EE450 Socket Programming Project, Spring 2025
Due Date: Sunday, April 27th, 2025, 11:59 PM (Midnight)
(The deadline is the same for all on-campus and DEN off-campus students)
Hard Deadline (Strictly enforced)

OBJECTIVE:

The objective of this assignment is to familiarize you with UNIX socket programming. **It is an individual assignment, and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).**
If you have any doubts/questions, post your questions on Piazza. You must discuss all project-related issues on Piazza. We will give extra points to those who actively help others out by answering questions on Piazza.

PROBLEM STATEMENT:

Stock trading, the practice of buying and selling shares of publicly traded companies, is a cornerstone of the global financial system. A reliable and efficient trading system is an important part of the process. We are designing a simplified program to simulate a stock trading system. This system will feature basic user authentication, record trading information, and user positions, and provide real-time stock quotes to users. For simplicity, this system assumes that:

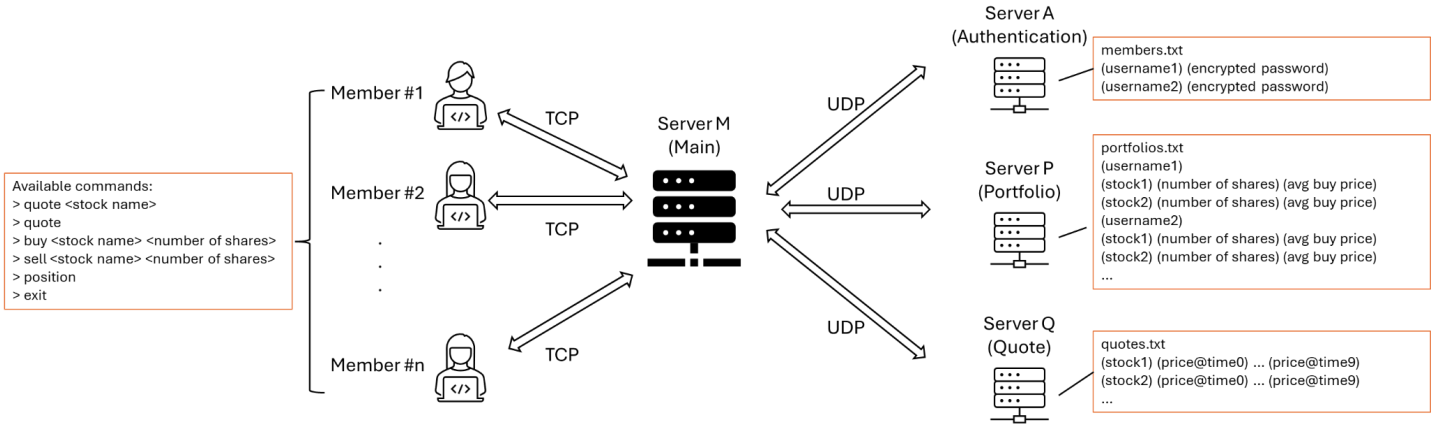
1. Users can only sell the quantity of stocks they currently own, ensuring they cannot sell stocks they do not have.
2. Users have unlimited buying power (cash) to purchase any stock.
3. An unlimited supply of stocks is available in the market for users to buy at any time.
4. All transactions occur at the market price.

Given the critical importance of user security, the system will encrypt passwords at client side during login, ensuring a secure, reliable, and user-friendly trading environment.

In this project, we need three backend servers: Server A, Server P, and Server Q, one main server: Server M, and client interfaces. Users will be able to perform various operations through the client interface, and these operations will be forwarded to the corresponding backend servers via the main server. Here are the functional descriptions of each server and client.

- Client: This is the only interface for user interaction. Through the client terminal, every member can perform actions listed below:

    - log in
    - view stock quotes
    - buy and sell stocks
    - check their own positions
    - close the connection (log out)
- Server M (Main): Handles all member actions by dispatching requests to the appropriate backend servers. Additionally, all communications between the three backend servers must go through Server M.
- Server A (Authentication): Verifies users' usernames and encrypted passwords and returns whether they are valid. It uses "members.txt" as its database.
- Server P (Portfolio): Records users' holdings and their current total profit and loss. It uses "portfolios.txt" as its database.
- Server Q (Quote): Provides the current price of each stock. It uses "quotes.txt" as its database.



