Articles / Programming Languages / C#

C# Windows .NET Visual-Studio TCP/IP

TCPIP Server and Client example

Patrick Eckler

6 Dec 2023 CPOL 9 min read 146K 14.8K 161

Example of a TCPIP server that listens and can serve multiple client connections

This article describes a classic TCP/IP server that can receive multiple client connections. The focus of the article is the unique packet collection and assembly process used by the server and client sides which allows the user to create specific commands for transmission to the server or another or all clients.

Download source code - 10.7 MB

UPDATED as of December 5, 2023...

* Made some changes to better handle multiple simultaneous incoming connection.

How It Works

When a `TCPIPClient` connects to the `TCPIPServer`, the `TCPIPServer` fires back to the newly connected `TCPIPClient` data about the other `TCPIPClient`s who are already connected AND THEN tells the already connected `TCPIPClient`s of the newly arrived `TCPIPClient`. So now, each client has a list of all the other clients who are on the system along with some details of who they are, like their IP addresses, their name, the name of the computer they're on and the connection ID that the `TCPIPServer` has given to the `TCPIPClient`.

When you select one or more `TCPIPClient`s to send a text message to, the text is put into packets... the `TCPIPServer` gets the packets and are re-routed to the specific `TCPIPClient`. The server knows where to redirect the packet because the '`idTo`' data field in the packet is set to the targeted `TCPIPClient`...

C#

```
/****************************************************************/
//prepare the start packet
xdata.Packet_Type = (UInt16)PACKETTYPES.TYPE_Message;
xdata.Data_Type = (UInt16)PACKETTYPES_SUBMESSAGE.SUBMSG_MessageStart;
xdata.Packet_Size = 16;
xdata.maskTo = 0;

// Set this so server will re-direct this message to the connected client.
// If it's '0' then it will only go to the server.
xdata.idTo = (uint)clientsHostId;

// Set this so the client who is getting your message will know who it's from.
xdata.idFrom = (uint)MyHostServerID;
```

NOTE: Sending data like this is not the most efficient way because the packets have to travel from the `TCPIPClient` computer to the `TCPIPServer` computer, then again to the targeted `TCPIPClient`... someone creative may want to create a server in each `TCPIPClient` so the only role the `TCPIPServer` would play would be to introduce the `TCPIPClient`s and provide enough data to each other that they could then make a point-to-point connections without having to send data via the `TCPIPServer`. But that's a lesson for another day!

