

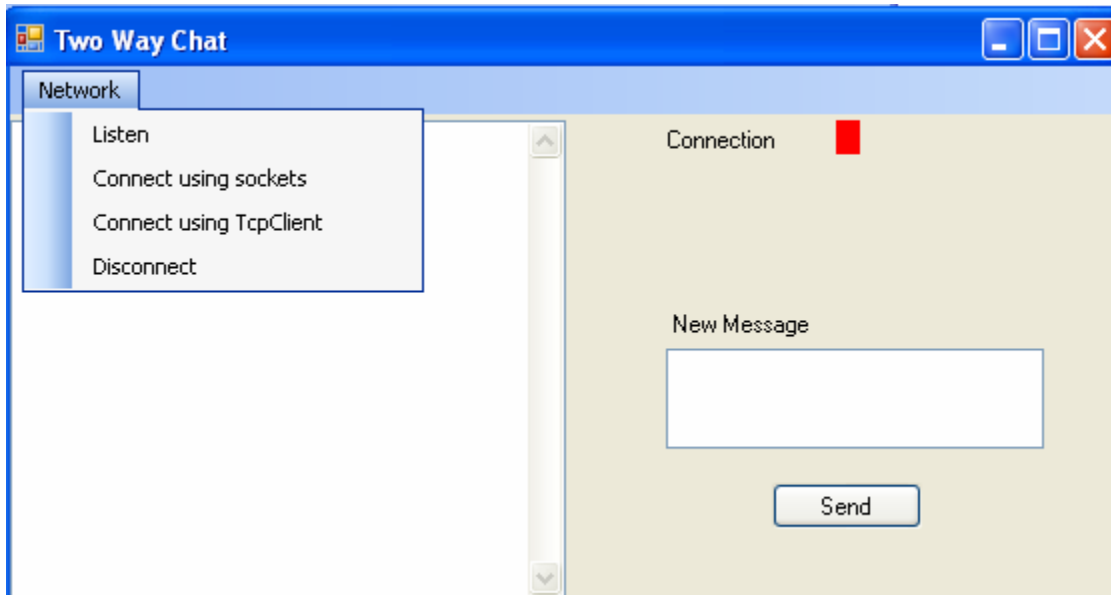
Second Network Programming Example: Two-way Chat

This example application lets two users (anywhere on the Internet) set up a two-way connection directly between their computers. (The world-wide web is not involved at all.) Of course, the two users need to know each other's IP addresses, which they could get by telephone or email. One of the users, say Alice, will start her copy of *TwoWayChat*, and choose *Listen* from the menu. The other user, say Bob, will start his copy of *TwoWayChat* and choose *Connect* from the menu. He will have to enter Alice's IP address and click OK. Then Bob's copy of *TwoWayChat* will connect to Alice's copy, and the two can "chat", exchanging text messages by pressing the *Send* button.

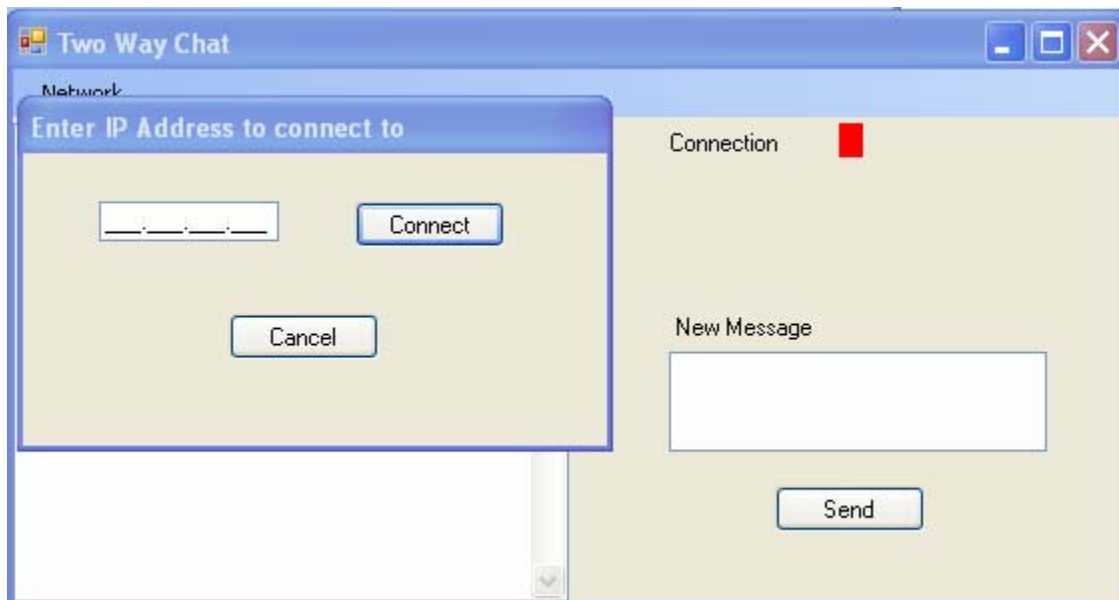
There are three parts to this example:

- The user interface (nothing new here)
- The code for the *Connect* button (we'll do this two ways)
- The code for the *Listen* button (the new part, asynchronous programming)

First let's set up the interface. Start with a *MenuStrip* containing a *Network* item on the menu bar and below it *Listen*, *Connect*, *Disconnect* items. (Actually, the screen shot shows two different *Connect* items because we're going to show two different methods to do it.) Add a textbox to display the chat text (call it *historyBox*). Make it multiline, anchor it to the top, bottom, and left, and give it vertical scroll bars. Make it read-only, and change the background color to something lighter than the default read-only color. Leave some space on the right (making the form itself bigger than default). In that space, put the controls you see in the screen shot below. The "red light" is just a label. To make it look as shown, set its *AutoSize* property to false, and make it a small square, and make its text empty and its name *ConnectionLight*. Then, set its *BackColor* to red (in source code, since you can't do that in the design editor). The textbox to hold a new message is called *sendBox*. The button is called *sendButton*.



When the user chooses *Connect*, they will get a modal dialog designed to collect an IP address from the user. On that dialog, use a *MaskedTextBox*. Set the mask to “000.000.000.000”. The zeroes stand for a digit. The *MaskedTextBox* type is used to perform data validation *within* a text box. Here’s a screen shot of this dialog:



Try out the masked text box!

Here's the first part of the code for the connect button handler, to bring the dialog up and initialize *m_ListenerIPAddress* when the dialog closes. You need this much to duplicate the screen shot.

```
private void connectToolStripMenuItem_Click(object sender,
                                           EventArgs e)
{
    Form2 dlg = new Form2();
    if (m_ListenerIPAddress != null)
        dlg.IPAddress = m_ListenerIPAddress.ToString();
    if (dlg.ShowDialog() != DialogResult.OK)
        return; // user cancelled
    m_ListenerIPAddress = IPAddress.Parse(dlg.IPAddress);
}
```

Now we're ready for the network programming. Make the first few lines of your source file look like this (after the *using* commands already present). These member variables will be explained as we go.

```
using System.Net;
using System.Net.Sockets;

namespace TwoWayChat
{
    public partial class Form1 : Form
    {
        Socket m_theSocket;
        int m_ListenPort = 8190;
        int m_ClientPort = 8191;
        IPAddress m_ListenerIPAddress = null;
        TcpListener m_Listener;
        NetworkStream m_theStream;
        byte[] m_data = new byte[1024];
        // buffer to receive incoming chat messages
        public Form1()
```

Asynchronous Programming for the Listener

The handler for the *Listen* button has to start the program listening, but without just making the program just hang until someone connects. This is the new part in this example. We will use the .NET class *TcpListener*. This class offers a *Start* method. That method (internally) starts a new thread to listen to the specified port. When a client tries to connect, it “raises an asynchronous event”. You will then have to handle that event just as you would handle a normal event. You write a handler for it, which we will call

listen. How does the system know to route this asynchronous event to your *Form1*? The crucial line of code for that is

```
m_Listener.BeginAcceptSocket(newAsyncCallback(listen),this);
```

Here *this* refers to our instance of *Form1*. This model of “asynchronous programming” is intended to spare you from the details of creating and managing your own threads.

