CS39006: Networks Lab
Lab Test 1
February 10, 2023
Time: 2 Hours

1. Write a TCP concurrent server (using fork(), not threads) that echoes back to the client whatever it receives from the client (so makes a recv() call and immediately send back to client whatever is received with a send() call), running perpetually until the client closes the connection, at which time the server process handling the client also exits. (start from the socket() call onwards, assume all variables you use are already defined). The server should close a connection from a client immediately after the connection is established (before even fork()'ing) if the client's IP address starts with anything other than 144.25. (for example, the connection should be dropped if the connection is from 145.30.20.2 or from 144.26.30.5 or from 210.200.34.5 etc.). (10)

   2 marks for checking for 144.25 (0 if you have directly compared client address returned with a string)
   2 marks for sending back whatever is received perpetually till close (1 mark for recv/send, 1 mark for exiting on connection close).
   6 marks for the rest

2. Consider a TCP iterative server that works as follows. Whenever it receives a connection request from a client, it accepts the request and stores the id of the new socket created in a free entry in an array S[ ]. S can store a maximum of 50 ids, so the server can be connected to at most 50 clients at a time. When a client disconnects, the server replaces that location in S with -1, so the socket ids stored in the array S need not be contiguous, there may be valid entries with zero or more -1 in between them.

   Write **only** the part of the server code that does the following: wait for receiving data on any of the connected clients, or a new connection. If a new connection comes, it puts the sockfd in a free entry in S. If any data comes to any of the clients, it makes a recv() call and prints the number of bytes received from that client in that call, and then closes the connection to that client; this should be done for all clients for whom any data has been received. It then goes back to wait for a new connection or data to come on some clients again. You can assume not more than 50 clients will be connected to the server at any point in time.

   Note that you do not have to write the code for creating sockets, binding etc. Assume all variables you use are already defined, you do not need to define them. Assume that all entries in S are initialized to -1. (10)

   3 marks for managing S[ ] array as connections are established and closed.
   *7 marks for the rest. You must use poll() to wait **together** for the socket receiving the connections and the sockets receiving data, so that any one enabled can be handled immediately or you got mostly 0. You cannot assume that once a connection is established with the client, it will send data immediately.*

3. Suppose we want to build a client server system in which a client wants to transfer a small file from a server. The steps to transfer the file are as follows. A file is seen as a number of chunks, each chunk (except possibly the last one) of size K bytes. Both client and server know K. The client sends the filename to the server (assume that the file always exists). The server first sends the total number of chunks in the file in a special message (assume that it can be identified as a special message). The client asks for this message again if it is not received within a timeout. The client then asks for the first chunk. The server sends the first chunk along with the chunk number. If the client receives the first chunk successfully within a timeout, it then asks for the second chunk, otherwise it asks for the first chunk again after a timeout. This continues until all chunks are received. Note that since there are chunk numbers in the chunks sent, the client can distinguish between different chunks or duplicate chunks.

   Since the transfer is chunk by chunk, suppose we decide to use UDP. Also, since many clients can try to transfer files, it is decided to use a concurrent UDP server to handle each client by a separate process. Do you feel this is a good design? Please be brief and to the point. Do not suggest any alternate design, just comment on why you feel it is good or why you feel it is not good. (5)

   In a UDP concurrent server, all processes created are waiting on the same <IP address, port>. So a chunk sent by one client can be received by the process handling another client. Handling this properly makes UDP concurrent servers very complex (other than the fact that for this example, overhead will be too much), so you will hardly see any concurrent UDP server. It is a bad design..
   3 to 5 marks if you got the above idea at least somewhat, 0 otherwise if you have said good or bad based on many other things but missed the fact that all are waiting on the same<IP, port> and things can get mixed up.

4. Consider a TCP iterative server that receives one character from a client, prints the character, and then closes the connection. For whatever reason, we want the server to behave like this: It should not block on the accept() call when waiting for client connections; if there is no client trying to connect to the server should go sleep for 2 seconds and come back and make the accept call. However, if a client does connect, the server wants to wait on the recv() call and block there until the character is received. Write the part of the code that will make this happen. You can assume that the socket is already created and bound to the proper address, and all variables you use are already declared. (5)

   2 marks given if you wait on poll before the accept call (waiting on poll() first has the same effect as waiting on accept(), no arguments please)
   2-3 marks given (depending on correctness) if you have used accept4, question clearly asks for accept() call.
   0 or 1 given (depending on what is written) if you have not used fcntl() to unblock accept().

If you used fcntl(), 2 marks for correct use of fcntl, 1 mark for checking errno, 1 mark for sleeping, 1 mark for recv.
Note that there was nothing special needed for the recv, as newsockfd does not inherit the non-blocking property of the original sockfd in Linux, but ok if you have made it blocking again.

5. (a) What will happen if a connect() call is made in an UDP client (with server address in proper format as in TCP clients)? (3)

   0 if you have written error or connection will be made (in any form).

   (b) Suppose a TCP server sends a sequence of null-terminated strings, each of size exactly 12 bytes (including the \0). The client has exactly one 12 bytes sized buffer, in which it should read one string, print it, then read the next one, print it and so on until all string are received. Also, the server cannot send the number of strings to the client. Can you write the part of client code for receiving and printing all the strings? Assume that the socket is already created and connected to the server. (7)

   Maximum 2 out of 7 given if you assumed that all 12 bytes will come together irrespective of what else you wrote, this is the main thing. 1 mark deducted if you have not clearly specified your assumption as to how you detect end of all strings, even if your code has it.