Socket Programming Project

Problem Statement:

Distributed lookup services (such as the Google File System, and Distributed Hash Tables) tackle the problem of locating data that is distributed over multiple nodes in the network. In this project we shall implement a simplified version of a lookup service that'll help us explore the core issues involved. Specifically, you'll be given a (distributed) database consisting of words in the English language, and their definitions. The database will be in plain text and consist of multiple key (the word), value (the corresponding definition) pairs.

In this project, you will implement a model of distributed lookup service where a single client issues a (dictionary) key search to a server which in turn searches for the key (and it's associated value) over 3 backend servers. The server facing the client then collects the results from the backend servers, performs additional computation on the results if required, and communicates it to the client in the required format (This is also an example of how a cloud-computing service such Amazon Web Services might speed up a large computation task offloaded by the client). The monitor also receives the results from the server and receive additional information if required (described in phase 2).

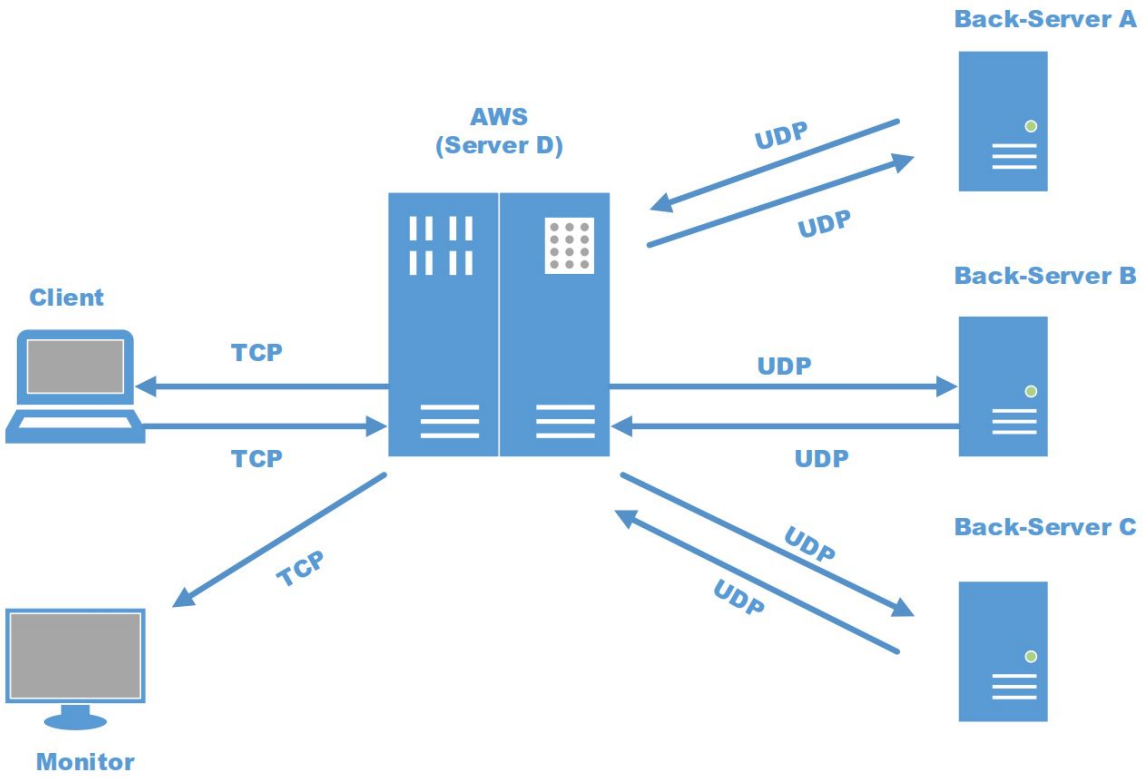

Figure 1. Illustration of the network

The server communicating with the client is called AWS (Amazon Web Server). The three backend servers are named Back-Server A, Back-Server B and Back-Server C. Back-Server A has access to a database file named `dict_A.txt`, Back-Server B has access to a database file named `dict_B.txt`, and Back-Server C chas access to a database file named `dict_C.txt`. The client, monitor and the AWS communicates over a TCP connection while the communication between AWS and the Back-Servers A, B & C is over a UDP connection. This setup is illustrated in Figure 1.

