EE450 Socket Programming Project, Spring 2022 Due Date : Thursday April 24th, 2022 11:59 PM (Midnight)

(The deadline is the same for all on-campus and DEN off-campus students)

Hard Deadline (Strictly enforced)

The objective of this assignment is to familiarize you with UNIX socket programming. This assignment is worth **10%** of your overall grade in this course. **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 those who actively help others out by answering questions on Piazza up to 10 bonus points.

Problem Statement:

Digital transactions and cryptocurrencies have been a hot topic lately. People prefer to use digital transactions due to their simplicity and they choose cryptos because they provide a sense of safety and anonymity that regular currencies cannot. While the latter are not bounded by a government or a country, their success relies on the distributed nature of their platforms. The technology behind them is known as blockchain, a chain of information blocks where we store all the transactions that have taken place since the start of the system. The amount of transactions stored on a block depends on the type of blockchain and the information can be provided to every member of the system (public blockchain) or to a few members of it (permissioned blockchain) . In this project we shall implement a simplified version of a blockchain system that'll help us understand how cryptocurrency transactions work. For this scenario, we will have three nodes on the blockchain where each stores a group of transactions. These will be represented by backend servers. While in the blockchain the transaction protocol deals with updating the digital wallet of each user, for this project we will have a main server in charge of running the calculations and updating the wallets for each user. Each transaction reported in the blockchain will include, in the following order, the transaction number, sender, receiver and amount being transferred.

In this project, you will implement a simplified version of a blockchain service called **Alichain**, where two clients issue a request for finding their current amount of **alicoins** in their account, transfer them to another client and provide a file statement with all the transactions in order. These requests will be sent to a Central Server which in turn interacts with three other backend servers for pulling information and data processing.

The Main server will connect to servers A, B and C, which have a group of transactions that are not in order. When required, the main server has to pull the transaction information from each backend server to find out the current wallet balance of the user, to transfer money from one user to another and to write back a file statement with all the transactions and store it on the Main server. The steps and explanation of each operation that will be required for the blockchain platform is provided in each phase of the project.