Source Code Files

Your implementation should include the source code files described below, for each component of the system:

- **Client:** The name of this piece of code must be `**client.c**` or `**client.cpp**` (all in lowercase). The header file (if you have one; it is not mandatory) must be called `**client.h**` (all in lowercase).

- **serverM (Main Server)**: You must name your code file: `**serverM.c**` or `**serverM.cpp**` (all lowercase except for the `M`). If you have a header file, it must be named `**serverM.h**` (all lowercase except for the `M`).

- **Backend Servers A, P, Q:** You are required to create three distinct files, choosing from the following naming conventions: server#.c or server#.cpp. The filename must utilize one of these formats, substituting "#" with the specific server identifier (either "A" or "P" or "Q") to reflect the server it represents, resulting in filenames like

serverA.c, serverA.cpp, serverP.c, or serverP.cpp (note that the name should be entirely in lowercase except for the letter replacing "#"). If available, you should also include a corresponding header file named server#.h, adhering to the same naming rule for the "#" replacement. This ensures a clear, organized naming structure for your code and its associated header file, if any.
Note: You are not allowed to use one executable for all four servers (i.e. a "fork" based implementation).

- **Optional files:** You may also include additional files that you write yourself with common functions, using .c, .cpp, or .h extensions.

Input Files:

- members.txt: Located in Server A (Authentication server) which maintains the user credentials. The format of this file consists of two columns, usernames and encrypted passwords. A space separates these columns. Each user's data is stored on a separate, new line. This is used for authentication purposes.
- portfolios.txt: Structured to list members and their stock portfolios. Each member is identified by a line with their name (e.g., username1), followed by multiple lines representing the stocks they own. Each stock line contains the stock name, the number of shares owned (as an integer), and the average buy price (as a decimal or floating-point number), separated by spaces. This pattern repeats for every member, with no blank lines between entries.
- quotes.txt: It contains ten prices for each stock, representing different points in time as the system progresses. Whenever any client (member) retrieves a stock quote from Server Q using the "buy," or "sell" command, the system advances **only the price of the stock involved in the "buy" or "sell" order** to the next price in the sequence **after each** "buy" or "sell" (**even if the buy or sell fails or is denied by the user**). If the end of the price list is reached, it loops back to the beginning.

  For example, consider the following line in quotes.txt:
  S1 100.0 101.0 102.0 103.0 104.0 105.0 106.0 107.0 108.0 109.0
  - ❖ At time 0 - the program starts, the price for stock S1 is $100.0.
  - ❖ At time 1 - after the 1st buy/sell command, the price of stock S1 is $101.0, but the first buy/sell was executed at $100.0.
  - ❖ At time 9 - after the 9th buy/sell command, the price is $109.0.
  - ❖ At time 10 - after the 10th buy/sell command, the sequence restarts, and the price returns to $100.0.
  - ❖ At time 11 - after the 11th buy/sell command, the price is $101.0.

Note: The length of a stock name is no greater than 5.

DETAILED EXPLANATION

Phase 1: Bootup

Please refer to the following order to start your programs: server M, server A, server P, server Q, and then multiple Clients. Your programs must start in this order. Each of the servers and the clients have boot-up messages that must be printed on the screen. Please refer to the on-screen messages section for further information.

When three backend servers (server A, server P, and server Q) are up and running, each backend server should read the corresponding input file (members.txt, portfolios.txt, and quotes.txt) and store the information in a certain data structure. You can choose any data structure that accommodates your needs. The communication between the backend servers and the main server (server M) should be via UDP over the port mentioned in the PORT NUMBER ALLOCATION section. In the following phases, you have to make sure that the correct backend server is being contacted by the main server for corresponding requests. You should print correct on-screen messages onto the screen for the main server and the backend servers, indicating the success of these operations as described in the “ON-SCREEN MESSAGES" section.

After the servers are booted up and the required pieces of information are stored on the backend servers, at least TWO clients will be started over TCP connections. Once the clients boot up and the initial boot-up messages are printed, the clients wait for the system to check the authentication and then enter the stock trading system.

Please check on-screen messages for the on-screen messages of different events on each server and client side.

Phase 2: Login and confirmation

In this phase, the clients will be asked to enter their username and unencrypted password into the terminal. A member can be authenticated by inputting the member’s username and unencrypted password. The client will send this information to the main server. The main server will then encrypt the password. Then, the main server would send the encrypted credentials (only the password would be encrypted) to serverA for authentication.

