“Network Applications and Design” Homework Assignment #2

“Socket Programming with Header Information” (3+1 points)

Due date: **April 10th (Sun) 2022, 11:59 PM**.
- Submit softcopy on the class server.

In this homework assignment, you’ll implement simple server-client socket programs using ‘Go’ language (Golang). The client will provide a menu of functions that it can support, send the corresponding command and data to the server, and the server will respond accordingly based on the requested command from the client. The key objectives are to get experience in socket programming and using application level headers.

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

Please download four “SimpleEcho/” programs, TCPClient.go, TCPServer.go, UDPClient.go, UDPServer.go. These are simple server-client programs that will serve as your skeleton/template code.

- What they do is very simple;
  - ✓ Server will wait for client connection
  - ✓ Client will take user keyboard input (in string format) and send that to the server.
  - ✓ Server will convert the string into all UPPER-case letters and return it back to the client.
  - ✓ Client will print the returned string and exit.
- As you can imagine from the names, TCPClient.go works with TCPServer.go, and UDPClient.go works with UDPServer.go. You cannot mix and match TCP and UDP. However, application level behavior of TCP version pair and UDP version pair are the same.
- Although you’ll be working on your code on our class server (nsl2.cau.ac.kr) for both client & server, these should work on any computer on the Internet even if client and server programs are on different machines. Specifically, it should be possible to run your **server** program on nsl2.cau.ac.kr and run your client program on another different computer such as nsl5.cau.ac.kr. You should not assume ‘localhost’ for the server.
- Make sure that your server socket uses your own fixed & designated personal port number. This is a MUST.
- For the client socket, you can simply use null (0), or not set the port number at all, which will let the operating system assign a random port number to your client socket.
- Below is an example of how it would work for UDP client and server. TCP version works in the same way.

client	server
	`$ go run UDPServer.go`
`$ go run UDPClient.go`	`Server is ready to receive on port 29999`
`Client is running on port 49642`	
`Input lowercase sentence: hello world!`	
`Reply from server:  HELLO WORLD!`	`Connection request from 165.194.35.202:49642`
`$`	

As you can see, what I’ve given you are very simple echo programs. Your task is to modify and extend these programs to add several new functions and features. Below are the high-level descriptions of what you need to do;

- Client program should show a menu with five options, and take user input for the selection.
  - ✓ option 1) convert text to UPPER-case letters. // a feature that SimpleEcho programs already have.
  - ✓ option 2) ask the server what the IP address and port number of the client is.
  - ✓ option 3) ask the server how many client requests(commands) it has served so far.
  - ✓ option 4) ask the server program how long it has been running for since it started.
  - ✓ option 5) exit client program
- On the client, if the user selects option 1, the client also takes user keyboard input.
- Client then sends the command (and maybe text, together) to the server.
  - ✓ You may need to design your own application-layer packet header for this purpose.

- Server will receive the command and reply with an appropriate response based on the command.
  - ✓ command 1) convert text to UPPER-case letters. // a feature that SimpleEcho programs already have.
  - ✓ command 2) tell the client what the IP address and port number of the client is.
  - ✓ command 3) tell the client how many client requests(commands) it has served so far.
  - ✓ command 4) tell the client how long it (server program) has been running for (unit: seconds).
- Client should print out the returned response;
  - ✓ for option 1, print out the reply text string
  - ✓ for option 2, print format should be "Reply from server: client IP = xx.xx.xx.xx, port = yyyy"
  - ✓ for option 3, print format should be "Reply from server: requests served = zz"
  - ✓ for option 4, print format should be "Reply from server: run time = HH:MM:SS"
- Client should also print out the round-trip response time, the time it took to get the response back.
  - ✓ Print out format should be "RTT = XX.XXX ms"
  - ✓ This time is the duration between when the client sent the command to the server and when the client received the reply from the server. Unit MUST be milliseconds.
    - o Sent time should be measured immediately *before* sending the command, and
    - o Receive time should be measured immediately *after* receiving the reply.
- Client should exit the program gracefully either when option 5 is selected, or user enters 'Ctrl-C'.
  - ✓ What I mean by 'gracefully' is that, when the user enters 'Ctrl-C', the program should not show any error messages. Instead, the program should print out "Bye bye~" and exit.
- Same for the server: Server should print "Bye bye~" and exit when the user enters 'Ctrl-C' on the server.
  - ✓ Note: When you do 'Ctrl-C' on the server program, the client does not know. To be precise, client knows when the connection disconnects for the TCP case, but client has no idea if UDP is used. Thus, if the server terminates, what to do on the client side is up to you.
- You should be able to restart any of your programs immediately after exiting/terminating.
- Unless explicitly terminated by option 5 or 'Ctrl-C', client should not terminate and should repeat the menu.
  - ✓ Same for the server; server should not terminate unless you do 'Ctrl-C' on the server.
- Below is an example of your program might look for the UDP version.