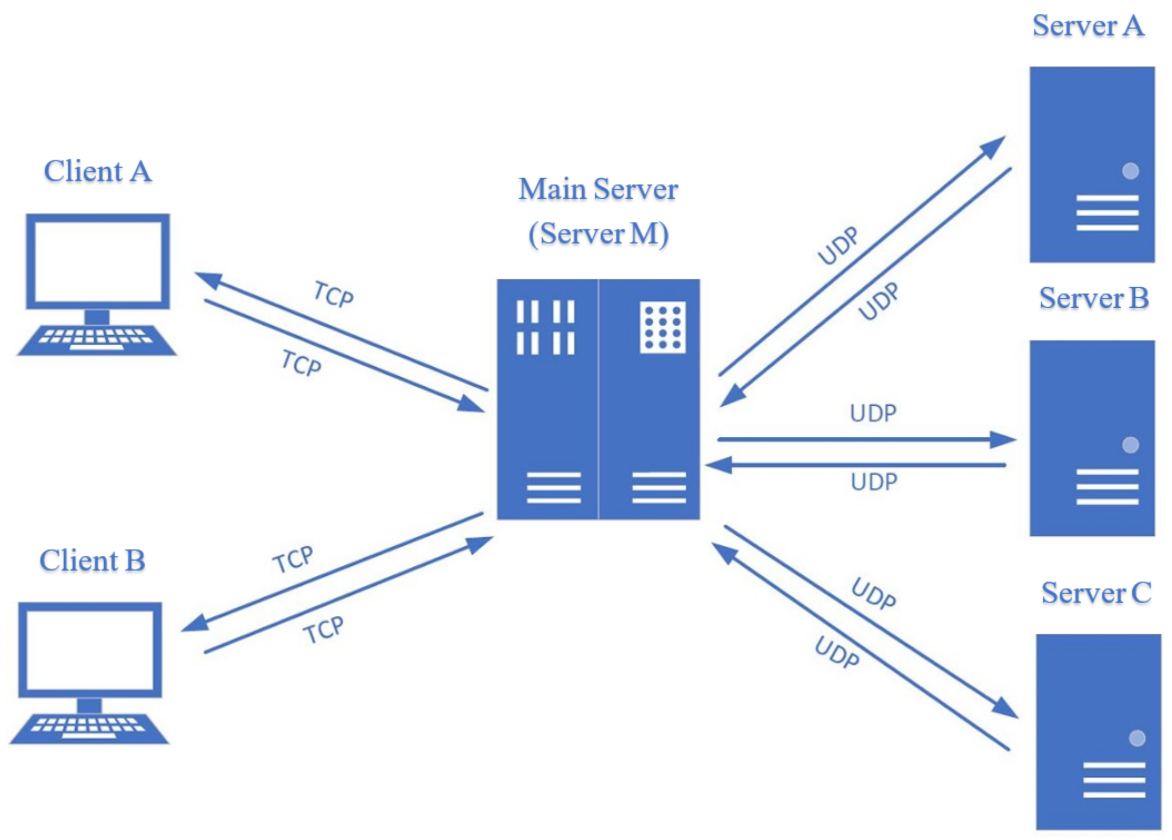

Figure 1. Illustration of the network

Server A has access to a file named block1.txt, server B has access to a file named block2.txt and server C has access to a file named block3.txt . Both clients and the main server communicate over a TCP connection while the communication between main server and the Back-Servers A, B & C is over a UDP connection. This setup is illustrated in Figure 1.

Source Code Files

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

1. <u>ServerM</u>: You must name your code file: **serverM.c** or **serverM.cc** or **serverM.cpp** (all small letters except 'M'). Also you must include the corresponding header file (if you have one; it is not mandatory) serverM.h (all small letters except 'M').
2. <u>Back-Server A, B and C:</u> You must use one of these names for this piece of code: **server#.c** or **server#.cc** or **server#.cpp** (all small letters except for #). Also you must include the corresponding header file (if you have one; it is not mandatory). **server#.h** (all small letters, except for #). The "#" character must be replaced by the server identifier (i.e. A or B or C), depending on the server it corresponds to. **Note**: You are not allowed to use one executable for all four servers (i.e. a "fork" based implementation).
3. <u>ClientA</u>: The name of this piece of code must be **clientA.c** or **clientA.cc** or clientA.cpp (all small letters) and the header file (if you have one; it is not mandatory) must be called clientA.h (all small letters).
4. <u>ClientB</u>: The code file for the monitor must be called **clientB.c** or clientB.cc or clientB.cpp (all small letters) and the header file (if you have one; it is not mandatory) must be called **clientB.h** (all small letters).

<u>Application workflow phase Description:</u>

<u>Phase 1: (30 points)</u>

Establish the connections between the Clients and Main Server

All four server programs (Main Server, Backend ServerA, Backend ServerB and Backend ServerC) boot up in this phase. While booting up, the servers must display a boot message on the terminal. The format of the boot message for each server is given in the onscreen messages tables at the later part of the document. As the boot message indicates, each server must listen on the appropriate port information for incoming packets/connections.

Once the server programs have booted up, two client programs should run. Each client displays a boot message as indicated in the onscreen messages table. Note that each client code takes an input argument from the command line that specifies the username(s).

Note: Once started, the main server and the three backend servers should always be running (unless stopped manually). The clients can terminate their connections with the main server once they receive the response for the operation they had requested (checking balance or performing a transaction) .

