EE450 Socket Programming Project, Fall 2021 Due Date : Sunday, Nov 28, 11:59PM (Midnight)

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

Hard Deadline (Strictly enforced, No submissions are accepted afterwards)

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.

Problem Statement:

Social networks are nowadays in every moment of our lives. The information built upon interactions among people has led to different types of applications. Crowdsourced apps such as Uber, Lyft, Waze use information for navigation purposes. Other apps such as dating apps provide matching algorithms to connect users who share similar behaviours and increase their compatibility chances for future success. In this project we shall implement a simplified version of a matching app that'll help us understand how matching systems work in the real world. Specifically, you'll be given a network topology consisting of social connections between different users in the network. This will consist of nodes representing users and the edges between them. Beside social network topology, you will also be given a database consisting of compatibility test scores. This database will be in plain text and consist of multiple key (the user), value (the corresponding score) pairs.

In this project, you will implement a model of a social matching service where two clients issue a request for finding their compatibility. This request will be sent to a Central Server which in turn interacts with three other backend servers for pulling information and data processing. The Central server will connect to the Topology server (server T) which has the user social network information. Central server has to connect as well to the Score server (server S) which stores the compatibility scores for each user. Finally, the server will use the network topology and scores to generate a graph that connects both users, and provide the smallest matching gap between them. The procedure to complete this task is provided in phase 2's description. Both the matching gap and the graph generated will be sent back to both clients.

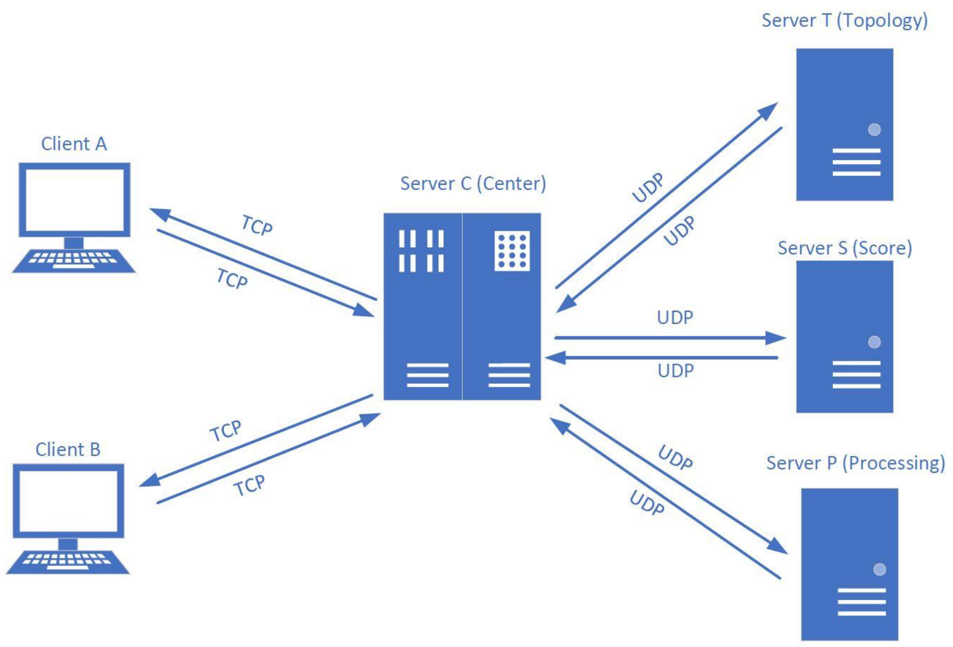

Figure 1. Illustration of the network

Server T has access to a database file named edgelist.txt, and Server S has access to a database file named scores.txt. Both clients and the Central server communicate over a TCP connection while the communication between Central and the Back-Servers T, S & P 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. Central: You must name your code file: **central.c** or **central.cc** or **central.cpp** (all small letters). Also you must include the corresponding header file (if you have one; it is not mandatory) central.h (all small letters).
2. Back-Server T, S and P: 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. T or S or P), 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. ClientA: 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. ClientB: 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).

Application workflow phase Description:

Phase 1: (30 points)

Phase 1A: establish the connections between the Clients and Server C

All four server programs (Central Server, Topology Server, Score Server and Processing Server) 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 end 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). The two input usernames from the two clients could calculate the matching score between these two users, which will be described in the next section. The format for running each client code is:

The command for the client A should be

```
./clientA <username>
```

The command for the client B should be

```
./clientB <username>
```

The usernames from the ClientA and ClientB are the inputs for computing the matching gap between themselves. As an example, to find the matching gap between Victor and Oliver, the two clients should be run as follows

```
./clientA Victor

./clientB Oliver
```

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 1A, and we now proceed to Phase 1B.

Phase 1B: establish the connections between Server C and all other backend servers as shown in the figure 1.

In Phase 1A, you read what should be sent from ClientA and ClientB to Server C over the TCP connections. In Phase 1B, Server C will send messages to the three back-servers (Server C, Server T and Server P) 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.

