ITP 484 Lab 2
TrojanChat Server and client

Purpose

The purpose of this lab is to create an application, TrojanChat, that can perform as both a chat server and chat client. It should be implemented in C++ and use TCP sockets for communication between hosts. It must support multiple connections from multiple clients. It must support the ability to relay a public message from one client to all clients, and also relay the name of the client sending the message. It should also allow the relay of a private message, sent only to one client, identified by name. Input and output should be through the windows console, using the WindowsConsole wrapper class that will be provided.

Specifications

The program should handle the following command lines as described. Note that TrojanChat is the name of the program, so argv[0] will always be TrojanChat- you don't need to check for it.

TrojanChat 45000

This should startup TrojanChat as a server and begin listening for connections on port 45000. Note that in this case 45000 will be the argv[1]

TrojanChat 192.168.2.10:45000 josh

This should startup TrojanChat as a client and attempt to connect to the server at ip address 192.168.2.10 on port 45000. It should also identify itself to the server as josh

Once a client has connected, that client should be able to type a line of text into the console, and send it to the server by pressing return. If a line of text does not start with a backtick `, then the line of text should be relayed to each connected clients. It should show up prefixed by the name of the client initiating the message. For instance, if client josh types

Hi everybody!

Then all connected clients receive the line of text

josh: Hi everybody!

A line of text that starts with a backtick ` should be considered a private message. When the server receives this line of text, it should consider the first token after the backtick to be the name of the client to which the message is addressed, and send that message to only that user, marking it as private. For instance, if client josh types

`Arnold How come you're late to class?

Then the server should relay a message to only Arnold such that Arnold 's console displays

josh (private) : How come you're late to class?

Steps

1) Open the TrojanChat.sln in Visual Studio 2013. Examine the layout- see how main either creates a Client or a Server and calls run. Throughout the lab, you will be adding code to the Client and to the Server, as well as creating other classes, to fully implement the required functionality.
2) Create a class called **TCPSocket** that will wrap a SOCKET and eventually have methods that wrap all the necessary socket api calls.
3) Create a class called **TCPSocketUtil** that is a friend of **TCPSocket**. Make sure that only **TCPSocketUtil** and **TCPSocket** can call the **TCPSocket** constructor. For the rest of this lab, make sure all socket api calls are wrapped inside methods in the **TCPSocketUtil** or **TCPSocket** classes.
    a. Implement a static method on **TCPSocketUtil** called **CreateSocket** that takes a port as a parameter. It should create a socket and, if the given port is non 0, bind it to the given port using any ip address. If the process is successful, it should return a std::shared_ptr< **TCPSocket** > that wraps the socket. If there is an error, return an empty std::shared_ptr< **TCPSocket** >.
    b. Implement static methods on **TCPSocketUtil** to wrap WSAStartup, WSACleanUp and WSAGetLastError. Call them when appropriate.
4) Build the functionality to parse the command line and determine if a server or client should be created
    a. Fill update main() so that it uses your logic to properly create a client or server. Feel free to add a constructor to both classes to take in any required parameters.
    b. In the server, use **TCPSocketUtil** to create a socket bound to the desired port and start listening for connections. Create a method **TCPSocket::Listen** to wrap the listen functionality.

    c. In the client, use TCPSocketUtil to create a socket but pass in 0 for the port. Implement **TCPSocket::Connect** to wrap the connect function, and attempt to connect to the desired address.
5) For now, for testing, build support into the server to support a single client
    a. Implement **TCPSocket::Accept** to wrap the accept socket function and return the new shared_ptr< TCPSocket >. On the server, call **Accept** right after listening. This will block until a connection comes in.
    b. Implement **TCPSocket::Send** that wraps send. When a connection does come in, send "**Hello World"** on the connection
6) For now, for testing, on the client, implement **TCPSocket::Receive** that wraps recv. Call this after the call to **Connect**, which will block until you receive some data. When you do, output the data to the windows console.
7) Start up a server and then start up a client (you might want to open two Visual Studio processes) and attempt to connect to the server. You should see "**Hello World**" in the client's console.
8) Implement **TCPSocket::SetNonBlocking**(bool inShouldBlock) to use ioctlsocket to disable blocking. Call this on the client socket so that it can send and receive without blocking.
    a. Edit the Client::Run function. Every trip through the loop, check if there is a pending line input to the console. If there is, send it down the connection to the server. Also check if there is anything to receive on the socket. If there is, display it on the console.
9) Now we want the server to be able to both listen for new incoming connections AND incoming data on each connection. Implement **TCPSocketUtil::Select** that wraps the socket api select function in a useful way (deliberately vague- lab is getting harder here)
10) You will find it useful (and thus you should) create a class called ClientProxy that the server class uses to track a connected client. You should have one of these for each connected client, and it should track the socket created for that client and any other information that you need to track per client.
11) Edit the run loop of the server. Using **Select**, handle accepting new connections and receiving data from existing connections. When new data comes in, send it to all the clients
12) Implement support for prefixing public messages with the sender's name. The client should only need to send its name to the server as part of the initial connection.
13) Implement support for private messaging by parsing the incoming messages on the server
14) Implement functionality such that when a client has disconnected from the server, the server eventually notices and cleans up its resources representing the client.
15) Test the server with 3 clients, sending public and private messages between them and making sure all output is correct. You'll need to either run your

compiled app multiple times by itself, or open many instances of visual studio. Make sure to test the following cases:

1) With 3 clients connected, have each send a public message and make sure it is received
2) With 3 clients connected, have each send a private message to another client
3) Have a client disconnect and have the other two send public messages
4) Have the two remaining clients try to private message the disconnected client and make sure nothing crashes
5) have the disconnected client reconnect with the same name and make sure all public and private messages work
6) have a client disconnect and reconnect with a different name and make sure everything works as expected

For testing, it's acceptable to run the server and client on the same computer. In this case, you can use 127.0.0.1 on the commandline as the target IP address, as that is the loopback address which always refers to the local host. (i.e. if a host tries to connect to 127.0.0.1 then it tries to connect to itself).

Make sure you do not leak memory. You will lose points for memory leaks, even if everything is functional.