EE450 Socket Programming Project

Part3

Fall 2021

Due Date:

Friday, December 3, 2021 11:59PM

(Hard Deadline, Strictly Enforced)

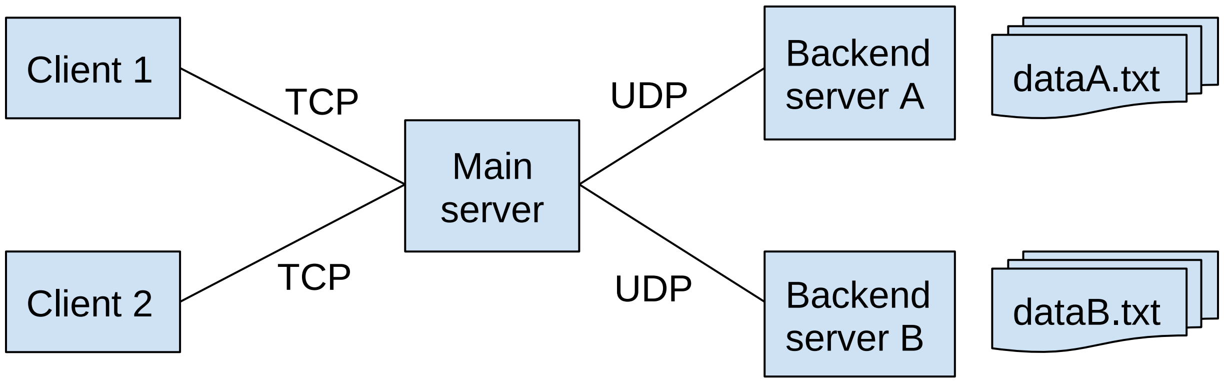
OBJECTIVE

The objective of project 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, email TA your questions or come by TA's office hours. **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 interests and mutual friends, 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 a social media platform that helps users meet new friends that live nearby (within the same state). They would send their queries to a social media server (i.e., main server) and receive the new friend suggestions as the reply from the same main server. Now since a social network is so large that it is impossible to store all the user information in a single machine. Thus, 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 social network for different states. For example, backend server A may store the user data of California and the Idaho, and backend server B may store the data of New York and Massachusetts. 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(s) 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 endpoints:

- Client 1 and Client 2: representing two different users, possibly in different states
- 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 dataA.txt and Backend server B stores dataB.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 dataA.txt and dataB.txt respectively and store the information in data structures.
    - Assume a "static" social network where contents in dataA.txt and dataB.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 data structures, not the text files.
        - For simplicity, there is no overlap of states between dataA.txt and dataB.txt.
2. [Communication]: after step 1, Main server will ask each of Backend servers which states they are responsible for. Backend servers reply with a list of states to the main server.
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]: Each client sends a query (a state name and a user ID) 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 computation to generate recommendations.
2. [Computation]: Once the query user ID is received, Backend server recommends users who belong in the same groups as the query user. You need to implement an algorithm on Backend servers to find all users that belong to the same social media groups as the query user.
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 recommendation correspond to which client query.
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 dataA.txt or dataB.txt is defined as follows. Below the name of state 1, there are multiple lines representing different social media groups. Each group (a line) contains the users that have the similar interests, and different lines represents different kinds of groups and different interests. A user is in at least one group and may in multiple groups.

```
<name of state 1>
<ID of user 1-1>,<ID of user 1-2>,...,< ID of user 1-k1>
<ID of user 2-1>,<ID of user 2-2>,...,<ID of user 2-k2>
...
<name of state 2>
...
```

Let’s consider an example:

Let’s say there are three states, “A”, “California” and “xYz”. In state A, there are three users with ID 0, 1 and 2. In state California, there are five users with IDs 78, 2, 8, 3 and 11. In state xYz, there are three users with IDs 1, 0 and 3. Although both state A and state xYz have the same user

ID 0 and 1, they are not the same user (See Assumption 3 below). In state A, user 0 and 1 are in the same social media group (this can be viewed as being part of the same Facebook group for example), and user 1 and 2 are in another group. Assume dataA.txt stores the user data for states A and xYz, and dataB.txt stores the user data for state California.

