EE450 Socket Programming Project

Fall 2019

Due Date:

Friday, November 29, 2019 11:59 PM

(Hard Deadline, Strictly Enforced)

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

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.

OBJECTIVE

The objective of this assignment is to familiarize you with UNIX socket programming. This assignment is worth **10%** of your overall grade in this course. **It is an individual assignment and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).**

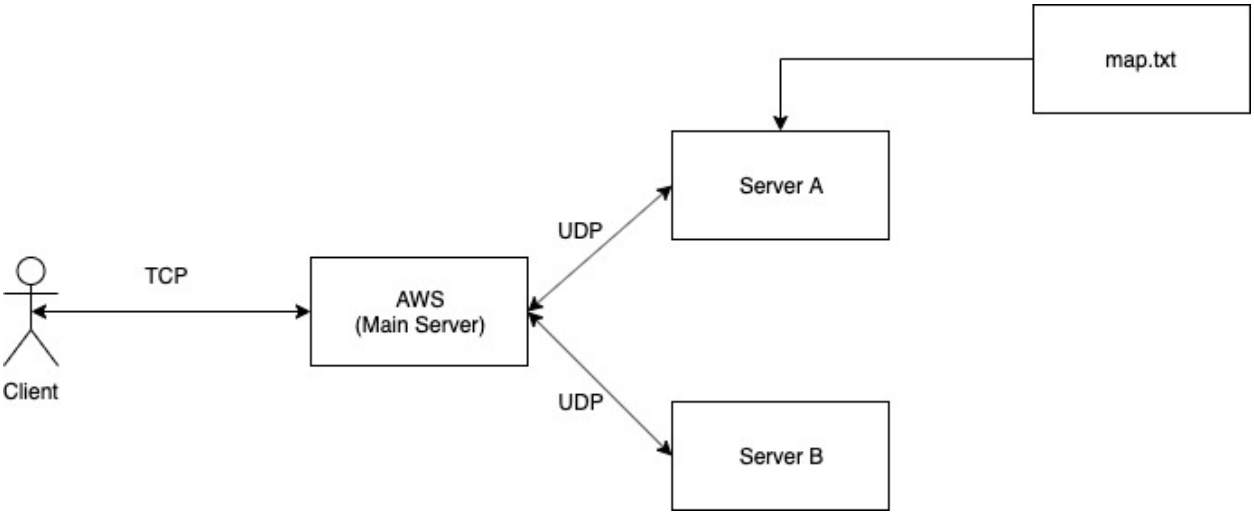
If you have any doubts/questions, post your questions on Piazza. **You must discuss all project related issues on Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points to the project.

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

PROBLEM STATEMENT

Many network related applications require fast identification of the shortest path between a pair of nodes to optimize routing performance. Given a weighted graph $G(V, E)$ consisting of a set of vertices V and a set of edges E, we aim at finding the path in G connecting the source vertex v_1 and the destination vertex v_n, such that the total edge weight along the path is minimized.

Dijkstra Algorithm is a procedure of finding the shortest path between a source and destination nodes. This algorithm will be discussed later in the semester. In this project, you will implement a distributed system to compute shortest path based on client's query. Suppose the system stores maps of a city, and the client would like to obtain the shortest path and the corresponding transmission delay between two points in the city. The figure below summarizes the system architecture. The distributed system consists of three computation nodes: a main server (AWS), connected to two backend servers (Server A and Server B). On the backend server A, there is a file named map.txt storing the map information of the city. The AWS server interfaces with the client to receive his query and to return the computed answer. The backend servers, A and B, perform the actual shortest path and transmission delay computation based on the message forwarded by AWS server.



Detailed computation and communication steps performed by the system is listed below:

1. [Communication] Client -> AWS: client sends the map ID, the source node in the map and the transmission file size (unit: bit) to AWS via TCP.

2. [Communication] AWS -> ServerA: AWS forwards the map ID and source node to serverA via UDP.

3. [Computation] ServerA reads map information from map.txt, uses Dijkstra to find the shortest path from input source to all the other nodes and print them out in a pre-defined format.

4. [Communication] ServerA -> AWS: ServerA sends the outputs of Dijkstra to AWS.

5. [Communication] AWS -> ServerB: AWS sends to ServerB the file size as well as the outputs of ServerA.

6. [Computation] ServerB calculates the transmission delay, propagation delay and end to end delay for each path.

7. [Communication] ServerB -> AWS: ServerB sends the calculated delay values to AWS.