Here is the code. Study it carefully!

```
public static void listen(IAAsyncResult ar)
// callback for the asynchronous event of a
// client connecting .
// since it's static it doesn't have access to Form1's
// nonstatic members.
// it gets an instance of Form1 as ar.AsyncState,
// because we pass one when we call BeginAcceptSocket.
{
    Form1 f = (Form1)ar.AsyncState; // we passed this when
    // we called BeginAcceptSocket
    TcpListener L = f.m_Listener;
    f.m_theSocket = L.EndAcceptSocket(ar);
    // now we have a connection to a client
    f.listenToolStripMenuItem.Enabled = false;
    f.connectToolStripMenuItem.Enabled = false;
    f.ConnectionLight.BackColor = Color.Green;
    f.Text = "Two Way Chat--Listener";
    f.m_theStream = new NetworkStream(f.m_theSocket);
    f.startChat(); // we'll write this soon
}

private void listenToolStripMenuItem_Click(object sender,
                                           EventArgs e)
{
    try
    {
        m_Listener = new TcpListener(m_ListenPort);
        m_Listener.Start(); // start the listening thread;
        m_Listener.BeginAcceptSocket(
            new AsyncCallback(listen),this
        );
        this.Text = "Two Way Chat--Listener";
        listenToolStripMenuItem.Enabled = false;
    }
    catch (Exception ex)
    {

```

```
        MessageBox.Show(ex.ToString());
    }
}
```

This works as follows: when the user clicks *Listen*, a new *TcpListener* is constructed. The next two lines start a thread and tell it to listen for a connection on the specified port. That thread goes off on its own to listen; execution continues in the main thread, setting the title bar and disabling a menu item. Nothing more happens until somebody tries to connect from elsewhere. Then the callback function *listen* executes. Note that when we called *BeginAcceptSocket*, we passed *this* in the second parameter. That's our instance of *Form1*. Thus when *listen* is called, it can access our member variables for the socket and controls, which it otherwise couldn't do, since a callback has to be static. When you do this at the thread level, it costs some effort to pass data from one thread to another like this!

Using TcpClient

TcpClient and *TcpListener* are new in .NET 2005. In 2003, you still had to use sockets, although the .NET socket library was a vast improvement over what was available to Windows programmers before that. Here's how simple *TcpClient* makes it to establish a connection with a listening computer:

```
private void tcpClientToolStripMenuItem1_Click(
    object sender, EventArgs e)
{
    Form2 dlg = new Form2();
    if (m_ListenerIPAddress != null)
        dlg.IPAddress = m_ListenerIPAddress.ToString();
    if (dlg.ShowDialog() != DialogResult.OK)
        return; // user cancelled
    m_ListenerIPAddress = IPAddress.Parse(dlg.IPAddress);
    TcpClient client;
    try
    {
        IPEndPoint listener =
            new IPEndPoint(m_ListenerIPAddress, m_ListenPort);
        client = new TcpClient();
        client.Connect(listener);
    }
}
```

```

    m_theStream = client.GetStream();
    ConnectionLight.BackColor = Color.Green;
    this.Text = "Two Way Chat--Client";
    connectToolStripMenuItem.Enabled = false;
    listenToolStripMenuItem.Enabled = false;
}
catch (Exception ex)
{
    MessageBox.Show(ex.ToString());
    return; // apparently other end was not listening
}
// and finally:
startChat();
}

```

This code shows a more compact way to do what we did with sockets in the first network programming example. Just for fun, I put a second menu item in this demo program so you could compare the two ways of doing it. Here's the socket-level code:

```

private void connectToolStripMenuItem_Click(object sender,
                                           EventArgs e)
{
    Form2 dlg = new Form2();
    if (m_ListenerIPAddress != null)
        dlg.IPAddress = m_ListenerIPAddress.ToString();
    if (dlg.ShowDialog() != DialogResult.OK)
        return; // user cancelled
    m_ListenerIPAddress = IPAddress.Parse(dlg.IPAddress);
    IPAddress bindAddress;
    bindAddress = IPAddress.Any;
    // in other words, not specified yet
    IPEndPoint bindEndPoint =
        new IPEndPoint(bindAddress, m_ClientPort);
    m_theSocket = new Socket(bindAddress.AddressFamily,
                             SocketType.Stream,
                             ProtocolType.Tcp);

    try
    {
        m_theSocket.Bind(bindEndPoint);
    }
    catch (SocketException ex)
    {
        // for example, if m_thePort is
        // already opened by some other process
        if (m_theSocket != null)

