TCP/IP Chat Application Using C#

ALLEXY

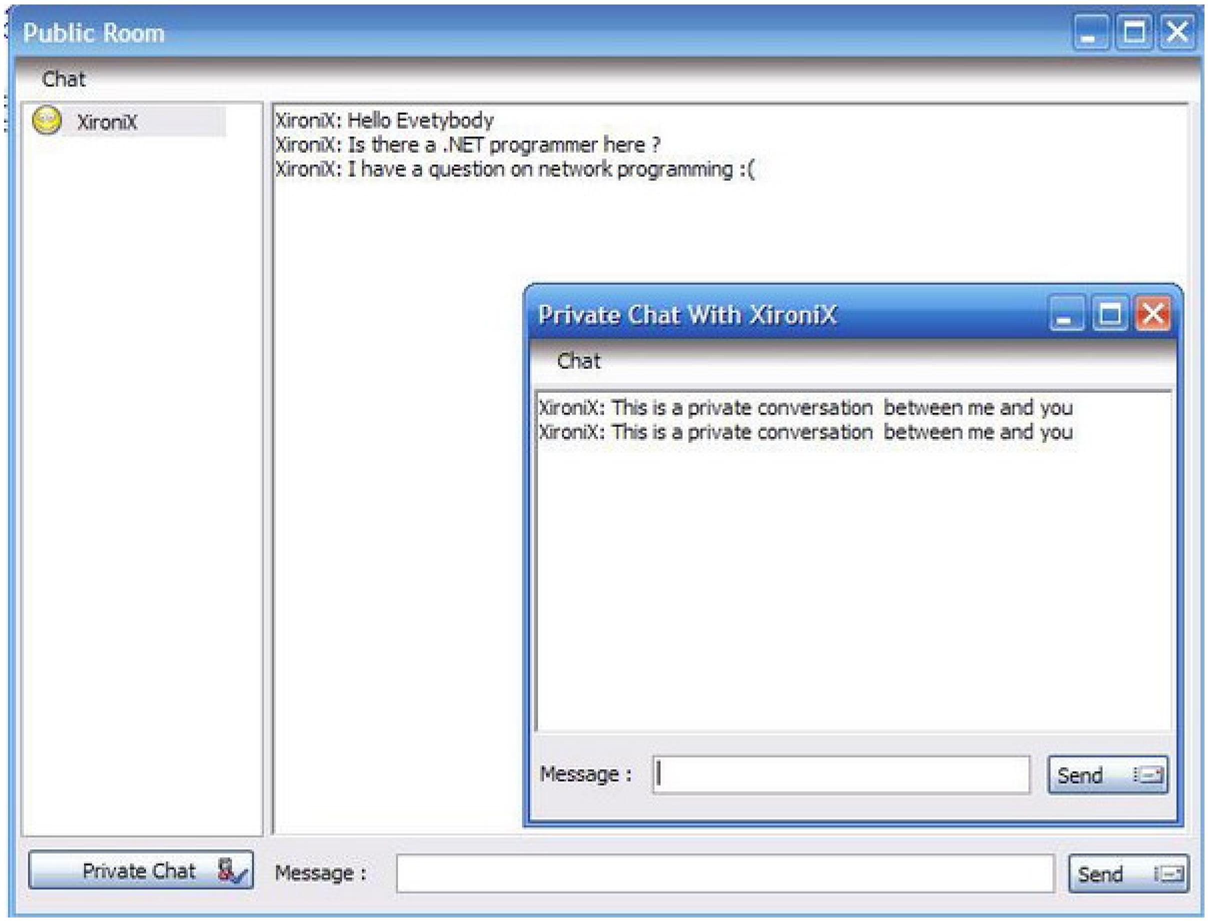
Rate me: 4.84/5 (212 votes)

30 Jan 2006 CPOL

This is a LAN chat application with TCP/IP socket programming technology in C#. This application is a multi thread network application and works in a non-blocking way. Public and private chat is also implemented in this code.

Download source files - 345 Kb

Download demo - 298 Kb



XironiX: Hello Evetybody
XironiX: Is there a .NET programmer here ?
XironiX: I have a question on network programming :(

XironiX: This is a private conversation between me and you
XironiX: This is a private conversation between me and you

Introduction

The power of network programming in .NET platform cannot be denied. Socket programming is the core of network programming in Windows and Linux, and today the .NET platform implements it in a powerful way. In this article, we will see the basics of socket programming in C#. To be precise, I have created a command client and a command server, to communicate between a remote server and up to 200 clients and send the specified commands to them. As a sample application, I have created a chat client application that uses this command client/server to implement chat functions. Before I start explaining my application, let me give you a small introduction on network programming and sockets taken from the book 'C# network programming', written by Richard Blum.

