Sochinda
Take and share any coding experience

Basic Client/Server Chat Application in C#

Welcome to the **Basic Client/Server Chat Application in C#**. In this tutorial I will provide the basics for a simple chat application in C# utilizing TCPClient, StreamReader, and the StreamWriter Classes in the .Net Framework.

In this application you have 3 components, the server (a class file), the communication component (a class file) and the client application. We will look at all 3 of these components individually, and how the can combine to create your basic chat application. The first component, the chat server, is where the messages are sent back and forth between the client and the server. Before writing any methods you need to add the following references to your class.

view sourceprint?

```
using System.IO;
using System.Net;
using System;
using System.Threading;
using Chat = System.Net;
using System.Collections;
```

I know some of you are going to look at the 5th reference and ask questions regarding Chat = System.Net. When adding references in C# you are allowed to add *aliases* to your references, thus allowing you to have multiple uses of the same Namespace at the same time, acting as 2 different objects.

NOTE: To use Aliases for the Namespace reference it has to be in conjunction with the **Using** Statement.

The first thing we do in our Server class is create 3 global variables, 2 are Hashtable variables, and the third is a TCPListener variable, which is used to listen for connections from TCP Clients.

view sourceprint?

```
System.Net.Sockets.TcpListener chatServer;
 public static Hashtable nickName;
public static Hashtable nickNameByConnect;
```

These three variables will be used throughout our ChatServer.cs class file. Next, is the Public ChatServer() method, this is where we start the chat server and connect. We will then use our **TCPListener** object to check if there are any pending connection requests. If there are pending requests we then create a new connection, let the user know they're connected, then create our **DoCommunication** Object.

We'll get to the **DoCommunication** object later in this tutorial. Here is the code for this method

view sourceprint?

```
public ChatServer()
{
//create our nickname and nickname by connection variables
nickName = new Hashtable(100);
nickNameByConnect = new Hashtable(100);
//create our TCPListener object
chatServer = new System.Net.Sockets.TcpListener(4296);
//check to see if the server is running
//while (true) do the commands
while (true)
{
//start the chat server
chatServer.Start();
//check if there are any pending connection requests
if (chatServer.Pending())
{
//if there are pending requests create a new connection
Chat.Sockets.TcpClient chatConnection = chatServer.AcceptTcpClient();
//display a message letting the user know they're connected
Console.WriteLine("You are now connected");
//create a new DoCommunicate Object
DoCommunicate comm = new DoCommunicate(chatConnection);
}
}
}
```

Next, since this is a *basic* chat application, we need a method for sending our messages to all that are connected. Here we create a **StreamWriter** object, used to write our messages to the chat window, a TcpClient Array, to hold all the TcpClients for all connected users, then we copy the users nickname to the chat server window. After that we create a loop, looping through all the TcpClients, we check if the message eing sent is empty or that index of our TcpClient array is empty. From there we send our message to the chat window, and flush to make sure the buffer is empty.

In your Catch, of our Try...Catch block, is where we handle the Exception that is caused when a user leaves or disconnects. We display a message letting the users know that that person has disconnected, we remove that nickname from the list, then dispose of that users TcpClient instance. Here is the code for this method

view sourceprint?

```
public static void SendMsgToAll(string nick, string msg)
{
//create a StreamWriter Object
StreamWriter writer;
ArrayList ToRemove = new ArrayList(0);
```

```
//create a new TCPClient Array
Chat.Sockets.TcpClient[] tcpClient = new
Chat.Sockets.TcpClient[ChatServer.nickName.Count];
//copy the users nickname to the CHatServer values
ChatServer.nickName.Values.CopyTo(tcpClient, 0);
//loop through and write any messages to the window
for (int cnt = 0; cnt < tcpClient.Length; cnt++)
{
try
{
//check if the message is empty, of the particular
//index of out array is null, if it is then continue
if (msg.Trim() == "" || tcpClient[cnt] == null)
continue;
//Use the GetStream method to get the current memory
//stream for this index of our TCPClient array
writer = new StreamWriter(tcpClient[cnt].GetStream());
//white our message to the window
writer.WriteLine(nick + ": " + msg);
//make sure all bytes are written
writer.Flush();
//dispose of the writer object until needed again
writer = null;
}
//here we catch an exception that happens
//when the user leaves the chatroow
catch (Exception e44)
{
e44 = e44;
string str = (string)ChatServer.nickNameByConnect[tcpClient[cnt]];
//send the message that the user has left
ChatServer.SendSysMsg("** " + str + " ** Has Left The Room.");
//remove the nickname from the list
ChatServer.nickName.Remove(str);
//remove that index of the array, thus freeing it up
//for another user
ChatServer.nickNameByConnect.Remove(tcpClient[cnt]);
}
}
}
```

