I226 Computer Networks

Manual on Socket Programming

This file is to introduce how to make a socket programming using C# (C sharp) language using Microsoft Visual Studio[1] software.

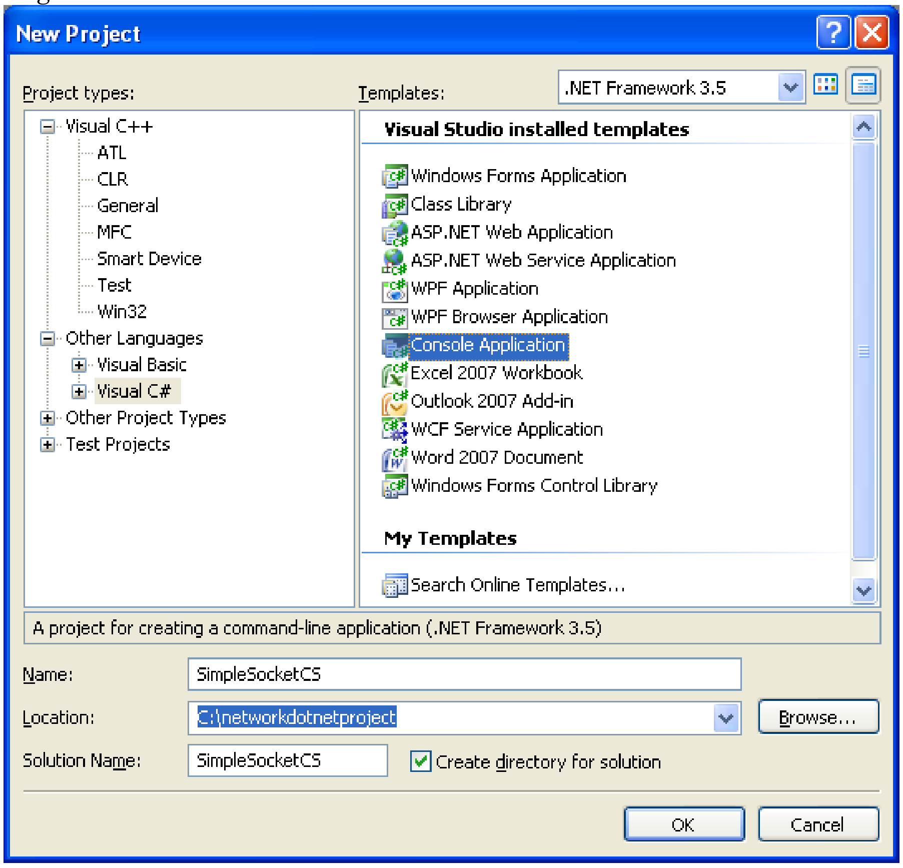
The overall step is below:
(1) Create server socket program
(2) Create client socket program
(3) Run both server and client programs simultaneously
    Note:
    In (3), the setting of server is complicated. To run on the same machine to see the same output, you can use the following two steps:
    Step1: run the server socket program using MS Visual Studio as usual
    Step2: run the client socket program using CLR (command line remote).

Prepared by
Yuto Lim (Ph.D.)

[1] The Microsoft Visual Studio version of this manual is 2008 version.

1.0 Create Server Socket Program

1. Create a console application project. You can use the project name as shown in the following Figure below:



A project for creating a command-line application (.NET Framework 3.5)

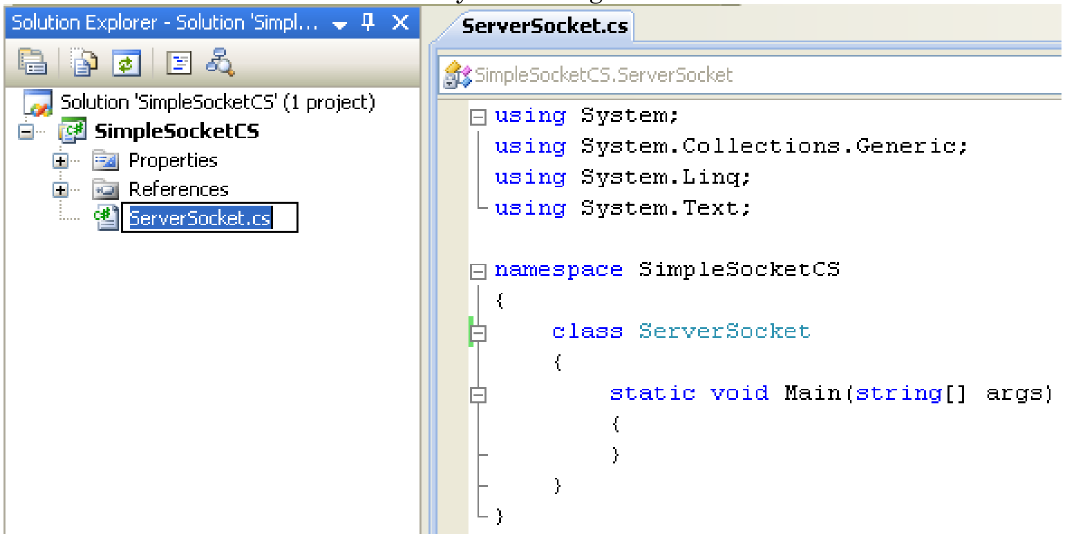
Name: SimpleSocketCS

Location: C:\networkdotnetproject

Solution Name: SimpleSocketCS

[✓] Create directory for solution

2. Rename the class to ServerSocket by renaming the source file.

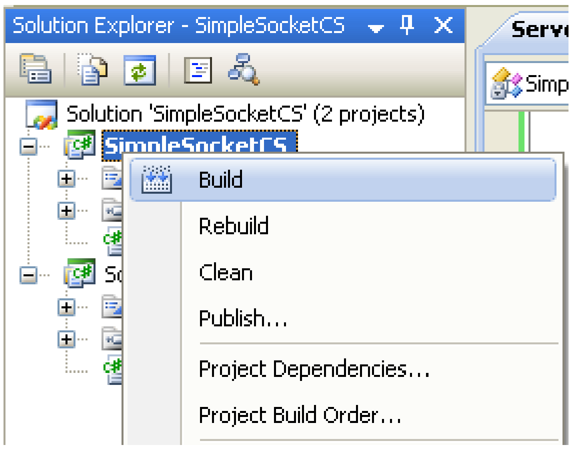

Solution Explorer - Solution 'Simpl...
Solution 'SimpleSocketCS' (1 project)
SimpleSocketCS
Properties
References
ServerSocket.cs
ServerSocket.cs
SimpleSocketCS.ServerSocket

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SimpleSocketCS
{
    class ServerSocket
    {
        static void Main(string[] args)
        {
        }
    }
}
```

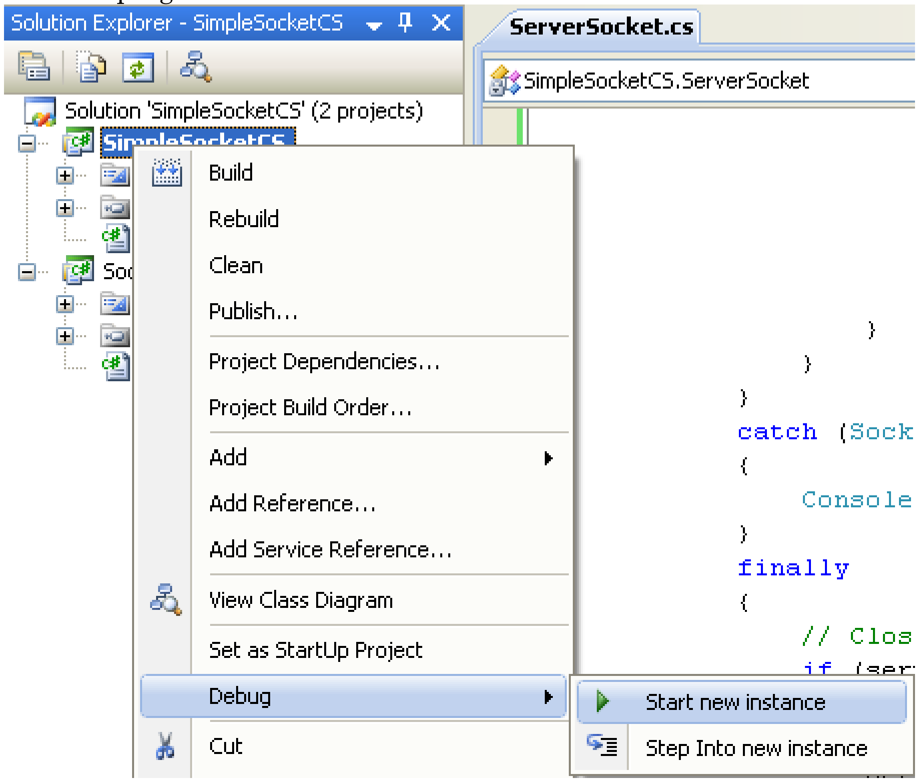
3. Add the following code to ServerSocket.cs.

