CN CSL317 Lab Assignment

Client Server Socket Programming

(One Client Program and four Server Program)

In this assignment, you will implement four types of client-server systems. The client will chat with a custom “calculator” server. The protocol between the client and server is as follows:

- Initially, the server program is started on a custom port (server IP and port number is provided on the command line). If the port is already occupied, then the server should throw and display the error.
- Next, the client program is started (server IP and port number is provided on the command line).
- The client then connects to the server, and asks the user for input. The user enters a simple arithmetic expression string (e.g., "9 + 8", "4 / 2", "4 * 7"). The user's input is sent to the server via the connected socket.
- The server reads the user's input from the client socket, evaluates the expression, and sends the result back to the client. If the client has sent an invalid input, the server should reply back accordingly.
- The client should display the server's reply to the user, and prompt the user for the next input, until the user terminates the client program with *Ctrl + c*.

Note:

- The sockets must use TCP and IPV4 protocols.
- All the messages of connection details and information passed should be displayed on the command line with respect to servers and clients.
- Marks reserved for indentation and documentation of the code.

You will write **one Client program** and **four versions of the server**:

- **server1.py** : Your server program "server1.py " will be a single process server that can handle only one client at a time. If a second client tries to chat with the server while some other client's session is already in progress, the second client's socket operations should see an error. After the first client closes the connection, the server should then accept connections from the other client.

- **server2.py** : Your server program "server2.py " will be a multi-threaded server that will create a new thread for every new client request it receives. Multiple clients should be able to simultaneously chat with the server.
- **server3.py** : Your server program "server3.py " will be a single process server that uses the "*select*" method to handle multiple clients concurrently.
- **server4.py** : Your server program "server4.py" will be an echo server (that replies the same message to the client that was received from the same client); it will be a single process server that uses the "*select*" method to handle multiple clients concurrently.

Note that the actual test cases we will use may be different from the ones shown above: your servers should correctly work with any numbers, not just the ones shown above, as long as they conform to this format. Handling non-integer operands, other arithmetic operations, or operations with more than 2 operands (e.g., "1 + 2 + 3") is purely optional.

Please write your code for this assignment in Python.

The servers

Part1: Server handling single connection at a time

First, you will write a simple server in a file called "server1.py". The server1 program should take the ip address and port number from the command line, and start a listening socket on that command line. Whenever a client request arrives on that socket, the server should accept the connection, read the client's request, and return the result. After replying to one message, the server should then wait to read the next message from the client, as long as the client wishes to chat. Once the client is terminated (socket read fails), the server should go back to waiting for another client. The server should terminate on Ctrl+C.

Your simple server1 should NOT handle multiple clients concurrently (only one after the other). That is, when server1 is engaged with a client, another client that tries to chat with the same server must see an error message. However, the second client should succeed if the first client has terminated.

Part2: Server handling multiple connections simultaneously

Note: For parts 2, 3 and 4 below, your server should behave like any real server. It should be able to handle several clients concurrently. It should work fine as clients come and go. Your server should always keep running (until terminated with Ctrl+C), and should not quit for any other reason. If you are in doubt about any functionality of the server, think of what a real server would do, and use that as a guide for your implementation.

Python Code *Sample* output example

For this example, the client code is first compiled. Then a server is run on port 5000 in another terminal. The client program is then given the server IP (localhost 127.0.0.1 in this case) and port (5000) as command line inputs.

```
$ python client.py 127.0.0.1 5000

Connected to server

Please enter the message to the server: 22 + 44

Server replied: 66

Please enter the message to the server: 3 * 4

Server replied: 12

...

...
```

In parallel, here is how the output at the server looks like this (you may choose to print more or less debug information). Note that the server's output shown below is only for illustration; the coder is free to choose any format of display.

```
$ python server1.py 127.0.0.1 5000
Connected with client socket number 4
Client socket 4 sent message: 22 + 44
Sending reply: 66
Client socket 4 sent message: 3 * 4
Sending reply: 12
...
...
```

Submission instructions:

Submit a *<roll no>_<name>.zip* file which consists of following 7 files:

- 4 server files: server1.py, ... server4.py
- 1 client file: client.py
- 1 pdf file (*<roll no>_report.pdf*) which will consist of:
  - instructions that *clearly* describe how to run your codes
  - screenshots of the outputs that were obtained when the test cases were applied
  - any additional or special features that you might have implemented/incorporated.
- 5 -10 minutes (approx.) video file clearly explaining the codes and demonstrating the various test cases that you have tried.