Example dataA.txt:

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

Example dataB.txt:

```
California
3,8
2,8
11
8,78,2,3
78,8
```

Assumptions on the data file:

1. A user is in at least one social media group and may be in multiple groups. There may be only one user in a group. (e.g., user 11 in state California).
2. There are at most 100 groups in a state.
3. The pair (state name, user ID) uniquely identifies a user around the world.
    - ○ Users in different states may have the same ID.
    - ○ Users in the same state do not have the same ID.
4. State names are letters. The length of a state name can vary from 1 letter to at most 20 letters. It may contain only capital and lowercase letters but does not contain any white spaces or other characters. State names "Abc" and "abc" are different.
5. There are at most 10 states in total.
6. User IDs are non-negative integer numbers and are separated by a comma in a line. 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.
7. Within the same state, user IDs do not need to be consecutive and may not start from 0.

i.e., if a state contains N users, their IDs do not need to be 0, 1, 2, …, N-1. See the case of California and xYz.

8. There is no additional empty line(s) at the beginning or the end of the file. That is, the whole dataA.txt and dataB.txt do not contain any empty lines.
9. For simplicity, there is no overlap of states between dataA.txt and dataB.txt.
10. There is no repeated user ID in a group.
11. The user IDs in the text are not sorted.
12. dataA.txt and dataB.txt will not be empty.
13. A state will have at least one user, and at most 100 users.

An example dataA.txt and dataB.txt is provided for you as a reference. Other dataA.txt and dataB.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. Main Server: 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). **There should be only one client file!!!**

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.

Backend servers should boot up first. A backend server needs to read the text file and store the state names and user IDs in appropriate data structures. There are many ways to store them, such as dictionary, array, vector, etc. You need to decide which format to use based on the requirement of the problem. You can use **any** format if you can generate the correct recommendation.

Main server then boots up after Backend servers are running and finish processing the files of dataA.txt and dataB.txt. Main server will request Backend servers for state lists so that Main server knows which Backend server is responsible for which states. 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 as in part 2.

```
std::unordered_map<std::string, int> state_backend_mapping;
state_backend_mapping["xYz"] = 0;
state_backend_mapping["California"] = 1;
state_backend_mapping["A"] = 0;
```

Above lines indicate that "xYz" and "A" stored in dataA.txt in Backend server A (represented by value 0) and "California" is stored in dataB.txt in Backend server B (represented by value 1). Again, you could use **any** data structure or format to store the information.

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 state name and a query user ID (e.g., implement using std::cin):

```
./client
…
Enter state name:
Enter user ID:
```

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

```
./client
…
Enter state name: California
Enter user ID: 78
```

After booting up, Clients establish TCP connections with Main server. After successfully establishing the connection, Clients send the input state 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.

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 if 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 state 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***). This is the same as in Project Part 1.

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 can 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 state name and a user ID. A backend server will generate one recommendation per request based on users that are in common groups as the provided user ID. If a user is in the same social media group as the query user, it should be taken as potential friends and be recommended to the query user. If the query user is in different groups, all users in these groups except the query user should be recommended. The recommendation result could be either None or ID(s) of other users.

Recall the example in the “PROBLEM STATEMENT” section. When the user queries 1 in A, user 0 and 2 should be recommended. When the user queries 11 in California, none of user can be recommended. When the user queries 0 in xYz, user 3 should be recommended. There should be no repeated user IDs in recommendation results. For example, when the user queries 78 in California, the recommendation result should be user 2, 3, 8, rather than 2, 3, 8, 8.