The next method we introduce is a way to send a system message, this method is almost identical to the SendMsgToAll method, except here we dont dispose of the TcpClient instance, since the message is being sent by the system, not a user.

view sourceprint?

```
public static void SendSystemMessage(string msg)
{
//create our StreamWriter object
StreamWriter writer;
ArrayList ToRemove = new ArrayList(0);
//create our TcpClient array
```

```
Chat.Sockets.TcpClient[] tcpClient = new
Chat.Sockets.TcpClient[ChatServer.nickName.Count];
//copy the nickname value to the chat servers list
ChatServer.nickName.Values.CopyTo(tcpClient, 0);
//loop through and write any messages to the window
for (int i = 0; i < tcpClient.Length; i++)
{
try
{
//check if the message is empty, of the particular
//index of out array is null, if it is then continue
if (msg.Trim() == "" || tcpClient[i] == null)
continue;
//Use the GetStream method to get the current memory
//stream for this index of our TCPClient array
writer = new StreamWriter(tcpClient[i].GetStream());
//send our message
writer.WriteLine(msg);
//make sure the buffer is empty
writer.Flush();
//dispose of our writer
writer = null;
}
catch (Exception e44)
{
```

```
e44 = e44;
ChatServer.nickName.Remove(ChatServer.nickNameByConnect[tcpClient[i]]);
ChatServer.nickNameByConnect.Remove(tcpClient[i]);
}
}
}
```

Believe it or not, thats the entirety of the ChatServer Class, simple isnt it. Working with Tcp objects can be fun, as you can do so much with them. In this simple application you could add the functionality to send files back and forth between users, and more. That may be the end of the ChatServer Class, but its not the end of creating our application.

The next component to look at is the **DoCommunicate** Class. This is the component that does the work for our server. For a chat application to work efficiently, and work as people expect a chat application to work, it needs to be a multi-threaded application. Meaning each user is running in their own thread, which allows for the messages to be sent and received in real time. Multi threading gives the illusion that multiple activities are happening at the same time.

The main purpose of multi threading is to improve performance. With each user in the chat application operating on their own thread, users don't have to wait for one user to be finished to send their message, they're able to send them simultaneously. C# has some powerful items in the **System.Threading Namespace**, which is used for, you guessed it, running multiple threads and synchronizing them.

For our **DoCommunicate.cs** class file we need the following references

view sourceprint?

```
using System.IO;
using System.Net;
using System;
using System.Threading;
using Chat = System.Net;
using System.Collections;
using PC;
```

Once again we add an alias to an instance of the **System.Net Namespace** reference, this prevents namespace collisions in our class. Like the ChatServer class, the first thing we do in our class is create some global variables, 4 of them:

- A **TCPClient** object
- A **StreamReader** object
- A **StreamWriter** object
- And a string object

In this method is where the new **Thread** is created and started, allowing this user to react in real time in the application.

view sourceprint?

```
public DoCommunicate(System.Net.Sockets.TcpClient tcpClient)
{
    //create our TcpClient
    client = tcpClient;
    //create a new thread
Thread chatThread = new Thread(new ThreadStart(startChat));
    //start the new thread
    chatThread.Start();
}
```

Notice when we create our new **Thread** we pass it a method called **startChat**. We'll get to this method momentarily, but first we need to do a couple things that **startChat** relies on. Once the thread is created and
started, we need to get the nickname the user wishes to use. For this we use the **GetNick** method we created. Here we simply ask the user what their nickname is, then return that value to the **startChat** method.

view sourceprint?

```
private string GetNick()
{
//ask the user what nickname they want to use
writer.WriteLine("What is your nickname? ");
//ensure the buffer is empty
writer.Flush();
//return the value the user provided
return reader.ReadLine();
}
```

Now lets look at the aforementioned **startChat** method. Here we create our **StreamReader** and **StreamWriter** objects and set the global string variable **nickName** to the value returned from the **GetNick** method. Next thing we do is check to ensure that the nickname provided by the user doesn't already exist, if it does we prompt them for a nickname until we find one thats not already in use.

