EE450 Socket Programming Project

Fall 2020

Due Date:

Thursday, November 5, 2020 11:59PM

(Hard Deadline, Strictly Enforced)

The deadline is for both on-campus and DEN off-campus students.

OBJECTIVE

The objective of this assignment is to familiarize you with UNIX socket programming. This assignment is worth **15%** 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).**

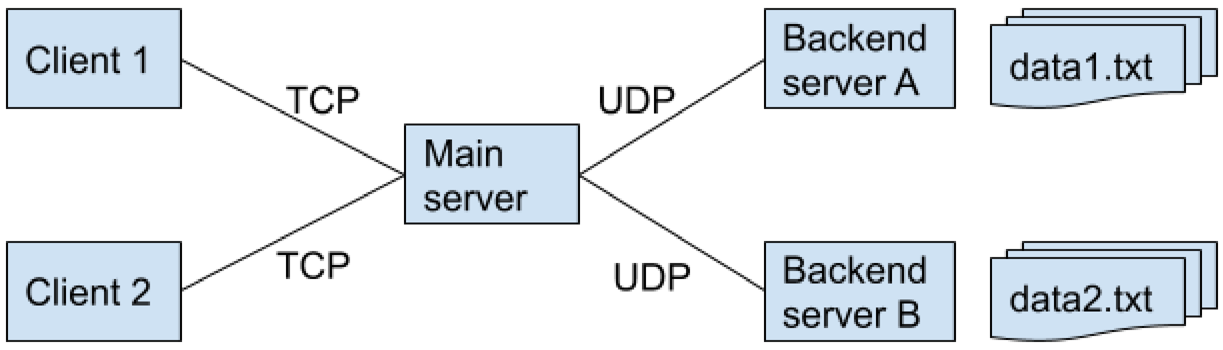
If you have any doubts/questions, post your questions on Piazza. **You must discuss all project related issues on Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points to the project.

You can ask TAs any question about the content of the project but TAs have the right to reject your request for debugging.

PROBLEM STATEMENT

Nowadays, social recommendation is an important task to enable online users to discover friends to follow, pages to read and items to buy. Big companies such as Facebook and Amazon heavily rely on high-quality recommendations to keep their users active. There exist numerous recommendation algorithms --- from the simple one based on common neighbor counting, to more complicated one based on deep learning models such as Graph Neural Networks. The common setup for those algorithms is that they represent the social network as a graph, and perform various operations on the nodes and edges.

In this project, you will implement a simple application to generate customized recommendations based on user queries. Specifically, consider that Facebook users in different countries want to follow new friends. They would send their queries to a Facebook server (i.e., main server) and receive the new friend suggestions as the reply from the same main server. Now since the Facebook social network is so large that it is impossible to store all the user information in a single machine. So we consider a distributed system design where the main server is further connected to many (in our case, two) backend servers. Each backend server stores the Facebook social network for different countries. For example, backend server A may store the user data of Canada and the US, and backend server B may store the data of Europe. Therefore, once the main server receives a user query, it decodes the query and further sends a request to the corresponding backend server. The backend server will search through its local data, identify the new friend to be recommended and reply to the main server. Finally, the main server will reply to the user to conclude the process.



The detailed operations to be performed by all the parties are described with the help of Figure 1. There are in total 5 communication end-points:

- Client 1 and Client 2: representing two different users, possibly in different countries
- Main server: responsible for interacting with the clients and the backend servers
- Backend server A and Backend server B: responsible for generating the new friend based on the query
    - For simplicity, user data is stored as plain text. Backend server A stores data1.txt and Backend server B stores data2.txt in their local file system.

The full process can be roughly divided into four phases (see also "DETAILED EXPLANATION" section), the communication and computation steps are as follows:

Bootup

1. [Computation]: Backend server A and B read the files data1.txt and data2.txt respectively, and construct a list of "graphs" (see "DETAILED EXPLANATION" for description of suggested data structures for graphs).
    - Assume a "static" social network where contents in data1.txt and data2.txt do not change throughout the entire process.
    - Backend servers only need to read the text files once. When Backend servers are handling user queries, they will refer to the internally represented graphs, not the text files.
    - For simplicity, there is no overlap of countries between data1.txt and data2.txt.
2. [Communication]: after step 1, Main server will ask each of Backend servers which countries they are responsible for.
3. [Computation]: Main server will construct a data structure to book-keep such information from step 2. When the client queries come, the main server can send a request to the correct Backend server.

