"Network Applications and Design" Homework Assignment #3

"Socket Programming with Multiple Clients" (3+2 points)

Due date: **May 3rd (Tue) 2022, 11:59 PM**.
- Submit softcopy on the class server.

In this homework assignment, you'll implement server-client socket programs using Golang where the server can handle multiple clients at the same time. That is, while the server is running, several clients will communicate with the server in a time-overlapping fashion.

Preliminary:

To help you understand the problem, please consider (as an example) the four 'SimpleEcho/' programs that you've already seen in your previous homework assignment; TCPClient.go, TCPServer.go, UDPClient.go, UDPServer.go.

- Try the following for both UDP version and TCP version:
  - ✓ run the server program
  - ✓ run one instance of the client program (client 1), but do not type the user keyboard input yet
  - ✓ run another instance of the client program (client 2), and try to type the user input
  - ✓ Does it work? *Why? Think!*
- You will notice that the UDP version works, but TCP does not. *Why? Think!*
  - ✓ This is because UDP has no real notion of 'connection', but TCP must 'connect'.
  - ✓ If you look at the code of the TCPServer.go program, it calls 'Accept' and 'Read' in a sequence. If it accepts one client, it will not accept another until that loop is finished.
- Thus, your goal in this assignment would be to make this work for ANY number of clients simultaneously.
  - ✓ Obviously, you won't need to do anything on the client side. All your work is only on the server side.
  - ✓ You might or might not need to do anything for UDP server; I'll leave this up to you to think about.

Then, how can we make the server accept connections from multiple clients while communicating with them without blocking any clients?

In most languages including "C, C++, Java, or Python", there are two ways in doing this; using (1) multi-thread method, and (2) non-blocking socket method. Most languages provide both methods, and the API's are very similar.

① **Multi-thread method**
- Server has a main thread with a server socket that is waiting for client connections.
- When a client connects to the server and server 'accept()'s,
  - server has a new socket for that client (client socket), and
  - server creates a new thread (client thread) for that connection.
    - client socket is given to the client thread
- Then, this client thread in the server will use the client socket to communicate with the client.
- If N client connects, there will be N client threads, N client sockets, along with 1 server socket in the main thread. In this way, threading will automatically take care of multiple clients.

② **Non-blocking socket method**
- No multi-threading. Server has only one thread with a server socket that is waiting for client connections.
- The server also maintains an array of client sockets that are connected.
  - If a new client connection comes in, will put that client socket into this array.
- Server uses the 'select()' function to wait for ALL sockets simultaneously
  - All sockets mean, all client's sockets + the server socket. If N clients, N+1 sockets.
- If any socket triggers an event on that socket, 'select()' will return.
  - check which socket triggered the event and handle the event.
  - The event can be reception of a packet, or disconnect event, or timer event, etc.

However, this is **slightly different in "Go" language (Golang)**.

- It is said that Go is designed for *concurrency*; the designers of the Go language considered concurrency as one of the most important design goals. For this purpose, Go has something called the **"Go routine"**.
- You may think of **"Go routine"** as **light-weight threads**.
  - You can use it with a similar idea and concept as threads, but in a simpler and easier way with less memory usage. They say it is more efficient. API's are a bit different from threads, but easier to use.
- So, you can use **"Go routine"** on the server to support multiple clients simultaneously.
  - You may consider this as the "**Multi-thread method**", but easier and more efficient.
  - It is possible to implement non-blocking socket method in Go language as well, but to do so, you must use more lower-level APIs and thus it is more complicated. → no need for this assignment.

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

You must use the two programs, EasyTCPClient.go and EasyTCPServer.go *that you've submitted for your previous assignment* as the base code for this assignment. In other words, you will be modifying and extending your own previous code for this assignment. Everything that was required in those programs are inherited in this assignment as well. If you haven't done it yet, you'll need to do it now.

Having those programs as your basis, your task is to extend the server program to add multi-client support using "Go routine" (multi-thread-like) method. High-level descriptions of what you need to do is as follows;

- In fact, you do not need to do anything for the client programs.
  - ✓ You can simply use EasyTCPClient.go client program, as is, for testing your servers.
- Implement MultiClientTCPServer.go which supports multiple clients using Go routine.
  - ✓ Hint: google for 'Go routine socket programming example'
- TCP server must also do the following:
  - ✓ Give each client a unique ID such as {client 1, client 2, client 3, …} when they connect.
    - This ID should not change even if a client disconnects. That is, if there were client 1, 2, 3, and if client 2 disconnects, the list of clients should be {client 1, client 3}, and if another client connects, it should be {client 1, client 3, client 4}.
    - Even if client 2 reconnects, that client is in fact client 5.
  - ✓ Whenever a new client connects, or an existing client disconnects, print out the (1) client ID and the (2) number of clients on the server as well as ALL connected clients as;
    - "Client X connected. Number of connected clients = N"
    - "Client Y disconnected. Number of connected clients = M"
- In addition, the server MUST print out the number of clients every 1 minute.
  - ✓ Be careful: this 1 minute should be really every 1 minute (+/-1 sec) regardless of whether there are other event or not (should not be delayed by more than 1 sec).
- Test your programs with more 3 or more clients to check that they work as you desired.

Other additional requirements:

- Your program must run on our class Linux server at nsl2.cau.ac.kr.
- Although you'll be working on your code on our class server (nsl2.cau.ac.kr) for both client and server, 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.
  - ✓ For example, you might run your server program on our class Linux server, and run your client program on a MacOS laptop. This should work without any modification to your code.
  - ✓ 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.
  - ✓ Instead, you should try using nsl5.cau.ac.kr for testing your clients.
    - For example, you may test while having (1) both server and client on nsl2, or (2) server on nsl2 and have client on nsl5. **Both should work without ANY code modification**.
- Make sure that you use your own designated port number for the server socket.

- For the client socket, you should not set the port number which will let the operating system assign a random port number to your socket.
- You should be able to restart any of your programs immediately after exiting/terminating
- 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 MultiClientTCPServer.go server program, TOGETHER WITH the EasyTCPClient.go client program that you have used.
- 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>_hw3"
    (ex> "/student/20229999/submit_net/submit_20229999_hw3")
    (do "pwd" to check your directory)
  - ✓ Put your "MultiClientTCPServer.go" and "EasyTCPClient.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 3 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 point 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/run/work will result in negative points.

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

- Do the same thing as above also in **C**.
  - ✓ That means, your C server program must support multiple clients, and provide same features as explained above for the Golang version.
  - ✓ However, since there is no 'Go routine' in C, you must implement the "**Non-blocking socket method**" using 'select()' function.
  - ✓ Hint: google for 'C socket select() example'
- Your file names should be same as Golang counterpart except the file extension (.go → .c). Your program should compile with gcc without any special option or extra files.
  - ✓ Please create a Makefile so that we can compile your code by just typing '**make**'.
- If you do this, not only your C client should work with C server, but your Python programs should also work with your C programs as well. That is, you should be able to mix and match C and Python 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!