The encryption scheme for member authentication would be as follows:

- Offset each character and/or digit by 3.
    - character: cyclically alphabetic (A-Z, a-z) update for overflow
    - digit: cyclically 0-9 update for overflow
- The scheme is case-sensitive.
- Special characters (including spaces and/or the decimal point) will not be encrypted or changed.

A few examples of encryption are given below:

Example	Original Text	Encrypted Text
#1	Welcome to EE450!	Zhofrph wr HH783!
#2	199xyz@$	422abc@$
#3	0.27#&	3.50#&

Constraints:

- The username is case-insensitive and will be converted to lowercase (1~50 chars).
- The password will be case-sensitive (1~50 chars).

Phase 2A:

A member client sends the authentication request to the main server over a TCP connection. Upon running the client using the following command, the user will be prompted to enter the username and password. This unencrypted information will be sent to the main server over TCP. The main server will then encrypt the password.

```
./client
(Please refer to the on-screen messages)
[Client] Logging in.
Please enter the username: <username>
Please enter the password: <unencrypted_password>
```

Phase 2B:

ServerA receives the username and encrypted password from the client via the main server. ServerA compares the username and encrypted password with the data in its database (BootUp phase) and sends the result of the authentication to the main server. Then the main server sends the result of the authentication request to the client over a TCP connection.
If the login information was not correct/found:

```
./client
(Please refer to the on-screen messages)
[Client] The credentials are incorrect. Please try again.
Please enter the username: <username>
Please enter the password: <unencrypted_password>
```

After the successful login:

```
[Client] You have been granted access.
```

Please note that both clients should be connected to the main server and if authenticated, they both can quote/buy/sell/ the stocks or check position as explained in the following phases.

Phase 3: Trading (quote/buy/sell)

Once authenticated, the client should be able to check stock prices and buy or sell the stocks.

Clients can put in one of the following commands in this phase:

> quote

> quote <stock name>

> buy <stock name> <number of shares>

> sell <stock name> <number of shares>

Please read the following instructions for each command.

quote

When the client enters the "quote" command, the request is sent to the main server, which then forwards it to ServerQ. ServerQ returns the price of the requested stock to the main server, and then the main server will send the price of the stock to the client. There are two ways that a client use a quote command:

quote : This command should return the current price of all stocks.

quote <stock name>: This command should return the current price of the specified stock.

If the <stock name> does not exist, it should return an error. Check the onscreen messages for the correct format of error messages.

buy

When the client enters the "buy" command, the main server contacts ServerQ to retrieve the stock's current price and returns the price to the client for confirmation. The main server then updates the client’s portfolio in ServerP (stocks that the client owns, the number of shares, and the average buy price for each stock) and instructs ServerQ to update this specific stock's next price. For calculating the average buy price after a buy command, use the following formulas:

We define the Avg_Buy_Price as follows:

$$\text{Avg\_Buy\_Price} = \frac{\text{Total Cost of the Stock Purchased}}{\text{Current Number of Shares Held}}$$

When buying additional shares of a stock that the client already owns, you should update the average buy price using the following formula after the transaction.

$$\text{Avg\_Buy\_Price}_{\text{new}} = \frac{(\text{Number of Shares} \times \text{Avg\_Buy\_Price}) + \text{Money Spent On New Purchase}}{\text{New Number of Shares Held}}$$

Additionally, the client receives a confirmation message from the main server indicating the purchase was successful (check the on-screen message).

If the purchase is unsuccessful (attempt to buy non-existing stock), the client receives a message from the main server stating that the transaction failed. Check the on-screen messages for the correct format of error messages.

-Please do not forget to update the average buy price and number of shares in the client's portfolio after a buy.

sell

When the client enters the "sell" command, the main server contacts ServerQ to retrieve the stock's current price and communicates with ServerP to check the client's holdings for that stock. If the client has enough shares to sell (equal to or more than the requested amount), the main server will ask the client for confirmation. If the client confirms then the sale is successful. The main server then updates the client's portfolio in ServerP by adjusting the number of shares held.

The main server also instructs ServerQ to update the stock's next price. Additionally, the client receives a confirmation message from the main server indicating that the sale was successful (check the on-screen message).

After a successful sale, we define the average buy price does not change. Please remember to update the number of shares of the stock.

If the sale is unsuccessful, the client receives a notification from the main server stating that the transaction failed. Even so, the stock price moves to the next price. Please refer to the on-screen messages for details regarding the unsuccessful sale.
-Note that a successful and unsuccessful buy or sell command from any client will update the stock price for the next buy or sell command, regardless of which client initiates it.