Query

1. [Communication]: Clients send their queries to the Main server.
    - A client can terminate itself only after it receives a reply from the server (in the Reply phase).
    - Main server may be connected to both clients at the same time.
2. [Computation]: Once the Main server receives the queries, it decodes the queries and decides which backend server(s) should handle the queries.

Recommendation

1. [Communication] Main server sends a message to the corresponding backend server so that the Backend server can perform local computation to generate recommendations.

2. [Computation]: Backend server performs some operations on the graph for friend recommendations, which is based on the number of common neighbors. You need to implement the algorithm on Backend servers to count the number of common neighbors.
3. [Communication]: Backend servers, after generating the recommendations, will reply to Main server.

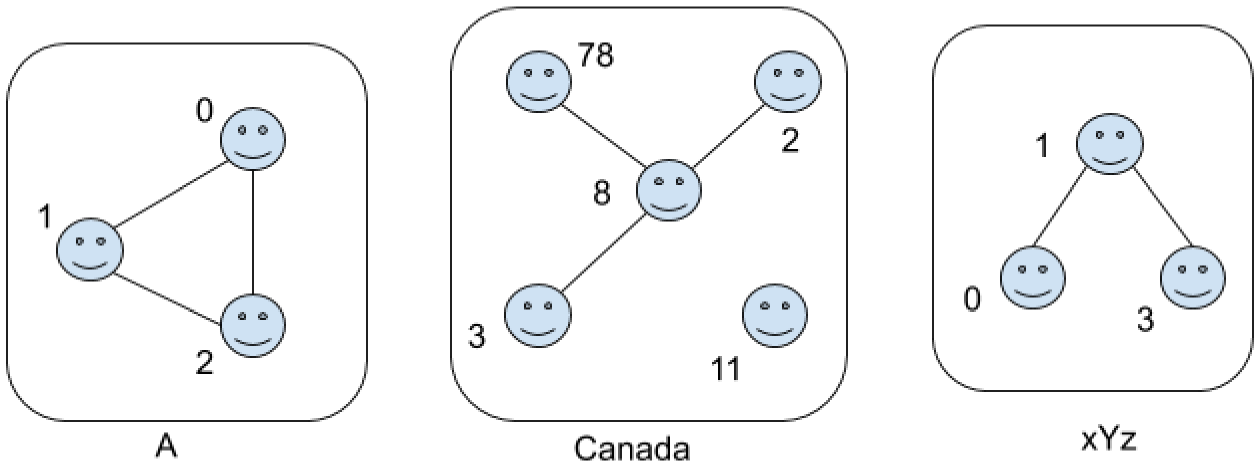
Reply

1. [Computation]: Main server decodes the messages from Backend servers and then decides which recommendations correspond to which client queries.
2. [Communication]: Main server prepares a reply message and sends it to the appropriate Client.
3. [Communication]: Clients receive the recommendation from Main server and display it. Clients should keep active for further inputted queries, until the program is manually killed.

The format of data1.txt or data2.txt is defined as follows.

```
<name of country 1>
<ID of user 1> <ID of neighbor 1 of user 1> … <ID of neighbor k
of user 1>
<ID of user 2> <ID of neighbor 1 of user 2> … <ID of neighbor k
of user 2>
...
<name of country 2>
...
```

Let's consider an example:



Let's say there are three countries, "A", "Canada" and "xYz". In country A, there are three users with ID 0, 1 and 2. In country Canada, there are five users with IDs 78, 2, 8, 3 and 11. In country xYz, there are three users with IDs 1, 0 and 3. Although both country A and country xYz have the same user ID 0 and 1, they are not the same user (See Assumption 7 below).

Assume data1.txt stores the user data for countries A and xYz, and data2.txt stores the user data for country Canada.