8. [Communication] AWS -> client: AWS sends to client the shortest path and delay results, and client prints the final results.

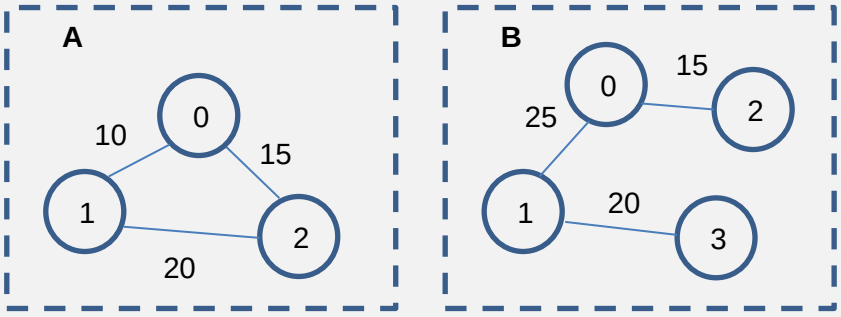
The map information of the city is stored in a file named map.txt, stored in ServerA. The map.txt file contains information of multiple maps (i.e. graphs), where each map can be considered as a community of the city. Within each map, the edge and vertex information are further specified, where an edge represents a communication link. We assume edges belonging to the same map have identical propagation speed and transmission speed.

The format of map.txt is defined as follows:

```
<Map ID 1>
<Propagation speed>
<Transmission speed>
<Vertex index for one end of the edge> <Vertex index for the
other end> <Distance between the two vertices>
… (Specification for other edges)
<Map ID 2>
…
```

Example:

```
A
10
10
0 1 10
0 2 15
1 2 20
B
20
10
0 1 25
0 2 15
1 3 20
…
```



Note:

1. For each map, number of vertices is between 1 and 10
2. We consider undirected, simple graphs:
   a. There are no repeated edges or self-loops
   b. An edge (p,q) in the graph means p and q are mutual neighbors
3. Units:
   a. Propagation speed: km/s
   b. Transmission speed: Bytes/s
   c. Distance: km

We provide a sample map.txt for you as a reference. We will use another map.txt for grading, so you are advised to prepare your own map files for testing purposes.

Source Code Files

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

1. AWS: The server can be viewed as a much simplified Amazon Web Service server. You must name your code file: **aws.c** or **aws.cc** or **aws.cpp** (all small letters). Also you must name the corresponding header file (if you have one; it is not mandatory) **aws.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).

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)

Boot up: 5 points
Client -> AWS: 5 points

All three server programs (AWS, Back-end Server 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.

Once the server programs have booted up, the client program runs. The client displays a boot up message as indicated in the onscreen messages table. Note that the client code takes input arguments from the command line. The format for running the client code is:

```
./client <Map ID> <Source Vertex Index> <File Size>
```

(Between two input arguments, there should be a space)

For example, if the client wants to calculate the end to end delay of each shortest path from source vertex 1 to any other vertices in Map A, with file size of 10 bits, then the command should be:

```
./client A 1 10
```

After booting up, the client establishes TCP connections with AWS. After successfully establishing the connection, the client sends the input (map ID, source vertex index and file size) to AWS. Once this is sent, the client should print a message in a specific format. This ends Phase 1 and we now proceed to Phase 2.

Phase 2 (40 points+20 points)

In Phase 1, you read the input arguments from the client and send them to the AWS server over a TCP connection. Now in phase 2, this AWS server will send selected input value(s) to the back-server A and B, depending on their functionalities.

The communication between the AWS server and both the backend servers is via **UDP**. The port numbers used by servers A and B are specified in Table 3. Since both the servers will run on the same machine in our project, both have the same IP address (the IP address of localhost is usually 127.0.0.1).

In Phase 2A, {map construction} operation will be implemented. In Phase 2B, the {path finding} operation will be implemented. In Phase 2C, the {calculation} operation will be implemented (see Table. 2).

Note that all messages required to be printed for the client, AWS server and backend server A, B can be found in the format given in the **ON SCREEN MESSAGES table**. You can also check the example output in the following part for reference. Please try your best to **follow the exact format when you print out the result**.

You are not required to have exact named functions (map construction, path finding, and calculation) in your code. These operation is named and divided to make the process clear. And as shown in the instruction, Phase 2A and 2B will together contribute to 40 points when your project is graded.