```
// This is a simple TCP and UDP server. A socket of the requested type is created that
// waits for clients. For TCP, the server waits for an incoming TCP connection after which
// it receives a "request". The request is terminated by the client shutting down the connection.
// After the request is received, the server sends data in response followed by shutting down its
// connection and closing the socket. The UDP server simply waits for a datagram request. The
// request consists of a single datagram packet. The server then sends a number of responses to
// the source address of the request followed by a number of zero byte datagrams. The zero
// byte datagrams will indicate to the client that no more data will follow.
//
// usage:
//      Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]");
//                       [-t tcp|udp] [-x size]
//        -l bind-address        Local address to bind to
//        -m message             Text message to format into send buffer
//        -n count               Number of times to send a message
//        -p port                Local port to bind to
//        -t udp | tcp           Indicates which protocol to use
//        -x size                Size of send and receive buffer
//
// sample usage:
//      The following command line invokes an IPv6 TCP server bound to the wildcard address (::)
//      and port 5150 that sends 10 messages of 1024 bytes filled with the string "hola". The
//      command line following the server's is the client side that can be used to connect to
//      the server. For the client substitute the appropriate IPv6 address of the server.
//
//          Executable_file_name -l :: -p 5150 -n 10 -m "hola" -x 1024 -t tcp
//          Executable_file_name -n 3ffe::1 -p 5150 -t tcp
//
//      The following command line invokes an IPv4 UDP server bound to a specific local interface
//      on port 5150 that sends 5 messages of 512 bytes filled with the string "response". The
//      second command line is an example of the client's command line used to connect to the server.
//      NOTE: For UDP the buffer size on client and server should match - otherwise, an exception
//            will be thrown since the smaller buffer won't be able to hold the larger received datagram.
//
//          Executable_file_name -l 10.10.10.1 -p 5150 -n 5 -m "response" -x 512 -t udp
//          Executable_file_name -n 10.10.10.1 -p 5150 -t udp -x 512
//

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

namespace SimpleSocketCS
{
    /// <summary>
    /// This is a simple TCP and UDP based server.
    /// </summary>
    class ServerSocket
    {
        /// <summary>
        /// Winsock ioctl code which will disable ICMP errors from being propagated to a UDP socket.
        /// This can occur if a UDP packet is sent to a valid destination but there is no socket
        /// registered to listen on the given port.
        /// </summary>

        public const int SIO_UDP_CONNRESET = -1744830452;

        /// <summary>
        /// This routine repeatedly copies a string message into a byte array until filled.
        /// </summary>
        /// <param name="dataBuffer">Byte buffer to fill with string message</param>
        /// <param name="message">String message to copy</param>

        static public void FormatBuffer(byte[] dataBuffer, string message)
        {
            byte[] byteMessage = System.Text.Encoding.ASCII.GetBytes(message);
            int index = 0;

            // First convert the string to bytes and then copy into send buffer
            while (index < dataBuffer.Length)
            {
                for (int j = 0; j < byteMessage.Length; j++)
                {
                    dataBuffer[index] = byteMessage[j];
                    index++;

                    // Make sure we don't go past the send buffer length
                    if (index >= dataBuffer.Length)
                    {
                        break;
                    }
                }
            }
        }

        /// <summary>
        /// Prints simple usage information.
        /// </summary>

        static void usage()
        {
            Console.WriteLine("Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]");
            Console.WriteLine("                 [-t tcp|udp] [-x size]");
            Console.WriteLine("  -l bind-address        Local address to bind to");
            Console.WriteLine("  -m message             Text message to format into send buffer");
            Console.WriteLine("  -n count               Number of times to send a message");
            Console.WriteLine("  -p port                Local port to bind to");
            Console.WriteLine("  -t udp | tcp           Indicates which protocol to use");
            Console.WriteLine("  -x size                Size  of send and receive buffer");
            Console.WriteLine(" Else, default values will be used...");
        }

        /// <summary>
        /// This is the main routine that parses the command line and invokes the server with the
        /// given parameters. For TCP, it creates a listening socket and waits to accept a client
        /// connection. Once a client connects, it waits to receive a "request" message. The
        /// request is terminated by the client shutting down the connection. After the request is
        /// received, the server sends a response followed by shutting down its connection and
        /// closing the socket. For UDP, the socket simply listens for incoming packets. The "request"
        /// message is a single datagram received. Once the request is received, a number of datagrams
        /// are sent in return followed by sending a few zero byte datagrams. This way the client
        /// can determine that the response has completed when it receives a zero byte datagram.
        /// </summary>

        /// <param name="args">Command line arguments</param>
        static void Main(string[] args)
        {
            string textMessage = "Server: ServerResponse";
            int localPort = 5150, sendCount = 10, bufferSize = 4096;
            IPAddress localAddress = IPAddress.Any;
            SocketType sockType = SocketType.Stream;
            ProtocolType sockProtocol = ProtocolType.Tcp;

            Console.WriteLine();
            usage();
            Console.WriteLine();

            // Parse the command line
            for (int i = 0; i < args.Length; i++)
            {
                try
                {
                    if ((args[i][0] == '-') || (args[i][0] == '/'))
                    {
                        switch (Char.ToLower(args[i][1]))
                        {
                            case 'l':       // Local interface to bind to
                                localAddress = IPAddress.Parse(args[++i]);
                                break;
                            case 'm':       // Text message to put into the send buffer
                                textMessage = args[++i];
                                break;
                            case 'n':       // Number of times to send the response
                                sendCount = System.Convert.ToInt32(args[++i]);
                                break;
                            case 'p':       // Port number for the destination
                                localPort = System.Convert.ToInt32(args[++i]);
                                break;
                            case 't':       // Specified TCP or UDP
                                i++;

                                if (String.Compare(args[i], "tcp", true) == 0)
                                {
                                    sockType = SocketType.Stream;
                                    sockProtocol = ProtocolType.Tcp;
                                }
                                else if (String.Compare(args[i], "udp", true) == 0)
                                {
                                    sockType = SocketType.Dgram;
                                    sockProtocol = ProtocolType.Udp;
                                }
                                else
                                {
                                    usage();
                                    return;
                                }
                                break;
                            case 'x':       // Size of the send and receive buffers
                                bufferSize = System.Convert.ToInt32(args[++i]);
                                break;
                            default:
                                usage();
                                return;
                        }
                    }
                }
                catch
                {
                    usage();
                    return;
                }
            }

            Socket serverSocket = null;

            try
            {
                IPEndPoint localEndPoint = new IPEndPoint(localAddress, localPort), senderAddress = new IPEndPoint(localAddress, 0);
                Console.WriteLine("Server: IPEndPoint is OK...");
                EndPoint castSenderAddress;
                Socket clientSocket;
                byte[] receiveBuffer = new byte[bufferSize], sendBuffer = new byte[bufferSize];
                int rc;

                FormatBuffer(sendBuffer, textMessage);

                // Create the server socket
                serverSocket = new Socket(localAddress.AddressFamily, sockType, sockProtocol);

                Console.WriteLine("Server: Socket() is OK...");

                // Bind the socket to the local interface specified
                serverSocket.Bind(localEndPoint);

                Console.WriteLine("Server: {0} server socket bound to {1}", sockProtocol.ToString(), localEndPoint.ToString());

                if (sockProtocol == ProtocolType.Tcp)
                {
                    // If TCP socket, set the socket to listening
                    serverSocket.Listen(1);
                    Console.WriteLine("Server: Listen() is OK, I'm listening for connection buddy!");
                }
                else
                {
                    byte[] byteTrue = new byte[4];

                    // Set the SIO_UDP_CONNRESET ioctl to true for this UDP socket. If this UDP socket
                    //    ever sends a UDP packet to a remote destination that exists but there is
                    //    no socket to receive the packet, an ICMP port unreachable message is returned
                    //    to the sender. By default, when this is received the next operation on the
                    //    UDP socket that send the packet will receive a SocketException. The native
                    //    (Winsock) error that is received is WSAECONNRESET (10054). Since we don't want
                    //    to wrap each UDP socket operation in a try/except, we'll disable this error
                    //    for the socket with this ioctl call.

                    byteTrue[byteTrue.Length - 1] = 1;
                    serverSocket.IOControl(ServerSocket.SIO_UDP_CONNRESET, byteTrue, null);
                    Console.WriteLine("Server: IOControl() is OK...");
                }

                // Service clients in a loop
                while (true)
                {
                    if (sockProtocol == ProtocolType.Tcp)
                    {
                        // Wait for a client connection
                        clientSocket = serverSocket.Accept();
                        Console.WriteLine("Server: Accept() is OK...");
                        Console.WriteLine("Server: Accepted connection from: {0}", clientSocket.RemoteEndPoint.ToString());

                        // Receive the request from the client in a loop until the client shuts
                        //    the connection down via a Shutdown.

                        Console.WriteLine("Server: Preparing to receive using Receive()...");
                        while (true)
                        {
                            rc = clientSocket.Receive(receiveBuffer);
                            Console.WriteLine("Server: Read {0} bytes", rc);
                            if (rc == 0)
                                break;
                        }

                        // Send the indicated number of response messages
                        Console.WriteLine("Server: Preparing to send using Send()...");

                        for (int i = 0; i < sendCount; i++)
                        {
                            rc = clientSocket.Send(sendBuffer);
                            Console.WriteLine("Server: Sent {0} bytes", rc);
                        }

                        // Shutdown the client connection
                        clientSocket.Shutdown(SocketShutdown.Send);

                        Console.WriteLine("Server: Shutdown() is OK...");
                        clientSocket.Close();
                        Console.WriteLine("Server: Close() is OK...");
                    }
                    else
                    {
                        castSenderAddress = (EndPoint)senderAddress;

                        // Receive the initial request from the client
                        rc = serverSocket.ReceiveFrom(receiveBuffer, ref castSenderAddress);
                        Console.WriteLine("Server: ReceiveFrom() is OK...");
                        senderAddress = (IPEndPoint)castSenderAddress;
                        Console.WriteLine("Server: Received {0} bytes from {1}", rc, senderAddress.ToString());

                        // Send the response to the client the requested number of times
                        for (int i = 0; i < sendCount; i++)
                        {
                            try
                            {
                                rc = serverSocket.SendTo(sendBuffer, senderAddress);
                                Console.WriteLine("Server: SendTo() is OK...");
                            }
                            catch
                            {
                                // If the sender's address is being spoofed we may get an error when sending
                                //    the response. You can test this by using IPv6 and using the RawSocket
                                //    sample to spoof a UDP packet with an invalid link local source address.
                                continue;
                            }
                            Console.WriteLine("Server: Sent {0} bytes to {1}", rc, senderAddress.ToString());
                        }

                        // Send several zero byte datagrams to indicate to client that no more data
                        //    will be sent from the server. Multiple packets are sent since UDP
                        //    is not guaranteed and we want to try to make an effort the client
                        //    gets at least one.
                        Console.WriteLine("Server: Preparing to send using SendTo(), on the way do sleeping, Sleep(250)...");

