"Network Applications and Design" Homework Assignment #4

"Simple Server-Client Chat App" (4+2 points)

Due date: **May 17th (Tue) 2022, 11:59 PM**.
- Submit softcopy on the server.

In this homework assignment, you'll implement a simple server-client **chatting** application where everyone (every client) connected to the server can chat with everyone else. For simplicity, there will be only one chat room. You will use TCP socket and Go language for this assignment.

Here are the descriptions and requirements on what you'll need to do.

You must implement two programs, ChatTCPClient.go for the client, and ChatTCPServer.go for the server. Your client-server program pair provides 'chatting' functionality.

Server accepts connections from any number of clients. For now, you may assume that the number of connected clients is always ≤ 8. All connected clients can chat with each other. If 9th client tries to connect, the server will send a "*chatting room full. cannot connect*" message to the client and refuse the connection. The client should show this message and quit gracefully.

The client can set its nickname(username) when it starts using a command line argument. This nickname is used for chatting. For example, when I run my client program to connect to the server, I can set my nickname as "*ironman*" and you can set yours as "*hulk*." Then these will be shown for all chatting. You may assume max. length of ≤ 32 characters for a nickname, English nickname, and no spaces or special characters (e.g. '\') in the nickname. However, if the server detects duplicate nickname, it should refuse the connection with a message "*that nickname is already used by another user. cannot connect*".

When a client connects to the server, the server will send back '*welcome*' message to the client. The client should print this message on the screen. Please also print the nickname, server IP address and port number together with the welcome message. For example, "*welcome <nickname> to CAU network class chat room at <serverIP:serverPort>. There are <N> users connected*". At the same time, the server should also print out "*<nickname> joined from <clientIP:clientPort>. There are <N> users connected*".

When a client connects to the server, it enters the chat room immediately. That is, everyone can chat with everyone else as soon as they connect to the server. When you type a chat message and press `Enter`, that message should be delivered to everyone (every client) connected, except yourself of course.

On the client, while running, if the user presses '**`Ctrl-C`**', the client should send an exit message to the server before quitting so that the server knows that you're gone. Then the client should exit gracefully with a message "*gg~*" printed on the screen. The server and ALL clients should also print out "*<nickname> left. There are <N> users now*" message on the screen. Note that 'N' is the number of currently connected users.

Chat room commands: While chatting, the user can type a text message and press `Enter` to send that message. The user can also send special commands to the server by typing the following strings as the message;

• `\list`	// show the <nickname, IP, port> list of all connected users
• `\dm <nickname> <message>`	// direct message only to <nickname>
• `\exit`	// disconnect from server, and exit
• `\ver`	// show server's software version
• `\rtt`	// show RTT (round trip time) from the client to the server and back

"`\list`" command should show the <nickname, IP, port> list of all connected users. "`\dm <nickname> <msg>`" command send direct message to <nickname>; which means it should send <msg> only to <nickname>, and no other user should see this message. Also, a direction message should be distinguishable from normal chat messages (e.g. like 'from xxx: <msg>'). Note that <msg> part may contain spaces. "`\exit`" command should do the same thing as when the user presses '`Ctrl-C`'. "`\ver`" should show the server's software version (you may freely choose your own version number yourself). "`\rtt`" should show the round-trip time (RTT) from the client to the server and back. I'll let you think about how to do this.

When sending a command to the server, the command part should not be sent in plain text from the client to the server. That is, you should never send the '`\xxxx`' part in ASCII string. This part should be encoded into 1 byte. I'll let you figure out and decide on how to do this.

If the user types an invalid command (a command that does not exist), for example "`\showmethemoney`", the client should print out a message *"invalid command"* and should not send this to the server.

When the server receives an invalid/unknown command, it should print out a message *"invalid command"* and do nothing more. (Ideally, this case should not happen if the client is doing its job. However in practice, this case happens when the client and server versions do not match; e.g. you upgraded the client to a newer version with more commands, but the server program has older version without those new commands.)

In addition to above, if anyone chats *"I hate professor"* (ignore case), the server must detect that, and disconnect that client. When doing this, the server and ALL clients must print out a message regarding this event, and the client must detect the fact that it is disconnected, print out a message regarding this event, and exit gracefully.

Example output from a client:

```
$ go run ChatTCPClient.go ironman

[Welcome ironman to CAU network class chat room at 165.194.35.202:29999.]
[There are 4 users connected.]

hi what's up?

hulk> don't you think prof. Paek is handsome?

couldn't agree more

wonderwoman> i think so too

superman> what? i hate professor. This homework is too difficult!
[superman is disconnected. There are 3 users in the chat room.]

\dm wonderwoman do you want to go out for a drink?

from: wonderwoman> no, I'm worried about COVID 19

\exit
```

A few important things to think about:

After reading above, one big question that should come to your mind is, which information/state to keep at the server and which to keep at the client. Should you make the server have all the intelligence and keep the client minimal/simple/dumb as possible? or should the client have sufficient information to reduce traffic between server and client. If latter, what if there is mismatch between the information at the server and the client? In this homework assignment, a lot of this is up to you; you have quite a bit of freedom to implement it in your own way. However, please think about the advantages and disadvantages of each design choice carefully.