For the first two phases the client can perform two kinds of request operations, namely “CHECK WALLET”, “TXCOINS” (transfer coins). Which operation is performed depends on the number of input arguments that the code takes from the command line:

Operation1: CHECK WALLET

In this operation, the input argument should be the client’s own name and it will be used to fetch the client’s balance in the wallet, which will be described in the next section. The format for running each client code in this operation is:

The command for the client A should be

```
./clientA <username1>
```

The command for the client B should be

```
./clientB <username2>
```

The usernames from the ClientA and ClientB are the inputs for getting the balance of each of their wallets. As an example, to check client A's wallet balance, the command should be run as follows:

```
./clientA Martin
```

Operation2: TXCOINS

In this operation, There will be three input arguments, namely the client name of the transfer out, the client name of the transfer in, and the number of transfers. The main server will complete and record this transaction according to the input arguments, more details will be described in the next section. The format for running each client code in this operation is:

The command for the client A should be

```
./clientA <username1> <username2> <transfer amount>
```

The command for the client B should be

```
./clientB <username2> <username1> <transfer amount>
```

The usernames from the Clients are the inputs for transferring coins from one user to another. As an example, to transfer 100 alicoins from client A to client B should be run as follows

```
./clientA Martin Luke 100
```

After booting up, the ClientA and ClientB establish TCP connections with the server. After successfully establishing the connections, the two clients first send the usernames to Server C. Once these are sent, each client should print a message in the format given in the table 8 & 9. This ends Phase 1 and we now proceed to Phase 2.

M

Phase 2: (40 points)

Phase 2A: Main Server connects to backend servers and proceeds to retrieve the information.

In Phase 1, you read what should be sent from ClientA and ClientB to server M (main server) over the TCP connections. For Phase 2, Server M will send messages to the three back-servers (Server A, Server B and Server C) with UDP connections. The request will be sent to their respective back-end server depending on which information they need to get and which operation they need to execute.

Table 1: An example of the format for each Alichain transaction

Serial No.	Sender	Receiver	Transfer Amount
1	Racheal	John	45
2	Rishil	Alice	30
3	Oliver	Rachit	94
4	Ben	Victor	85
5	Chinmay	Oliver	129
6	Racheal	Alice	49
7	Martin	Luke	25
8	Rishil	Chinmay	10

9	Ali	Luke	155

Every member of the platform received **1000 alicoins** as incentive to join this blockchain. On table 1 we can see the format of each row in the block files stored at the backend servers.

Operation1: CHECK WALLET

For this operation the main server will use the input *username* provided on phase 1 and check if the username is found on the transaction records while communicating with the backend servers. If the username is not found the main server will provide a message as indicated below on the On-screen messages portion of the document. If the username exists we will calculate the amount the *username* has on Alichain.

Operation2: TXCOINS

For this operation the main server will use the input *username1* , *username2* and *amount* provided on phase 1 and check if both usernames can be found on the records while communicating with the backend servers. If any of the usernames is not found the main server will provide a message as indicated below on the On-screen messages portion of the document. If both usernames exist, we will add the transaction to any of the block files that are located in the backend servers.

Phase 2B: Main Server computes the operation and replies to the client.

Once the connection server (or the main server) receives the relevant data for the desired operation from the other three servers (namely serverA, serverB and serverC), it will perform the required computation and send the results to the clients. The computation performed by the main server depends on the service requested by the clients (i.e, CHECK WALLET or TXCOINS)

Operation1: CHECK WALLET

In this operation, the main server would be receiving only those transaction logs from each of the three backend servers (serverA, serverB and serverC) in which the client was involved. There can be multiple transaction logs related to a particular client distributed randomly across the three log files. Each backend server would be handling it's own log file.

Based on the log data (mentioned above) received, the main server will compute the current balance of the client based on the following formula :

$$Current\ Balance\ =\ Initial\ Balance\ +\ \sum Coins\ Received\ -\ \sum Coins\ Sent$$

As already mentioned in phase 2A, initially all the members would be having 1000 alicoins. So, 'Initial Balance' for everyone would be 1000.
After computing the current balance for a client, the main server will send this info to that respective client.

Operation2: TXCOINS

In this operation, the main server would notify the client about the transaction status. The transaction can either be successful or unsuccessful. In either case, the main server should notify the client about the status.

The transaction can be unsuccessful mainly because of two reasons. The first one being insufficient current balance of the sender. Transactions can also fail if either sender or recipient (or both) is not part of the network.