Phase 4: Checking the positions

When a client uses the "position" command, the main server will contact serverP to return the portfolio

information and contact serverQ for stocks' current prices, then calculate the profit/loss. A client portfolio consists of the following information:

-Stocks owned and number of shares

-The average buy price for each stock.

The "position" command returns the client portfolio (mentioned above) and also the current profit and loss of the client.
To calculate profit/loss use the following formula.

$$Profit = \sum_{i}^{n} (N_i \cdot P_{current,i} - N_i \cdot P_{avg,i})$$

Where

- i represents each stock

- N_i is the number of shares of stock i

- P_avg,i is the average purchase price per share of stock i

- P_current,i is the current price per share of stock i

- n is the total number of stocks in the portfolio

The negative profit will be inferred as a loss. Here, we calculate unrealized profit exclusively for stocks currently held in the portfolio.

Please check the on-screen messages for the servers and clients.

Phase 5: Closing clients' connections

Clients can use either the "exit" command or ctrl+c to close the connections. There are no specific on-screen messages for this phase.

Required Port Number Allocation

The ports to be used by the clients and the servers for the exercise are specified in the following table (Major points will be lost if the port allocation is not as per the below description):

Static and Dynamic assignments for TCP and UDP ports.		
Process	Dynamic Ports	Static Ports
Server A	-	UDP, 41000+xxx
Server P	-	UDP, 42000+xxx
Server Q	-	UDP, 43000+xxx
Server M	-	UDP (with servers), 44000+xxx TCP (with clients), 45000+xxx
Clients	2 TCPs	

NOTE: xxx is the last 3 digits of your USC ID. For example, if the last 3 digits of your USC ID are “319”, you should use the port: 41000+319 = 41319 for the Backend-Server (A). It is NOT going to be 41000319. Note that the serverM has only one UDP port. The same port is used to connect to all the backend servers.

ON-SCREEN MESSAGES

*Note: The order of the on-screen messages shown below in the tables does not carry any specific meaning.

Table 1. Server A (Authentication Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting):	[Server A] Booting up using UDP on port <port number>.
Upon Receiving the auth request:	[Server A] Received username <USERNAME> and password ******.
If the auth request is member user:	[Server A] Member <USERNAME> has been authenticated.
If either username or password is incorrect:	[Server A] The username <USERNAME> or password ****** is incorrect.

Table 2. Server P (Portfolio Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting):	[Server P] Booting up using UDP on port <port number>.
Position request	
Upon receiving position requests from the main server	[Server P] Received a position request from the main server for Member: <User Name>
After returning the user's portfolio	[Server P] Finished sending the gain and portfolio of <User Name> to the main server.
Sell Request	
Upon receiving a sell request from the main server	[Server P] Received a sell request from the main server.
After checking the user's portfolio, if sufficient shares are available to sell	[Server P] Stock <stock name> has sufficient shares in <user name>'s portfolio. Requesting users' confirmation for selling stock.
Upon receiving a Y response to the sell confirmation	[Server P] User approves selling the stock.
Upon receiving a N response to the sell confirmation	[Server P] Sell denied.
After checking the user's portfolio, if sufficient shares are not available to sell	[Server P] Stock <stock name> does not have enough shares in <user name>'s portfolio. Unable to sell <number of shares> shares of <stock name>.
After selling the stock	[Server P] Successfully sold <number of shares> shares of <stock name> and updated <user name>'s portfolio.
Buy Request	
Upon receiving a Y response to the buy confirmation	[Server P] Received a buy request from the client.
After buying the stock	[Server P] Successfully bought <number of shares> shares of <stock name> and updated <user name>'s portfolio.