                        for (int i = 0; i < 3; i++)
                        {
                            serverSocket.SendTo(sendBuffer, 0, 0, SocketFlags.None, senderAddress);
                            // Space out sending the zero byte datagrams a bit. UDP is unreliable and
                            //   the local stack can even drop them before even hitting the wire!
                            System.Threading.Thread.Sleep(250);
                        }
                    }
                }
            }
            catch (SocketException err)
            {
                Console.WriteLine("Server: Socket error occurred: {0}", err.Message);
            }
            finally
            {
                // Close the socket if necessary
                if (serverSocket != null)
                {
                    Console.WriteLine("Server: Closing using Close()...");
                    serverSocket.Close();
                }
            }
        }
      }
    }
```

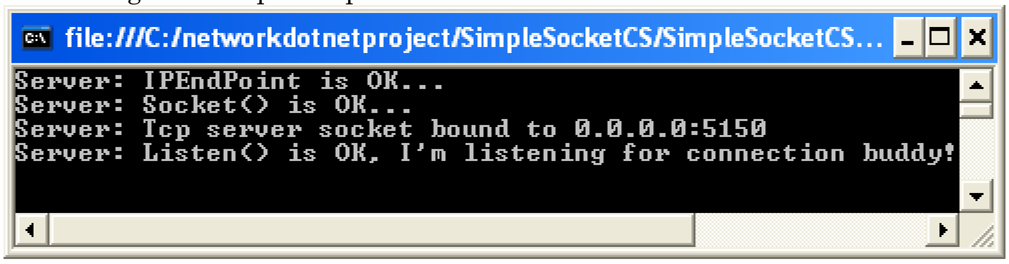
4. Then, build the server program and make sure there is no error.



5. Then, run this program.

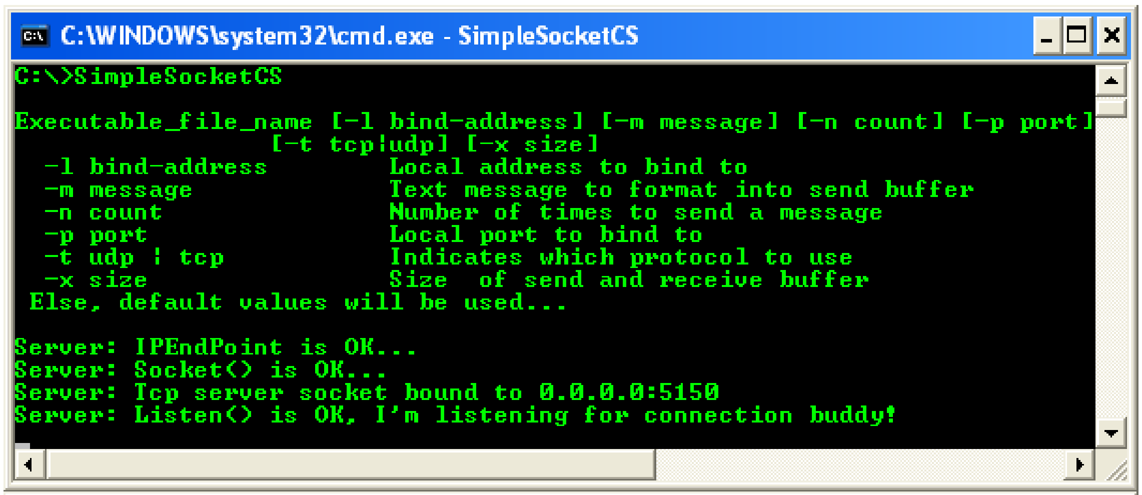


6. The following is a sample output.

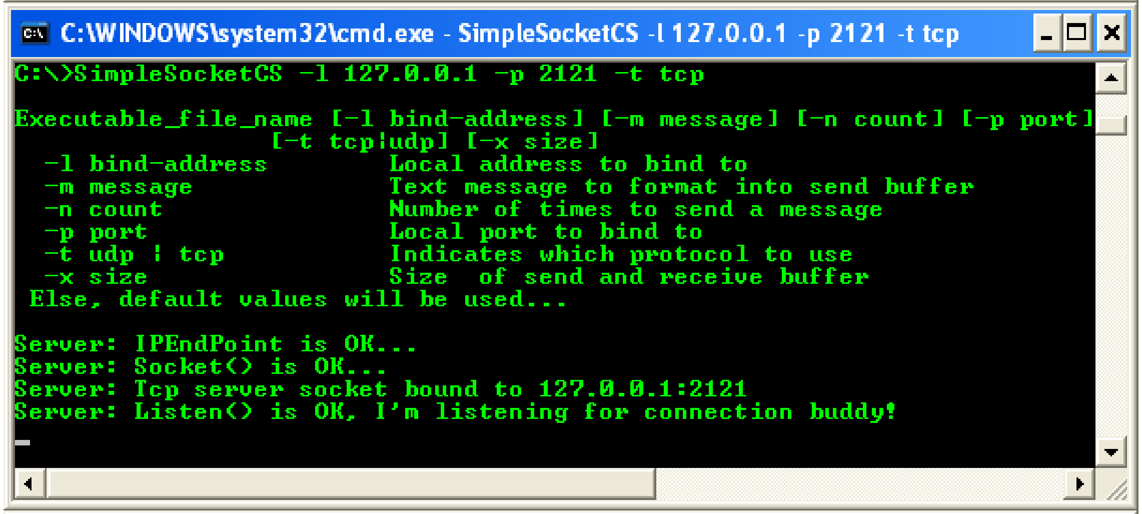

file:///C:/networkdotnetproject/SimpleSocketCS/SimpleSocketCS...

```
Server: IPEndPoint is OK...
Server: Socket() is OK...
Server: Tcp server socket bound to 0.0.0.0:5150
Server: Listen() is OK, I'm listening for connection buddy!
```

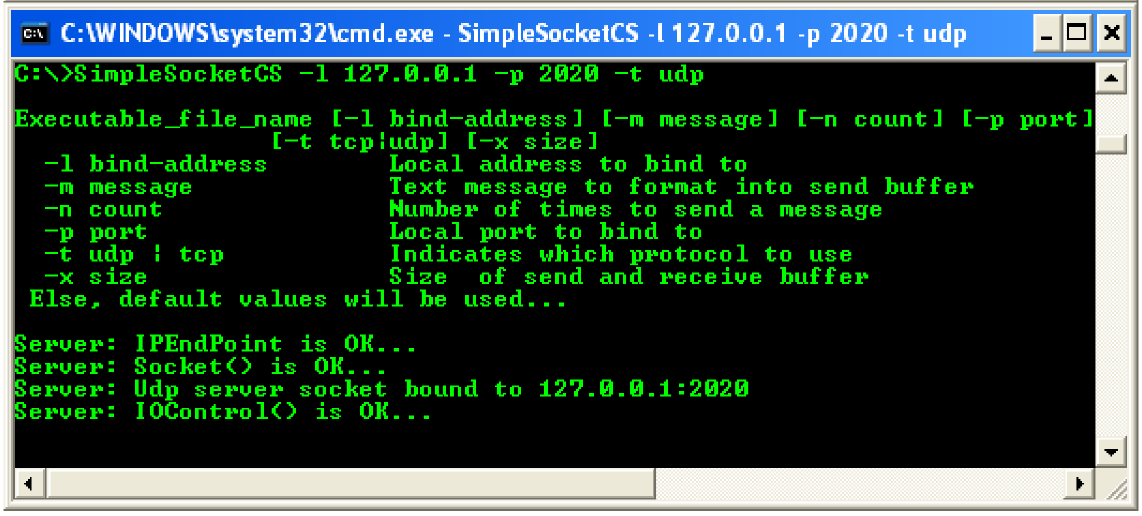
7. The following is a sample output when the program runs at the command prompt.



```
C:\>SimpleSocketCS

Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]
                 [-t tcp|udp] [-x size]
  -l bind-address         Local address to bind to
  -m message              Text message to format into send buffer
  -n count                Number of times to send a message
  -p port                 Local port to bind to
  -t udp | tcp            Indicates which protocol to use
  -x size                 Size  of send and receive buffer
 Else, default values will be used...

Server: IPEndPoint is OK...
Server: Socket() is OK...
Server: Tcp server socket bound to 0.0.0.0:5150
Server: Listen() is OK, I'm listening for connection buddy!
```


C:\WINDOWS\system32\cmd.exe - SimpleSocketCS -l 127.0.0.1 -p 2121 -t tcp

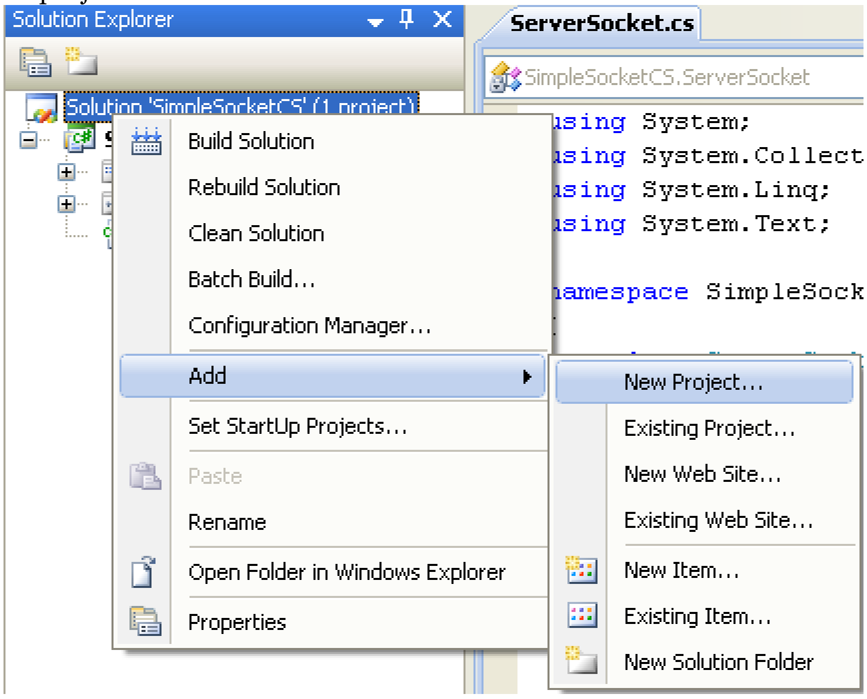
```
C:\>SimpleSocketCS -l 127.0.0.1 -p 2121 -t tcp

Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]
                 [-t tcp|udp] [-x size]
  -l bind-address         Local address to bind to
  -m message              Text message to format into send buffer
  -n count                Number of times to send a message
  -p port                 Local port to bind to
  -t udp | tcp            Indicates which protocol to use
  -x size                 Size  of send and receive buffer
 Else, default values will be used...