Example data1.txt:

```
A
0 1 2
1 2 0
2 1 0
xYz
1 0 3
0 1
3 1
```

Example data2.txt:

```
Canada
3 8
2 8
11
8 78 2 3
```

```
78 8
```

Assumptions on the data file:

1. We consider undirected connections between any pair of users. For example, if there is a connection from user 1 to user 0, then there must be a connection from user 0 to user 1.
2. There may be isolated users. That is, a user may not have any connection to other users (e.g., user 11 in country Canada).
3. Each user will have one line in the text file. Even though 11 in Canada has no other connections, there is still a line for 11 in data2.txt.
4. Country names are in letters. The length of the name can vary from 1 letter to at most 20 letters. There can be both capital and lowercase letters in the name. Country name does not contain any white spaces.
5. User IDs are represented by non-negative integer numbers.
6. Within the same country, user IDs do not need to be consecutive. I.e., if a country contains N users, their IDs do not need to be 0, 1, 2, …, N-1. See the case of Canada and xYz.
7. The pair (country name, user ID) uniquely identifies a user around the world.
   - Users in different countries may have the same ID.
   - Users in the same country do not have the same ID.
8. User ID may not start from 0. See the case of Canada.
9. The maximum possible user ID is (2^31 - 1). The minimum possible user ID is 0.
   - This ensures that you can always use int32 to store the user ID.
   - Backend servers may also want to re-index the user IDs when constructing the graph.
10. There are at most 10 countries in total.
11. There is no additional empty line(s) at the beginning or the end of the file. That is, the whole data1.txt and data2.txt do not contain any empty lines.
12. For simplicity, there is no overlap of countries between data1.txt and data2.txt.
13. The user IDs in data1.txt and data2.txt are separated by white space(s). That is, there will be at least one white space between two IDs, but there can also be multiple spaces.
14. A user will not connect to him/herself.
15. For a given user, his/her corresponding line in the text file may contain repeated neighbor ID.
    - For example, it is still a valid file if we replace the second line of data1.txt with “0 1 2 1”
16. The user IDs in the text are not sorted.
17. data1.txt and data2.txt will not be empty.
18. A country will have at least one user, and at most 100 users.

A sample data1.txt and data2.txt is provided for you as a reference. Please refer to the "DOWNLOAD SAMPLES" Section to download them. Other data1.txt and data2.txt will be used for grading, so you are advised to prepare your own files for testing purposes.

Source Code Files

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

1. servermain: You must name your code file: **servermain.c** or **servermain.cc** or **servermain.cpp** (all small letters). Also you must name the corresponding header file (if you have one; it is not mandatory) **servermain.h** (all small letters).

2. Backend-Server A and B: 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 name 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), e.g., serverA.c.

3. Client: The name for this piece of code must be **client.c** or **client.cc** or **client.cpp** (all small letters) and the header file (if you have one; it is not mandatory) must be called **client.h** (all small letters).

Note: Your compilation should generate separate executable files for each of the components listed above.

DETAILED EXPLANATION

Phase 1 (10 points) -- Bootup

All three server programs (Main server, Backend servers A & B) boot up in this phase. While booting up, the servers must display a boot up message on the terminal. The format of the boot up message for each server is given in the onscreen message tables at the end of the document. As the boot up message indicates, each server must listen on the appropriate port for incoming packets/connections.

As described in the previous section, the backend server needs to read the text file and convert the social network into graphs --- one graph per country. There are many ways to represent a graph. For example, adjacency matrix, adjacency list or Compressed Sparse Row (CSR) format. You need to decide which format to use based on the requirement of the problem. You can use **any** format as long as you can generate the correct recommendation.

For example, suppose Backend serve 2 re-indexed the Canada users as

Original user ID	Re-indexed user ID
3	0
2	1
11	2
8	3
78	4

Then the adjacency matrix for the graph will be:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Where the element at the i-th row and j-th column will be 1 if there is a connection between the (re-indexed) user i and user j.

Again, you are welcome to use other ways to represent the graph. Once Backend server finishes processing the files of data1.txt and data2.txt, Main server will ask Backend servers so that Main server knows which Backend server is responsible for which countries. The communication between Main server and Backend servers is using UDP. For example, Main server may use an unordered_map to store the following information:

```
std::unordered_map<std::string, int> country_backend_mapping;

country_backend_mapping["xYz"] = 0;

country_backend_mapping["Canada"] = 1;

country_backend_mapping["A"] = 0;
```

Above lines indicate that "xYz" and "A" stored in data1.txt in Backend server A (represented by value 0) and "Canada" is stored in data2.txt in Backend server B (represented by value 1).