Sockets

In socket-based network programming, you don't directly access the network interface device to send and receive packets. Instead, an intermediary connector is created to handle the programming interface to the network. Assume that a socket is a connector that connects your application to a network interface of your computer. For sending and receiving data to and from the network you should call the socket's methods.

Socket programming in C#

The 'System.Net.Sockets' namespace contains the classes that provide the actual .NET interface to the low-level Winsock APIs. In network programming, apart from which programming language to use there are some common concepts like the IP address and port. IP address is a unique identifier of a computer on a network and port is like a gate through which applications communicate with each other. In brief, when we want to communicate with a remote computer or a device over the network, we should know its IP address. Then, we must open a gate (Port) to that IP and then send and receive the required data.

IP addresses in C#

One of the biggest advantages you will notice in the .NET network library is the way IP address/port pairs are handled. It is a fairly straightforward process that presents a welcome improvement over the old, confusing UNIX way. .NET defines two classes in the System.Net namespace to handle various types of IP address information:

- IPAddress
- IPEndPoint

IPAddress

An IPAddress object is used to represent a single IP address. This value is then used in various socket methods to represent the IP address. The default constructor for IPAddress is as follows:

C#

```
public IPAddress(long address)
```

The default constructor takes a long value and converts it to an IPAddress value. In practice, the default is almost never used. Instead, several methods in the IPAddress class can be used to create and manipulate IP addresses. The Parse() method is often used to create IPAddress instances:

C#

```
IPAddress newaddress = IPAddress.Parse("192.168.1.1");
```

IPEndPoint

The .NET Framework uses the IPEndPoint object to represent a specific IP address/port combination. An IPEndPoint object is used when binding sockets to local addresses, or when connecting sockets to remote addresses.

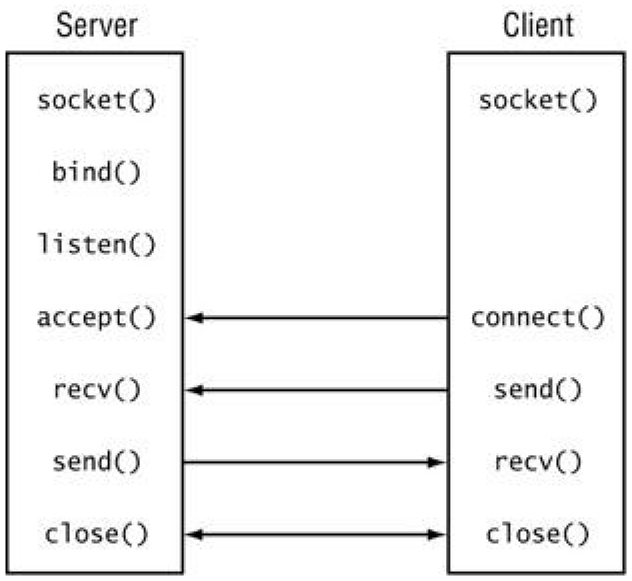
Connection-oriented and connectionless sockets

The world of IP connectivity revolves around two types of communication: connection-oriented and connectionless. In a connection-oriented socket, the TCP protocol is used to establish a session (connection) between two IP address endpoints. There is a fair amount of overhead involved with establishing the connection, but once it is established, the data can be reliably transferred between the devices.

Connectionless sockets use the UDP protocol. Because of that no connection information is required to be sent between the network devices and it is often difficult to determine which device is acting as a "server", and which is acting as a "client". We will focus on the first type of socket programming in this article.

Using connection-oriented sockets

In the .NET Framework, you can create connection-oriented communications with remote hosts across a network. To create a connection-oriented socket, separate sequences of functions must be used for server programs and client programs:



Server

You have four tasks to perform before a server can transfer data with a client connection:

1. Create a socket.
2. Bind the socket to a local `IPEndPoint`.
3. Place the socket in listen mode.
4. Accept an incoming connection on the socket.

Creating the server

The first step to constructing a TCP server is to create an instance of the `Socket` object. The other three functions necessary for successful server operations are then accomplished by using the methods of `Socket` object. The following C# code snippet includes these steps:

C#

```
IPEndPoint localEndPoint = new IPEndPoint(IPAddress.Any, 8000);
Socket newsock = Socket(AddressFamily.InterNetwork,
                   SocketType.Stream, ProtocolType.Tcp);
newsock.Bind(localEndPoint);
newsock.Listen(10);
Socket client = newsock.Accept();
```

The `Socket` object created by the `Accept()` method can now be used to transmit data in either direction between the server and the remote client.