Source Code Files

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

1. AWS: You must name your code file: **aws.c** or **aws.cc** or **aws.cpp** (all small letters). Also you must call the corresponding header file (if you have one; it is not mandatory) **aws.h** (all small letters).
2. Back-Server A, B and C: 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 call the corresponding header file (if you have one; it is not mandatory)

server#.h (all small letters, except for **#**). The "#" character must be replaced by the server identifier (i.e. A or B or C), depending on the server it corresponds to.

Note: *You are **not** allowed to use one executable for all four servers (i.e. a "fork" based implementation).*

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

More Detailed Explanations:

Phase 1A: (30 points)

All four server programs (AWS -a.k.a. server-D, Back-Server A, B, & C) 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, the client program is run. The client displays a boot message as indicated in the onscreen messages table. Note that the client code takes an input argument from the command line, that specifies the computation that is to be run and another argument for the desired input value to compute. The format for running the client code is

```
./client <function> <input>
```

where `<function>` can take a value from `{search, prefix}` and `<input>` will be a a word consisting of 27 characters or less. As an example, to find the dictionary definition of the word `muggle`, the client should be run as follows:

```
./client search muggle
```

After booting up, the client and monitor establish TCP connections with AWS. After successfully establishing the connection, the client first sends the `<function> <input>` to AWS. Once this is sent, the client should print a message in the format given in the table 8. This ends Phase 1A and we now proceed to Phase 2.

Phase 1B: (40 points)

In Phase 1A, you read the function and input value and sent them to the AWS server over a TCP connection. Now in phase 2, this AWS server will send the value to the 3 back-servers. The value will be sent to their respective back-end server depending on the function to be executed.

The communication between the AWS server and the back-servers happens over UDP. The AWS server will send the `<input>` to the server. The port numbers used by back-servers A, B and C are specified in Table 2. Since all the servers will run on the same machine in our project, all have the same IP address (the IP address of localhost is usually `127.0.0.1`).
The back-end servers are in charge of performing the operations `{search, prefix}`, on their respective dictionary files. Note that the final response for the client will be created by the AWS server after obtaining the results from all the backed servers. The operations `search` and `prefix` are detailed in Table 1:

Table 1. Database Operations	
`search`	In this function, you search for an exact match of the given `<input>` in the database. If a match is found, return the dictionary value (word definition) associated with `<input>`
`prefix`	In this function, you search for an partial match of the given `<input>` in the database. Specifically, you are searching for all words in the dictionary that starts with or exactly matches `<input>.`

Each back-server will perform their respective database operation and then return their result to AWS (server D) using UDP.

Phase 2: (30 points)

After receiving the collected results from the back-server A/B/C, AWS should combine the results. If the required function is **`prefix`**, AWS should combine the list of words (only words, no deninitions) obtained from each backend server and send the combined list to both client and

monitor. If the required function is **`search`**, AWS should send the word and definition to the client. Then, AWS needs to to some additional operation for the monitor.
If the required operation is "**`search`**", AWS needs to provide some more information to the monitor. For example, you make "**`./client search week`**" in the client. In phase 1, you should look up the files in three backend servers and find the corresponding definition of the word "week". In phase 2, you need to look up the dictionaries to find one word that only has one replaced letter with comparing to the original word, no insertion, no deletion, ONLY replacement. For example, "seek" and "weak" are all valid words as long as they are in the dictionaries. But "work" (two changes), "eek" (one deletion) and "area" (one insertion if the input is "are") are all not valid even though they are in the dictionaries. You are suggested to scan the dictionaries and check every word to see if it's valid. You don't need to generate these words at the AWS. You are only asked to find one of these valid words and send it and its definition to the monitor along with the original query about the definition of "week". You can do the original search and this query in one pass and send the original word with definition to the client and these two entries to the monitor.
For any of the database operations, if there are no results from any of the backend servers, the AWS server should send the message, that no matches are found, to the client. This message should be printed at the client side, when received from the AWS server.

Phase 3: (10 points extra, not mandatory)

If you want to earn 10 extra points, you can implement another operation other than **`search`** and **`prefix`**. The operation **`suffix`** is similar to **`prefix`**. For "**`./client suffix <word>`**", You need to find all words ending with or exactly matching the given "**`word`**". Then, similar to **`prefix`**, the list of words should be collected by AWS and sent to both client and monitor.

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 Numer Allocation

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

Table 3. Static and Dynamic assignments for TCP and UDP ports.