Once the server programs have booted up, two client programs run. Each client displays a boot up message as indicated in the onscreen messages table. Note that the client code takes no input argument from the command line. The format for running the client code is:

```
./client
```

After running it, it should display messages to ask the user to enter a query country name and a query user ID (e.g., implement using std::cin):

```
./client
Enter country name:
Enter user ID:
```

For example, if the client 1 is booted up and asks for the friend recommendation for user ID 78 in Country Canada, then the terminal displays like this:

```
./client
Enter country name: Canada
Enter user ID: 78
```

After booting up, Clients establish TCP connections with Main server. After successfully establishing the connection, Clients send the input country name and user ID) to Main server. Once this is sent, Clients should print a message in a specific format. Repeat the same steps for Client 2.

Each of these servers and the main server have its unique port number specified in "PORT NUMBER ALLOCATION" section with the source and destination IP address as localhost/127.0.0.1/::1.

Clients, Main server, Backend server A and Backend server B are required to print out on screen messages after executing each action as described in the "ON SCREEN MESSAGES" section. These messages will help with grading in the event that the process did not execute successfully. Missing some of the on screen messages might result in misinterpretation that your process failed to complete. Please follow the exact format when printing the on screen messages.

Phase 2 (40 points) -- Query

In the previous phase, Client 1 and Client 2 receive the query parameters from the two users and send them to Main server over TCP socket connection. In phase 2, Main server will have to receive requests from two Clients. If the country name or user ID are not found, the main server will print out a message (see the "On Screen Messages" section) and return to standby.

For a server to receive requests from several clients at the same time, the function **fork()** should be used for the creation of a new process. Fork() function is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (***parent process***).

For a TCP server, when an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using accept(). After the connection with the client is successfully established, the accept() function returns a non zero descriptor for a socket called the child socket. The server can then fork off a process using fork() function to handle connection on the new socket and go back to waiting on the original socket. Note that the socket that was originally created, that is the parent socket, is going to be used only to listen to the client requests, and it is not going to be used for communication between client and Main server. Child sockets that are created for a parent socket have the identical well-known port number IP address at the server side, but each child socket is created for a specific client.

Through using the child socket with the help of fork(), the server is able to handle the two clients without closing any one of the connections.

Phase 3 (40 points) -- Recommendation

In this phase, each Backend server should have received a request from Main server. The request should contain a user ID and a country name. The backend server will generate one recommendation per request based on the common neighbor counting. If two nodes are adjacent to each other, they are neighbors. The common neighbors are the intersection between the neighbors of two nodes. The recommendation result should be either None or an ID of another user (the ID should be the original ID as stored in data1.txt or data2.txt, not the ID re-indexed by the backend server).

Take the request for user u at country C as an example. Consider the following three cases:

1. u is already connected to all the other users in country C. Then there is not any new user to be recommended. So Backend server should recommend None.
    a. If u is the only user in the graph, it should also recommend None.
2. u is not yet connected to some users in C. Let's denote those users **unconnected** to u as a set N. For each user n in N, count the number of common neighbors between u and n.
    a. If no n shares any common neighbor with u (i.e., $max_{n \in N}\ commonNeighbor(u, n) = 0$), then the Backend server will recommend a n with the highest degree. The degree of n is the number of nodes connected to n. Tie-breaker: if multiple users have the same highest degree, you should recommend the one with the smallest ID (again, original ID as stored in the text file).
    b. If some n shares some common neighbors with u. Then the Backend server will pick the n such that it has the most common neighbors with u (i.e., recommend $argmax_{n \in N}\ commonNeighbor(u, n)$). Tie-breaker: if multiple users have the

same highest number of common neighbors, you should recommend the one with the smallest ID (again, original ID as stored in the text file).

Recall the example in the "PROBLEM STATEMENT" section. When the user queries 1 in A (or 0 in A, or 2 in A), this corresponds to case 1. When the user queries 11 in Canada, this corresponds to case 2.a, and the recommendation should be user ID 8 (since 8 has degree 3, which is the highest).When the user queries 8 in Canada, this also corresponds to case 2.a, and the recommendation should be 11. When user queries 0 in xYz, this corresponds to case 2.b, and the recommendation should be 3 (in this case user 0 and 3 share one common neighbor that is user 1).