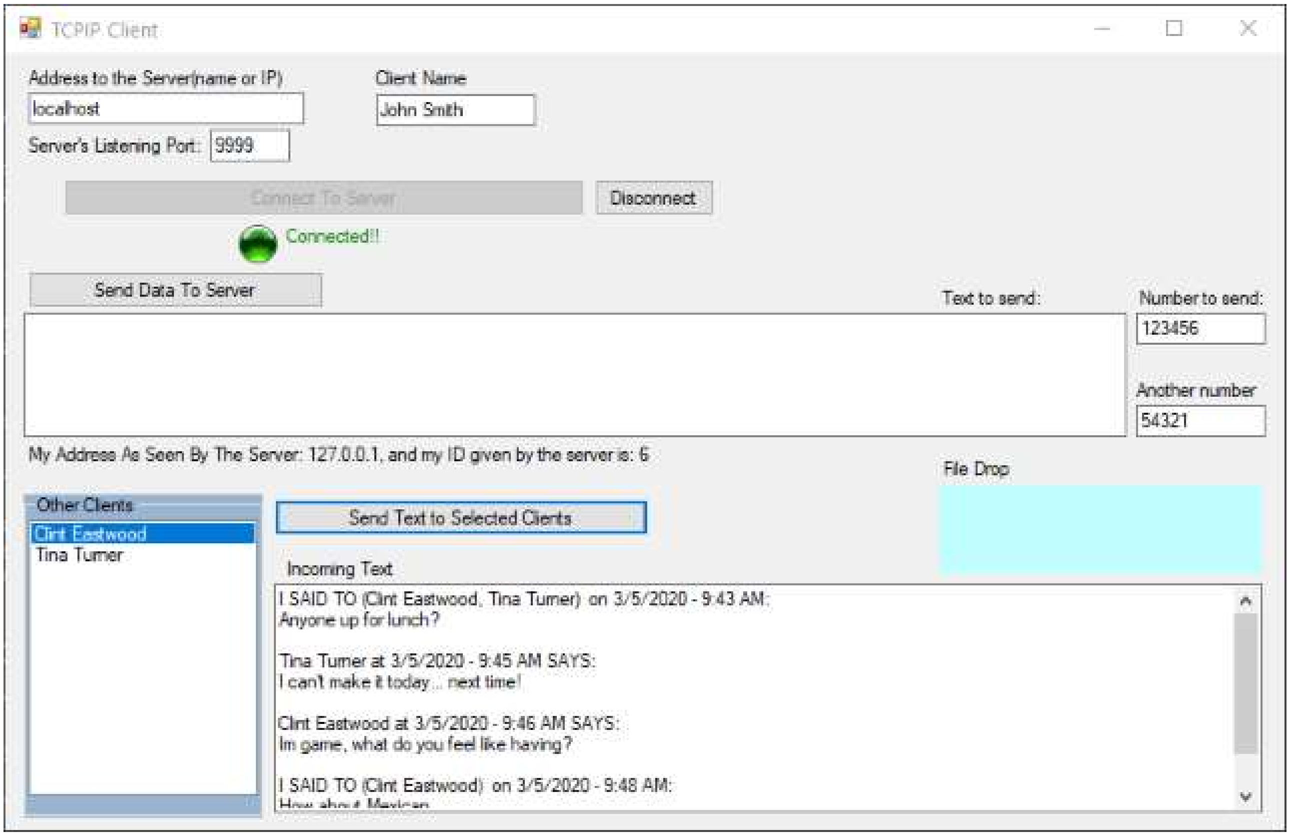
File Transfer

Also note the blue **'File Drop'** area.... drag and drop some files on there and see what happens. :)

Introduction

The solution contains a `TCPIPServer` project, a `TCPIPClient` project and a `CommonClassLibs` DLL project that the first two projects share.

The solution was created using Microsoft Visual Studio 2015, .NET Framework 4.5... The server is set to listen on port 9999 so it will ask to open the firewall for that.



Address to the Server(name or IP): localhost

Client Name: John Smith

Server's Listening Port: 9999

Text to send:

Number to send: 123456

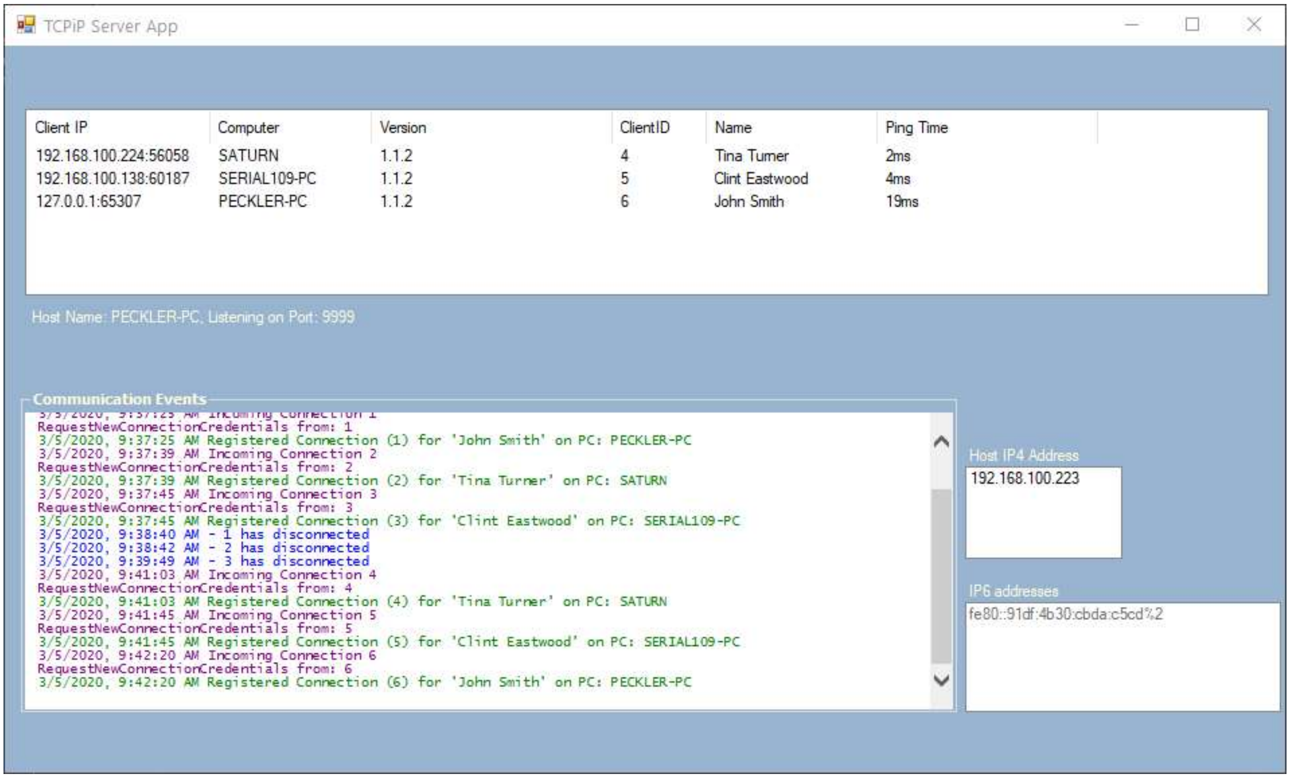
Another number: 54321

My Address As Seen By The Server: 127.0.0.1, and my ID given by the server is: 6

Other Clients:
- Clint Eastwood
- Tina Turner

```
I SAID TO (Clint Eastwood, Tina Turner)  on 3/5/2020 - 9:43 AM:
Anyone up for lunch?

Tina Turner at 3/5/2020 - 9:45 AM SAYS:
I can't make it today... next time!

Clint Eastwood at 3/5/2020 - 9:46 AM SAYS:
Im game, what do you feel like having?

I SAID TO (Clint Eastwood)  on 3/5/2020 - 9:48 AM:
How about Mexican
```

`TCPIPClient` ... you can run several of these on a network!



Client IP	Computer	Version	ClientID	Name	Ping Time
192.168.100.224:56058	SATURN	1.1.2	4	Tina Turner	2ms
192.168.100.138:60187	SERIAL109-PC	1.1.2	5	Clint Eastwood	4ms
127.0.0.1:65307	PECKLER-PC	1.1.2	6	John Smith	19ms

Host Name: PECKLER-PC, Listening on Port: 9999

```
3/5/2020, 9:37:25 AM Incoming Connection 1
RequestNewConnectionCredentials from: 1
3/5/2020, 9:37:25 AM Registered Connection (1) for 'John Smith' on PC: PECKLER-PC
3/5/2020, 9:37:39 AM Incoming Connection 2
RequestNewConnectionCredentials from: 2
3/5/2020, 9:37:39 AM Registered Connection (2) for 'Tina Turner' on PC: SATURN
3/5/2020, 9:37:45 AM Incoming Connection 3
RequestNewConnectionCredentials from: 3
3/5/2020, 9:37:45 AM Registered Connection (3) for 'Clint Eastwood' on PC: SERIAL109-PC
3/5/2020, 9:38:40 AM - 1 has disconnected
3/5/2020, 9:38:42 AM - 2 has disconnected
3/5/2020, 9:39:49 AM - 3 has disconnected
3/5/2020, 9:41:03 AM Incoming Connection 4
RequestNewConnectionCredentials from: 4
3/5/2020, 9:41:03 AM Registered Connection (4) for 'Tina Turner' on PC: SATURN
3/5/2020, 9:41:45 AM Incoming Connection 5
RequestNewConnectionCredentials from: 5
3/5/2020, 9:41:45 AM Registered Connection (5) for 'Clint Eastwood' on PC: SERIAL109-PC
3/5/2020, 9:42:20 AM Incoming Connection 6
RequestNewConnectionCredentials from: 6
3/5/2020, 9:42:20 AM Registered Connection (6) for 'John Smith' on PC: PECKLER-PC
```

Host IP4 Address: 192.168.100.223

IP6 addresses: fe80::91df:4b30:cbda:c5cd%2

The `TCPIPServer` with a `listview` of the client connections and a fancy event viewer so we can see what's going on inside.

Background

As a developer of windows programs, there was a need to be able to communicate and send real-time data to other users who were using the same application at the same time. Perhaps you develop an application that allows you to create and edit common network documents... if another person is also viewing that same document and making changes, then you will want to make sure that the other user is not overwriting the changes you are making at that moment. So, there should be a way to communicate information between the users so they know that changes are being made.

This article describes a classic TCP/IP server that can receive several client connections and handles all the data packets that come from the client side connections. The packets are assembled and processed on the server OR they can be forwarded to other single clients or sent to all of them simultaneously.

If you are really creative, you can use the server to simply let clients know about the other connected clients and pass enough information to each other that they can just talk directly to one other (typically called a rendezvous server). Here is a picture of what I call 'GComm'(Group Communicator) using this principle. This app is a tool for sending files and RTF messages to one or more people on a network. The core mechanism for receiving and constructing data packets is what this article will describe.

The nuts and bolts are described below...