If the transaction is feasible (both the sender and the receiver are part of the network and the sender's balance is equal to or more than what he/she is intending to send), the main server would perform the following computations:

- The main server would figure out what should be the serial number of the current transaction. Every transaction would have a serial number which should be assigned in a contiguous fashion as they occur. In the log files, these transactions would not be arranged in some specific order of their serial number as the transaction log data is randomly distributed across three log databases handled by the three backend servers (serverA, serverB and serverC) separately. So, if the last transaction had a serial number 'n', then the current transaction should have the serial number of 'n+1'.
- The main server would then generate the log entry (which would be similar to the the other logs):

  <Serial_No> <Sender_Username> <Receiver_Username> <Transfer_Amount>

- The main server then randomly selects any of the three backend servers (serverA, serverB and serverC) and sends this log entry to the selected server.

The server which receives this log information records this log information as a new entry in its log database and sends a confirmation message to the main server.

- Upon receiving the confirmation, the main server enquiries for the updated balance of the sender and sends the transaction status along with the updated balance (current balance after this transaction) to the sender client.

Phase 3: (30 points)

In phases 1 and 2 we have described two operations that required us to do calculations to check each user's account balance and also to add transactions to the blockchain. For phase 3, we will add one more operation named TXLIST

Operation3: TXLIST

For this operation the client will send a keyword to indicate that the client is asking to get the full text version of all the transactions that have been taking place in Alichain and save it on a file named "alichain.txt" located on the main server. The format of the operation will be as follows:

./clientA TXLIST or ./clientB TXLIST

Either client should be able to handle this operation. When the client runs this operation, main server will receive this request and connect to the backend servers to gather all the information from the transactions. Main server will sort the list of transactions and generate the "alichain.txt" file with all the transactions made up to that point (including any new transactions made from the moment we booted up the servers). Students will have the freedom to choose any algorithm they prefer for this sorting operation.

Phase 4: (10 points extra, not mandatory)

If you want to earn 10 extra points, you can implement an extra operation where either client provides a statistical result with a summary of all the transactions and provides a list of all the users the client has made transactions with usernameA. This operation cannot be done separately, i.e, it should be on the same command line "**`./clientA <usernameA> stats`**". For example, for username Ali when running the command:

./clientA Ali stats

Table 2: An example of the stats file format for phase 4			
Rank	**Username**	**Number of transactions made with user**	**Transfer Amount**
1	Racheal	15	355
2	Alice	10	280
3	Oliver	8	35
4	Ben	5	-240
5	Chinmay	3	-100
6	Luke	1	30

ClientA will receive this information and print it on screen. The results should be ranked based on the number of times the user has made or received a transaction from each user. The transfer amount is the resulting balance from all the transactions made between the username we are requesting the stats from and the other it has interacted with.

NOTE: The extra points will only work when you don't get full 100 points. The maximum points for this socket programming project is only 100. For example, you get 97 + 10 = 100.

Required Port Number Allocation

The ports to be used by the clients and the servers for the exercise are specified in the following table:

Table 3. Static and Dynamic assignments for TCP and UDP ports.		
Process	**Dynamic Ports**	**Static Ports**
Backend-Server (A)	-	1 UDP, 21000+xxx
Backend-Server (B)	-	1 UDP, 22000+xxx

Backend-Server (C)	-	1 UDP, 23000+xxx
Main Server (M)	-	1 UDP, 24000+xxx 1 TCP with client A, 25000+xxx 1 TCP with client B, 26000+xxx
Client A	1 TCP	<Dynamic Port assignment>
Client B	1 TCP	<Dynamic Port assignment>

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: **21000+319 = 21319** for the Backend-Server (A). **It is NOT going to be 21000319.**

ON SCREEN MESSAGES:
Table 4. Backend-Server A on screen messages

Event	On Screen Message (inside quotes)
Booting Up (Only while starting):	"The ServerA is up and running using UDP on port <port number>."
Upon Receiving the request from main server	"The ServerA received a request from the Main Server."
After sending the results to the main server:	"The ServerA finished sending the response to the Main Server."