Server: IPEndPoint is OK...
Server: Socket() is OK...
Server: Tcp server socket bound to 127.0.0.1:2121
Server: Listen() is OK, I'm listening for connection buddy!
```


C:\WINDOWS\system32\cmd.exe - SimpleSocketCS -l 127.0.0.1 -p 2020 -t udp

```
C:\>SimpleSocketCS -l 127.0.0.1 -p 2020 -t udp

Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]
                 [-t tcp|udp] [-x size]
  -l bind-address         Local address to bind to
  -m message              Text message to format into send buffer
  -n count                Number of times to send a message
  -p port                 Local port to bind to
  -t udp | tcp            Indicates which protocol to use
  -x size                 Size  of send and receive buffer
 Else, default values will be used...

Server: IPEndPoint is OK...
Server: Socket() is OK...
Server: Udp server socket bound to 127.0.0.1:2020
Server: IOControl() is OK...
```

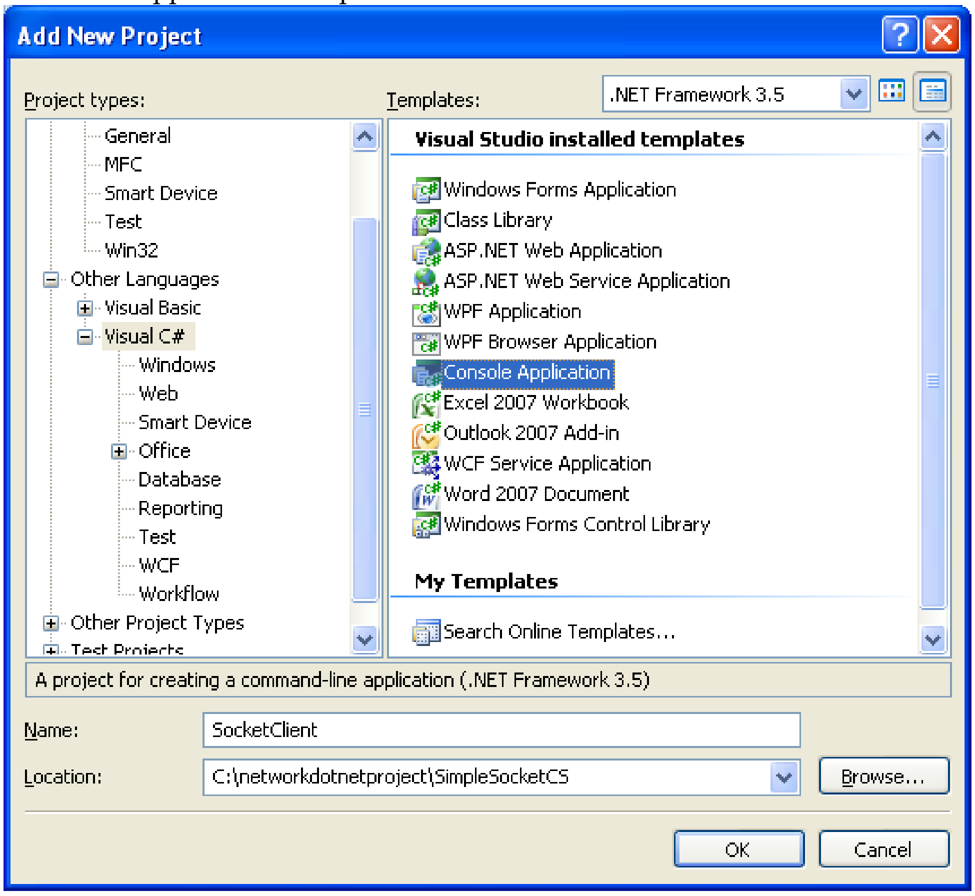
2.0 Create Client Socket Program

1. Add a new project.



```
sing System;
sing System.Collect
sing System.Linq;
sing System.Text;

amespace SimpleSock
```

2. Select console application template and name it as SocketClient.

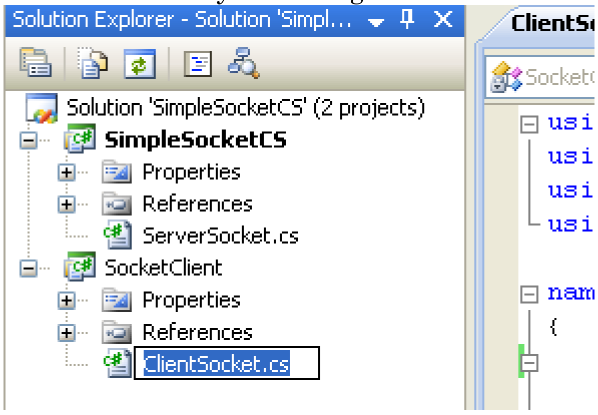


A project for creating a command-line application (.NET Framework 3.5)

Name: SocketClient

Location: C:\networkdotnetproject\SimpleSocketCS

3. Rename the class to ClientSocket by renaming the source file.

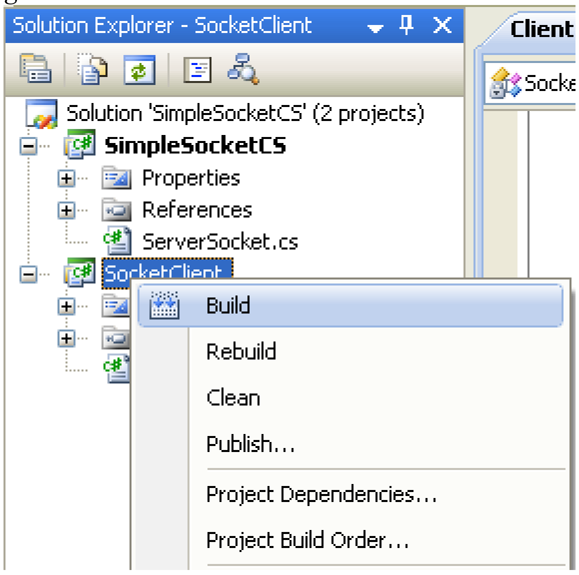


4. Next, let do the client coding. Add the following code to ClientSocket.cs

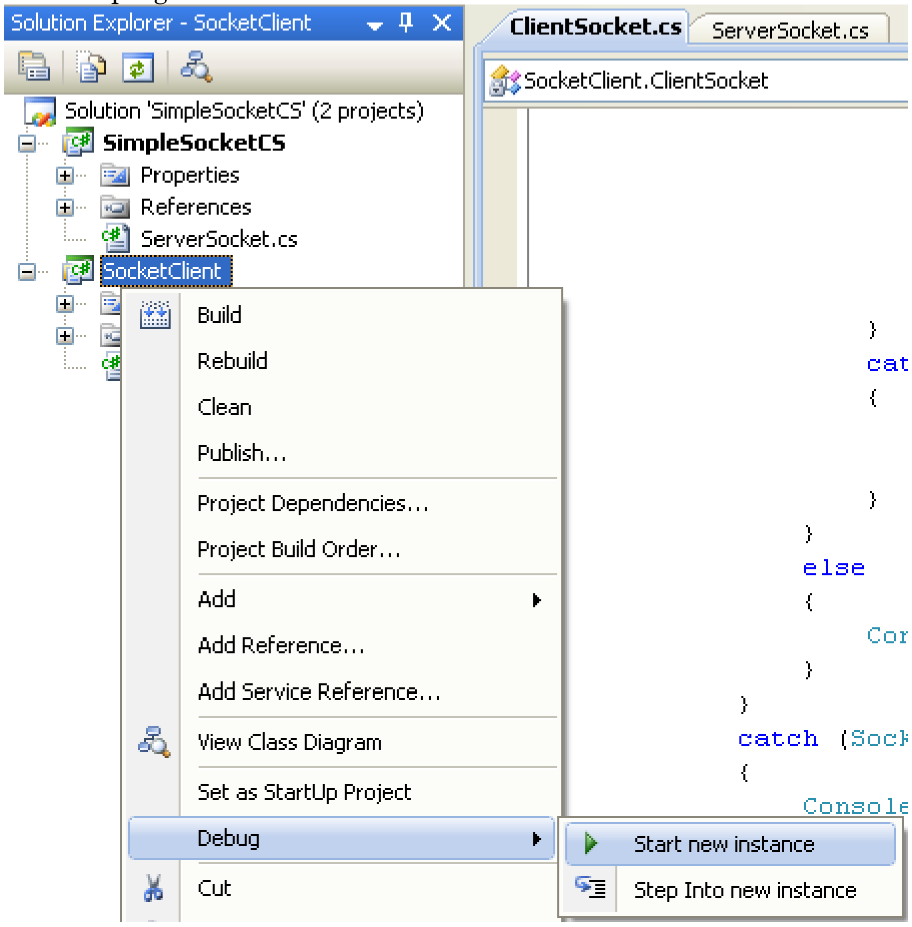
```
// This is a simple TCP and UDP client application. For TCP, the server name is
// resolved and a socket is created to attempt a connection to each address
// returned until a connection succeeds. Once connected the client sends a "request"
// message to the server and shuts down the sending side. The client then loops
// to receive the server response until the server closes the connection at which
// point the client closes its socket and exits. For UDP, the server name is resolved
// and the first address returned is used (since there is no indication that there
// is a UDP server at the endpoint). The UDP client then sends a single datagram
// "request" message and then waits to receive a response from the server. The client
// continues to receive until a zero byte datagram is received. Note that the server
// sends several zero byte datagrams but if they are lost, the client will never
// exit.
//
// usage:
//      Executable_file_name [-c] [-n server] [-p port] [-m message]
//                        [-t tcp|udp] [-x size]
//           -c                         If UDP connect the socket before sending
//           -n server              Server name or address to connect/send to
//           -p port                  Port number to connect/send to
//           -m message      String message to format in request buffer
//           -t tcp|udp            Indicates to use either the TCP or UDP protocol
//           -x size                  Size of send and receive buffers
//
// sample usage:
//      The following command line establishes a TCP connection to the given server
//      on port 5150. The other two command lines are sample server command lines --
//      one for IPv4 and one for IPv6. Since the client will attempt to resolve
//      the server's name, it should attempt to connect over IPv4 and IPv6 as long
//      as the addresses are registered in DNS.
//
//          Executable_file_name -n server -p 5150 -t tcp
//          Executable_file_name -l :: -p 5150 -t tcp
//          Executable_file_name -l 0.0.0.0 -p 5150 -t tcp
//
//      The following command line creates a connected UDP socket that sends
//      data to the server x.y.z.w on port 5150. While the second entry is the
//      server command line used.
//      NOTE: For UDP sockets, the buffer size on the client and server should match
//            as an exception will be thrown if the receiving buffer is smaller than
//            the incoming datagram.
//
//          Executable_file_name -n x.y.z.w -p 5150 -t udp -c -x 512
//          Executable_file_name -l x.y.z.w -p 5150 -t udp -x 512
//