Using the Code

The TCPIPServer Program

Let me preface by saying that in this example, I'm using a fixed packet size which is inefficient since typically most of the packet space won't be used, but as you will see it's not the end of the world.... as pointed out by a commenter using 'length prefixing' would be the best way... in this case the user would stuff in the size of the packet as well a type so you would then just assemble the tcpip chunks to that length and cast the whole thing to its original state.

Let's have a look at the server side of the `TCPIPServer` project... the main purpose of this application is to listen for and connect to client connections. We have a set of global variables for the project:

C#

```
/*******************************************************/
/// <summary>
/// TCPiP server
/// </summary>
```

```
Server svr = null;

private Dictionary<int, MotherOfRawPackets> dClientRawPacketList = null;
private Queue<FullPacket> FullPacketList = null;
static AutoResetEvent autoEvent;//mutex
static AutoResetEvent autoEvent2;//mutex
private Thread DataProcessThread = null;
private Thread FullPacketDataProcessThread = null;
/*******************************************************/
```

- The '`Server`' is the TCP layer class that establishes a Socket that listens on a port for incoming client connection and gives up the raw data packets to the interface via an Event callback... it also maintains a few items of information on each client and note a defined **Packet class** that contains a data buffer of **1024** bytes.

  C#

  ```
  private Socket _UserSocket;
  private DateTime _dTimer;
  private int _iClientID;
  private string _szClientName;
  private string _szStationName;
  private UInt16 _UserListeningPort;
  private string _szAlternateIP;
  private PingStatsClass _pingStatClass;

  /// <summary>
  /// Represents a TCP/IP transmission containing the socket it is using, the 
  clientNumber
  ///  (used by server communication only), and a data buffer representing the message.
  /// </summary>
  private class Packet
  {
      public Socket CurrentSocket;
      public int ClientNumber;
      public byte[] DataBuffer = new byte[1024];

       /// <summary>
       /// Construct a Packet Object
       /// </summary>
       /// <param name="sock">The socket this Packet is being used on.</param>
       /// <param name="client">The client number that this packet is from.</param>
       public Packet(Socket sock, int client)
       {
          CurrentSocket = sock;
          ClientNumber = client;
       }
  }
  ```

- The '`dClientRawPacketList`' is a `Dictionary` that handles each clients raw data packets. As a client attaches to the server, the server creates and assigns a unique integer value(starting at 1) to each client... a `Dictionary` entry is made for that client where the Key value in the `Dictionary` is the unique value. As those clients fire data packets to the server, it collects that clients packets in the dictionary's `MotherOfRawPackets` class which manages a queue type list of classes called `RawPackets`.

C#

```
public class RawPackets
{
    public RawPackets(int iClientId, byte[] theChunk, int sizeofchunk)
    {
        _dataChunk = new byte[sizeofchunk]; //create the space
        _dataChunk = theChunk;              //ram it in there
        _iClientId = iClientId;             //save who it came from
        _iChunkLen = sizeofchunk;           //hang onto the space size
    }

    public byte[] dataChunk { get { return _dataChunk; } }
    public int iClientId { get { return _iClientId; } }
    public int iChunkLen { get { return _iChunkLen; } }

    private byte[] _dataChunk;
    private int _iClientId;
    private int _iChunkLen;
}
```

- The '`FullPacketList`' is a Queue type list. Its purpose is to hold onto the incoming packets in the order by which they arrived. If you have 10 client connections all firing data at the server, the server's `DataProcessingThread` function will assemble those packets into full packets and store them into this list for processing shortly thereafter.
- There are 2 AutoEvent mutexes used in packet assembly threads, `autoEvent` and `autoEvent2` (sorry for the generic names). These allow those threaded function to efficiently sleep when data is being processed.
- As mentioned above, the '`DataProcessThread`' and the '`FullPacketDataProcessThread`' are 2 threads that work hand in hand to assemble data packets in the exact order they were sent.

As the `TCPIPServer` application starts up, we initialize the above defined variables:

C#

```
private void StartPacketCommunicationsServiceThread()
{
    try
    {
        //Packet processor mutex and loop
        autoEvent = new AutoResetEvent(false); //the RawPacket data mutex
        autoEvent2 = new AutoResetEvent(false);//the FullPacket data mutex
        DataProcessThread = new Thread(new ThreadStart(NormalizeThePackets));
        FullPacketDataProcessThread = new Thread(new ThreadStart(ProcessRecievedData));

        //Lists
        dClientRawPacketList = new Dictionary<int, MotherOfRawPackets>();
        FullPacketList = new Queue<FullPacket>();

        //Create HostServer
        svr = new Server();
```

```
        svr.Listen(MyPort);//MySettings.HostPort);
        svr.OnReceiveData += new Server.ReceiveDataCallback(OnDataReceived);
        svr.OnClientConnect += new Server.ClientConnectCallback(NewClientConnected);
        svr.OnClientDisconnect += new Server.ClientDisconnectCallback(ClientDisconnect);

        DataProcessThread.Start();
        FullPacketDataProcessThread.Start();
        OnCommunications($"TCPiP Server is listening on port {MyPort}", INK.CLR_GREEN);
    }
    catch(Exception ex)
    {
        var exceptionMessage = (ex.InnerException != null) ?
                                ex.InnerException.Message : ex.Message;
        //Debug.WriteLine($"EXCEPTION IN: StartPacketCommunicationsServiceThread -
        // {exceptionMessage}");
        OnCommunications($"EXCEPTION: TCPiP FAILED TO START,
                         exception: {exceptionMessage}", INK.CLR_RED);
    }
}
```

Note the '`NormalizeThePackets`' and '`ProcessRecievedData`'(yes misspelled) threads... as the TCP Socket layer throws up its incoming packets, we get a hold of them in the `NormalizeThePackets` function loop. As long as the application is listening (`while(svr.IsListening)`), the function thread stays alive and sits at the `autoEvent.WaitOne()` mutex until data comes in on the TCP layer and we call `autoEvent.Set()`, which allows the application process to drop through and process the data that is being collected in the `dClientRawPacketList Dictionary`. Each client's dictionary entry (`MotherOfRawPackets`) is examined for data, if one of the attached clients have sent packets, then `RawPackets` Queue list will have items to be processed. The Packets are concatenated together and once 1024 bytes are strung together, we know we have 1 full packet! That packet is Enqueued into the `FullPacket` Queue List, then the 2nd mutex is triggered(`autoEvent2.Set()`) to drop past the loop in the other threaded function(`ProcessRecieveData`)... see below. :)

NOTE: With TCPIP, we know that the data packets are guaranteed to arrive intact and in order of how they were sent...

Knowing that then we can assume that if we send packets of data then we know that we can assemble them when we get them on the receiving side... but the trickery of the TCP layer is that the packets can come in varying chunks before we get the whole thing, so we need a way to stick the chunks together to reassemble what was originally sent...

C#

```
private void NormalizeThePackets()
{
    if (svr == null)
        return;

    while (svr.IsListening)
    {
        autoEvent.WaitOne();//wait at mutex until signal
```

```
/***********************************************/
lock (dClientRawPacketList)//http://www.albahari.com/threading/part2.aspx#_Locking
{
    foreach (MotherOfRawPackets MRP in dClientRawPacketList.Values)
    {
        if (MRP.GetItemCount.Equals(0))
            continue;
        try
        {
            byte[] packetplayground = new byte[11264];//good for
                                //10 full packets(10240) + 1 remainder(1024)
            RawPackets rp;

            int actualPackets = 0;

            while (true)
            {
                if (MRP.GetItemCount == 0)
                    break;

                int holdLen = 0;

                if (MRP.bytesRemaining > 0)
                    Copy(MRP.Remainder, 0, packetplayground, 0, MRP.bytesRemaining);

                holdLen = MRP.bytesRemaining;

                for (int i = 0; i < 10; i++)//only go through a max of
                          //10 times so there will be room for any remainder
                {
                    rp = MRP.GetTopItem;//dequeue

                    Copy(rp.dataChunk, 0, packetplayground, holdLen, rp.iChunkLen);

                    holdLen += rp.iChunkLen;

                    if (MRP.GetItemCount.Equals(0))//make sure there is more
                                                   //in the list before continuing
                        break;
                }

                actualPackets = 0;

                if (holdLen >= 1024)//make sure we have at least one packet in there
                {
                    actualPackets = holdLen / 1024;
                    MRP.bytesRemaining = holdLen - (actualPackets * 1024);

                    for (int i = 0; i < actualPackets; i++)
                    {
                        byte[] tmpByteArr = new byte[1024];
                        Copy(packetplayground, i * 1024, tmpByteArr, 0, 1024);
                        lock (FullPacketList)
                            FullPacketList.Enqueue(new FullPacket
                                       (MRP.iListClientID, tmpByteArr));
                    }
```

```
                        }
                        else
                        {
                            MRP.bytesRemaining = holdLen;
                        }

                        //hang onto the remainder
                        Copy(packetplayground, actualPackets * 1024, MRP.Remainder,
                                               0, MRP.bytesRemaining);

                        if (FullPacketList.Count > 0)
                            autoEvent2.Set();

                    }//end of while(true)
                }
                catch (Exception ex)
                {
                    MRP.ClearList();//pe 03-20-2013
                    string msg = (ex.InnerException == null) ?
                                  ex.Message : ex.InnerException.Message;

                    OnCommunications
                       ("EXCEPTION in  NormalizeThePackets - " + msg, INK.CLR_RED);
                }
            }//end of foreach (dClientRawPacketList)
        }//end of lock
        /**********************************************/
        if (ServerIsExiting)
            break;
    }//Endof of while(svr.IsListening)

    Debug.WriteLine("Exiting the packet normalizer");
    OnCommunications("Exiting the packet normalizer", INK.CLR_RED);
}
```