Process	Dynamic Ports	Static Ports
Backend-Server (A)	-	1 UDP, 21000+xxx
Backend-Server (B)	-	1 UDP, 22000+xxx

Backend-Server (C)	-	1 UDP, 23000+xxx
AWS (D)	-	1 UDP, 24000+xxx 1 TCP with client, 25000+xxx 1 TCP with monitor, 26000+xxx
Client	1 TCP	<Dynamic Port assignment>
Monitor	1 TCP	<Dynamic Port assignment>

NOTE: xxx is the last 3 digits of your USC ID. For example, if the last 3 digits of your USC ID are "319", you should use the port: **21000+319 = 21319** for the Backend-Server (A). **It is NOT going to be 21000319.**

ON SCREEN MESSAGES: **Table 4. Backend-Server A on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	"The ServerA is up and running using UDP on port <port number>."
Upon Receiving the input string:	"The ServerA received input <INPUT> and operation <FUNCTION>"
For `search`, after calculating	"The serverA has found <m> match and <n> similar words" (m = 0 or 1, n = 0 ,1, 2 ...)
For `prefix/suffix`, after calculating	"The ServerA has found <n> matches" (n = 0, 1, 2, ...)
After sending the results to the AWS server (D):	"The ServerA finished sending the output to AWS"

ON SCREEN MESSAGES: **Table 4. Backend-Server B on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	"The ServerB is up and running using UDP on port <port number>."
Upon Receiving the input string:	"The ServerB received input <INPUT> and operation <FUNCTION>"
For `search`, after calculating	"The serverB has found <m> match and <n> similar words" (m = 0 or 1, n = 0 ,1, 2 ...)

For `prefix/suffix`, after calculating	"The ServerB has found <n> matches" (n = 0, 1, 2, ...)
After sending the results to the AWS server (D):	"The ServerB finished sending the output to AWS"

ON SCREEN MESSAGES: **Table 4. Backend-Server C on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	"The ServerC is up and running using UDP on port <port number>."
Upon Receiving the input string:	"The ServerC received input <INPUT> and operation <FUNCTION>"
For `search`, after calculating	"The serverC has found <m> match and <n> similar words" (m = 0 or 1, n = 0 ,1, 2 ...)
For `prefix/suffix`, after calculating	"The ServerC has found <n> matches" (n = 0, 1, 2, ...)
After sending the results to the AWS server (D):	"The ServerC finished sending the output to AWS"

ON SCREEN MESSAGES: **Table 7. AWS on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up (only while starting):	"The AWS is up and running."
Upon Receiving the number and function from the client:	"The AWS received input=<INPUT> and function=<FUNCTION> from the client using TCP over port <port number>"
After querying each Backend-Server	"The AWS sent <INPUT> and <FUNCTION> to Backend-Server A" "The AWS sent <INPUT> and <FUNCTION> to Backend-Server B" "The AWS sent <INPUT> and <FUNCTION> to Backend-Server C"
`For prefix/suffix:` After receiving result from backend server i): i is one of A, B, or C	"The AWS received <m> matches from Backend-Server <i> using UDP over port <port number>" (m = 0, 1, 2, …)

`For search:` After receiving result from backend server i): i is one of A, B, or C	“The AWS received <n> similar words from Backend-Server <i> using UDP over port <port number>” (n = 0, 1)
After sending the final result to the client:	“The AWS sent <n> matches to client.
After sending the final result to the monitor	“The AWS sent <n> matches to the monitor via TCP port <port number>” for `prefix/suffix` “The AWS sent <WORD1> and <WORD2> to the monitor via TCP port <port number>” for `search`

ON SCREEN MESSAGES: **Table 8. Client 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 <INPUT> and <FUNCTION> to AWS.”
After receiving the `search` result from AWS	“Found a match for <INPUT>: <VALUE>”
After receiving the `prefix` result from AWS	“Found <n> matches for <INPUT>: <VALUE1> <VALUE2> . <VALUEn>”
After receiving the `suffix` result from AWS (optional)	“Found <n> matches for <INPUT>: <VALUE1> <VALUE2> . <VALUEn>”
If no matches are found for either `search` or `prefix`	“Found no matches for <INPUT>”

ON SCREEN MESSAGES: **Table 9. Monitor on screen messages**

Event	On Screen Message (inside quotes)
Booting Up:	"The Monitor is up and running."
After receiving the `search` result from AWS	"Found a match for <INPUT>: <VALUE> One edit distance match is <WORD>: <DEFINITION>"
After receiving the `prefix` result from AWS	"Found <n> matches for <INPUT>: <VALUE1> <VALUE2> . <VALUEn>"
After receiving the `suffix` result from AWS (optional)	"Found <n> matches for <INPUT>: <VALUE1> <VALUE2> . <VALUEn>"
If no matches are found for either `search` or `prefix`	"Found no matches for <INPUT>"