Client

Now that you have a working TCP server, you can create a simple TCP client program to interact with it. There are only two steps required to connect a client program to a TCP server:

1. Create a socket.
2. Connect the socket to the remote server address.

Creating the client

As it was for the server program, the first step for creating the client program is to create a `Socket` object. The `Socket` object is used by the socket `Connect()` method to connect the socket to a remote host:

C#

```
IPEndPoint ipep =
    new IPEndPoint(Ipaddress.Parse("127.0.0.1"), 8000);
Socket server = new Socket(AddressFamily.InterNetwork,
                  SocketType.Stream, ProtocolType.Tcp);
server.Connect(ipep);
```

This example attempts to connect the socket to the server located at address 127.0.0.1.This is the IP address of the local host (current computer) and is a loopback IP for testing a network application without a network. Of course, you can also use hostnames along with the `Dns.Resolve()` method in a real network. (Dns is in `System.Net` namespace). Once the remote server TCP program accepts the connection request, the client program is ready to transmit data with the server using the standard `Send()` and `Receive()` methods.

Blocking problem of network applications

Sockets are in blocking mode by default. In this mode they will wait forever to complete their functions, holding up other functions within the application program until they are complete. Many programs can work quite competently in this mode, but for applications that work in the Windows programming environment, this can be a problem. There are some ways to solve this problem. The first thing that comes to a programmer's mind is multi threading. I chose this solution in my application too. This is a simple way when compared to asynchronous network programming or the old 'Non-Blocking sockets' way.

Our command client/server

After a brief introduction on network programming in C#, I should give you more details about our command client/server application here. Of course, I can't write a book on network programming in this little article. This is only an introduction to network programming. You can find many samples and tutorials on MSDN and CodeProject explaining this concept in detail.

About the command server

The server application is a console program. After starting, it will bind to the '127.0.0.1' local IP and wait on the port 8000 by default for clients. You can pass the IP and the port of the server as the first and second command line parameters when starting the server, if you have a real network. For example: *c:\> ConsoleServer 192.198.0.100 8960*.

I used `BackgroundWorker` to implement multithreading in time consuming functions in the server and client. One of these actions includes the acceptance part of the server:

C#

```
bwListener = new BackgroundWorker();
bwListener.DoWork += new DoWorkEventHandler(StartToListen);
bwListener.RunWorkerAsync();

private void StartToListen(object sender , DoWorkEventArgs e)
{
   this.listenerSocket = new Socket(AddressFamily.InterNetwork,
                           SocketType.Stream, ProtocolType.Tcp);
   this.listenerSocket.Bind(
               new IPEndPoint(this.serverIP , this.serverPort));
   this.listenerSocket.Listen(200);
   while ( true )
      this.CreateNewClientManager(this.listenerSocket.Accept());
}
```

I have a class named `ClientManager`. When the server is connected to a remote client it passes the communication socket to this class and adds this new `ClientManager` object to a list of current connected remote clients. Each `ClientManager` object in the list is responsible for communicating with its remote client. The `ClientManager` object announces the server with various events defined in this class when an action takes place between the remote client and the server. These events are:

C#

```
public event CommandReceivedEventHandler CommandReceived;
```

Occurs when a command is received from a remote client.

C#

```
public event CommandSentEventHandler CommandSent;
```

Occurs when a command had been sent to the remote client successfully.

C#

```
public event CommandSendingFailedEventHandler CommandFailed;
```

Occurs when a command sending action fails. This is may be because of disconnection or sending exception.

C#

```
public event DisconnectedEventHandler Disconnected;
```

Occurs when a client is disconnected from this server.

Sending and receiving data

Since we have a command client/server application we should have a command object to send and receive data. This is implemented in a '`Command`' class. This class is the same in client and server. When the server wants to send a command to the client it builds a `Command` object and then sends it to the client and vice versa.