ON SCREEN MESSAGES:
Table 5. Backend-Server B on screen messages

Event	On Screen Message (inside quotes)
Booting Up (Only while starting):	"The ServerB is up and running using UDP on port <port number>."
Upon Receiving the request from Main Server	"The ServerB received a request from the Main Server."
After sending the results to the main server:	"The ServerB finished sending the response to the Main Server."

ON SCREEN MESSAGES:
Table 6. Backend-Server C on screen messages

Event	On Screen Message (inside quotes)

Booting Up (Only while starting):	"The ServerC is up and running using UDP on port <port number>."
Upon Receiving the request from main server	"The ServerC received a request from the Main Server."
After sending the results to the main server:	"The ServerC finished sending the response to the Main Server."

ON SCREEN MESSAGES: Table 7. Main Server on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (only while starting):	"The main server is up and running."
Upon Receiving the username from the clients for checking balance:	"The main server received input=<USERNAME> from the client using TCP over port <port number>."
Upon Receiving the username from the clients for transferring coins:	"The main server received from <SENDER_USERNAME> to transfer <TRANSFER_AMOUNT> coins to <RECEIVER_USERNAME> using TCP over port <port number>."
After querying Backend-Server i for checking balance (i is one of A,B, or C):	"The main server sent a request to server <i>."
After receiving result from backend server i for checking balance (i is one of A,B, or C):	"The main server received transactions from Server <i> using UDP over port <PORT_NUMBER>."

After querying Backend-Server i for transferring coins (i is one of A,B, or C):	“The main server sent a request to server <i>.”
After receiving result from backend server i for transferring coins (i is one of A,B, or C):	“The main server received the feedback from server <i> using UDP over port <PORT_NUMBER>.”
After sending the current balance to the client j (j is either A or B):	“The main server sent the current balance to client <j>.”
After sending the result of transaction to the client j (j is either A or B):	“The main server sent the result of the transaction to client <j>.”

ON SCREEN MESSAGES: Table 8. Client A on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The client A is up and running.”
Upon sending the input to main server for checking balance	"<USERNAME> sent a balance enquiry request to the main server.”
Upon sending the input(s) to the main server for making a transaction.	“<SENDER_USERNAME> has requested to transfer <TRANSFER_AMOUNT> coins to <RECEIVER_USERNAME>.”
After receiving the balance information from the main server	“The current balance of <USERNAME> is : <BALANCE_AMOUNT> alicoins.”
After receiving the transaction information from the main server (if successful)	“<SENDER_USERNAME> successfully transferred <TRANSFER_AMOUT> alicoins to <RECEIVER_USERNAME>. The current balance of <SENDER_USERNAME> is : <BALANCE_AMOUNT> alicoins.”
After receiving the transaction information from the main server (if transaction fails due to insufficient balance)	“<SENDER_USERNAME> was unable to transfer <TRANSFER_AMOUT> alicoins to <RECEIVER_USERNAME> because of insufficient balance. The current balance of <SENDER_USERNAME> is : <BALANCE_AMOUNT> alicoins.”
After receiving the transaction information from the main server (if one of the clients is not part of the network)	“Unable to proceed with the transaction as <SENDER_USERNAME/RECEIVER_USERNAME> is not part of the network.”
After receiving the transaction information from the main server (if both the clients are not part of the network)	“Unable to proceed with the transaction as <SENDER_USERNAME> and <RECEIVER_USERNAME> are not part of the network.”
Upon sending the input to main server for requesting a sorted list	“<USERNAME> sent a sorted list request to the main server.”
Upon sending the input to main server for requesting statistics	"<USERNAME> sent a statistics enquiry request to the main server.”
After receiving the transaction information from the main server (list of transactions)	"<USERNAME> statistics are the following.:” Rank--Username--NumofTransacions--Total

