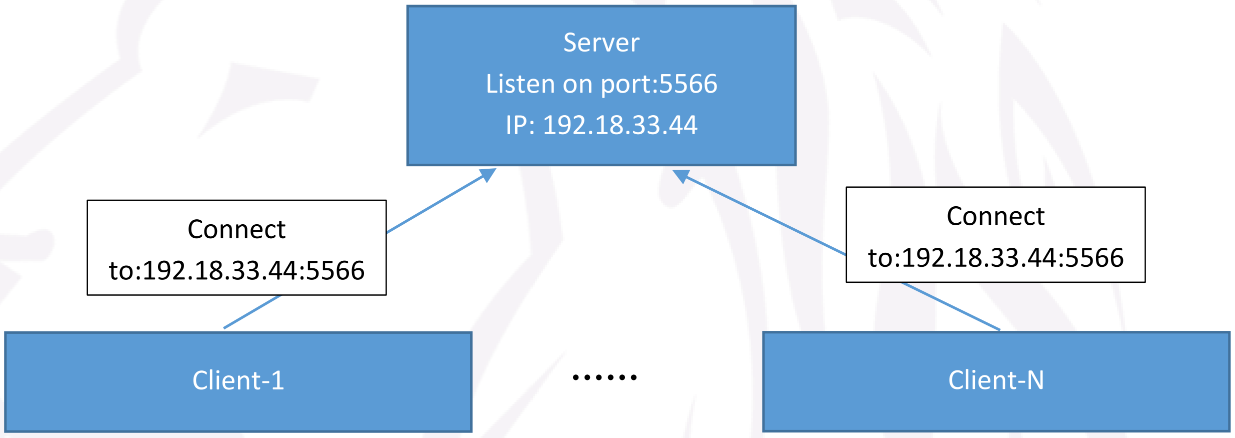
# CS-232: Operating Systems

Assignment 04 (20% of your total grade).

Duration: 2 Weeks (Please submit it by Wednesday 4th of December, 2019, before 1155 pm)

This assignment is a practice in socket programming, threads, and synchronization. You are supposed to implement a rudimentary chatting application. The application will run on a client/server model which means that you'll build two different programs: a server and a client. For simplicity we'd say that one instance of the server will be running and multiple clients can connect to the server and chat to each other using this server program as an intermediary. The server and clients may run on the same PC or on different ones.



## TO DO:

- You basically run the server passing it a port number:
  **./server 5566**
  The server program creates a socket and starts listening on the port passed to it at command line.

- You run a client passing it the server programs computer IP, the port it's listening on, and a name for your client:
  **./client 192.18.33.44 5566 client1**
  The client should make a socket and try to make a connection to server program using the server's PC's IP and the port number the server is listening on. Once the connection is established, the client should send its identifier ("client1" in this case) to the server. In case there is no server running, the client should terminate with an appropriate message.

- The server program should accept connections from multiple clients. For each connection established the server should store the client's identifier and keep a record of which port is being used for communication with this client.

- The server and client programs can be on the same PC or different ones.

- In such applications, usually, a sever creates a separate thread for each connection with the client.

- The server would maintain a linked-list of all the clients connected to it and each node of the linked list would store the information about a client, i.e., it's name, it's file descriptor, etc, any other thing that you find useful. This linked-list should be accessible via multiple threads in a thread-safe manner.

- When the server creates a thread for a client, this client's node should be added to the linked list.

- When a client exits, its node should be removed from the linked list.

- If a new connection request comes from a a client with a client identifier which is already in use, the server should inform the client with an appropriate error message and close that connection.

- If you make the client multithreaded as well, you will be able to receive messages from other clients, even when your own client program is blocked waiting for an input from user.

- The client can send "commands" to the server. Each command starts with a '/' character. When the server program receives a command from a client, it should take the appropriate action and send a response to the client. Few commands:

  **/list** - if the client sends a "/list" command to the server, it means the client is asking for a list of all the connected clients. Upon receiving this command, the server should put the names of all the clients connected to itself in string and send it back to the client. The client shall receive this string and display all these names to the user.

  **/msg** – users can type this command to send messages to other clients. The general syntax is "/msg clientname message". i.e. if the user at client-1 types "/msg client-2 hi there", the server should receive this message from client-1 and send it to client-2. The threaded the server created for client-1 should access the linked-list, search for the node containing client-2 info, and send the message to client-2 by writing to its descriptor. The user at client-2 should receive the message "hi there" along with the information that it came from client-1. You should do proper error handling for this command at server and client end i.e. missing destination (client) identifier or incorrect destination identifier should generate appropriate response.

  **/quit** – users can type this command to disconnect from the server and quit the client. Upon reception of this command from a client, that particular server thread should close the connection with that client, remove it's data from the linked-list, free any resources, and terminate itself.

You'll need socket programming, multi-threading, dynamic memory allocation/de-allocation, and synchronization for this assignment.

Some resources can be found on in the syllabus, the announcement for lab13, etc. The chapters on threading and socket programming in the book (Advanced programing in unix environment) APUE 3rd edition should help as wel. For the rest google is your friend.

You'll also need string manipulation functions in C: https://en.wikibooks.org/wiki/C_Programming/String_manipulation

IMPORTANT: You've got two weekends and the deadline is close to end of semester so extensions would be difficult this time!!

Do it incrementally i.e., first try to have a single-threaded server communicate with a single-threaded client, and then build on it.

**Start working on it. I'll be giving you the rubric next week.**

**Submission guidelines:**

1. You are to work in groups of two for this assignment. You should declare at start who would submit the assignment. One or both members can be called for viva arbitrarily. Both should be knowing working of all the code even if they've divided the work b/w themselves.
2. You'll mention any and all help (people, friends, books, weblinks, etc) you've received in doing this assignment in the comments section of the PDF mentioned below.
3. You should submit a single tar-zipped file whose name should be your ID.
4. When extracted it should extract to a single directory whose name should be your ID.
5. In that directory there should be only files for the source code of each part and a makefile to build them as well as a PDF containing the problem description, all the codes, and any comments you might have.
6. Write clean code, comment it, follow good coding practices like, giving sensible names to variables, de-allocate all resources after their use, etc.

**Make sure that you DO NOT create any memory leaks, dangling pointers, zombie threads, race conditions, unfree'd resources, etc, when the program terminates.**