You may use the `std::set` data structure to find the number of common neighbors.

Phase 4 (10 points) -- Reply

At the end of Phase 3, the responsible Backend server should have the recommendation result ready. The result is the recommended user IDs (the original node index). The result should be sent back to the Main server using UDP. When the Main server receives the result, it needs to forward all the result to the corresponding Client using TCP. The clients will print out the recommended user ID and then print out the messages for a new request as follows:

```
...

Recommended User ID is 2.

-----Start a new request-----
Enter country name:
Enter user ID:
```

See the ON SCREEN MESSAGES table for an example output table.

<u>DOWNLOAD SAMPLES</u>

Samples of data1.txt and data2.txt for this project are available online for download. The data in these data.txt files are generated randomly for each download, but the structure and data type of the data.txt files are consistent. data1.txt and data2.txt are expected to be read and stored into serverA and serverB respectively.

The link to download a copy of data.txt files will be posted on DEN soon.

<u>PORT NUMBER ALLOCATION</u>

The ports to be used by the client and the servers 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		UDP: 30xxx
Backend-Server B		UDP: 31xxx
Main Server		UDP(with server): 32xxx TCP(with client): 33xxx
Client 1	TCP	
Client 2	TCP	

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: **<u>30319</u>** for the Backend-Server (A), etc.

Port number of all processes print port number of their own

<u>ON SCREEN MESSAGES</u>

Table 4. Backend-Server A on-screen messages

Event	**On-screen Messages**
Booting up (Only while starting):	The server A is up and running using UDP on port <server A port number>
Sending the country list that contains in “data1.txt” to Main Server:	The server A has sent a country list to Main Server
For friends searching, upon receiving the input query:	The server A has received request for finding possible friends of User<user ID> in <Country Name>
If we could not find this user ID in this country, send “not found” back to Main Server:	User<user ID> does not show up in <Country Name>
	The server A has sent “User<user ID> not found” to Main Server
If we find this user ID in this country, searching possible friends for this user and send result(s) back to Main Server:	The server A is searching possible friends for User<user ID> … Here are the results: User<user ID1>, User<user ID2>...
	The server A has sent the result(s) to Main Server

Table 5. Backend-Server B on-screen messages

Event	**On-screen Messages**
Booting up (Only while starting):	The server B is up and running using UDP on port <server B port number>
Sending the country list that contains in “data2.txt” to Main Server:	The server B has sent a country list to Main Server
For neighbor finding, upon receiving the input query:	The server B has received request for finding possible friends of User<user ID> in <Country Name>

If we could not find this user ID in this country, send “not found” back to Main Server:	User<user ID> does not show up in <Country Name>
	The server B has sent “User<user ID> not found” to Main Server
If we find this user ID in this country, searching possible friends for this user and send result(s) back to Main Server:	The server B is searching possible friends for User<user ID> … Here are the results: User<user ID1>, User<user ID2>...
	The server B has sent the result(s) to Main Server

Table 6. Main Server on-screen messages

Event	**On-screen Messages**
Booting up(only while starting):	The Main server is up and running.
Upon receiving the country lists from server A:	The Main server has received the country list from server A using UDP over port <Main server UDP port number>
Upon receiving the country lists from server B:	The Main server has received the country list from server B using UDP over port <Main server UDP port number>
List the results of which country serverA/serverB is responsible for:	Server A | Server B <Country Name 1> | <Country Name 3> <Country Name 2> |
Upon receiving the input from the client:	The Main server has received the request on User <user ID> in <Country Name> from client<client ID> using TCP over port <Main server TCP port number>
If the input country name could not be found, send the error message to the client:	<Country Name> does not show up in server A&B
	The Main Server has sent “Country Name: Not found” to client1/2 using TCP over port <Main server TCP port number>
If the input country name could be found, decide which server	<Country Name> shows up in server A/B

contains related information about the input userID and send a request to serverA/B	
	The Main Server has sent request from User <user ID> to server A/B using UDP over port <Main server UDP port number>
If we could find this user ID in the graph, the Main Server will receive the searching results from serverA/B and send them to client1/2	The Main server has received searching result(s) of User <user ID> from server<A/B>
	The Main Server has sent searching result(s) to client using TCP over port <Main Server TCP port number>
If we could not find this user ID in the graph, send the error message back to client	The Main server has received “User ID: Not found” from server <A/B>
	The Main Server has sent error to client using TCP over <Main Server UDP port number>