client	server
	$ go run EasyUDPServer.go
$ go run EasyUDPClient.go	The server is ready to receive
The client is running on port 54809	on port 29999
<Menu>	
1) convert text to UPPER-case	
2) get my IP address and port number	
3) get server request count	
4) get server running time	
5) exit	
Input option: 1	
Input sentence: hello world!	Connection request from
	165.194.35.202:54809
Reply from server: HELLO WORLD!	Command 1
RTT = 1.123 ms	
<Menu>	
1) convert text to UPPER-case	
2) get my IP address and port number	
3) get server request count	
4) get server running time	
5) exit	
Input option: 2	
	Connection requested from
Reply from server: client IP = 165.194.35.202, port = 54809	165.194.35.202:54809
RTT = 0.965 ms	Command 2
<Menu>	
1) convert text to UPPER-case	
2) get my IP address and port number	
3) get server request count	
4) get server running time	
5) exit	
Input option: // ← 'Ctrl-C'	
Bye bye~	

You should do all above for both UDP and TCP;

- Create new programs
  - ✓ EasyTCPClient.go that works with EasyTCPServer.go,
  - ✓ EasyUDPClient.go that works with EasyUDPServer.go
  - ✓ File names must be exact. Otherwise, they will not be graded.
- The high-level application behavior should be the same regardless of whether you use UDP or TCP.
  - ✓ The code will almost be the same as well, with one important difference…
- An important high-level difference between UDP version and TCP version is that,
  - ✓ TCP client will 'connect' **only once** at the beginning of the program. After that, even if the menu repeats, it simply uses that same connection. If you connect again, you'll be deducted points.
  - ✓ UDP client does not have a notion of connection, and thus use the same socket to send commands.

Other additional requirements:

- Your programs 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** (by changing only the server address on the client program). In our case, if you run your server on nsl2.cau.ac.kr, your client should run on any computer.
  - ✓ 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.
  - ✓ You may test on nsl5.cau.ac.kr. In fact, I recommend testing two cases:
    - o 1) have both server and client on nsl2, or
    - o 2) have server on nsl2 and have client on nsl5.
    - o Both should work without ANY code modification.
- Again, make sure that you use your own designated port number for the server program. For the client program, do not set the port number so that the OS assigns a random port number to your client 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 "EasyTCPClient.go", "EasyTCPServer.go", "EasyUDPClient.go", and "EasyUDPServer.go" files on nsl2.
- 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>_hw2"
    (ex> "/student/20229999/submit_net/submit_20229999_hw2")
  - ✓ Put your "EasyTCPClient.go", "EasyTCPServer.go", "EasyUDPClient.go", and "EasyUDPServer.go" files and only those in that directory. Do not put any code that does not work!

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 possible exceptional cases that might occur.
- Otherwise, partial deduction may apply.
- You may get optional extra credit of up to 1 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 or code that does not run will result in negative points.

[Optional] Extra credit task: (up to 1 point)

- Do the same thing as above also in **Java**.
  - ✓ Your file names should be EasyTCPClient.java, EasyTCPServer.java, EasyUDPClient.java, and EasyUDPServer.java.
  - ✓ Your program should compile with javac without any -cp config.
  - ✓ Please submit a 'Makefile' together with your Java code so that we can compile using 'make'.
- If you do this, not only your Java client and server should work with each other, but your Golang programs should also work with your Java programs. That is, you should be able to mix and match Java and Golang programs for server and client, in any combination. Do not submit if this does not work.

Note on running your socket programs on CAU campus:

- Above I wrote, "Ideally, your server-client program should work on any machine connected to the Internet. For example, it should be possible to run your server program on our class server (nsl2.cau.ac.kr), and your client program on your own computer at home".
  - ✓ Yes, I still mean this. You should be able run your server program on any computer on the Internet, and your client program on any (other) computer on the Internet.
- 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. For this reason, don't panic even if this is the case. That may just be due to port blocking by CAU due to security reasons. (The reverse might work, and you may try if you're interested)