Phase 2A (2A + 2B = 40 points)

Phase 2A begins when the backend server A boots up. Afterwards, server A will execute the {map construction} operation and read the map data file (map.txt, see the problem statement section). Reading this file will allow A to construct a list of maps. Every map will be identified by its unique ID. After Phase 2A, backend server A will keep this list so that the client can query on any map in this list. AWS will then wait for appropriate user input to let server A perform the {path finding} operation.

Phase 2B (2A + 2B = 40 points)

Phase 2B is initiated when AWS receives all required data from the client. The AWS will forward the <Map_Name>, <Start_Node> from the client to backend server A. Backend server A will have a list of maps in its memory upon finishing Phase 2A. After the backend server A receives the two parameters, it will perform {path finding} operation (see Table. 2) by Dijkstra algorithm to find the shortest path from the <Start_Node> to all other node in the same map.

Once finished, the server A will print out a table to demonstrate the minimum length found. The table will include 2 columns (see an example output table in the "ON SCREEN MESSAGES" table). The first column will be the destination node index, the second column will be the minimum length from start node to the destination. Please format your output so the table is clear and

readable. Then server A will send all required map information back to AWS.

Phase 2C (20 points)

Phase 2C starts when the AWS receives the corresponding map information from server A. The AWS server will forward both the result and the <File_Size> to backend server B.

The backend server B is a computing server. It performs the operation {calculation} (see Table 2) based on the data sent by the AWS server (please think carefully on your own what information is necessary from AWS, to enable B to complete the calculation). With the given <File_Size>, the backend server B will compute the delay for the start node to send the file to all other node. The server B will compute both T_{trans} and T_{prop}. The final step for Phase 2C is sending 3 delay results (T_{trans}, T_{prop}, *end to end delay*) back to AWS server.

Table 2. Server Operations

Map Construction	In this operation, you will read the data file (map.txt) to construct a list of maps (i.e., graphs). For each map, you will book-keep the edge information including length, propagation speed and bit rate.
Path Finding	In this operation, you will find the path of minimum length from a given start vertex to all other vertices in the selected map, using Dijkstra algorithm. You also need to print out a simple 2-columns table to show the result.
Calculation	In this operation, you compute the transmission delay (in ms), the propagation delay (in ms), and the end-to-end delay (in ms) for transmitting a file of given size in the selected map.

Phase 3 (15 points)

At the end of Phase 2C, backend server B should have the calculation results ready. Those results should be sent back to the AWS using UDP. When the AWS receives the calculation results, it needs to forward all the result to the client using TCP. The results should include minimum path length to all destination node and 3 delays to transfer the file to corresponding destinations. The clients will print out a table to display the response. The table should include 5 columns. One for destination node index, one for path length and the other three for delays.

Make sure you round the results of three delay time to the 2nd decimal point for display. Round the result **after** summing T_{trans} and T_{prop} along a path. Do not sum rounded T_{trans} and rounded T_{prop} as your total delay.

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 3. Static and Dynamic assignments for TCP and UDP ports		
Process	**Dynamic Ports**	**Static Ports**
Backend-Server (A)	-	1 UDP, 21xxx
Backend-Server (B)	-	1 UDP, 22xxx
AWS	-	1 UDP, 23xxx 1 TCP with client, 24xxx
Client	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: **21319** for the Backend-Server (A), etc.

ON SCREEN MESSAGES

Table 4. Backend-Server A on screen messages

Event	On Screen Message
Booting up (Only while starting):	`The Server A is up and running using UDP on port <port number>.`

For map construction, after constructing the list of maps	`The Server A has constructed a list of <number> maps:`
	`----------`
	`Map ID  Num Vertices  Num Edges`
	`----------`
	`A  9  15`
	`B  7  25`
	`…`
	`----------`

For path finding, upon receiving the input query:	`The Server A has received input for finding shortest paths: starting vertex <index> of map <ID>.`

For path finding, after finishing Dijkstra:	`The Server A has identified the following shortest paths:`
	`----------`
	`Destination  Min Length`
	`----------`
	`1  10`
	`2  20`
	`7  21`
	`…`
	`----------`

For path finding, after sending to AWS:	`The Server A has sent shortest paths to AWS.`

Table 5. Backend-Server B on screen messages

Event	On Screen Message
Booting up (Only while starting):	`The Server B is up and running using UDP on port <port number>.`