Table 7. Client 1 on-screen messages

Event	**On-screen Messages**
Booting up(only while starting)	Client1 is up and running
	Please enter the User ID: Please enter the Country Name:
After sending User ID to Main Server:	Client1 has sent User<user ID> and <Country Name> to Main Server using TCP
If input country not found	<Country Name> not found
If input User ID not found	User<user ID> not found
If input User ID and country can be found:	Client1 has received results from Main Server: User<user ID1>, User<user ID2> is/are possible friend(s) of User<user ID> in <Country Name>

Table 8. Client 2 on-screen messages

Event	**On-screen Messages**
Booting up(only while starting)	The client is up and running
	Please enter the User ID: Please enter the Country Name:
After sending User ID to Main Server:	Client2 has sent User<user ID> and <Country Name> to Main Server using TCP
If input country not found	<Country Name> not found
If input User ID not found	User<user ID> not found
If input User ID and country can be found:	Client2 has received results from Main Server: User<user ID1>, User<user ID2> is/are possible friend(s) of User<user ID> in <Country Name>

ASSUMPTIONS

1. You have to start the processes in this order: **Backend-server (A), Backend-server (B), Main-server, and Client 1, Client 2.**

2. The data1.txt and data2.txt files are created before your program starts.

3. 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.

4. You are allowed to use code snippets 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.

5. 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.

6. 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 developer
```

Identify the zombie processes and their process number and kill them by typing at the command-line:

```
kill -9 <process number>
```

REQUIREMENTS

1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table 3 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 client should keep running and ask to enter a new request after displaying the previous result, until the TAs manually terminate it by Ctrl+C. The backend servers and the Main server should keep 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.

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 use fork() or similar system calls to create concurrent processes is not mandatory if you do not feel comfortable using them. However, the use of fork() for the creation of a child process when a new TCP connection is accepted is mandatory and everyone should support it. This is useful when different clients are trying to connect to the same server simultaneously. If you don't use fork() in the Main server when a new connection is accepted, the Main Server won't be able to handle the concurrent connections.

8. 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 and we don't care.

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 or gedit 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 may need to include these header files in addition to any other header file you used:

```
#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

Along with your code files, include a **README file and a Makefile**. In the README file write:

- Your **Full Name** as given in the class list
- Your Student ID
- What you have done in the assignment.
- 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).
- The format of all the messages exchanged.

- Any idiosyncrasy of your project. It should say under what conditions the project fails, if any.
- 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 BE SUBJECT TO A SERIOUS PENALTY.

About the Makefile

Makefile Tutorial:

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

Makefile should support following functions:

Compile **all** your files and creates executables	make all
Run server A	make serverA
Run server B	make serverB
Run Main Server	make mainserver
Run client 1	./client
Run client 2	./client

TAs will first compile all codes using **make all**. They will then open 5 different terminal windows. On 3 terminals they will start servers A, B and Main Server using commands **make serverA**, **make serverB**, and **make mainserver**. **Remember that servers should always be on once started.** On the 5th terminal they will start the client as "**./client**". TAs will check the outputs for multiple queries. The terminals should display the messages specified above.

1. 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:

   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:

   ```
   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 the first command.

2. Do NOT include anything not required in your tar.gz file. Do NOT use subfolders. 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 drop box, 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 keep a history of all your submissions. If you make multiple submissions, we will grade your latest valid submission. Submission after the 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 Main Server. But the result is not received by the client because Main-server 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, especially 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 you add subfolders or compress files in the wrong way, you will lose 2 points each.

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

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

11. 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.

12. 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.

13. **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. Also, you will NOT get credit by repeating others' answers.

14. 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.

15. Your code will not be altered in any way 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 checks whether it works correctly or not. If your README is not consistent with the description, we will follow the description.

FINAL WORDS

1. Start on this project early. Hard deadline is strictly enforced. No grace periods. No grace days. No exceptions.

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. Check Piazza regularly for additional requirements and latest updates about the project guidelines. Any project changes announced on Piazza are final and overwrites the respective description mentioned in this document.

4. Plagiarism will not be tolerated and will result in an "F" in the course.