Table 3. Server Q (Quote Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting):	[Server Q] Booting up using UDP on port <port number>.
Quote Request	
Upon receiving a general quote request from the main server	[Server Q] Received a quote request from the main server.
Upon returning the general stock quote to the main server	[Server Q] Returned all stock quotes.
Upon receiving a specific quote request from the main server	[Server Q] Received a quote request from the main server for stock <Stock Name>.
Upon returning the specific stock quote to the main server	[Server Q] Returned the stock quote of <stock name>.
Time Shift Request	
Upon receiving a time forward request from the main server	[Server Q] Received a time forward request for <stock name>, the current price of that stock is <current price> at time <time index>. *Note: time index start from 0*

Table 4. Server M (Main Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting):	[Server M] Booting up using UDP on port <port number>.
Authentication	
Upon receiving the authentication request from the member	[Server M] Received username <USERNAME> and password ****.
Upon sending the authentication request to server A	[Server M] Sent the authentication request to Server A
Upon receiving the	[Server M] Received the response from server A using UDP over

authentication response from server A	<Main Server UDP port number>
Upon sending the authentication response to the client	[Server M] Sent the response from server A to the client using TCP over port <main server TCP port number>.
Position request	
Upon receiving a position request from a member	[Server M] Received a position request from Member to check <username>'s gain using TCP over port <main server TCP port number>.
After forwarding the position request to server P	[Server M] Forwarded the position request to server P.
After receiving the gain response from server P	[Server M] Received user's portfolio from server P using UDP over <Main Server UDP port number>
After forwarding the gain response to the client	[Server M] Forwarded the gain to the client.
Quote request	
After sending a quote request to server Q	[Server M] Sent quote request to server Q.
After receiving a quote response from server Q	[Server M] Received quote response from server Q.
Upon receiving a general quote request from a client	[Server M] Received a quote request from <username>, using TCP over port <main server TCP port number>.
Upon receiving a specific quote request from a client	[Server M] Received a quote request from <username> for stock <stock name>, using TCP over port <main server TCP port number>.
After forwarding the quote request to Server Q	[Server M] Forwarded the quote request to server Q.
Upon receiving a general quote response from Server Q	[Server M] Received the quote response from server Q using UDP over <Main Server UDP port number>
Upon receiving a specific quote response from Server Q	[Server M] Received the quote response from server Q for stock <stock name> using UDP over <Main Server UDP port number>
After forwarding the response to the client	[Server M] Forwarded the quote response to the client.
Sell Request	
Upon receiving a sell request	[Server M] Received a sell request from member <username> using

from the Member User	TCP over port <main server TCP port number>.
After sending the quote request to server Q (to check the stock price)	[Server M] Sent the quote request to server Q.
After forwarding the sell request to server P (to check the user's current hold)	[Server M] Forwarded the sell request to server P.
After forwarding the sell confirmation request to the client	[Server M] Forwarded the sell confirmation to the client.
After forwarding the sell confirmation response to Server P	[Server M] Forwarded the sell confirmation response to Server P.
After forwarding the sell result to the client	[Server M] Forwarded the sell result to the client.
Buy Request	
Upon receiving a buy request from the Member User	[Server M] Received a buy request from member <username> using TCP over port <main server TCP port number>.
After sending the quote request to server Q (to check the stock price)	[Server M] Sent the quote request to server Q.
After sending the buy confirmation request to the client	[Server M] Sent the buy confirmation to the client.
If user reply Y	[Server M] Buy approved.
If user reply N	[Server M] Buy denied.
After forwarding the buy confirmation response to Server P	[Server M] Forwarded the buy confirmation response to Server P.
After forwarding the buy result to the client	[Server M] Forwarded the buy result to the client.
Time Shift Request	
Upon sending a time forward request to server Q	[Server M] Sent a time forward request for <stock name>.

Table 5. Client on-screen messages

Event	On Screen Message
Booting Up	[Client] Booting up.
Prompt to enter credentials	[Client] Logging in. Please enter the username: <username> Please enter the password: <unencrypted_password>
Enter the correct credentials	[Client] You have been granted access.
Enter wrong credentials	[Client] The credentials are incorrect. Please try again. Please enter the username: <username> Please enter the password: <unencrypted_password>
Asking for commands	[Client] Please enter the command: <quote> <quote <stock name>> <buy <stock name> <number of shares>> <sell <stock name> <number of shares>> <position> <exit> *Note: The menu can be printed in horizontal or vertical format.*
Quote request	
Upon sending a quote request	[Client] Sent a quote request to the main server.
After receiving the response from the main server (no stock name specify)	[Client] Received the response from the main server using TCP over port <client port number>. <stock name> <current price> <stock name> <current price> <stock name> <current price> … —--Start a new request—--
After receiving the response from the main server (the stock name exists)	[Client] Received the response from the main server using TCP over port <client port number>. <stock name> <current price>