Now the `ProcessRecievedData` function:

C#

```
private void ProcessReceivedData()
{
    if (svr == null)
        return;

    while (svr.IsListening)
    {
        autoEvent2.WaitOne();//wait at mutex until signal

        try
        {
            while (FullPacketList.Count > 0)
            {
                FullPacket fp;
                lock (FullPacketList)
                    fp = FullPacketList.Dequeue();
                //Console.WriteLine(GetDateTimeFormatted +" - Full packet fromID: " +
```

```
            //fp.iFromClient.ToString() + ", Type: " +
            //((PACKETTYPES)fp.ThePacket[0]).ToString());
            UInt16 type = (ushort)(fp.ThePacket[1] << 8 | fp.ThePacket[0]);
            switch (type)//Interrogate the first 2 Bytes to see what the packet TYPE is
            {
                case (UInt16)PACKETTYPES.TYPE_MyCredentials:
                    {
                        PostUserCredentials(fp.iFromClient, fp.ThePacket);
                        SendRegisteredMessage(fp.iFromClient, fp.ThePacket);
                    }
                    break;
                case (UInt16)PACKETTYPES.TYPE_CredentialsUpdate:
                    break;
                case (UInt16)PACKETTYPES.TYPE_PingResponse:
                    //Debug.WriteLine(DateTime.Now.ToShortDateString() + ", " +
                    //DateTime.Now.ToLongTimeString() + " - Received Ping from: " +
                    //fp.iFromClient.ToString() + ", on " +
                    //DateTime.Now.ToShortDateString() + ", at: " +
                    //DateTime.Now.ToLongTimeString());
                    UpdateTheConnectionTimers(fp.iFromClient, fp.ThePacket);
                    break;
                case (UInt16)PACKETTYPES.TYPE_Close:
                    ClientDisconnect(fp.iFromClient);
                    break;
                case (UInt16)PACKETTYPES.TYPE_Message:
                    {
                        AssembleMessage(fp.iFromClient, fp.ThePacket);
                    }
                    break;
                default:
                    PassDataThru(type, fp.iFromClient, fp.ThePacket);
                    break;
            }
        }//END  while (FullPacketList.Count > 0)
    }//END try
    catch (Exception ex)
    {
        try
        {
            string msg = (ex.InnerException == null) ?
                          ex.Message : ex.InnerException.Message;
            OnCommunications($"EXCEPTION in  ProcessRecievedData - {msg}", INK.CLR_RED);
        }
        catch { }
    }

    if (ServerIsExiting)
        break;
}//End while (svr.IsListening)

string info2 = string.Format("AppIsExiting = {0}", ServerIsExiting.ToString());
string info3 = string.Format("Past the ProcessRecievedData loop");

Debug.WriteLine(info2);
Debug.WriteLine(info3);

try
```

```
    {
        OnCommunications(info3, INK.CLR_RED); // "Past the ProcessRecievedData Loop"
                                              // also is logged to InfoLog.log
    }
    catch { }

    if (!ServerIsExiting)
    {
        //if we got here then something went wrong, we need to shut down the service
        OnCommunications("SOMETHING CRASHED", INK.CLR_RED);
    }
}
```

Ok, we have described how data is received from the clients on the TCPIP server application! Let's look at the packet of data that is transmitted... both the server and client have this packet defined in the `CommonClassLib` DLL... I decided that I would just create a generic class called `PACKET_DATA` of a fixed size of a computer friendly number of 1024. You can create as many classes as you like. Just make sure that they are 1024 bytes. **Note that that matches the size of the Packet class described in the Service class above.**

So! For each full packet that comes in and is EnQueued in the `FullPacketList`, this is the class we are getting.

The very first variable is an unsigned `short(UInt16)` called `Packet_Type`. If we interrogate the first 2 bytes as seen in the `ProcessRecievedData` function above, we can then figure out what the rest of the data in the class contains.

