# Socket programming Questions
# (Note: you can submit these questions in any language you prefer)

1. Using UDP sockets in Python (files UDP_Server.py and UDP_Client.py). Make sure that you understand the purpose of UDP sockets, and the steps for creating and using these sockets. Run the Client and Server processes and observe the output.

2. Run the Client and the Server processes on different computers and check if they work as expected (you will need to specify the server's IP address).

3. Design and implement a Client-Server system that uses UDP sockets to do the following:
    a. The client sends the server a request. The request string can either be: "SEND_DATE" or "SEND_TIME".
    b. The server runs in a infinite loop where it keeps waiting for requests. Whenever it sees a request, it responds by sending either the current DATE or the current TIME in (HH:MM:SS) format as specified in the request.
    c. When the Client receives a response, it prints it.
    d. The Client runs in a loop where it generates multiple such requests, and the time between successive requests varies randomly between 1-2 seconds. HINT: You can use the following line of code to generate a random amount of delay, uniformly distributed between 1-2 seconds:

```
import time, random
...
time.sleep(random.uniform(1,2))
```

4. Using TCP sockets in Python (files TCP_Server.py and TCP_Client.py). Make sure that you understand the purpose of TCP sockets and the steps for creating and using these sockets. Observe the differences between UDP and TCP sockets and the steps for their use. Run the TCP Client and Server processes and observe the output.

5. Start the Wireshark and apply a filter such that only the traffic generated by your Client and Server processes is displayed. Identify the messages used by TCP during the Handshake and the actual text sent by the two processes. Are the "contents" of the packet (the message strings) visible within Wireshark? (This is what we'd expect since the strings aren't encrypted before sending.)

6. Design and implement a Client-Server system that uses TCP sockets to do the following:
    a. The client initiates communication with a Server by sending the server it's name. (Choose some name for your client process). The Server

remembers this name for the entire duration of the communication session.

b. The client then runs in an infinite loop where it accepts a line of input from the user. The user is expected to enter a string consisting of two numbers and a simple arithmetic operation (separated by spaces), for example: “12 + 42” or “3.24 45” or “4.5 / -6” . If the input is not correctly formatted, a warning is displayed to the user. If correctly formatted, the Client sends this string to the Server.
c. The Server runs an infinite loop where it keeps waiting for requests from this client. Upon receiving a request, the server prints the received message, computes the answer by performing the arithmetic operation and sends it back as a string. The Client prints the answer it received from the server.
d. When the user wishes to stop, they enter “q”. The client process forwards the “q” to the Server upon which the server ends the communication session and prints “Session ended”. The Client processes stops.

7. When you decide upon a “format” for the messages that can be correctly understood by the Client/Server processes, you have implicitly designed a “Protocol”. A protocol can be “Stateful” of “Stateless”. Find out and understand what these terms mean. (Ref: https://en.wikipedia.org/wiki/Stateless_protocol)
    a. Is the application-layer protocol designed by you for question 3 Stateless?
    b. Is the application-layer protocol designed by you for question 6 Stateless?
    c. Is TCP a Stateless protocol?
    d. Is UDP a Stateless protocol?
8. For systems such as those described in question 6, think of how you could modify the Server so that it can handle multiple clients at the same time. For each connection (using TCP sockets), the Server should provide the same service as described. There are several ways to implement this. One way is to use mutiple threads for the Server, one per connection. A skeleton for a multi-threaded server program can be found in this answer on StackOverflow: https://stackoverflow.com/a/40351010/1329325 In this question, we wish to design such a multi-Client system to implement a **“Chat Room”** as follows:
    a. The Client process acts like a **chat window**. It takes user input, sends appropriate requests to the Server, and displays the messages sent by the server to the human user.
    b. The Server process acts like a **chat room manager**. It allows client processes to login to the chat room (each client needs to have a unique name). The server keeps track of all the clients that are currently logged in. Whenever any interesting event happens, (such as new user logging in or leaving the chat room) the status is broadcast to all connected clients. Also, whatever each user types is broadcast to all clients.
    c. At the beginning, the user is requested for a “login name”. The client process then sends a login request to the chat room (Server) with this name.
    d. After logging in, whatever lines the user types is broadcast to all clients along with the sender’s name. The following lines show an example of

output that might be displayed to two different clients. Client 1 is the first to join the chat room.

```
Client 1

Enter login name:
> Batman

Server: time=10.01 Batman has joined. Member count=1
Server: time=11.01 Voldemort has joined. Member count=2
> Anyone here?
Voldemort: Crucio!
> Aaaaaaahrrhhhhhh..
> quit
```

```
Client 2

Enter login name:
> Voldemort

Server: time=11.01 Voldemort has joined. Member count=2
Batman: Anyone here?
> Crucio!
Batman: Aaaaaaahrrhhhhhh..
Server: time=13.01 Batman has left. Member count=1
```