The `command` class is good for the user of this code. But in the network, we can't send and receive an object or a type. Everything should be converted to `byte` array. So, we should convert this object to a `byte` array part by part and send or receive it over the network in real `Send` and `Receive` functions inside our code. The following code shows the `send` command method. '`cmd`' is the command that we want to send to the remote client:

C# Shrink ▲

```
//Type
byte [] buffer = new byte [4];
buffer = BitConverter.GetBytes((int)cmd.CommandType);
this.networkStream.Write(buffer , 0 , 4);
this.networkStream.Flush();

//Sender IP
byte [] senderIPBuffer =
     Encoding.ASCII.GetBytes(cmd.SenderIP.ToString());
buffer = new byte [4];
buffer = BitConverter.GetBytes(senderIPBuffer.Length);
this.networkStream.Write(buffer , 0 , 4);
this.networkStream.Flush();
this.networkStream.Write(senderIPBuffer, 0,
                               senderIPBuffer.Length);
this.networkStream.Flush();

//Sender Name
byte [] senderNameBuffer =
   Encoding.Unicode.GetBytes(cmd.SenderName.ToString());
buffer = new byte [4];
buffer = BitConverter.GetBytes(senderNameBuffer.Length);
this.networkStream.Write(buffer , 0 , 4);
this.networkStream.Flush();
this.networkStream.Write(senderNameBuffer, 0,
                               senderNameBuffer.Length);
this.networkStream.Flush();

//Target
byte [] ipBuffer =
  Encoding.ASCII.GetBytes(cmd.Target.ToString());
buffer = new byte [4];
buffer = BitConverter.GetBytes(ipBuffer.Length);
this.networkStream.Write(buffer , 0 , 4);
this.networkStream.Flush();
this.networkStream.Write(ipBuffer , 0 , ipBuffer.Length);
this.networkStream.Flush();

//Meta Data.
if ( cmd.MetaData == null || cmd.MetaData == "" )
    cmd.MetaData = "\n";

byte [] metaBuffer =
          Encoding.Unicode.GetBytes(cmd.MetaData);
buffer = new byte [4];
buffer = BitConverter.GetBytes(metaBuffer.Length);
this.networkStream.Write(buffer , 0 , 4);
this.networkStream.Flush();
this.networkStream.Write(metaBuffer, 0, metaBuffer.Length);
this.networkStream.Flush();
```

The `send` and `receive` are bidirectional operations. For example, when we send 4 bytes to the client, the client should read the 4 bytes. We should repeat this operation until all the sent data is read. See the receive code of the client here:

C# Shrink ▲

```
while ( this.clientSocket.Connected )
{
    //Read the command's Type.
    byte [] buffer = new byte [4];
    int readBytes = this.networkStream.Read(buffer , 0 , 4);
    if ( readBytes == 0 )
      break;
    CommandType cmdType =
         (CommandType)( BitConverter.ToInt32(buffer , 0) );

    //Read the command's sender ip size.
    buffer = new byte [4];
    readBytes = this.networkStream.Read(buffer , 0 , 4);
    if ( readBytes == 0 )
    break;
    int senderIPSize = BitConverter.ToInt32(buffer , 0);

    //Read the command's sender ip.
    buffer = new byte [senderIPSize];
    readBytes =
        this.networkStream.Read(buffer , 0 , senderIPSize);
    if ( readBytes == 0 )
      break;
    IPAddress senderIP = IPAddress.Parse(
             System.Text.Encoding.ASCII.GetString(buffer));

    //Read the command's sender name size.
    buffer = new byte [4];
    readBytes = this.networkStream.Read(buffer , 0 , 4);
    if ( readBytes == 0 )
      break;
    int senderNameSize = BitConverter.ToInt32(buffer , 0);

    //Read the command's sender name.
    buffer = new byte [senderNameSize];
    readBytes = this.networkStream.Read(buffer, 0, senderNameSize);
    if ( readBytes == 0 )
      break;
    string senderName =
        System.Text.Encoding.Unicode.GetString(buffer);

    //Read the command's target size.
    string cmdTarget = "";
    buffer = new byte [4];
    readBytes = this.networkStream.Read(buffer , 0 , 4);
    if ( readBytes == 0 )
       break;
    int ipSize = BitConverter.ToInt32(buffer , 0);

    //Read the command's target.
    buffer = new byte [ipSize];
    readBytes = this.networkStream.Read(buffer , 0 , ipSize);
    if ( readBytes == 0 )
      break;
    cmdTarget = System.Text.Encoding.ASCII.GetString(buffer);
    //Read the command's MetaData size.
    string cmdMetaData = "";
    buffer = new byte [4];
    readBytes = this.networkStream.Read(buffer , 0 , 4);
    if ( readBytes == 0 )
      break;
    int metaDataSize = BitConverter.ToInt32(buffer , 0);

    //Read the command's Meta data.
    buffer = new byte [metaDataSize];
    readBytes = this.networkStream.Read(buffer , 0 , metaDataSize);
    if ( readBytes == 0 )
         break;
    cmdMetaData = System.Text.Encoding.Unicode.GetString(buffer);

    Command cmd = new Command(cmdType,
                        IPAddress.Parse(cmdTarget), cmdMetaData);
    cmd.SenderIP = senderIP;
    cmd.SenderName = senderName;
    this.OnCommandReceived(new CommandEventArgs(cmd));
    }
    this.OnServerDisconnected(new ServerEventArgs(this.clientSocket));
    this.Disconnect();
}
```