C#

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public class PACKET_DATA
{
    /****************************************************************/
    //HEADER is 18 BYTES
    public UInt16 Packet_Type;  //TYPE_??
    public UInt16 Packet_Size;
    public UInt16 Data_Type;    // DATA_ type fields
    public UInt16 maskTo;        // SENDTO_MY_SHUBONLY and the like.
    public UInt32 idTo;            // Used if maskTo is SENDTO_INDIVIDUAL
    public UInt32 idFrom;        // Client ID value
    public UInt16 nAppLevel;
    /****************************************************************/

    public UInt32 Data1;        //miscellaneous information
    public UInt32 Data2;        //miscellaneous information
    public UInt32 Data3;        //miscellaneous information
    public UInt32 Data4;        //miscellaneous information
    public UInt32 Data5;        //miscellaneous information

    public Int32 Data6;         //miscellaneous information
    public Int32 Data7;         //miscellaneous information
    public Int32 Data8;         //miscellaneous information
    public Int32 Data9;         //miscellaneous information
```

```
        public Int32 Data10;        //miscellaneous information

        public UInt32 Data11;       //miscellaneous information
        public UInt32 Data12;       //miscellaneous information
        public UInt32 Data13;       //miscellaneous information
        public UInt32 Data14;       //miscellaneous information
        public UInt32 Data15;       //miscellaneous information

        public Int32 Data16;        //miscellaneous information
        public Int32 Data17;        //miscellaneous information
        public Int32 Data18;        //miscellaneous information
        public Int32 Data19;        //miscellaneous information
        public Int32 Data20;        //miscellaneous information

        public UInt32 Data21;       //miscellaneous information
        public UInt32 Data22;       //miscellaneous information
        public UInt32 Data23;       //miscellaneous information
        public UInt32 Data24;       //miscellaneous information
        public UInt32 Data25;       //miscellaneous information

        public Int32 Data26;        //miscellaneous information
        public Int32 Data27;        //miscellaneous information
        public Int32 Data28;        //miscellaneous information
        public Int32 Data29;        //miscellanious information
        public Int32 Data30;        //miscellaneous information

        public Double DataDouble1;
        public Double DataDouble2;
        public Double DataDouble3;
        public Double DataDouble4;
        public Double DataDouble5;

        /// <summary>
        /// Long value1
        /// </summary>
        public Int64 DataLong1;
        /// <summary>
        /// Long value2
        /// </summary>
        public Int64 DataLong2;
        /// <summary>
        /// Long value3
        /// </summary>
        public Int64 DataLong3;
        /// <summary>
        /// Long value4
        /// </summary>
        public Int64 DataLong4;

        /// <summary>
        /// Unsigned Long value1
        /// </summary>
        public UInt64 DataULong1;
        /// <summary>
        /// Unsigned Long value2
        /// </summary>
        public UInt64 DataULong2;
```

```
    /// <summary>
    /// Unsigned Long value3
    /// </summary>
    public UInt64 DataULong3;
    /// <summary>
    /// Unsigned Long value4
    /// </summary>
    public UInt64 DataULong4;

    /// <summary>
    /// DateTime Tick value1
    /// </summary>
    public Int64 DataTimeTick1;

    /// <summary>
    /// DateTime Tick value2
    /// </summary>
    public Int64 DataTimeTick2;

    /// <summary>
    /// DateTime Tick value1
    /// </summary>
    public Int64 DataTimeTick3;

    /// <summary>
    /// DateTime Tick value2
    /// </summary>
    public Int64 DataTimeTick4;

    /// <summary>
    /// 300 Chars
    /// </summary>
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 300)]
    public Char[] szStringDataA = new Char[300];

    /// <summary>
    /// 300 Chars
    /// </summary>
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 300)]
    public Char[] szStringDataB = new Char[300];

    /// <summary>
    /// 150 Chars
    /// </summary>
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 150)]
    public Char[] szStringData150 = new Char[150];

    //18 + 120 + 40 + 96 + 600 + 150 = 1024
}
```

Creating an `enum` and defining a set of packet types allows us to know what the data is that's coming in from a client.

C#

```
public enum PACKETTYPES
{
    TYPE_Ping = 1,
    TYPE_PingResponse = 2,
    TYPE_RequestCredentials = 3,
    TYPE_MyCredentials = 4,
    TYPE_Registered = 5,
    TYPE_HostExiting = 6,
    TYPE_ClientData = 7,
    TYPE_ClientDisconnecting = 8,
    TYPE_CredentialsUpdate = 9,
    TYPE_Close = 10,
    TYPE_Message = 11,
    TYPE_MessageReceived = 12,
    TYPE_FileStart = 13,
    TYPE_FileChunk = 14,
    TYPE_FileEnd = 15,
    TYPE_DoneRecievingFile = 16
}
```

Again, this `PACKETTYPES enum` is also part of the `CommonClassLib` DLL that are shared between the `TCPIPServer` and `TCPIPClient` programs.

The TCPIPClient Program

The `TCPIPClient` program is almost identical to the server as far as how it processes data packets but it only has to worry about what it's getting from the server, rather than several TCP streams from several clients.

The `TCPIPClient` also has a client side version of the TCP layer that does a connect to attach to the listening server.

C#