ON SCREEN MESSAGES:
Table 9. Client B on screen messages

Event	On Screen Message (inside quotes)
Booting Up:	"The client B is up and running."
Upon sending the input to main server for checking balance	"<USERNAME> sent a balance enquiry request to the main server."
Upon sending the input(s) to the main server for making a transaction.	"<SENDER_USERNAME> has requested to transfer <TRANSFER_AMOUNT> coins to <RECEIVER_USERNAME>."
After receiving the balance information from the main server	"The current balance of <USERNAME> is : <BALANCE_AMOUNT> alicoins."
After receiving the transaction information from the main server (if successful)	"<SENDER_USERNAME> successfully transferred <TRANSFER_AMOUT> alicoins to <RECEIVER_USERNAME>. The current balance of <SENDER_USERNAME> is : <BALANCE_AMOUNT> alicoins."
After receiving the transaction information from the main server (if transaction fails due to insufficient balance)	"<SENDER_USERNAME> was unable to transfer <TRANSFER_AMOUT> alicoins to <RECEIVER_USERNAME> because of insufficient balance. The current balance of <SENDER_USERNAME> is : <BALANCE_AMOUNT> alicoins."
After receiving the transaction information from the main server (if one of the clients is not part of the network)	"Unable to proceed with the transaction as <SENDER_USERNAME/RECEIVER_USERNAME> is not part of the network."
After receiving the transaction information from the main server (if both the clients are not part of the network)	"Unable to proceed with the transaction as <SENDER_USERNAME> and <RECEIVER_USERNAME> are not part of the network."
Upon sending the input to main server for requesting a sorted list	"<USERNAME> sent a sorted list request to the main server."
Upon sending the input to main server for requesting statistics	"<USERNAME> sent a statistics enquiry request to the main server."
After receiving the transaction information from the main server (list of transactions)	"<USERNAME> statistics are the following.:" Rank--Username--NumofTransacions--Total

Example Output to Illustrate Output Formatting:

For operation check wallet:

Backend-Server A Terminal:

```
The ServerA is up and running using UDP on port 21319.
The ServerA received a request from the Main Server.
The ServerA finished sending the response to the Main Server.
```

Backend-Server B Terminal:

```
The ServerB is up and running using UDP on port 22319.
The ServerB received a request from the Main Server.
The ServerB finished sending the response to the Main Server.
```

Backend-Server C Terminal:

```
The ServerC is up and running using UDP on port 23319.
The ServerC received a request from the Main Server.
The ServerC finished sending the response to the Main Server.
```

Main Server Terminal:

```
The main server is up and running.
The main server received input=“Racheal” from the client using TCP over port 25319.
The main server received from “Racheal” to transfer 45 coins to “John” using TCP over port 25319.
The main server sent a request to server A.
The main server received transactions from Server A using UDP over port 21319.
The main server sent a request to server A.
The main server received the feedback from server A using UDP over port 21319.
The main server sent the current balance to client A.
The main server sent the result of the transaction to client A.
```

Client A Terminal:

```
The client A is up and running.
“Racheal” sent a balance enquiry  request to the main server.
The current balance of “Racheal” is : 1000 alicoins.
```

```
"Racheal" has requested to transfer 45 coins to "John".
"Racheal" has successfully transferred 45 alicoins to "John".
The current balance of "Racheal" is : 955 alicoins.
"Racheal" sent a sorted list request to the main server.
"Racheal" sent a statistics enquiry  request to the main server.
```

Client B Terminal:

```
The client B is up and running.
"John" sent a balance enquiry  request to the main server.
The current balance of "John" is : 1000 alicoins.
```

Assumptions:

1. You have to start the processes in this order: **Server C, Server T, Server S, Server P, Client A and Client B.**

2. 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.

3. 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 mark the copied part in your code.

4. 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** (see following). 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.

5. 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 kills them by typing at the command-line: `>>kill -9 processNumber.`

Requirements:

1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table 1 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 (127.0.0.1)** in all codes.
3. Your clients should terminate themselves after all is done. And the clients can run multiple times to send requests. However, the backend servers and the Central server should keep being running and be waiting for another request until the TAs terminate them by Ctrl+C. If they terminate before that, you will lose some points for it.
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 parameter or value or string or character as a command-line argument except while running the clients in Phase 1.
6. All the on-screen messages must conform exactly to the project description. You should not add anymore on-screen messages. If you need to do so for the 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.
2. All submissions will only be graded on the provided Ubuntu. TAs 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 do 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 yourfile.c or yourfile.cpp. 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 file 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 Rules:

1. Along with your code files, include a **README file and a Makefile**. In the README file write
    a. Your **Full Name** as given in the class list
    b. Your Student ID
    c. What you have done in the assignment, if you have completed the optional part (suffix). If it's not mentioned, it will not be considered.
    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.
    g. Any idiosyncrasy of your project. It should say under what conditions the project fails, if any.
    h. Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.)

