connect ( ipEnd );
//watch for data ( asynchronously )...
WaitForData();
}
public void WaitForData()
{
if ( pfnCallBack == null )
pfnCallBack = new AsyncCallback (OnDataReceived);
// now start to listen for any data...
m_asynResult =
m_socClient.BeginReceive (m_DataBuffer,0,m_DataBuffer.Length,SocketFlags.None, pfnCallBack,null);
}
public void OnDataReceived(IAsyncResult asyn)
{
//end receive...
int iRx = 0 ;
iRx = m_socClient.EndReceive (asyn);
char[] chars = new char[iRx + 1];
System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
int charLen = d.GetChars(m_DataBuffer, 0, iRx, chars, 0);
System.String szData = new System.String(chars);
WaitForData();
}
```

The OnConnect function makes a connection to the server and then makes a call to WaitForData. WaitForData creates the callback function and makes a call to BeginReceive passing a global buffer and the callback function. When data arrives the OnDataReceive is called and the m_socClient's EndReceive is called which returns the number of bytes received and then the data is copied over to a string and a new call is made to WaitForData which will call BeginReceive again and so on. This works fine if you have one socket in you application.

MULTIPLE SOCKETS

Now lets say you have two sockets connecting to either two different servers or same server(which is valid) . One way is to create two different delegates and attach a different delegate to different BeginReceive function. What if you have 3 sockets or for that matter n sockets , this approach of creating multiple delegates does not fit well in

such cases. So the solution should be to use only one delegate callback. But then the problem is how do we know what socket completed the operation.

Fortunately there is a better solution. If you look at the BeginReceive function again, the last parameter is a state is an object. You can pass anything here . And whatever you pass here will be passed back to you later as the part of parameter to the callback function. Actually this object will be passed to you later as a IAsyncResult.AsyncState. So when your callback gets called, you can use this information to identify the socket that completed the operation. Since you can pass any thing to this last parameter, we can pass a class object that contains as much information as we want. For example we can declare a class as follows:

```
public class CSocketPacket
{
public System.Net.Sockets.Socket thisSocket;
public byte[] dataBuffer = new byte[1024];
}
```

and call BeginReceive as follows:

```
CSocketPacket theSocPkt = new CSocketPacket ();
theSocPkt.thisSocket = m_socClient;
// now start to listen for any data...
m_asynResult = m_socClient.BeginReceive (theSocPkt.dataBuffer ,0,theSocPkt.dataBuffer.Length ,
SocketFlags.None,pfnCallBack,theSocPkt);
```

and in the callback function we can get the data like this:

```
public void OnDataReceived(IAsyncResult asyn)
{
  try
  {
CSocketPacket theSockId = (CSocketPacket)asyn.AsyncState ;
//end receive...
int iRx = 0 ;
iRx = theSockId.thisSocket.EndReceive (asyn);
char[] chars = new char[iRx + 1];
System.Text.Decoder d = System.Text.Encoding.UTF8.GetDecoder();
int charLen = d.GetChars(theSockId.dataBuffer, 0, iRx, chars, 0);
System.String szData = new System.String(chars);
txtDataRx.Text = txtDataRx.Text + szData;
WaitForData();
  }
  catch (ObjectDisposedException )
  {
System.Diagnostics.Debugger.Log(0,"1","\nOnDataReceived: Socket has been closed\n");
  }
  catch(SocketException se)
  {
MessageBox.Show (se.Message );
  }
}
```

To see the whole application download the code and you can see the code.

There is one thing which you may be wondering about. When you call BeginReceive , you have to pass a buffer and the number of bytes to receive. The question here is how big should the buffer be. Well, the answer is it depends. You can have a very small buffer size say, 10 bytes long and if there are 20 bytes ready to be read, then you would require 2 calls to receive the data. On the other hand if you specify the length as 1024 and you know you are always going to receive data in 10−byte chunks you are unnecessarily wasting memory. So the length depends upon your application.

Server Side

If you have understood whatever I have described so far, you will easily understand the Server part of the socket application. So far we have been talking about a client making connection to a server and sending and receiving data.

On the Server end, the application has to send and receive data. But in addition to adding and receiving data, server has to allow the clients to make connections by listening at some port. Server does not need to know client I.P. addresses. It really does not care where the client is because its not the server but client who is responsible for making connection. Server's responsibility is to manage client connections.

On the server side there has to be one socket called the Listener socket that listens at a specific port number for client connections. When the client makes a connection, the server needs to accept the connection and then in order for the server to send and receive data from that connected client it needs to talk to that client through the socket that it got when it accepted the connection . Following code illustrates how server listens to the connections and accepts the connection:

```
public Socket m_socListener;
public void StartListening()
{
 try
 {
//create the listening socket...
m_socListener = new Socket(AddressFamily.InterNetwork,SocketType.Stream,ProtocolType.Tcp);
IPEndPoint ipLocal = new IPEndPoint ( IPAddress.Any ,8221);
//bind to local IP Address...
m_socListener.Bind( ipLocal );
//start listening...
m_socListener.Listen (4);
// create the call back for any client connections...
m_socListener.BeginAccept(new AsyncCallback ( OnClientConnect ),null);
cmdListen.Enabled = false;
 }
 catch(SocketException se)
 {
MessageBox.Show ( se.Message );
 }
}
```

If you look at the above code carefully you will see that its similar to we did in the asynchronous client. First of all the we need to create a listening socket and bind it to a local IP address. Note that we have given Any as the IPAddress . I will explain what it means later. and also we have passed port number as 8221. Next we made a call to Listen function. The 4 is a parameter indicating *backlog* indicating the maximum length of the queue of pending connections.

Next we made a call to BeginAccept passing it a delegate callback. BeginAccept is a non−blocking method that returns immediately and when a client has made requested a connection, the callback routine is called and you can accept the connection by calling EndAccept. The EndAccept returns a socket object which represents the incoming connection. Here is the code for the callback delegate:

```
public void OnClientConnect(IAsyncResult asyn)
{
try
{
  m_socWorker = m_socListener.EndAccept (asyn);
WaitForData(m_socWorker);
}
catch(ObjectDisposedException)
{
        System.Diagnostics.Debugger.Log(0,"1","\n OnClientConnection: Socket has been closed\n");
}
catch(SocketException se)
{
```

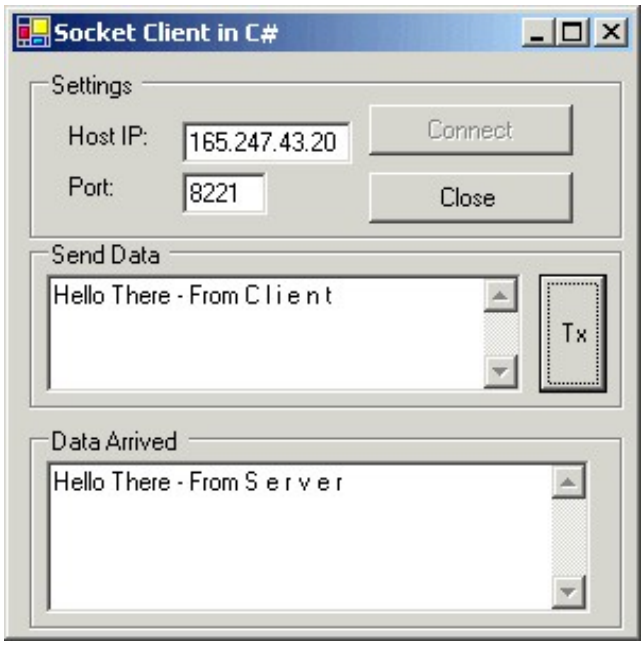
```
MessageBox.Show ( se.Message );
}
}
```

Here we accept the connection and call WaitForData which in turn calls BeginReceive for the m_socWorker.

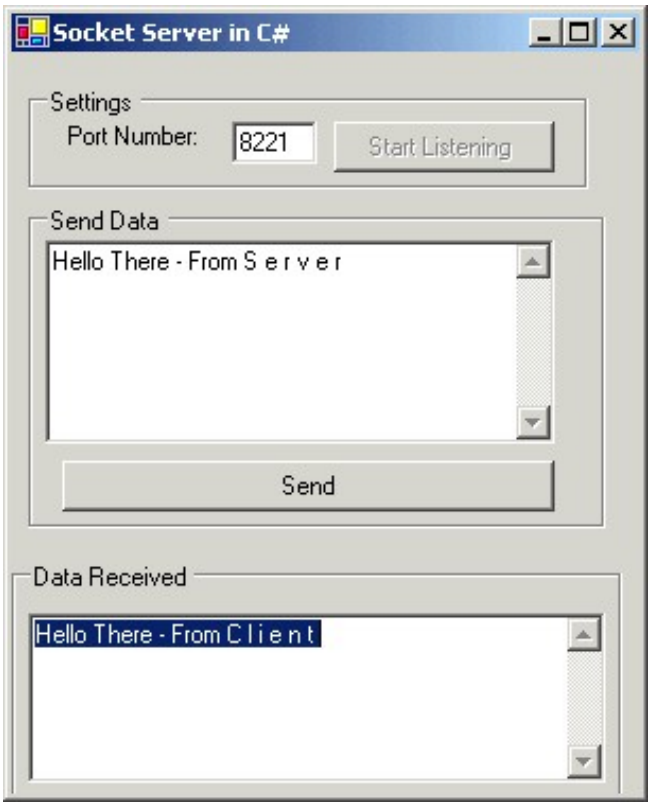
If we want to send data some data to client we use m_socWorker socket for that purpose like this:

```
Object objData = txtDataTx.Text;
byte[] byData = System.Text.Encoding.ASCII.GetBytes(objData.ToString ());
m_socWorker.Send (byData);
```

Here is how our client looks like



Here is how our server looks like



That is all there is to the socket programming.