About the command client

The command client is very similar to the server. Everything is in the '`CommandClient`' class. Since our application is an event driven program this class also has some events to announce the user of the occurred actions. Here is a brief definition of these events:

C#

```
public event CommandReceivedEventHandler CommandReceived;
```

Occurs when a command is received from a remote client.

C#

```
public event CommandSentEventHandler CommandSent;
```

Occurs when a command has been sent to the remote server successfully.

C#

```
public event CommandSendingFailedEventHandler CommandFailed;
```

Occurs when a command sending action fails. This is because of disconnection or sending exception.

C#

```
public event ServerDisconnectedEventHandler ServerDisconnected;
```

Occurs when the client is disconnected.

C#

```
public event DisconnectedEventHandler DisconnectedFromServer;
```

Occurs when this client is disconnected from the remote server.

C#

```
public event ConnectingSuccessedEventHandler ConnectingSuccessed;
```

Occurs when this client is connected to the remote server successfully.

C#

```
public event ConnectingFailedEventHandler ConnectingFailed;
```

Occurs when this client fails on connecting to the server.

C#

```
public event NetworkDeadEventHandler NetworkDead;
```

Occurs when the network fails.

C#

```
public event NetworkAlivedEventHandler NetworkAlived;
```

Occurs when the network starts to work.

Conclusion

In this application, you can find the following concepts of .NET programming:

- Socket programming, server side and client side.
- Working with resources at runtime.
- Concurrency management in multi threaded environment.
- Calling windows API functions within C# code.
- Creating custom events and eventargs, and throwing events in a UI safe mode.
- Creating custom exceptions and throwing them as and when needed.
- Generating an HTML page at runtime dynamically.

And many other .NET programming concepts that I couldn't explain in detail here. The code is fully XML commented and very clear to understand. Please contact me if there is any ambiguous point or you need any help on my code.

License

Comments and Discussions

You must Sign In to use this message board.

Search Comments

First Prev Next

Very Well Written
mash1230 20-Oct-10 5:55

a small problem
vbvp 9-Sep-10 11:13

same code but for vb.net
abusharkas 22-Aug-10 18:20

Re: same code but for vb.net
thatraja 28-Jan-12 5:33

My vote of 5
tigerwang202 3-Aug-10 0:49

Great Project! 5++++ Stars!
dmofo 9-Jul-10 14:10

Connection problems
zimmwarrior 2-May-10 0:12

Re: Connection problems
SERVODROID 2-Aug-11 21:53

Thankyou, friend!
e_legion 16-Apr-10 9:30

Thanks
elinta 9-Apr-10 19:37

Thank you so much :)
simal10 27-Feb-10 13:21

getting error "Unable to read data from the transport connection: An existing connection was forcibly closed by the remote host."
MayurPanchal 22-Feb-10 6:55

problem with demo
iyerakshay1 17-Feb-10 4:46

A blocking operation was interrupted by a call to WSACancelBlockingCall [modified]
christine sarsonas 15-Jan-10 15:23

External IP [modified]
rajni pal 15-Jan-10 1:29

how to write server side for a tcp sender device (as client side)
fa-ha 7-Dec-09 6:46

شبکه
omar1388 29-Sep-09 2:37

How can I send Picture with this application?
phata4u 15-Sep-09 1:01

IPAddress.Parse
maxrpgg 22-Aug-09 20:13

Re: IPAddress.Parse
dmofo 9-Jul-10 14:04

How to integrate with Asp.net?
ajay_zenta 13-Aug-09 1:52

Some One please help ME
bobby_u13 23-Jul-09 17:13

Re: Some One please help ME [modified]
ALLEXY 23-Jul-09 20:08

Re: Some One please help ME

bobby_u13 27-Jul-09 11:30

Re: Some One please help ME

Mike Diack 12-Aug-09 1:01

Refresh ◁ Prev 1 2 3 4 **5** 6 7 8 9 10 11 Next ▷

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

Permalink
Advertise
Privacy
Cookies
Terms of Use

Layout: fixed | fluid

Web02 2.8:2023-08-14:1