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

/// <summary>
/// This is a simple TCP and UDP based client.
/// </summary>

namespace SocketClient
{
    class ClientSocket
    {
        /// <summary>
        /// This routine repeatedly copies a string message into a byte array until filled.
        /// </summary>
        /// <param name="dataBuffer">Byte buffer to fill with string message</param>
        /// <param name="message">String message to copy</param>

        static public void FormatBuffer(byte[] dataBuffer, string message)
        {
            byte[] byteMessage = System.Text.Encoding.ASCII.GetBytes(message);
            int index = 0;

            // First convert the string to bytes and then copy into send buffer
            while (index < dataBuffer.Length)
            {
                for (int j = 0; j < byteMessage.Length; j++)
                {
                    dataBuffer[index] = byteMessage[j];
                    index++;

                    // Make sure we don't go past the send buffer length
                    if (index >= dataBuffer.Length)
                    {
                        break;
                    }
                }
            }
        }

        /// <summary>
        /// Prints simple usage information.
        /// </summary>

        static public void usage()
        {
            Console.WriteLine("usage: Executable_file_name [-c] [-n server] [-p port] [-m message]");
            Console.WriteLine("                        [-t tcp|udp] [-x size]");
            Console.WriteLine("     -c              If UDP connect the socket before sending");
            Console.WriteLine("     -n server       Server name or address to connect/send to");
            Console.WriteLine("     -p port         Port number to connect/send to");
            Console.WriteLine("     -m message      String message to format in request buffer");
            Console.WriteLine("     -t tcp|udp      Indicates to use either the TCP or UDP protocol");
            Console.WriteLine("     -x size         Size of send and receive buffers");
            Console.WriteLine(" Else, default values will be used...");
        }

        /// <summary>
        /// This is the main function for the simple client. It parses the command line and creates
        /// a socket of the requested type. For TCP, it will resolve the name and attempt to connect
        /// to each resolved address until a successful connection is made. Once connected a request
        /// message is sent followed by shutting down the send connection. The client then receives
        /// data until the server closes its side at which point the client socket is closed. For
        /// UDP, the socket is created and if indicated connected to the server's address. A single
        /// request datagram message. The client then waits to receive a response and continues to
        /// do so until a zero byte datagram is receive which indicates the end of the response.
        /// </summary>
        /// <param name="args">Command line arguments</param>

        static void Main(string[] args)
        {
            SocketType sockType = SocketType.Stream;
            ProtocolType sockProtocol = ProtocolType.Tcp;
            string remoteName = "localhost", textMessage = "Client: This is a test";
            bool udpConnect = false;
            int remotePort = 5150, bufferSize = 4096;

            Console.WriteLine();
            usage();
            Console.WriteLine();

            // Parse the command line
            for (int i = 0; i < args.Length; i++)
            {
                try
                {
                    if ((args[i][0] == '-') || (args[i][0] == '/'))
                    {
                        switch (Char.ToLower(args[i][1]))
                        {
                            case 'c':       // "Connect" the UDP socket to the destination
                                udpConnect = true;
                                break;
                            case 'n':       // Destination address to connect to or send to
                                remoteName = args[++i];
                                break;
                            case 'm':       // Text message to put into the send buffer
                        textMessage = args[++i];
                        break;
                    case 'p':       // Port number for the destination
                        remotePort = System.Convert.ToInt32(args[++i]);
                        break;
                    case 't':       // Specified TCP or UDP
                        i++;
                        if (String.Compare(args[i], "tcp", true) == 0)
                        {
                            sockType = SocketType.Stream;
                            sockProtocol = ProtocolType.Tcp;
                        }
                        else if (String.Compare(args[i], "udp", true) == 0)
                        {
                            sockType = SocketType.Dgram;
                            sockProtocol = ProtocolType.Udp;
                        }
                        else
                        {
                            usage();
                            return;
                        }
                        break;
                    case 'x':       // Size of the send and receive buffers
                        bufferSize = System.Convert.ToInt32(args[++i]);
                        break;
                    default:
                        usage();
                        return;
                }
            }
        }
        catch
        {
            usage();
            return;
        }
    }

    Socket clientSocket = null;
    IPHostEntry resolvedHost = null;
    IPEndPoint destination = null;
    byte[] sendBuffer = new byte[bufferSize], recvBuffer = new Byte[bufferSize];
    int rc;

    // Format the string message into the send buffer
    FormatBuffer(sendBuffer, textMessage);

    try
    {
        // Try to resolve the remote host name or address
        resolvedHost = Dns.GetHostEntry(remoteName);
        Console.WriteLine("Client: GetHostEntry() is OK...");

        // Try each address returned
        foreach (IPAddress addr in resolvedHost.AddressList)
        {
            // Create a socket corresponding to the address family of the resolved address
            clientSocket = new Socket(
                addr.AddressFamily,
                sockType,
                sockProtocol
                );
            Console.WriteLine("Client: Socket() is OK...");
            try
            {
                // Create the endpoint that describes the destination
                destination = new IPEndPoint(addr, remotePort);
                Console.WriteLine("Client: IPEndPoint() is OK. IP Address: {0}, server port: {1}", addr, remotePort);

                if ((sockProtocol == ProtocolType.Udp) && (udpConnect == false))
                {
                    Console.WriteLine("Client: Destination address is: {0}", destination.ToString());
                    break;
                }
                else
                {
                    Console.WriteLine("Client: Attempting connection to: {0}", destination.ToString());
                }
                clientSocket.Connect(destination);
                Console.WriteLine("Client: Connect() is OK...");
                break;
            }
            catch (SocketException)
```

```
                    {
                        // Connect failed, so close the socket and try the next address
                        clientSocket.Close();
                        Console.WriteLine("Client: Close() is OK...");
                        clientSocket = null;
                        continue;
                    }
                }

                // Make sure we have a valid socket before trying to use it
                if ((clientSocket != null) && (destination != null))
                {
                    try
                    {
                        // Send the request to the server
                        if ((sockProtocol == ProtocolType.Udp) && (udpConnect == false))
                        {
                            clientSocket.SendTo(sendBuffer, destination);
                            Console.WriteLine("Client: SendTo() is OK...UDP...");
                        }
                        else
                        {
                            rc = clientSocket.Send(sendBuffer);
                            Console.WriteLine("Client: send() is OK...TCP...");
                            Console.WriteLine("Client: Sent request of {0} bytes", rc);

                            // For TCP, shutdown sending on our side since the client won't send any more data
                            if (sockProtocol == ProtocolType.Tcp)
                            {
                                clientSocket.Shutdown(SocketShutdown.Send);
                                Console.WriteLine("Client: Shutdown() is OK...");
                            }
                        }

                        // Receive data in a loop until the server closes the connection. For
                        //    TCP this occurs when the server performs a shutdown or closes
                        //    the socket. For UDP, we'll know to exit when the remote host
                        //    sends a zero byte datagram.
                        while (true)
                        {
                            if ((sockProtocol == ProtocolType.Tcp) || (udpConnect == true))
                            {
                                rc = clientSocket.Receive(recvBuffer);
                                Console.WriteLine("Client: Receive() is OK...");
                                Console.WriteLine("Client: Read {0} bytes", rc);
                            }
                            else
                            {
                                IPEndPoint fromEndPoint = new IPEndPoint(destination.Address, 0);
                                Console.WriteLine("Client: IPEndPoint() is OK...");
                                EndPoint castFromEndPoint = (EndPoint)fromEndPoint;
                                rc = clientSocket.ReceiveFrom(recvBuffer, ref castFromEndPoint);
                                Console.WriteLine("Client: ReceiveFrom() is OK...");
                                fromEndPoint = (IPEndPoint)castFromEndPoint;
                                Console.WriteLine("Client: Read {0} bytes from {1}", rc, fromEndPoint.ToString());
                            }