Here are additional important things that you should think about while doing this assignment;

- Should the client send nickname to the server only once when it connects? or every time for every message (since you need it to show chatting screen anyway)? Should client keep its nickname at all?
- How should the server identify a client? using <ip, port> pair? or nickname? or some other unique identifier (an integer)? You need to be careful about these identifiers since some clients may connect and disconnect at any time (e.g. 1 connect, 2 connect, 3 connect, 1 disconnect, 4 connect, …), there may be same IP address or same port number (although not both together at the same time), same nickname, etc.
- What data structure would you like to use for the list of clients? What information to put for each client? Would your data structure be fast enough (for search/insert/delete) even if you have 20000 clients in the future?
- When sending commands from the client to the server (and vice versa also), you have two options: You can
    - use a text string (e.g. "\version", "\list") to represent a command (but not for this homework), or
    - encode that into binary information (you should do this for this homework).
    - What are the advantages and disadvantages of these? (string vs. binary)

Most of these questions are open ended; the decision is up to you. And since our software supports only small number of clients and the amount of data generated by chatting is also small, performance will not be a problem regardless of what choice you make. However, what if? what if you have 20000 clients? It would be interesting to think about this… and may be keep in mind when you do the next homework assignment.

Additional note and requirement regarding command line argument:

Ideally, it would be nice if both your programs, ChatTCPClient.go and ChatTCPServer.go take IP address and port number as command line arguments so that we can run them on any machine and any port without modifying the code. However, if I do that, it would be very difficult to grade your submissions. For this reason only, (although it is not nice), you should hardcode the 'server port number' in both programs with your personal designated port number. Also, you should hardcode the server address (nsl2.cau.ac.kr) in your client program. The only command line argument is the 'nickname' for the client program. That is, the client should run using the command "`$ go run ChatTCPClient.go <nickname>`", and the server should run using the command "`$ go run ChatTCPServer.go`",

Other additional requirements:

- Your program must run on our class Linux server at nsl2.cau.ac.kr. (both server and client on nsl2)
- Running your client on nsl5.cau.ac.kr and your server on nsl2.cau.ac.kr should work.
    - ✓ Both should work without ANY code modification.
- Ideally, your client and server programs should work on any computer on the Internet even if the client and the server programs are **on different machines**. This also means that all your programs should work on any operating system such as Windows, Linux, or MacOS.
    - ✓ However, due to security reasons, our university (CAU) blocks all ports when coming into the university from outside, except for a few that have explicit permission (e.g. 7722).
        - ▪ Thus, it may not be possible to connect to your server program on our class server (nsl2.cau.ac.kr) from your client program on your own computer at home.
- Make sure that you use your own designated port number for the server socket.
- For the client socket, you should either not set the port number, or must use null (0) port number which will let the OS assign a random port number to your socket.
- You should be able to restart any of your programs immediately after exiting/terminating
- You should close all sockets when exiting.
- Your code must include your name and student ID at the beginning of the code as a comment.
- Your code should be easily readable and include sufficient comments for easy understanding.
- Your code must be properly indented. Bad indentation/formatting will result in score deduction.
- Your code should not include any Korean characters, not even your names. Write your name in English.

What and how to submit

- You must submit softcopy of ChatTCPClient.go and ChatTCPServer.go programs.
- Here is the instruction on how to submit the softcopy files:
    - ✓ Login to your server account at nsl2.cau.ac.kr.
    - ✓ In your home directory, create a directory "submit_net/submit_<student ID>_hw4"
      (ex> "/student/20229999/submit_net/submit_20229999_hw4")
      (do "pwd" to check your directory)
    - ✓ Put your "ChatTCPClient.go" and "ChatTCPServer.go" files in that directory. Do not put any code that does not work! You will get negative points for that.

Grading criteria:

- You get 4 points
    - ✓ if all your programs work correctly, AND if you meet all above requirements, AND
    - ✓ if your code handles all the exceptional cases that might occur, AND
    - ✓ if your code is concise, clear, well-formatted, and good looking.
- Otherwise, partial deduction may apply.
- You may get optional extra credit of up to 2 points if you do the optional extra credit task.
- No delayed submissions are accepted.
- Copying other student's work will result in negative points.
- Code that does not compile or code that does not run will result in negative points.

[Optional] Extra credit task: (up to 2 points)

- Do the same thing as above also in **C**.
    - ✓ That means, your C programs must provide same features as the Golang version.
    - ✓ However, since there is no 'Go routine' in C, you must implement the "**Non-blocking socket method**" using 'select()' function.
    - ✓ Your file names should be same as Golang version except the file extension (.go → .c)
    - ✓ You should provide a Makefile that can compiles both your server and client by just typing '**make**'.
- If you do this, not only your C client should work with C server, but your Golang programs should also work with your C programs as well. That is, you should be able to mix and match C and Golang programs for server and client, in any combination. Do not submit if it does not work. You will get negative points for submitting something that does not work.
- Think about why I ask you to do this: network applications should work with each other independent of language or operating system!