	—Start a new request—
After receiving the response from the main server (the stock name does not exist)	[Client] Received the response from the main server using TCP over port <client port number>. <stock name> does not exist. Please try again. —Start a new request—
Position Request	
Upon sending a position request	[Client] <username> sent a position request to the main server.
After receiving the gain from the main server	[Client] Received the response from the main server using TCP over port <client port number>. stock shares avg_buy_price <stock name1> <number of shares of stock name1> <avg_buy_price> … <user name>'s current profit is <gain value>. *Note: If the quantity of a stock held is zero, it does not need to be displayed*.
Sell Request	
Stock name does not exist	[Client] Error: stock name does not exist. Please check again.
No stock name or shares are specified	[Client] Error: stock name/shares are required. Please specify a stock name to sell.
Receive stock price and ask for approval	[Client] <stock name>'s current price is <stock price>. Proceed to sell? (Y/N)
Do not have enough shares to sell	[Client] Error: <user name> does not have enough shares of <stock name> to sell. Please try again —Start a new request—
Sell successfully	[Client] <user name> successfully sold <number of shares> shares of <stock name>. —Start a new request—
Buy Request	
Stock name does not exist	[Client] Error: stock name does not exist. Please check again.

No stock name or shares are specified	[Client] Error: stock name/shares are required. Please specify a stock name to buy.
Receive stock price and ask for approval	[Client] <stock name>'s current price is <stock price>. Proceed to buy? (Y/N)
After receiving the response from the main server	[Client] Received the response from the main server using TCP over port <client port number>. <user name> successfully bought <number of shares> shares of <stock name>. —Start a new request—

Assumptions

1. You have to start the processes in this order: serverM, serverA, serverP, serverQ, and at least two clients. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and mention them all in your README file.

2. You are allowed to use blocks of code from Beej's socket programming tutorial (Beej's guide to network programming) in your project. However, you need to cite the copied part in your code. Any signs of academic dishonesty will be taken very seriously.

3. When you run your code, if you get the message port already in use or "address already in use," please first check to see if you have a zombie process. If you do not have such zombie processes or if you still get this message after terminating all zombie processes, try changing the static UDP or TCP port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you have to change the port number, please do mention it in your README file and provide reasons for it.

4. You do not need to consider the racing condition for the price list. Assume that when two or more clients are connected, any given operation completes **fully** before another client begins. In other words, we do not consider cases where a price might change mid-purchase/mid-sell due to another client's action.

Requirements

1. 1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table Port Number Allocation to see which ports are statically defined and which ones are dynamically assigned. Use `getsockname()` function to retrieve the locally-bound port number wherever ports are assigned dynamically as shown below:

   /* Retrieve the locally-bound name of the specified socket and store it in the sockaddr structure */

```
getsock_check = getsockname(TCP_Connect_Sock, (struct sockaddr*)&my_addr, (socklen_t
*)&addrlen);
// Error checking
if (getsock_check == -1) {
      perror("getsockname");
      exit(1);
  }
```

2. The host name must be hard coded as "localhost" or "127.0.0.1" in all codes.

3. Your client, backend servers, and main server should remain running and continue waiting for additional requests until the TAs terminate the servers and clients using Ctrl+C or terminate the clients with the "exit" command. If any of them terminate prematurely, you will lose points as a result.

4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.

5. You are not allowed to pass any parameters or values or strings or characters as a command-line argument except what is already described in the project document.

6. All the on-screen messages must conform exactly to the project description. You should not add any more on-screen messages. If you need to do so for debugging purposes, you must comment out all of the extra messages before you submit your project.

7. Please do remember to close the socket and tear down the connection once you are done using that socket.

Programming Platform and Environment

1. All your submitted code MUST work well on the provided virtual machine Ubuntu. No additional package installation is allowed.

2. All submissions will only be graded on the provided Ubuntu. TAs/Graders won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code works well on the provided Ubuntu. "It works well on my machine" is not an excuse.

3. Your submission MUST have a Makefile. Please follow the requirements in the following "Submission Rules" section.

Programming Languages and Compilers

You must use only C/C++ on UNIX as well as UNIX Socket programming commands and functions. Here are the pointers for Beej's Guide to C Programming and Network Programming (socket programming):
- http://www.beej.us/guide/bgnet/
(If you are new to socket programming, please study this tutorial carefully as soon as possible and before starting the project)
- http://www.beej.us/guide/bgc/

You can use a UNIX text editor like `emacs` to type your code and then use compilers such as `g++` (for C++) and `gcc` (for C) that are already installed on Ubuntu to compile your code.
You must use the following commands and switches to compile your `.c` or `.cpp` files. It will make an executable by the name of "yourfileoutput":

```
gcc -o yourfileoutput yourfile.c
g++ -o yourfileoutput yourfile.cpp
```

Do NOT forget the mandatory naming conventions mentioned before!