Once they provide a valid nickname we add their nickname to the server, preventing another user from using it, then we send a system message letting the other users know there is a new user. From there we create a new **Thread**, which calls the **runChat** method. Lets first look at the **startChat** method

view sourceprint?

```
private void startChat()
{
//create our StreamReader object to read the current stream
reader = new System.IO.StreamReader(client.GetStream());
//create our StreamWriter objec to write to the current stream
writer = new System.IO.StreamWriter(client.GetStream());
writer.WriteLine("Welcome to PCChat!");
//retrieve the users nickname they provided
nickName = GetNick();
//check is the nickname is already in session
//prompt the user until they provide a nickname not in use
while (PC.ChatServer.nickName.Contains(nickName))
{
//since the nickname is in use we display that message,
//then prompt them again for a nickname
writer.WriteLine("ERROR - Nickname already exists! Please try a new one");
nickName = GetNick();
}
//add their nickname to the chat server
PC.ChatServer.nickName.Add(nickName, client);
PC.ChatServer.nickNameByConnect.Add(client, nickName);
//send a system message letting the other user
//know that a new user has joined the chat
PC.ChatServer.SendSystemMessage("** " + nickName + " ** Has joined the room");
writer.WriteLine("Now Talking.....\r\n-------------------------------");
//ensure the buffer is empty
writer.Flush();
//create a new thread for this user
Thread chatThread = new Thread(new ThreadStart(runChat));
//start the thread
chatThread.Start();
}
```

The last method in our **DoCommunicate.cs Class** is the **runChat** method called by the new thread in **startChat**. This is simply for reading the current stream and sending our messages to the chat window.

view sourceprint?

```
private void runChat()
//use a try...catch to catch any exceptions
{
try
{
//set out line variable to an empty string
string line = "";
while (true)
{
//read the curent line
line = reader.ReadLine();
//send our message
PC.ChatServer.SendMsgToAll(nickName, line);
}
}
catch (Exception e44)
{
Console.WriteLine(e44);
}
}
```

That is the end of our **DoCommunicate** class. So far you have seen how to create a chat server, a class to handle the work of the chat application. You have learned about **TcpClients**, **TcpListeners**, **StreamReaders**, **StreamWriters**, and **Threads**. We discussed the purpose of a multi threaded application, and how to create one, and you have learned about adding an alias to your reference to prevent namespace collision in your application.

Now that we have our chat server completely defined, we need a client application to chat with. In this application I have a single form, ChatClient, but I did this a little differently. I didn't add any controls via drag and drop, I added them at runtime, personally I wouldn't recommend this for new programmers.

First thing i our client application is a **Windows API** call, the reference we need is the **ExitProcess** function. That looks like this

view sourceprint?

```
[DllImport("kernel32.dll")]
private static extern void ExitProcess(int a);
```

In **void Main** is where I create a new form, add my controls, set the properties of the window, WindowState, Text, and my **TcpClient** and call the **Connect** method of the **System.Net.Sockets.TcpClient Class**.

With the **Connect** method you provide the IP address, or host name, along with the port number to connect to, then it connects you to that information. Since this is a basic application, that information is hard coded into the application, with a real application you would have an area to give the user the option to specify which chat server they wish to connect to. Aside from the Main method we have three more methods:

- **ChatClient_Closing**: This handles what needs to be done once the user closes the application. This all happens as the form is closing.
- **key_up**: This is what sends our message to the chat window. Since I do it on the key up event, they will see what you're typing as you trype. For an actual application this functionality would be added to a **Send** button, or when the user hits Enter.
- **Run**: This is the running of the chat application, reading the current stream and appending it to the current contents of the chat window, and placing the cursor at the end of the text already in the textbox you're typing your message into

How I'm appending the text to the current contents of the chat window is by using the **AppendText Method** of the **TextBox Class**.

First lets look at the code for the **Closing Event** of the form.

view sourceprint?

```
private static void ChatClient_Closing(object s, CancelEventArgs e)
{
    e.Cancel = false;
//exit the application
Application.Exit();
//call the ExitProcess API
ExitProcess(0);
}
```

When the form closes, it calls the **Application.Exit Method**, then the call to the **ExitProcess Function**.