For calculation, after receiving data from AWS:	`The Server B has received data for calculation:`
	`* Propagation speed: <speed1> km/s;`
	`* Transmission speed <speed2> Bytes/s;`
	`* Path length for destination <vertex index>: <length1 length2 …>;`
	`* Path length for destination <vertex index>: <length1 length2 …>;`
	`* …`

After calculation:	`The Server B has finished the calculation of the delays:`
	`----------`
	`Destination  Delay`
	`----------`
	`1  0.30`
	`2  0.35`
	`7  0.33`
	`…`
	`----------`

ending the results to the AWS server:	`The Server B has finished sending the output to AWS`

Table 6. AWS on screen messages

Event	On Screen Message
Booting up (only while starting):	`The AWS is up and running.`
Upon Receiving the input from the client:	`The AWS has received map ID <map ID>, start vertex <vertex ID> and file size <file size> from the client using TCP over port <port number>`
After sending information to server A	`The AWS has sent map ID and starting vertex to server A using UDP over port <port number>”`

After receiving results from server A	`The AWS has received shortest path from server A:`
	`----------`
	`Destination  Min Length`
	`----------`
	`1  10`
	`2  20`
	`7  21`
	`…`
	`----------`

After sending information to server B	`The AWS has sent path length, propagation speed and transmission speed to server B using UDP over port <port number>.`

After receiving results from server B	`The AWS has received delays from server B:`
	`----------`
	`Destination  Tt  Tp  Delay`
	`----------`
	`1  0.10  0.10  0.20`
	`2  0.10  0.20  0.30`
	`7  0.10  0.21  0.31`
	`…`
	`----------`

After sending results to client	`The AWS has sent calculated delay to client using TCP over port <port number>.`

Table 7. Client on screen messages

Event	On Screen Message
Booting Up:	`The client is up and running.`
After sending query to AWS	`The client has sent query to AWS using TCP over port <port number>: start vertex <vertex index>; map <map ID>; file size <file size>.`

After receiving output from AWS	`The client has received results from AWS:`
	`----------`
	`Destination  Min Length  Tt  Tp  Delay`
	`----------`
	`1  10  0.10  0.10  0.20`
	`2  20  0.10  0.20  0.30`
	`7  21  0.10  0.21  0.31`
	`…`
	`----------`

ASSUMPTIONS

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

2. The map.txt file is created before your program starts.

3. Client always sends valid queries. In other words, the map ID corresponds to existing maps, and start vertex is an existing vertex in the map.

4. 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.

5. 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.

6. 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.

7. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command:

```
ps –aux | grep ee450
```

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

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

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 hardcoded as **localhost (127.0.0.1)** in all codes.

3. Your client should terminate itself after all done. And the client can run multiple times to send requests. However, the backend servers and the AWS should keep running and be waiting for another request until the TAs terminate them by Ctrl+C. It 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 client 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 working 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 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:

Compiles **all** your files and creates executables	`make all`
Runs server A	`make serverA`
Runs server B	`make serverB`
Runs AWS	`make aws`
Query the AWS	`./client <Map ID> <Source Vertex Index> <File Size>`

TAs will first compile all codes using **make all**. They will then open 4 different terminal windows. On 3 terminals they will start servers A, B and AWS using commands **make serverA**, **make serverB**, and **make aws**. **Remember that servers should always be on once started.** Client can connect again and again with different input query arguments. On the 4th terminal they will start the client as "**./client <Map ID> <Source Vertex Index> <File Size>**". 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 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 and keep a history of all your submissions. If you make multiple submission, we will grade your latest valid submission. Submission after deadline is considered as invalid.

5. D2L will send you a "Dropbox submission receipt" to confirm your submission. So please do check your emails to make sure your submission is successfully received. If you don't receive a confirmation email, try again later and contact your TA if it always fails.

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

7. Please DO NOT wait till the last 5 minutes to upload and submit because some technical issues might happen and you will miss the deadline. And a kind suggestion, if you still get some bugs one hour before the deadline, please make a submission first to make sure you will get some points for your hard work!

8. After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. If the outcome is not what you expected, try to resubmit and confirm again. We will only grade what you submitted even though it's corrupted.

9. **You have plenty of time to work on this project and submit it in time hence there is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a zero for the project.**

GRADING CRITERIA

Notice: We will only grade what is already done by the program instead of what will be done. For example, the TCP connection is established and data is sent to the AWS. But result is not received by the client because the AWS got some errors. Then you will lose some points for phase 1 even though it might work well.

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, 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 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. Also, you will NOT get credit by repeating others' answers.

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.

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.