Also, inside your code, you need to include these header files in addition to any other header files you think you may need:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
#include <arpa/inet.h>
#include <sys/wait.h>
```

Submission File and Folder Structure:

Your submission must have the following folder structure and the files (the examples are of .cpp, but it can be .c files as well):

- ee450_lastname_firstname.tar.gz
    - ee450_lastname_firstname
        - client.cpp
        - serverM.cpp
        - serverA.cpp
        - serverP.cpp
        - serverQ.cpp
        - Makefile
        - readme.txt (or) readme.md
        - <Any additional header or implementation files YOU WROTE YOURSELF>

Please make sure that your name matches the one in the class list. The TAs will extract the tar.gz file, and will place all the input data files in the same directory as your source files. The executable files should also be generated in the same directory as your source files.

Do NOT include any input data files or executable files, as this will result in losing points.

Submission Rules

Along with your code files, include a README file and a Makefile. The Makefile requirements are mentioned in the section below.

The README file can be in any format (Markdown or txt). The only requirement is that the TAs should be able to open the file and read it in the studentVM (the VM your project will be graded in) without installing any additional software.

In the README file, please include:

a. Your Full Name as given in the class list
b. Your Student ID
c. What you have done in the assignment
d. What your code files are and what each one of them does. (Please do not repeat the project description; just name your code files and briefly mention what they do).
e. The format of all the messages exchanged, e.g., username and password are concatenated and delimited by a comma, etc.
f. Any idiosyncrasy of your project. It should specify under what conditions the project fails, if any.

g. Reused Code: Did you use code from anywhere for your project? If not, say so. If so, state what functions and where they're from. (Also identify this with a comment in the source code).
h. Which version of Ubuntu (only the Ubuntu versions that we provided to you) are you using?

Reusing functions that are directly obtained from a source that does not belong to you (e.g., from the internet or generated by AI) with little to no modifications, is considered plagiarism—except for code from Beej's Guide. Whenever you are referring to an online resource, make sure to only look at the source, understand it, close it, and then write the code by yourself. The TAs will perform plagiarism checks on your code, so make sure to follow this step rigorously for every piece of code you submit.

Submissions WITHOUT README and Makefile WILL NOT BE GRADED.

Makefile

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html

About the Makefile: makefile should support following functions:

make all	Compiles all your files and creates executables
make clean	Removes all the executable files

After "make all', TAs will run the following commands in sequence:

./serverM	start Main server
./serverA	start Backend server A
./serverP	start Backend server P
./serverQ	start Backend server Q
./client	start the client

TAs will first compile all codes using **make all**. They will then open at least 6 different terminal windows. On 4 terminals they will start servers M, A, P and Q. On the remaining terminals, they will start the clients using "./client". Remember that all programs should always be on once started. TAs will check the outputs for multiple values of input. The terminals should display the messages shown in On-screen Messages tables in this project writeup.

Here are the instructions:

On your VM, go to the directory which has all your project files. Remove all executable and other unnecessary files. **ONLY** include the required source code files, Makefile and the README file. Now run the following two commands:
>> tar cvf ee450_YourLastName_YourFirstName.tar *
>> gzip ee450_YourLastName_YourFirstName.tar
Now, you will find a file named
"ee450_YourLastName_YourFirstName.tar.gz" in the same directory.
Please notice there is a space and a star(*) at the end of the first command.

An example submission would be:
First Name: John
Last Name: Doe
So the submission would be: ee450_Doe_John.tar.gz
Any compressed format other than .tar.gz will NOT be graded!

Upload "ee450_YourLastName_YourFirstName.tar.gz" to the Digital Dropbox on the BrightSpace website (BrightSpace -> EE450 -> Activities -> Assignments -> Project). After the file is uploaded to the dropbox, you must click on the "Submit" button to actually submit it. If you do not click on "Submit", the file will not be submitted.
BrightSpace will keep only the last submission of your submissions. Submission after the deadline is considered as invalid. BrightSpace will send you a "Dropbox submission receipt" to confirm your submission. So please do check your emails to make sure your submission is successfully received. If you have not received a confirmation mail, contact your TA if it always fails. After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. This is exactly what your designated TA would do, So please grade your own project from the perspective of the TA. If the outcome is not what you expected, try to resubmit and confirm again. We will only grade what you submitted even though it's corrupted.
Please take into account all kinds of possible technical issues and do expect a huge traffic on the BrightSpace website very close to the deadline which may render your submission or even access to BrightSpace unsuccessful.