Next we have the code for the **Control.KeyUp Event**, which is what sends our messages to the chat window. In this method, we create a **StreamWriter** for writing to the current stream. To do this we call the **GetStream Method** of the **System.Net.Sockets.TcpClient** class. **GetStream** retrieves the current **NetworkStream**, used for sending and receiving messages across a network.

view sourceprint?

```
private static void key_up(object s, KeyEventArgs e)
{
//create our textbox value variable
TextBox txtChat = (TextBox)s;
//check to make sure the length of the text
//in the TextBox is greater than 1 (meaning it has text in it)
if (txtChat.Lines.Length > 1)
{
//create a StreamWriter based on the current NetworkStream
StreamWriter writer = new StreamWriter(tcpClient.GetStream());
//write our message
writer.WriteLine(txtChat.Text);
//ensure the buffer is empty
writer.Flush();
//clear the textbox for our next message
txtChat.Text = "";
txtChat.Lines = null;
}
}
```

Next we have the code for our **run** method. This creates a **StreamReader** Object, using **GetStream** to retrieve the current **NetworkStream**, this will be used for reading the messages in the stream. We then append the value in the current stream, line by line, to the chat window.

view sourceprint?

```
private static void run()
{
//create our StreamReader Object, based on the current NetworkStream
StreamReader reader = new StreamReader(tcpClient.GetStream());
while (true)
{
//call DoEvents so other processes can process
//simultaneously
Application.DoEvents();
//create a TextBox reference
TextBox txtChat = (TextBox)client.Controls[0];
//append the current value in the
//current NetworkStream to the chat window
txtChat.AppendText(reader.ReadLine() + "\r\n");
//place the cursor at the end of the
//text in the textbox for typing our messages
txtChat.Selectionstart = txtChat.Text.Length;
}
}
```

That is the end of the tutorial Basic Client/Server Chat Application in C#. I am enclosing all three files with this tutorial. They are under the **Public GNU License** which means you can modify the code to suit your needs, but you need to provide a reference to the original creator of the code. Also, you are not allowed to remove the license header at the beginning of all the files in this solution.

I hope you enjoyed this tutorial, and found it useful. I will next write a tutorial for an advanced client/server chat application, to show what can be done with the techniques we learned in this tutorial.

Thank you so much for reading :)

NOTE: You're going to want to take the **ChatServer Class** and possibly make an application out of that as well. I have it as a class file as I'm using a different implementation of the server.

Attached File Basic_ClientServer_Chat.zip (132.21K)

Referenced by: http://www.dreamincode.net/forums/topic/33396-basic-clientserver-chat-application-in-c%23/

http://www.codeproject.com/KB/IP/dotnettcp.aspx

http://www.codeproject.com/KB/IP/chatserver.aspx

sochinda **January 26, 2011** **C#, Window Application**

Published by sochinda

I really like new technology!! **View more posts**

6 thoughts on “Basic Client/Server Chat Application in C#”

Sarath
September 23, 2011 at 9:43 PM

Hi, my name is Sarath. My team has registered into a software design contest at RUPP. We want to create a Chatting and Video Conferencing Application, but we don’t have any idea of where to start, and to do. Can u please drop me your idea, your help could support me a lot. We have only 3 months more, so hope u can respond my message soon.

Yours,
Sarath

Reply

sochinda
September 28, 2011 at 6:37 AM

Oh! I have no experience in Video Conferencing Application, but can please check this link http://xmpp.org/, i hope it can help u because XMPP is protocol that Google use to to develop Google Talk

Reply

saffron extract weight loss does it work
May 22, 2013 at 12:19 AM

Acquiring more than enough natural vitamins and minerals in your physique can also make your hair shinier and a lot more workable.
Since of this it is recommended for expecting girls, as it will help to reduced the chance of spina bifida.

Reply

dragons of atlantis ruby hack
January 25, 2014 at 1:15 PM

Hi! Would you mind if I share your blog with my zynga group?
There's a lot of folks that I think would really appreciate your content.
Please let me know. Thanks

Reply

http://www.sagami-kadan.jp/userinfo.php?uid=106015
February 15, 2014 at 8:40 AM

I enjoy what you guys are usually up too. Such clever work and reporting!
Keep up the terrific works guys I've added you guys to my
personal blogroll.

Reply

youtube.com
July 21, 2014 at 4:04 AM

If you are planning to engage in a personal injury settlement, seek the services of our experienced Los
Angeles attorneys. It is recommended to hire a professional because the claim laws can vary greatly from state to state, and if you
are not familiar with these nuances, you may not get your fully entitled payout.
Far worse is the pain in personal injury cases when individual's rights are not effectively represented and innocent parties wind up getting shafted.

Reply

Leave a Reply

Enter your comment here...