```
/*******************************************************/
private Client client = null;//Client Socket class

private MotherOfRawPackets HostServerRawPackets = null;
static AutoResetEvent autoEventHostServer = null;//mutex
static AutoResetEvent autoEvent2;//mutex
private Thread DataProcessHostServerThread = null;
private Thread FullPacketDataProcessThread = null;
private Queue<FullPacket> FullHostServerPacketList = null;
/*******************************************************/
```

Here is a client side example of the client responding to a `TYPE_Ping` message from the server:

C#

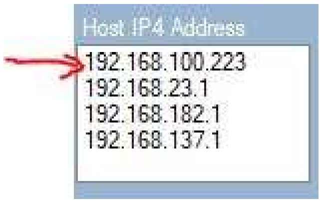
```
private void ReplyToHostPing(byte[] message)
{
    try
```

```
    {
        PACKET_DATA IncomingData = new PACKET_DATA();
        IncomingData = (PACKET_DATA)PACKET_FUNCTIONS.ByteArrayToStructure
                       (message, typeof(PACKET_DATA));


/**********************************************************************************/
        //calculate how long that ping took to get here
        TimeSpan ts = (new DateTime(IncomingData.DataLong1)) - (new DateTime(ServerTime));
        Console.WriteLine($"{GeneralFunction.GetDateTimeFormatted}:
        {string.Format("Ping From Server to client: {0:0.##}ms", ts.TotalMilliseconds)}");

/**********************************************************************************/

        ServerTime = IncomingData.DataLong1;// Server computer's current time!

        PACKET_DATA xdata = new PACKET_DATA();

        xdata.Packet_Type = (UInt16)PACKETTYPES.TYPE_PingResponse;
        xdata.Data_Type = 0;
        xdata.Packet_Size = 16;
        xdata.maskTo = 0;
        xdata.idTo = 0;
        xdata.idFrom = 0;

        xdata.DataLong1 = IncomingData.DataLong1;

        byte[] byData = PACKET_FUNCTIONS.StructureToByteArray(xdata);

        SendMessageToServer(byData);

        CheckThisComputersTimeAgainstServerTime();
    }
    catch (Exception ex)
    {
        string exceptionMessage = (ex.InnerException != null) ?
                                   ex.InnerException.Message : ex.Message;
        Console.WriteLine($"EXCEPTION IN: ReplyToHostPing - {exceptionMessage}");
    }
}
```

Compiling and Running the Apps in the Solution

To start, compile the `CommonClassLibs` project. This creates the DLL that the `TCPIPServer` and the `TCPIPClient` will need. It contains the classes and enumerations that each side will need along with a few common functions. Make sure that you reference this DLL in the other projects.

Compile the `TCPIPServer` and the `TCPIPClient` projects, then run the `TCPIPServer`... it will likely want to make a rule in the computers firewall to allow port 9999 through so go ahead and allow that. Take note of the computer's IP address on the network:



192.168.100.223
192.168.23.1
192.168.182.1
192.168.137.1

(If more than one, then it's likely the first one.)

Once it's running, fire up the `TCPIPClient` application... Set the IP address of the `TCPIPServer` in the **'Address to the Server'** textbox. If you are running this on the same computer using *localhost* should work. Run this app on as many computers as you like and click the **'Connect to Server'** button. If the red indicator turns green, then the connection was made... it turns green when the client gets a `TYPE_Registered` message from the server.

Points of Interest

I've used this method between applications for years and it's pretty solid!

History

- A rainy November the 15th, 2017 day in Livonia Michigan

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Written By

Patrick Eckler

Software Developer (Senior)

🇺🇸 United States

Born and raised in the city of Detroit...

C, C++, C# application and web developer.

https://PESystemsllc.com/

Email: EcklerPa@yPESystemsllc.com

Comments and Discussions

60 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1215517/TCPIP-Server-and-Client-example** to post and view comments on this article, or click **here** to get a print view with messages.

Posted 17 Nov 2017

Article Copyright 2017 by Patrick Eckler
Everything else Copyright © CodeProject, 1999-2023

Web03 2.8:2023-10-29:1