This ends Phase 1B.

Phase 2: (40 points)

After receiving messages from both clients, server C will firstly contact server T (where the social network graph is stored) and server S (where compatibility test scores are stored) to retrieve related data to compute the final results. After server C receives related data from both server T and S, server C will forward these to server P. Server P will then use this information to find a social network path that has the smallest matching gap. Let's take a look at an example below. Figure 2 and Table 1 demonstrate examples of data that can be stored in server T and server S.

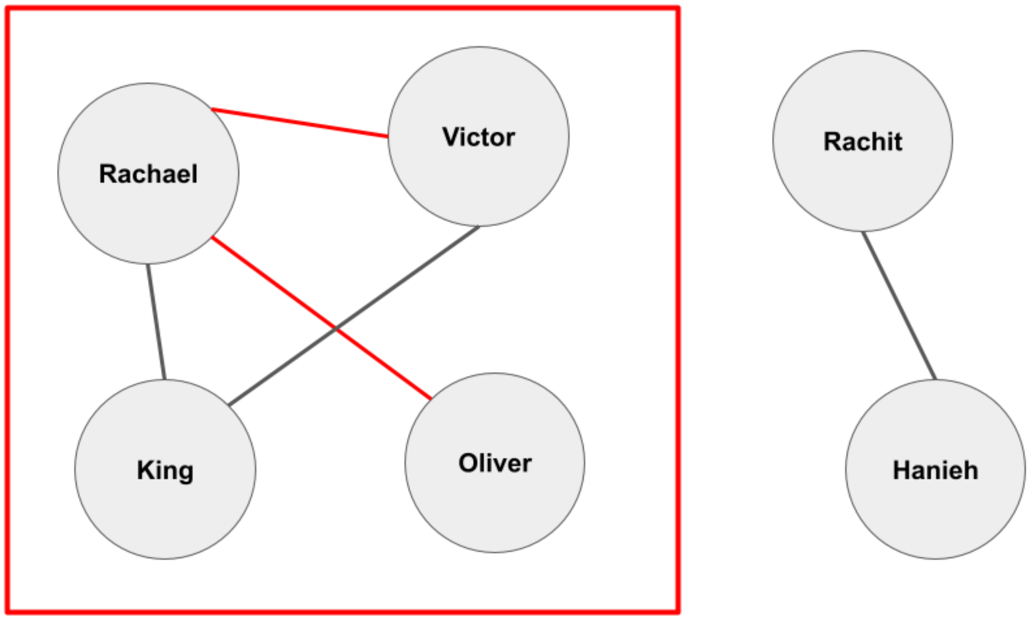

Figure 2: An example of graphs stored in Server T

Table 1: An example of compatibility test scores stored in Server S	
Racheal	43
King	3
Oliver	94
Victor	8
Rachit	129
Hanieh	49

If the clients' usernames are Victor and Oliver, server T should send the graph in the red square to server C. The highlighted graph contains both Oliver and Victor. Server C should then ask Server S for the necessary compatibility test scores. In this example, server S shall send compatibility test scores of Rachael, Victor, King and Oliver to server C. After server C received all necessary messages, it will forward them to server P. Server P will find a social connection that bridges Victor and Oliver with the smallest compatibility gap. Server P should also be able to realize connecting certain people are impossible(King and Rachit). Given two compatibility test scores $S_1$and S_2 , matching gap between two people are computed using equations below:

$$matching\ gap = \frac{|S_1 - S_2|}{S_1 + S_2}.$$

Them atching gap between Victor and Rachael is:$\frac{|43-8|}{43+8}$= 0.686 and matching gap between Rachael and Oliver is $\frac{|94-43|}{94+43}$= 0.372. If there are more than two edges connecting two people, the matching gap of that path will be the summation of the matching gap of all the edges. The matching gap of connecting Victor and Oliver through Rachael is 1.06.

Phase 3: (30 points)

After processing the results in the backend server P, results have to be sent back to the Central server. As mentioned in phase 2, the processing server will calculate the matching gap between the two clients' usernames requested.

Using the previous example, we will have as a result a graph with Victor --- Rachel --- Oliver and the matching gap equal to 1.06 .This information will be sent to the Central server and will be forwarded to each of the clients. The format can be seen on the on-screen messages table shown later.

If there is no path that connects both username's requests on the network, the processing server has to provide to the central server that no matching can occur.

Phase 4: (10 points extra, not mandatory)

If you want to earn 10 extra points, you can implement an extra operation where client B provides two usernames and the system compares both of their matching gaps with the client A username. This operation cannot be done separately, i.e, it should be on the same command line "`./clientB <usernameX> <usernameY>`", You need to find both matching gaps and provide the results to client A and client B.