```

```

        m_theSocket.Close();
        MessageBox.Show(ex.ToString());
    }
    // OK, now we have a socket, let's connect
    IPEndPoint serverEndPoint;
    serverEndPoint = new IPEndPoint(m_ListenerIPAddress,
                                   m_ListenPort);

    try
    {
        m_theSocket.Connect(serverEndPoint);
        ConnectionLight.BackColor = Color.Green;
        this.Text = "Two Way Chat--Client";
        m_theStream = new NetworkStream(m_theSocket);
    }
    catch (SocketException err)
    {
        MessageBox.Show(err.Message);
        historyBox.Text = "Connection failed.";
        return; // nobody listening
    }
    // and finally:
    startChat();
}

```

Note that we used *m_ClientPort*, which was unused in the *TcpClient* version of this code. The *TcpClient* object must invent its own port and use it to create a socket. God knows what port it uses, as that is not a public member.

As a footnote (that means you can skip this paragraph if you want), here's a brief discussion of a still-unresolved problem. If you try to use the same port for *m_ClientPort* and *m_ListenPort*, you get an exception thrown. There would not be a problem doing that in .NET 2003. Investigating this problem I found, on page 188 of *Network Programming for the .NET Framework* (by Jones, Ohlund, and Olson), three paragraphs of detailed information. The bottom line is that prior to 2003, sockets were shareable by default, but in 2005, they're not. There are some options specified on that page that should make them shareable, but one of those options doesn't seem to exist in .NET 2005 (the book was written prior to .NET 2005), so I couldn't make them shareable. For this program, it doesn't matter, but for server programs intended to make many connections, it is an important point. I haven't checked this yet in .NET 2008.

Sending and Receiving Data

Now that we have a connection, we need to send and receive data. Sending is easy. Receiving is harder, because like waiting for connections, it's a "blocking" operation—you don't want your main thread to hang up while it waits to receive data. Again, asynchronous programming comes to our rescue and saves us from dealing directly with threads.

Since it's easier, we'll first give the code for sending. Note that the above code gives us a *NetworkStream* object. Such an object can be used much like a regular stream—we could create a *BinaryWriter* from it, etc. Instead we'll just use its native *Send* method.

```
private void sendButton_Click(object sender, EventArgs e)
{
    byte[] data = new byte[sendBox.Text.Length + 1];
    int i;
    for (i = 0; i < sendBox.Text.Length; i++)
        data[i] = (byte)sendBox.Text[i];
    try
    {
        m_theStream.Write(data, 0, sendBox.Text.Length);
        historyBox.Text += "\r\n" + sendBox.Text;
        sendBox.Text = "";
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}
```

You couldn't ask for simpler, more straightforward code.

Now, let's turn to the reading code. *NetworkStream* does have a *Read* method, but that method "blocks", i.e. waits and does not return until it actually reads data, and you don't know when the data is coming. So .NET 2005 breaks this operation into two pieces. There is *BeginRead*, which starts a new thread that calls *Read*. When there is data available, *Read* unblocks (starts executing again) and calls *your callback function*. Your main thread can go on doing something else immediately after calling *BeginRead*, because *BeginRead* terminates immediately after starting the new thread. Your callback function then calls *EndRead* to do the actual reading.

Note: the online documentation of *EndRead* is inaccurate, in fact, to put it bluntly, the documentation is wrong. It says that your callback function executes in a separate thread after *BeginRead* returns. Well, it is *called* from a separate thread that is created by *BeginRead*. But it will *execute* in your main thread, the one that called *BeginRead*. And of course that will happen *sometime* “after *BeginRead* returns”, but not *immediately* after. Here is an accurate statement: *your callback function is executed in the calling thread when the asynchronous [running in a worker thread] Read returns.* You can verify these things for yourself by setting a breakpoint in your callback in this program.

