CS 39006: Lab Test 1

Date: February 28, 2022, Time: 2-15 pm to 4-00 pm

Problem Statement:

You have to develop a multicast chat application using stream sockets, where there will be multiple chat clients connected to a server. Whenever a client sends some message, the server will forward the message to all other connected clients and the message will be printed on the screen by all other clients. For simplicity, you will have a maximum of 3 clients.

IMPORTANT: Please write your name and roll no. in a comment as the beginning of every file you write.

Server Functionality:

The chat server will be an iterative TCP server. The server will wait for new clients to connect, or receive message from connected clients to be forwarded to other connected clients. You can assume that there can be maximum three clients connected to the server and a connected client never disconnects (i.e, closes its socket) after connection. Each client will be identified by its <IP, port>.

The server maintains an array of socket file descriptors called `clientsockfd[]` of size 3 (each element is initialized to -1), array of clients `clientId[]` of size 3, and an integer `numclient` (initialized to 0). The integer `numclient` indicates how many clients are currently connected to the server. `clientsockfd[i]` contains the socket id to communicate with the `i`-th connected client and `clientId[i]` contains the <IP, port> of the corresponding connected client (if `clientsockfd[i]` $\neq -1$).

The server runs in an infinite loop. In each iteration, it waits over a `select()` call for new incoming client connections or messages from already connected clients. The `select()` call can exit either because a new connection request from a new client has arrived, or some connected client has sent a message to the server. The action taken for each of these cases is as follows:

1. If the server receives a new connection (this should be checked **first** after coming out of the `select()` call), it adds the socket id from the return value of the `accept()` call to `clientsockfd[]`, updates the corresponding entry of `clientId[]` with the <client IP, client port> of the new client (returned in the accept() call) , and increments `numclient` by 1. The server also prints the following message at the console.

   Server: Received a new connection from client <client IP: client Port>

   Note that every client will be uniquely identified through the <IP, Port> pair.
2. If the server receives a message from any of the connected clients, it simply sends the same message to all other connected clients except the one from where it has received the message, in the following format: *the client IP (4 bytes) and the port number (2*

bytes) of the client sending the original message, followed by the actual message (null-terminated string). The server also prints the following message at the console.

Server: Received message "<message>" from client <client IP: client port>
Server: Sent message "<message>" from client <from_client IP: from_client port> to <to_client IP: to_client port>
Server: Sent message "<message>" from client <from_client IP: from_client port> to <to_client IP: to_client port>

where <message> is the actual message sent by the client. The above message will be printed for all the clients to whom the message has been sent. Note that if `numclient` is 1, there is no other client to send the message to. In that case, it will print the following message:

Server: Insufficient clients, "<message>" from client <client IP: client port> dropped

`So there is no buffering of old messages; a client only gets messages that are sent by some other client after it is connected.`

3. The server then resets select() read file descriptors properly again to receive new connections or receive messages from connected clients, and goes to the next iteration to wait on the select() call.

Client Functionality:

You have to implement the client socket as a ***nonblocking TCP socket*** using **`fcntl()`** call. The client maintains a counter `N` (initialized to 1). Once you start a chat client, the client will first create a connection to the server. After that the client will run in an infinite loop. In each iteration, it will execute the following tasks **exactly in this sequence**:

1. Generate a random number `T` between 1 and 3 (both included) and sleep for `T` seconds (see the *rand()* function if by chance you do not know how to generate a random number)**.**
2. After waking up from sleep, if `N < 6,` send a message **"**`Message <N>`**"** `where <N> is the value of N,` to the server. The client then prints the following text to the console.

   Message N sent

   It then increments the value of `N` by 1.
3. The client then checks whether there is any incoming message from the server. If there is an incoming message, the client reads the message and prints the following at the console.

   Client: Received <message> from <Client IP: Client Port>
   Client: Received <message> from <Client IP: Client Port>
   Client: Received <message> from <Client IP: Client Port>

   Note that here the *<Client_IP: Client Port>* is the *IP:Port* of the client that originally sent the message to the server. The client prints all the messages that it has received from different clients in separate lines using the format as shown above. **Note that this receive is non-blocking.**

Note that each client will send five messages. However, the client will never close its socket, and continue to loop forever executing the above sequence of operations.

Also, write as a comment at the end of the client file how would you change the client code if the client did not use non-blocking receive AND read the messages to be sent from the keyboard (to be entered by the user) instead of generating fixed messages after random sleep.

Submission Instruction:

You have to submit the following two files: <your roll no>`_server.c` containing the server code and <your roll no>`_client.c` containing the client code. For example, if your roll number is 19CS3000, then the files should be named 19CS3000_server.c and 19CS3000_client.c. Upload both the files through the assignment submission link in the Moodle submission page (The assignment will accept up to 2 files).

You have to follow the submission instructions exactly, otherwise a 5% penalty will be imposed.