Please DO NOT wait till the last 5 minutes to upload and submit because some technical issues might happen and you will miss the deadline. And a kind suggestion, if you still get some bugs one hour before the deadline, please make a submission first to make sure you will get some points for your hard work!
There is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a ZERO for the project.

Grading Criteria

Notice: We will only grade what is already done by the program instead of what will be done. The grading criteria are subject to change.

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, specially the communications through UDP and TCP sockets.
2. Inline comments in your code. This is important as this will help in understanding what you have done.
3. Whether your programs work as you say they would in the README file.
4. Whether your programs print out the appropriate error messages and results.
5. Your code will only be tested on a fresh copy of the provided Virtual Machine (either studentVM (64-bit) or Ubuntu 22.04 ARM64 for M1/M2 Mac users). If your programs are not compiled or executed on these VM, you will receive only minimum points as described below. Be careful if you are going to use other environments!!! Do not update or upgrade the provided VM as well!!!
6. If your submitted codes do not even compile, you will receive 5 out of 100 for the project.
7. If your submitted codes compile using make but when executed, produce runtime errors without performing any tasks of the project, you will receive 10 out of 100.
8. The minimum points for compiled and executable codes is 15 out of 100.
9. If your code does not correctly assign the TCP or UDP port numbers (in any phase), you will lose points each.
10. We will use similar test cases to test all the programs. These test cases cover all situations including edge cases, referring to the on-screen messages section.
11. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend about 2 weeks on this project and it doesn't even compile, you will receive only 5 out of 100.
12. You must discuss all project related issues on the Piazza Discussion Forum. We will give extra points to those who actively help others out by answering questions on Piazza. (If you want to earn the extra credits, do remember to leave your names visible to instructors when answering questions on Piazza.)

13. Your code will not be altered in any way for grading purposes and however it will be tested with different inputs. Your TA/Grader runs your project as is, according to the project description and your README file and then checks whether it works correctly or not. If your README is not consistent with the description, we will follow the description.

Cautionary Words

1. Start on this project early!!!

2. In view of what is a recurring complaint near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is studentVM (64-bit) or Ubuntu 22.04 ARM64 for M1/M2 Mac users. It is strongly recommended that students develop their code on this virtual machine. In case students wish to develop their programs on their personal machines, possibly running other operating systems, they are expected to deal with technical and incompatibility issues (on their own) to ensure that the final project compiles and runs on the requested virtual machine. If you do

development on your own machine, please leave at least three days to make it work on Ubuntu. It might take much longer than you expect because of some incompatibility issues.

3. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command:
   >>ps –aux | grep ee450
   Identify the zombie processes and their process number and kill them by typing at the command-line:
   >>kill -9 <process number>

Academic Integrity

All students are expected to write all their code on their own!!!

Do not post your code on Github, especially in a public repository before the deadline!!!
Double check the setting and do some testing before posting in a private repository!!!

Copying code from friends or from any unauthorized resources (webpages, github, etc.) is called plagiarism not collaboration and will result in an F for the entire course. **Any libraries or pieces of code that you use or refer and you did not write must be listed in your README file.** Students are only allowed to use the code from Beej's socket programming tutorial. Copying the code from any other resources may be considered as plagiarism. Please be careful!!! All programs will be compared with automated tools to detect similarities; examples of code copying will get an F for the course. **IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR ISN'T ALLOWED ABOUT PLAGIARISM, TALK TO THE TA. "I didn't know" is not an excuse.**

AI Policy

We allow the use of AI to assist you in understanding how to write code, utilize libraries, and similar tasks. However, you are not permitted to copy the AI-generated content directly. It can only be used as a reference. Additionally, you must include comments in the code specifying the prompt used and the AI model involved. **You should never attempt to present or include content created by others, including generative AI as your own. Attempting to take credit for content generated by AI or others without proper acknowledgement is a violation of USC's policies and standards for academic integrity and can result in disciplinary action.** For any other details not explicitly mentioned, please refer to the USC AI Policy webpage: https://libguides.usc.edu/generative-AI/scholarship-research