Example Output to Illustrate Output Formatting:

For operation `prefix/suffix`:

Backend-Server A Terminal:

The Server A is up and running using UDP on port 21319.
The Server A received input <mug> and operation <prefix>
The Server A has found < 3 > matches
The Server A finished sending the output to AWS

Backend-Server B Terminal:

The Server B is up and running using UDP on port 22319.
The Server B received input <mug> and operation <prefix>
The Server B has found < 3 > matches
The Server B finished sending the output to AWS

Backend-Server C Terminal:

The Server C is up and running using UDP on port 23319.

The Server C received input <mug> and operation <prefix>
The Server C has found < 3 > matches
The Server C finished sending the output to AWS

AWS Terminal:
The AWS is up and running.
The AWS received input=<mug> and function=<prefix> from the client using TCP over port 25319
The AWS sent < mug > and <prefix> to Backend-Server A
The AWS sent < mug > and <prefix> to Backend-Server B
The AWS sent < mug > and <prefix> to Backend-Server C
The AWS received < 1 > matches from Backend-Server < A > using UDP over port < 24319 >
The AWS received < 3 > matches from Backend-Server < B > using UDP over port < 24319 >
The AWS received < 10 > matches from Backend-Server < C > using UDP over port < 24319 >
The AWS sent < 14 > matches to client.
The AWS sent < 14 > matches to the monitor via TCP port 26319.

Client Terminal:
The client is up and running.
The client sent < mug > and < prefix > to AWS
Found < 3 > matches for < mug >:
< mug >
< mugs >
< muggle >

Monitor Terminal:
The monitor is up and running.
Found < 3 > matches for < mug >:
< mug >
< mugs >
< muggle >

For operation search:

Backend-Server A Terminal:
The Server A is up and running using UDP on port 21319.

The Server A received input <week> and operation <search>
The Server A has found < 1 > matches and < 1 > similar words
The Server A finished sending the output to AWS

Backend-Server B Terminal:
The Server B is up and running using UDP on port 22319.
The Server B received input <week> and operation <search>
The Server B has found < 0 > matches and < 1 > similar words
The Server B finished sending the output to AWS

Backend-Server C Terminal:
The Server C is up and running using UDP on port 23319.
The Server C received input <week> and operation <search>
The Server C has found < 0 > matches and < 1 > similar words
The Server C finished sending the output to AWS

AWS Terminal:
The AWS is up and running.
The AWS received input=<week> and function=<search> from the client using TCP over port 25319
The AWS sent < week > and <search> to Backend-Server A
The AWS sent < week > and <search> to Backend-Server B
The AWS sent < week > and <search> to Backend-Server C
The AWS received < 1 > similar words from Backend-Server < A > using UDP over port < 24319 >
The AWS received < 1 > similar words from Backend-Server < B > using UDP over port < 24319 >
The AWS received < 1 > similar words from Backend-Server < C > using UDP over port < 24319 >
The AWS sent < 1 > matches to client.
The AWS sent <week> and <weak> to the monitor via TCP port 26319.

Client Terminal:
The client is up and running.
The client sent < week > and < search > to AWS
Found a match for < week >:
< a series of regular working, business, or school days during each 7-day period >

Monitor Terminal:

The monitor is up and running.
Found a match for < week >:
< a series of regular working, business, or school days during each 7-day period >
One edit distance match is <weak>:
< lacking strength >

Assumptions:

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

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 kill 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 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 be 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 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 serverA`	**Run**s server A
`make serverB`	**Run**s server B
`make serverC`	**Run**s server C
`make aws`	**Run**s AWS
`make monitor`	**Runs** monitor
`./client <function> <input>`	Starts the client

TAs will first compile all codes using **`make all`**. They will then open 6 different terminal windows. On 5 terminals they will start servers A, B, C and AWS using commands **`make serverA`**, **`make serverB`**, **`make serverC`**, **`make aws`** and **`make monitor`**. **Remember that servers and monitor should always be on once started.** Client can connect again and again with different input values and function. On the sixth terminal they will start the client as **`./client search muggle`** or **`./client prefix mug`**. Note that input value of `muggle/mug` is just an example. TAs will check the outputs for multiple values of input. The terminals should display the messages shown in table 4, 5, 6, 7 and 8.

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.

6.

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. 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.

13. 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`**