CS 39006: Lab Test 1- Set A

Date: February 11, 2025

Important Note:

You have to follow the instructions and variable names given in this problem statement. Anything which is not given can be assumed; however, you should clearly write your assumptions at the beginning of the code.

Problem Statement:

You have to develop a multicast chat application using stream sockets, where there will be multiple (two or more than that) chat clients connected to a server. Whenever a client sends some message, the message will be received by all other clients.

Server Functionality: To do this, you have to implement a chat server which will initially wait for the clients over a stream socket. It will wait for at least two chat clients before the clients are ready to send/receive messages. Consider that there will be *a maximum of five clients*. The chat server maintains an array of socket file descriptors called `clientsockfd[]`, and waits over a select call for the incoming connections or messages. It can be noted that the first two instances when `select()` will exit are the two connection requests, as you need at least two chat clients for the communication to begin with. After that, the `select()` can exit either because a new connection request from a new client has arrived, or some client has sent a message.

If the server receives a new connection, it adds the connection to `clientsockfd[]`, increments a counter `numclient`, and include the new client socket file descriptor `clientsockfd[numclient-1]` to the `select()` read file descriptors. 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.

If the server receives a message from one of the clients, it simply sends the same message to all other clients except the one from where it has received the message, appending the client IP and the port number from where the message has been received. The server prints the following message at the console.

```
Server: Received message "<message>" from client <client IP: client port>
Server: Send message "<message>" from client <from_client IP: from_client port> to <to_client IP: to_client port>
Server: Send message "<message>" from client <from_client IP: from_client port> to <to_client IP: to_client port>
...
```

The above message will be printed for all the clients to which the message has been sent. If the server receives a message before at least two clients have joined, it will print the following message.

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

Client Functionality: The client uses a stream socket to connect to the server. After the connection is established, it uses a `select()` call within an infinite loop to check one of the following.

1. The user has given a message input through the keyboard. Note that the `STDIN` file descriptor is a standard POSIX file descriptor that can be checked to read data from the keyboard. You need to add `STDIN` (`STDIN_FILENO` macro defined in `unistd.h`) to `FD_SET` as a read file descriptor, and then use the `select()` call to wait on it. After the `select()` returns, you need to check whether something is available on `STDIN`, and if so, you can use a `read()` call to read the data from the keyboard. If the user inputs some message, then the client socket sends the message to the server and prints the following at the terminal.
   `Client <Client IP: Client Port> Message <Input Message Text> sent to server`
2. The client socket has received a message from the server. Note that in this case, the server has actually relayed the message sent by another client. If a message is received, the client prints the following at the terminal.
   `Client: Received Message <Message Text> from <original_client_IP:Port>`
   Note that here the `original_client_IP:Port` is the `IP:Port` of the client that actually sent the message to the server.

NB: Assume that the clients remain connected once they join the network. You do not need to handle connection closure. Also you may use the function `int getpeername(int socket, struct sockaddr *restrict address, socklen_t *restrict address_len)` to get the address for a client associated with a socket FD (sockfd returned from the `accept()` call). For details, check the man page.

Submission Instruction:

You have to submit the following two files: `chatserver.c` containing the server code and `chatclient.c` containing the client code. Put the files in a folder named `LT1_<Your Roll Number>` (for example, if your roll number is 22CS90098, then the folder name will be `LT1_22CS90098`. Compress the folder in a `zip` format and upload it on the MS Teams submission page.

You have to follow the submission instructions exactly, otherwise a 20% penalty will be imposed.