                            // Exit loop if server indicates shutdown
                            if (rc == 0)
                            {
                                clientSocket.Close();
                                Console.WriteLine("Client: Close() is OK...");
                                break;
                            }
                        }
                    }
                    catch (SocketException err)
                    {
                        Console.WriteLine("Client: Error occurred while sending or receiving data.");
                        Console.WriteLine("   Error: {0}", err.Message);
                    }
                }
                else
                {
                    Console.WriteLine("Client: Unable to establish connection to server!");
                }
            }
            catch (SocketException err)
            {
                Console.WriteLine("Client: Socket error occurred: {0}", err.Message);
            }
        }
    }
}
```

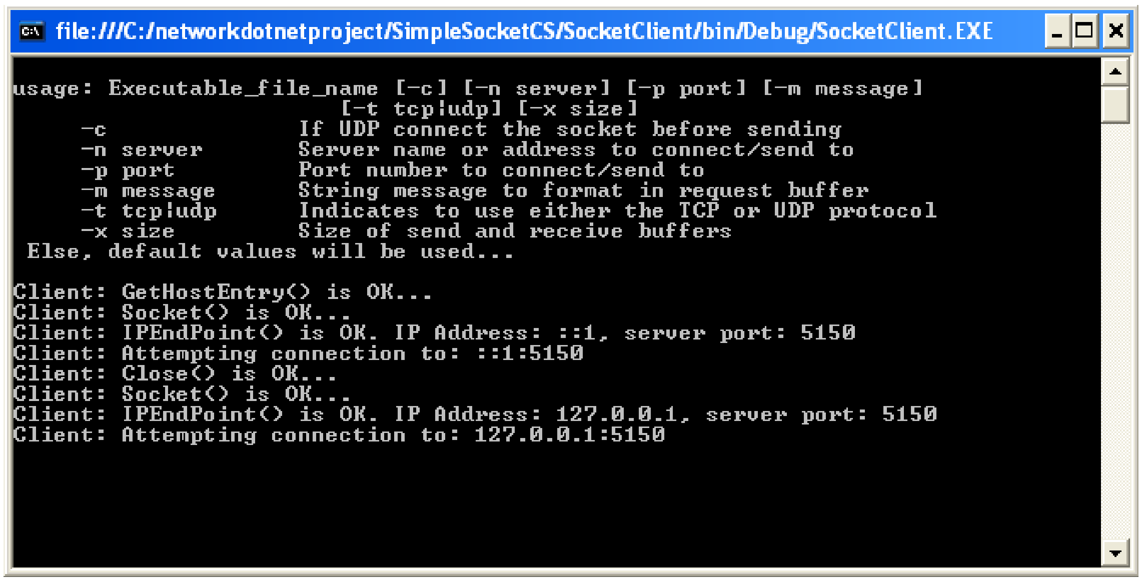
5. Build the client program and make sure there is no error.



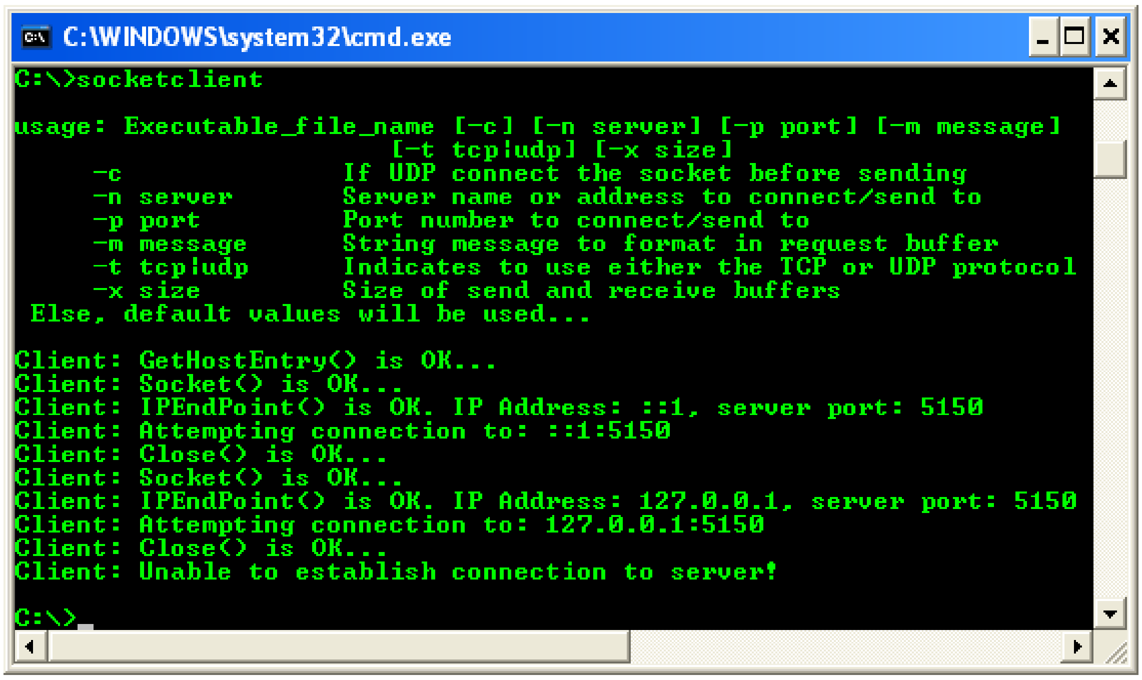
6. Run the client program.



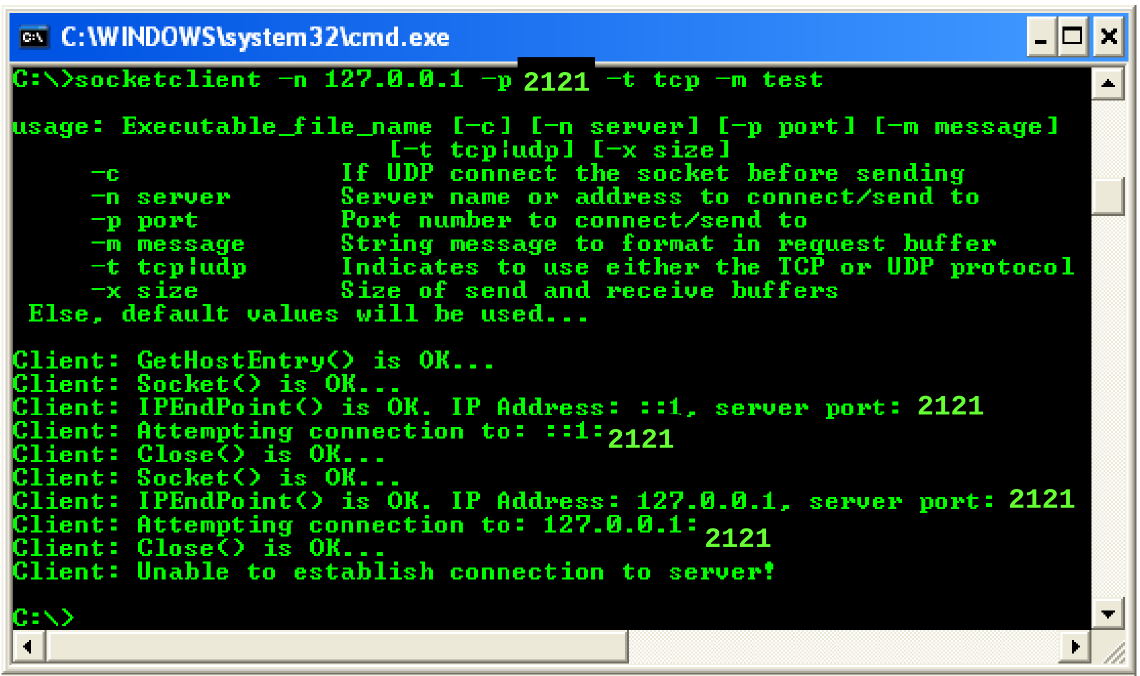
```
            }
            cat
            {
            }
        }
        else
        {
            Cor
        }
    }
    catch (Sock
    {
        Console
```


file:///C:/networkdotnetproject/SimpleSocketCS/SocketClient/bin/Debug/SocketClient.EXE

```
usage: Executable_file_name [-c] [-n server] [-p port] [-m message]
                        [-t tcp|udp] [-x size]
     -c                 If UDP connect the socket before sending
     -n server          Server name or address to connect/send to
     -p port            Port number to connect/send to
     -m message         String message to format in request buffer
     -t tcp|udp         Indicates to use either the TCP or UDP protocol
     -x size            Size of send and receive buffers
 Else, default values will be used...

Client: GetHostEntry() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: ::1, server port: 5150
Client: Attempting connection to: ::1:5150
Client: Close() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: 127.0.0.1, server port: 5150
Client: Attempting connection to: 127.0.0.1:5150
```

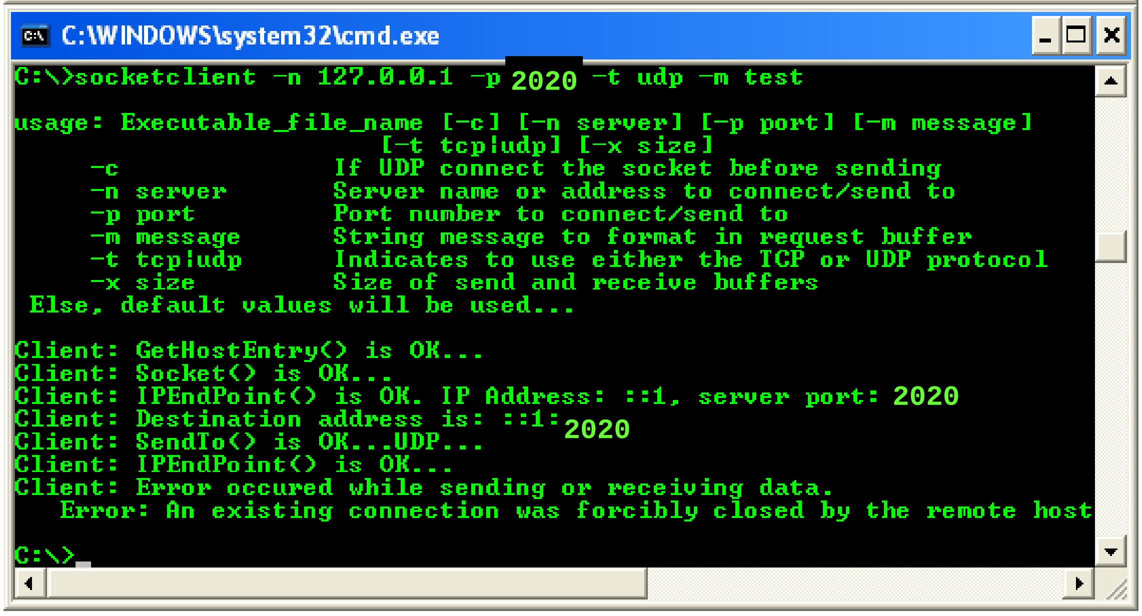


```
C:\>socketclient

usage: Executable_file_name [-c] [-n server] [-p port] [-m message]
                        [-t tcp|udp] [-x size]
     -c                 If UDP connect the socket before sending
     -n server          Server name or address to connect/send to
     -p port            Port number to connect/send to
     -m message         String message to format in request buffer
     -t tcp|udp         Indicates to use either the TCP or UDP protocol
     -x size            Size of send and receive buffers
 Else, default values will be used...