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 2. Static and Dynamic assignments for TCP and UDP ports.		
Process	**Dynamic Ports**	**Static Ports**
Backend-Server (T)	-	1 UDP, 21000+xxx
Backend-Server (S)	-	1 UDP, 22000+xxx

Backend-Server (P)	-	1 UDP, 23000+xxx
Central (C)	-	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 3. Backend-Server T on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	"The ServerT is up and running using UDP on port <port number>."
Upon Receiving the request from Central server:	"The ServerT received a request from Central to get the topology."
After sending the results to the Central server (C):	"The ServerT finished sending the topology to Central."

ON SCREEN MESSAGES: **Table 5. Backend-Server S on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	"The ServerS is up and running using UDP on port <port number>."
Upon Receiving the request from Central server:	"The ServerS received a request from Central to get the scores."

After sending the results to the Central server (C):	“The ServerS finished sending the scores to Central.”

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

Event	On Screen Message (inside quotes)
Booting Up (Only while starting):	“The ServerP is up and running using UDP on port <port number>.”
Upon Receiving the input string:	“The ServerP received the topology and score information.”
After sending the results to the Central server (C):	“The ServerP finished sending the results to the Central.”

ON SCREEN MESSAGES:
Table 7. Central on screen messages

Event	On Screen Message (inside quotes)
Booting Up (only while starting):	“The Central server is up and running.”
Upon Receiving the username from the client A:	“The Central server received input=<INPUT1> from the client using TCP over port <port number>.”
Upon Receiving the username from the client B:	“The Central server received input=<INPUT2> from the client using TCP over port <port number>.”
After querying each Backend-Servers	“The Central server sent a request to Backend-Server T”. “The Central server sent a request to Backend-Server S” “The Central server sent a processing request to Backend-Server P.”
After receiving result from backend server i): i is one of T or S	“The Central server received information from Backend-Server <i> using UDP over port <port number>.”

After receiving result from backend server P	“The Central server received the results from backend server P.”
After sending the final result to the client j): j is either A or B	“The Central server sent the results to client <j>.”

ON SCREEN MESSAGES: Table 8. Client A on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The client is up and running.”
Upon sending the input and function to AWS	"The client sent <INPUT1> to the Central server.”
After receiving the result from Central server	“Found compatibility for <INPUT1> and <INPUT2>: <INPUT1> --- <USERY> ---<USERX> --- <INPUT2> Matching Gap : <VALUE>”
If no compatibility between both	“Found no compatibility for <INPUT1> and <INPUT2>”

ON SCREEN MESSAGES: Table 9. Client B on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The client is up and running.”
Upon sending the input and function to AWS	"The client sent <INPUT2> to the Central server.”
After receiving the result from Central server	“Found compatibility for <INPUT2> and <INPUT1>: <INPUT2> --- <USERX> ---<USERY> --- <INPUT1> Matching Gap : <VALUE>”
If no compatibility between both	“Found no compatibility for <INPUT2> and <INPUT1>”

Example Output to Illustrate Output Formatting:

User Inputs are Victor and Oliver. Topologies and scores are shown in the figure x and table 1.

For operation search:

Backend-Server T Terminal:

```
The ServerT is up and running using UDP on port 21319.
The ServerT received a request from Central to get the topology.
The ServerT finished sending the topology to Central.
```

Backend-Server S Terminal:

```
The ServerS is up and running using UDP on port 22319.
The ServerS received a request from Central to get the scores.
The ServerS finished sending the scores to Central.
```

Backend-Server P Terminal:

```
The ServerP is up and running using UDP on port 23319.
The ServerP received the topology and score information.
The ServerP finished sending the results to the Central.
```

Central Server Terminal:

```
The Central server is up and running.
The Central server received input=”Victor” from the client using TCP over port 25319.
The Central server received input=”Oliver” from the client using TCP over port 26319.
The Central server sent a request to Backend-Server T.
The Central server received information from Backend-Server T using UDP over port
24319.
The Central server sent a request to Backend-Server S.
The Central server received information from Backend-Server S using UDP over port
24319.
The Central server sent a processing request to Backend-Server P.
The Central server received the results from backend server P.
The Central server sent the results to client A.
The Central server sent the results to client B.
```

Client A Terminal:

```
The client is up and running.

The client sent Victor to the Central server.
Found compatibility for Victor and Oliver:
Victor --- Rachael --- Oliver
Compatibility score: 1.06
```

Client B Terminal:

```
The client is up and running.

The client sent Oliver to the Central server.
Found compatibility for Victor and Oliver:
Oliver --- Rachael --- Victor
Compatibility score: 1.06
```

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
`make serverC`	**Run**s central server
`make serverT`	**Run**s server T
`make serverS`	**Run**s server S
`make serverP`	**Run**s server P
`./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 C, T, S and P using commands `make serverC`, `make serverT`, `make serverS`, and `make serverP`. On other two terminals (three if you do the extra credit), they will start client processes using `./client <name>`. **Remember that servers should always be on once started.** Client 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 the 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.