You can think of calling *BeginRead* as telling a new thread, “wait here and call me back when you’ve read something”, and give it a callback function to use.

The following code cost me several hours to write, and contains lines that cure some nasty bugs. So study it carefully, don’t just skim it.

```
private void startChat()
// begin an asynchronous read operation on m_theStream
{
    m_theStream.BeginRead(m_data, // where to put the data
        0, // the initial offset
        m_data.Length, // max bytes to read
        receiveData, // name of the callback
        this); // this instance of Form1
}

public static void receiveData(IAsyncResult ar)
// callback for the asynchronous read.
{
    Form1 f = (Form1)ar.AsyncState; // we passed this when
        // we called BeginAcceptSocket
    NetworkStream s = f.m_theStream;
    // Now f.m_data contains the string that has been read.
    // But the rest of f.m_data is full of '\0' characters
    // that we need to get rid of. A C# string can contain
    // such characters, but we don't want them.
    int count = 0;
    while(f.m_data[count] != '\0' && count < f.m_data.Length)
        ++count;
    String received =
        Encoding.UTF8.GetString(f.m_data,0,count).Trim();
    if(received.Length > 0)
```

```

        f.historyBox.Text += "\r\n" + received;
    try
    {
        int nBytesRead = s.EndRead(ar);
        if (nBytesRead == 0)
        {
            f.ConnectionLight.BackColor = Color.Red;
            return; // This only happens when the connection
                    // has been broken. So don't start
                    // another listening thread. Give up.
        }
        // clean old data out of the buffer:
        for (int i = 0; i < f.m_data.Length; i++)
            f.m_data[i] = 0;
        // f.m_data.Initialize() does not work for this!
        // Now start waiting for the next message:
        s.BeginRead(f.m_data, 0, f.m_data.Length,
                    receiveData, f
                    );
    }
    catch (System.IO.IOException)
    {
        // this occurs when the connection is broken, from
        the other side.
        if (f.m_theSocket != null)
        {
            f.m_theSocket.Shutdown(SocketShutdown.Both);
            f.m_theSocket.Close();
        }
    }
    catch (Exception ex)
    {
        if (f.m_theSocket != null)
        {
            f.m_theSocket.Shutdown(SocketShutdown.Both);
            f.m_theSocket.Close();
        }
        MessageBox.Show(ex.ToString());
    }
}

```

When I first programmed this, I was unpleasantly surprised to find that the line that sets *historyBox.Text* threw an exception. This turns out to be a threading issue that was discovered sometime between .NET 2003 and .NET 2005. Because this is a threading issue, not a networking issue, we sidestep it here by setting the *CheckForIllegalCrossThreadCalls* property of *Form1* to false (see the top of the code given at the beginning of the lecture).

This could only be a problem anyway when both users are on the same computer, which is not the intended use of this program, although of course it occurs during testing.

Cleaning Up

We can't rely on garbage collection to dispose of all our resources. Garbage collection only disposes of resources allocated by C#, but sockets are allocated by the operating system or a low-level library, and are not part of "managed code". The *Shutdown* method ensures that data waiting to be sent or received is in fact sent or received, but doesn't destroy the socket. The *Close* method releases the socket. The C# *Socket* object that wraps the underlying socket is of course subject to garbage collection—just not the actual socket that it wraps.

We need to do this cleanup when the user chooses to disconnect, and when the form is closing. So, we put the code in a separate method:

```
private void cleanUpSocket()
{
    if (m_theSocket == null)
        return;
    try
    {
        if (m_theSocket.Connected)
            m_theSocket.Shutdown(SocketShutdown.Both);
        // if we don't check m_theSocket.Connected first
        // this throws an ObjectDisposed exception
        // sometimes.
        m_theSocket.Close();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
    }
}
```