Client: GetHostEntry() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: ::1, server port: 5150
Client: Attempting connection to: ::1:5150
Client: Close() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: 127.0.0.1, server port: 5150
Client: Attempting connection to: 127.0.0.1:5150
Client: Close() is OK...
Client: Unable to establish connection to server!

C:\>
```



C:\WINDOWS\system32\cmd.exe

```
C:\>socketclient -n 127.0.0.1 -p 2121 -t tcp -m test

usage: Executable_file_name [-c] [-n server] [-p port] [-m message]
                        [-t tcp|udp] [-x size]
     -c                If UDP connect the socket before sending
     -n server         Server name or address to connect/send to
     -p port           Port number to connect/send to
     -m message        String message to format in request buffer
     -t tcp|udp        Indicates to use either the TCP or UDP protocol
     -x size           Size of send and receive buffers
 Else, default values will be used...

Client: GetHostEntry() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: ::1, server port: 2121
Client: Attempting connection to: ::1:2121
Client: Close() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: 127.0.0.1, server port: 2121
Client: Attempting connection to: 127.0.0.1:2121
Client: Close() is OK...
Client: Unable to establish connection to server!

C:\>
```

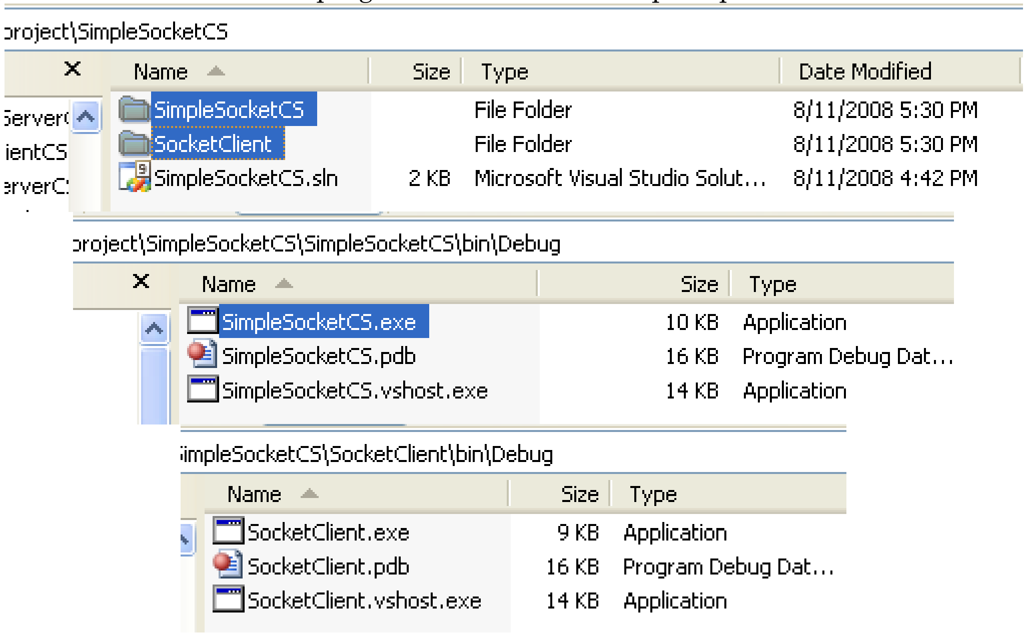


C:\WINDOWS\system32\cmd.exe

```
C:\>socketclient -n 127.0.0.1 -p 2020 -t udp -m test

usage: Executable_file_name [-c] [-n server] [-p port] [-m message]
                        [-t tcp|udp] [-x size]
     -c                If UDP connect the socket before sending
     -n server         Server name or address to connect/send to
     -p port           Port number to connect/send to
     -m message        String message to format in request buffer
     -t tcp|udp        Indicates to use either the TCP or UDP protocol
     -x size           Size of send and receive buffers
 Else, default values will be used...

Client: GetHostEntry() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: ::1, server port: 2020
Client: Destination address is: ::1:2020
Client: SendTo() is OK...UDP...
Client: IPEndPoint() is OK...
Client: Error occured while sending or receiving data.
   Error: An existing connection was forcibly closed by the remote host