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 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 IDs and then print out the messages for a new request as follows:

```
...

User 2, User 3, user 8 is/are possible friend(s) of User 78 in
Canada

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

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

PORT NUMBER ALLOCATION

The ports to be used by the client and the servers are specified in the following table:

Table 1. 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: **30319** for the Backend-Server (A), etc. Port number of all processes print port number of their own.

ON SCREEN MESSAGES

Table 2. Backend-Server A on-screen messages

Event	**On-screen Messages**
Booting up (Only while starting):	Server A is up and running using UDP on port <server A port number>
Sending the state list that contains in "dataA.txt" to Main Server:	Server A has sent a state list to Main Server

For friends searching, upon receiving the input query:	Server A has received a request for finding possible friends of User<user ID> in <State Name>
If this user ID cannot be found in this state, send “not found” back to Main Server:	User<user ID> does not show up in <State Name>
	Server A has sent “User<user ID> not found” to Main Server
If this user ID is found in this state, searching possible friends for this user and send result back to Main Server:	Server A found the following possible friends for User<user ID> in <State Name>: <user ID1>, <user ID2>...
	Server A has sent the result to Main Server

Table 3. Backend-Server B on-screen messages

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

Table 4. Main Server on-screen messages

Event	**On-screen Messages**
Booting up(only while starting):	Main server is up and running.
Upon receiving the state lists from Server A:	Main server has received the state list from server A using UDP over port <Main server UDP port number>
Upon receiving the state lists from Server B:	Main server has received the state list from server B using UDP over port <Main server UDP port number>
List the results of which state server A/B is responsible for:	Server A <State Name 1> < State Name 2> Server B < State Name 3>
Upon receiving the input from the client:	Main server has received the request on User <user ID> in <State Name> from client <client ID> using TCP over port <Main server TCP port number>
If the input state name could not be found, send the error message to the client:	<State Name> does not show up in server A&B
	Main Server has sent “<State Name>: Not found” to client <client ID> using TCP over port <Main server TCP port number>
If the input state name could be found, decide which server contains related information about the input state and send a request to server A/B	<State Name> shows up in server <A or B>
	Main Server has sent request of User <user ID> to server A/B using UDP over port <Main server UDP port number>
If this user ID is found in a backend server, Main Server will receive the searching results from server A/B and send them to client1/2	Main server has received searching result of User <user ID> from server<A or B>
	Main Server has sent searching result(s) to client <client ID> using TCP over port <Main Server TCP port number>

If this user ID cannot be found in a backend server, send the error message back to client	Main server has received “User <user ID>: Not found” from server <A or B>
	Main Server has sent message to client <client ID> using TCP over <Main Server TCP port number>

Table 5. Client 1 or Client 2 on-screen messages

Event	**On-screen Messages**
Booting up(only while starting)	Client is up and running
	Enter state name: Enter user ID:
After sending User ID to Main Server:	Client has sent < State Name> and User<user ID> to Main Server using TCP over port <dynamic TCP port>
If input state not found	<State Name>: Not found
If received message from main server saying that input User ID not found	User <user ID>: Not found
If input User ID and state can be found and the result is received:	User<user ID1>, User<user ID2>, … is/are possible friend(s) of User<user ID> in <State Name>
After the last query ends:	-----Start a new request----- Enter state name: Enter user ID:

ASSUMPTIONS

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

2. The dataA.txt and dataB.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 can 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>
   ```

7. You may use the following command to double check the assigned TCP and UDP port numbers:
   ```
   sudo lsof -i -P -n
   ```

REQUIREMENTS

1. Do not hardcode the TCP 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 client should keep running and ask to enter a new request after displaying the previous result, until the TA 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 commandline 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 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**.
Submissions without README and Makefile will be subject to a serious penalty.

In the README file write:

- Your **Full Name** as given in the class list
- Your Student ID
- Briefly summarize what you have done in the assignment. (Please do not repeat the project description).
- List all your code files and briefly summarize their fulfilled functionalities. (Please do not repeat the project description).
- The format of all the messages exchanged between servers.
- 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.)

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
Compile server A	make serverA
Compile server B	make serverB
Compile Main Server	make servermain

Compile client	make client
Run Server A	./serverA
Run Server B	./serverB
Run Main Server	./servermain
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 three terminals they will start servers A, B and Main Server using commands **./serverA**, **./serverB**, and **./servermain**. **Remember that servers should always be on once started.** On another two 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.tar.gz (all small letters) e.g. an example filename would be ee450_nanantha.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 such as data files!!! Only include the required source code files, Makefile and the README file. Now run the following commands:

   ```
   tar cvf ee450_yourUSCusername.tar *
   gzip ee450_yourUSCusername.tar
   ```

   Now, you will find a file named "ee450_yourUSCusername.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!

1. Upload "ee450_yourUSCusername.tar.gz" to Blackboard -> Assignments. After the file is submitted, you must click on the "submit**"** button to submit it. If you do not click on "submit", the file will not be submitted.

2. Blackboard 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.

3. Please consider all kinds of possible technical issues and do expect a huge traffic on the Blackboard website very close to the deadline which may render your submission or even access to Blackboard unsuccessful.

4. 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!

5. After submitting, 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.

6. You have sufficient 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 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 10 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 15 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 data file path is not the same as the code files, you will lose 5 points.

10. Do not submit datafile (three .txt files) used for test, otherwise, you will lose 10 points.

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

12. Detailed points assignments for each functionality will be posted after finishing grading.

13. The minimum grade for an on-time submitted project is 10 out of 100, the submission includes a working Makefile and a README.

14. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend plenty of time on this project and it doesn't even compile, you will receive only 10 out of 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 project description, we will follow the project 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 Blackboard (Announcements and Discussion Board) regularly for additional requirements and latest updates about the project guidelines. Any project changes announced on Blackboard 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.