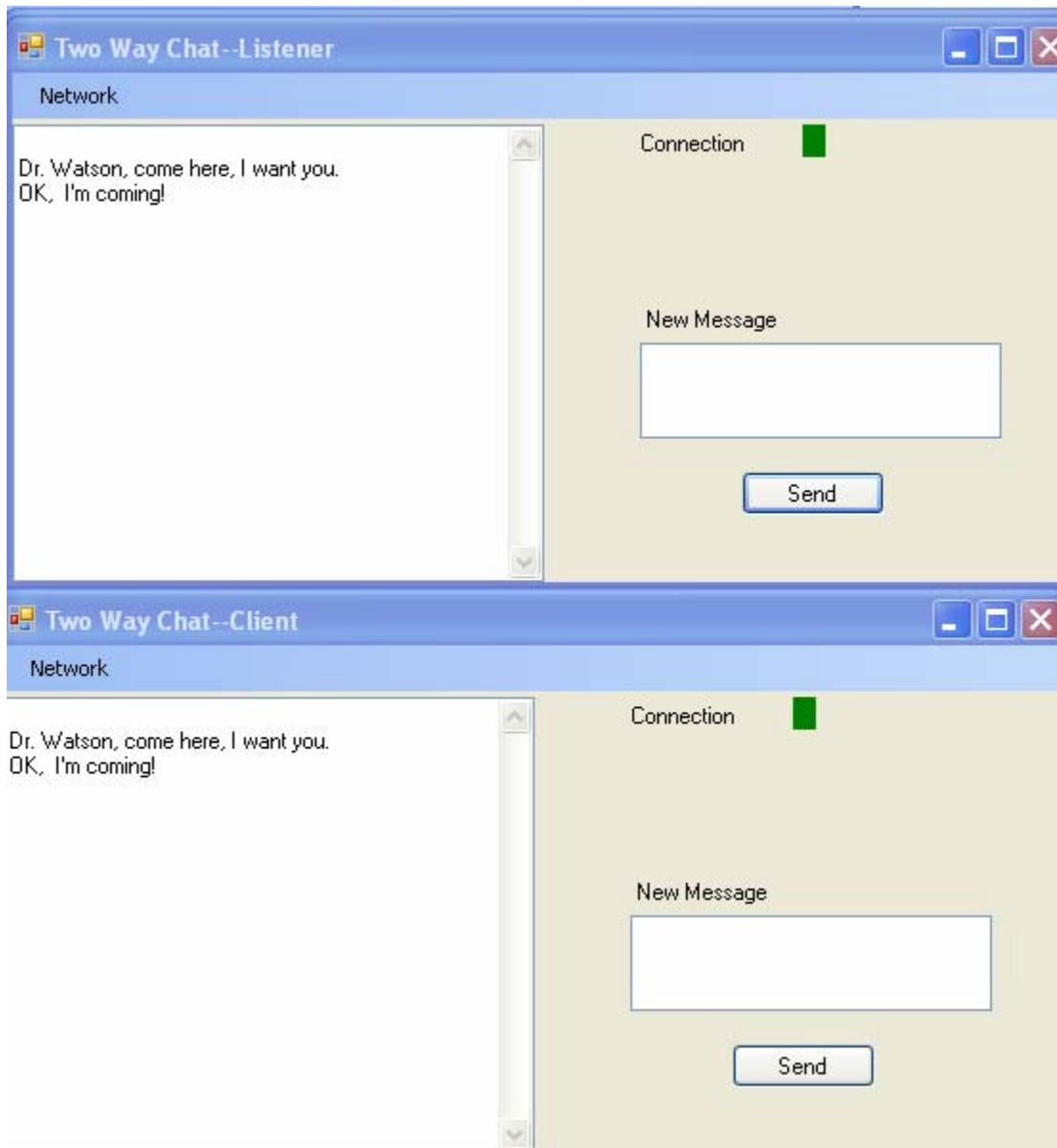
```
private void disconnectToolStripMenuItem_Click(object
sender, EventArgs e)
{
```

```
ConnectionLight.BackColor = Color.Red;
cleanUpSocket();
}

private void Form1_FormClosing(object sender,
                               FormClosingEventArgs e)
{
    cleanUpSocket();
}
```

Now it works!

Here's a screen shot showing two copies of the program that have exchanged a couple of messages. Obviously, it wouldn't be hard to label the messages to show who said what before you put them into *historyBox*.



Further tests, which earlier versions did not pass but the above code will pass: Send a few more short messages (after the initial long one shown); close the listener and then the client; close the client and then the listener.