C:\>_
```

3.0 Run Both Server and Client Programs Simultaneously

1. We can't see the real communication without running both client and server programs simultaneously. In this case we copy both executables (under the Debug folder – SimpleSocketCS.exe - server and SocketClient.exe - client) to the C:¥, run the server program and then the client program at the command prompt.



project\SimpleSocketCS

Name	Size	Type	Date Modified
SimpleSocketCS		File Folder	8/11/2008 5:30 PM
SocketClient		File Folder	8/11/2008 5:30 PM
SimpleSocketCS.sln	2 KB	Microsoft Visual Studio Solut...	8/11/2008 4:42 PM

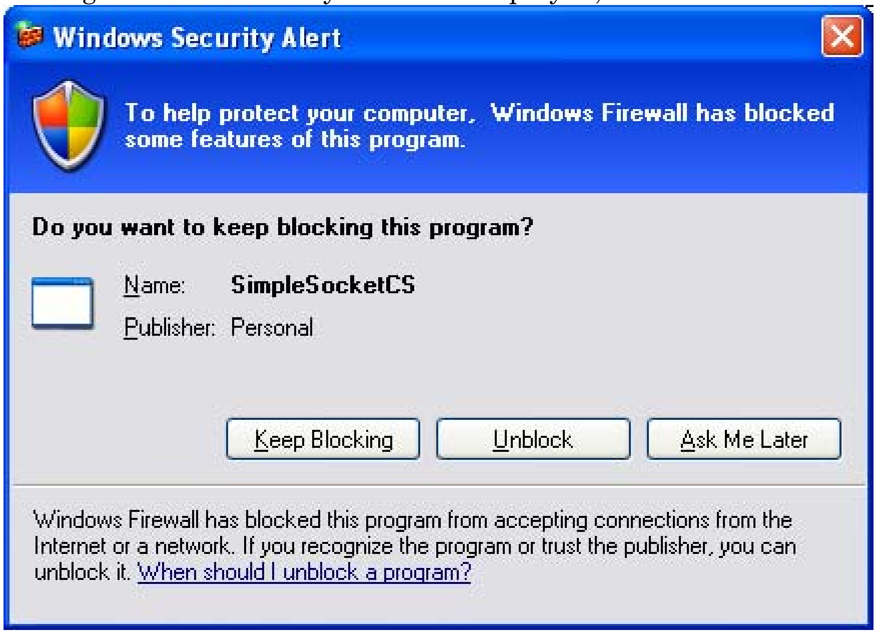
project\SimpleSocketCS\SimpleSocketCS\bin\Debug

Name	Size	Type
SimpleSocketCS.exe	10 KB	Application
SimpleSocketCS.pdb	16 KB	Program Debug Dat...
SimpleSocketCS.vshost.exe	14 KB	Application

impleSocketCS\SocketClient\bin\Debug

Name	Size	Type
SocketClient.exe	9 KB	Application
SocketClient.pdb	16 KB	Program Debug Dat...
SocketClient.vshost.exe	14 KB	Application

2. If the following Windows Security Alert box displayed, click the Unblock button.



Windows Security Alert

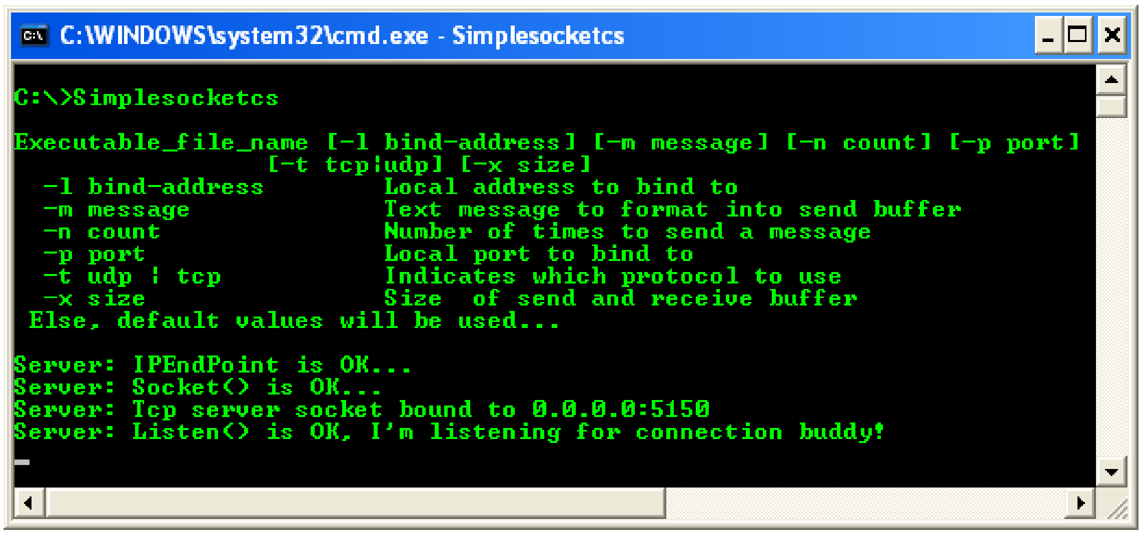
To help protect your computer, Windows Firewall has blocked some features of this program.

Do you want to keep blocking this program?

Name: SimpleSocketCS
Publisher: Personal

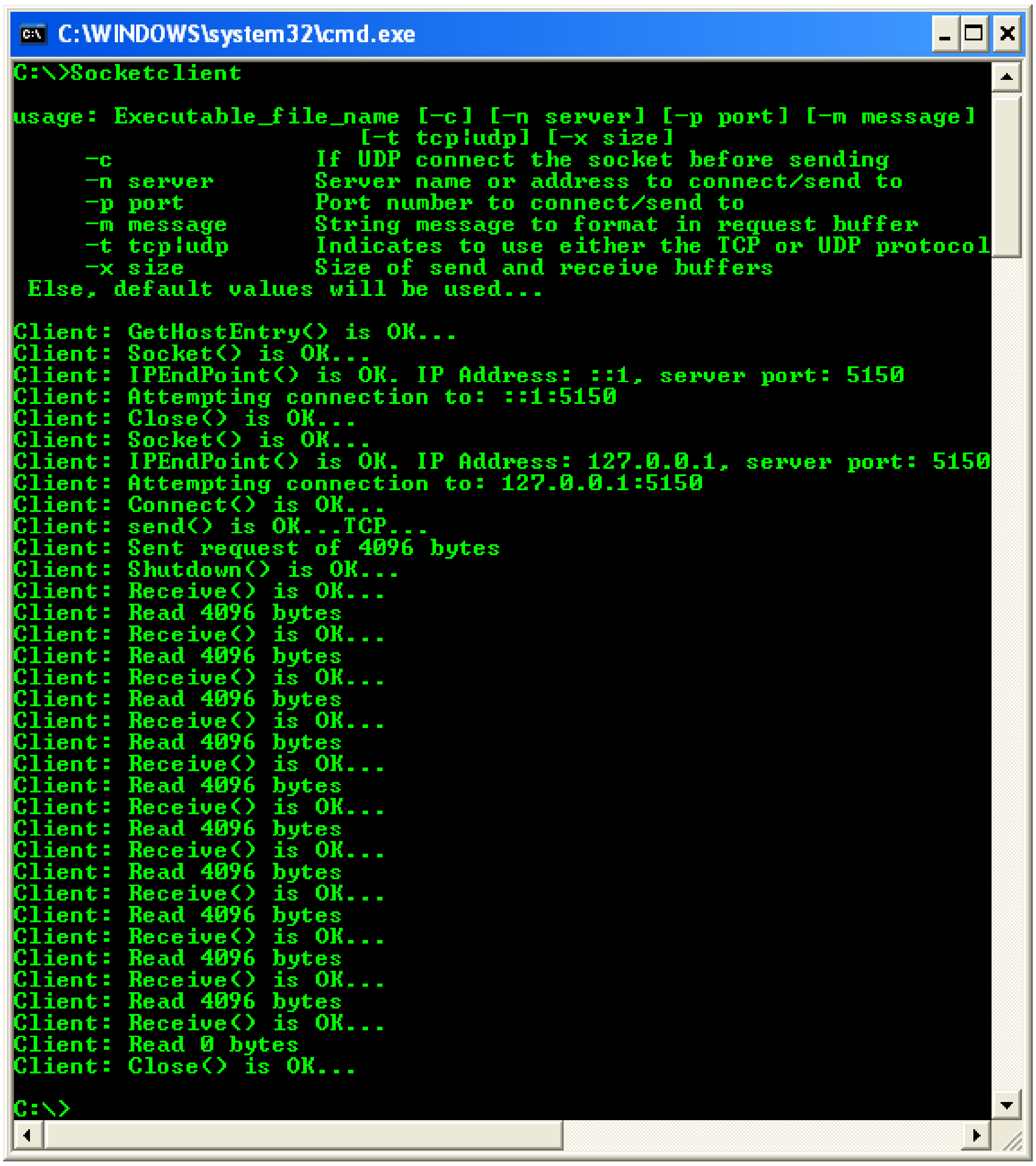
Windows Firewall has blocked this program from accepting connections from the Internet or a network. If you recognize the program or trust the publisher, you can unblock it. When should I unblock a program?

3. Run the server program and the following is a sample output.



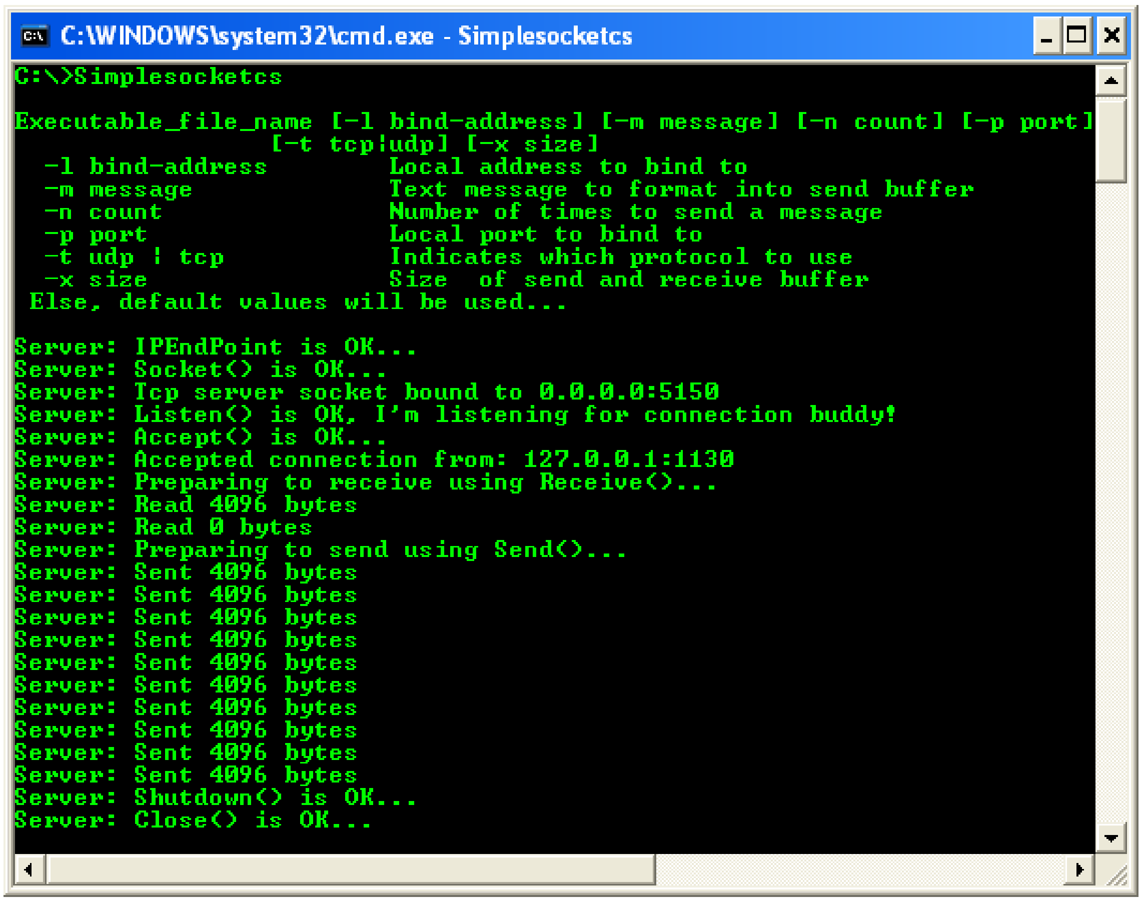
```
C:\>Simplesocketcs

Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]
                 [-t tcp|udp] [-x size]
  -l bind-address        Local address to bind to
  -m message             Text message to format into send buffer
  -n count               Number of times to send a message
  -p port                Local port to bind to
  -t udp | tcp           Indicates which protocol to use
  -x size                Size  of send and receive buffer
 Else, default values will be used...

Server: IPEndPoint is OK...
Server: Socket() is OK...
Server: Tcp server socket bound to 0.0.0.0:5150
Server: Listen() is OK, I'm listening for connection buddy!
_
```

4. Next, run the client program.



```
C:\>Socketclient

usage: Executable_file_name [-c] [-n server] [-p port] [-m message]
                        [-t tcp|udp] [-x size]
     -c                  If UDP connect the socket before sending
     -n server           Server name or address to connect/send to
     -p port             Port number to connect/send to
     -m message          String message to format in request buffer
     -t tcp|udp          Indicates to use either the TCP or UDP protocol
     -x size             Size of send and receive buffers
 Else, default values will be used...

Client: GetHostEntry() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: ::1, server port: 5150
Client: Attempting connection to: ::1:5150
Client: Close() is OK...
Client: Socket() is OK...
Client: IPEndPoint() is OK. IP Address: 127.0.0.1, server port: 5150
Client: Attempting connection to: 127.0.0.1:5150
Client: Connect() is OK...
Client: send() is OK...TCP...
Client: Sent request of 4096 bytes
Client: Shutdown() is OK...
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 4096 bytes
Client: Receive() is OK...
Client: Read 0 bytes
Client: Close() is OK...

C:\>
```

6. The following is the server output screenshot when the communication was completed.



```
C:\>Simplesocketcs

Executable_file_name [-l bind-address] [-m message] [-n count] [-p port]
                 [-t tcp|udp] [-x size]
  -l bind-address         Local address to bind to
  -m message              Text message to format into send buffer
  -n count                Number of times to send a message
  -p port                 Local port to bind to
  -t udp | tcp            Indicates which protocol to use
  -x size                 Size  of send and receive buffer
 Else, default values will be used...

Server: IPEndPoint is OK...
Server: Socket() is OK...
Server: Tcp server socket bound to 0.0.0.0:5150
Server: Listen() is OK, I'm listening for connection buddy!
Server: Accept() is OK...
Server: Accepted connection from: 127.0.0.1:1130
Server: Preparing to receive using Receive()...
Server: Read 4096 bytes
Server: Read 0 bytes
Server: Preparing to send using Send()...
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Sent 4096 bytes
Server: Shutdown() is OK...
Server: Close() is OK...
```