Submissions WITHOUT README AND Makefile WILL NOT BE GRADED.

Makefile tutorial:
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
`./serverM`	**Run**s central server
`./serverA`	**Run**s server A
`./serverB`	**Run**s server B
`./serverC`	**Run**s server C
`./clientA <username1>`	Starts the clientA
`./clientB <username2>`	Starts the clientB

TAs will first compile all codes using `make all`. They will then open 6 different terminal windows. On 4 terminals they will start servers M, A, B and C . On the other two terminals, they will start client processes using `./client <name>`. **Remember that servers should always be on once started.** Clients can connect again and again with different input values. TAs will check the outputs for multiple values of input. The terminals should display the messages shown in tables in this project writeup.

2. Compress all your files including the README file into a single "tar ball" and call it: **ee450_yourUSCusername_session#.tar.gz** (all small letters) e.g. my filename would be **ee450_nanantha_session1.tar.gz**. Please make sure that your name matches the one in the class list. Here are the instructions:

   a. 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 commands:

   b.
   >> tar cvf **ee450_yourUSCusername_session#.tar** *
   >> gzip **ee450_yourUSCusername_session#.tar**
   Now, you will find a file named "ee450_yourUSCusername_session#.tar.gz" in the same directory. Please notice there is a star(*) at the end of first command.
   Any compressed format other than .tar.gz will NOT be graded!

3. Upload "ee450_yourUSCusername_session#.tar.gz" to the Digital Dropbox on the DEN website (DEN -> EE450 -> My Tools -> Assignments -> Socket Project). After the file is uploaded to the dropbox, you must click on the "**send"** button to actually submit it. If you do not click on "**send**", the file will not be submitted.

4. D2L will and keep a history of all your submissions. If you make multiple submission, we will grade your latest valid submission. Submission after deadline is considered as invalid.

5. D2L 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 don’t receive a confirmation email, try again later and contact your TA if it always fails.

6. Please take into account all kinds of possible technical issues and do expect a huge traffic on the DEN website very close to the deadline which may render your submission or even access to DEN unsuccessful.

7. 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!

8. After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. 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.

9. **You have plenty of time to work on this project and submit it in time hence 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.
For example, the TCP connection is established and data is sent to the AWS. But result is not received by the client because the AWS got some errors.Then you will lose some points for phase 1 even though it might work well.

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. If your submitted codes do not even compile, you will receive 5 out of 100 for the project.

6. 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.

7. If you forget to include the README file or Makefile in the project tar-ball that you submitted, you will lose 15 points for each missing file (plus you need to send the file to the TA in order for your project to be graded.)

8. If your code does not correctly assign the TCP or UDP port numbers (in any phase), you will lose 10 points each.

9. You will lose 5 points for each error or a task that is not done correctly.

10. The minimum grade for an on-time submitted project is 10 out of 100, assuming there are no compilation errors and the submission includes a working Makefile and a README.

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 months 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 Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points. (If you want to earn the extra credits, do remember to leave your names visible to instructors when answering questions on Piazza.)

13. The maximum points that you can receive for the project with the bonus points is 100. In other words the bonus points will only improve your grade if your grade is less than 100.

14. Your code will not be altered in any ways for grading purposes and however it will be tested with different inputs. Your designated TA runs your project as is, according to the project description and your README file and then check 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 *the provided **Ubuntu (16.04)***. 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 processnumber`**

Academic Integrity:

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

Copying code from friends is called **plagiarism** not **collaboration** and will result in an F for the entire course. Any libraries or pieces of code that you use and you did not write must be listed in your README file. 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.