EE450 Socket Programming Project, Spring 2019
Due Date: Friday April 12th, 2019 11:59 PM (Midnight)
(The deadline is the same for all on-campus and DEN off-campus students)

Hard Deadline (Strictly enforced)

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:

In information theory, the Shannon-Hartley theorem tells the maximum rate at which information can be transmitted over a communications channel of a specified bandwidth in the presence of noise. It is an application of the noisy-channel coding theorem to the archetypal case of a continuous-time analog communications channel subject to Gaussian noise. The theorem establishes Shannon's channel capacity for such a communication link, a bound on the maximum amount of error-free information per time unit that can be transmitted with a specified bandwidth in the presence of the noise interference. The law is named after Claude Shannon and Ralph Hartley. Formulated by:

$$C = B\log_2(1+\frac{S}{N})$$

where

C is the channel capacity in bits per second, a theoretical upper bound on information rate;

B is the bandwidth of the channel in hertz (pass-band bandwidth in case of a band-pass signal);

S is the average received signal power over the bandwidth measured in watts (or volts squared);

N is the average noise power over the bandwidth, measured in watts (or volts squared).

In this project, you will implement a model computational offloading where a single client issues two functions (compute and write) and parameters (depends on the function) to the AWS server and expects the reply for the end-to-end delay of the designated link.

The setup of the network is illustrated in Figure 1. The server communicating with the client acts similar as AWS (Amazon Web Server). There are two back-end servers, named Back-Server A and Back-Server B. Back-Servers A is a storage server. It possesses a database file, `database.txt,` in which attribute values regarding information of links are stored (Original data and data wrote by client). A sample database is demonstrated in Table 1. There are five attributes in the database, which are {Link ID, Bandwidth, Length, Velocity (Propagation Speed), Noise Power}, from left to right. In `database.txt,` there will be only five columns of numbers without names of attributes.

The Back-end Server B is a computing server. It receives data from the AWS server, performs computational tasks, and returns the result to the AWS server. The monitor connecting to the AWS server is used to record results of every steps and print them out. The client, monitor and the AWS communicate over TCP connections while the AWS and the Back-Servers A & B communicate over UDP connections.

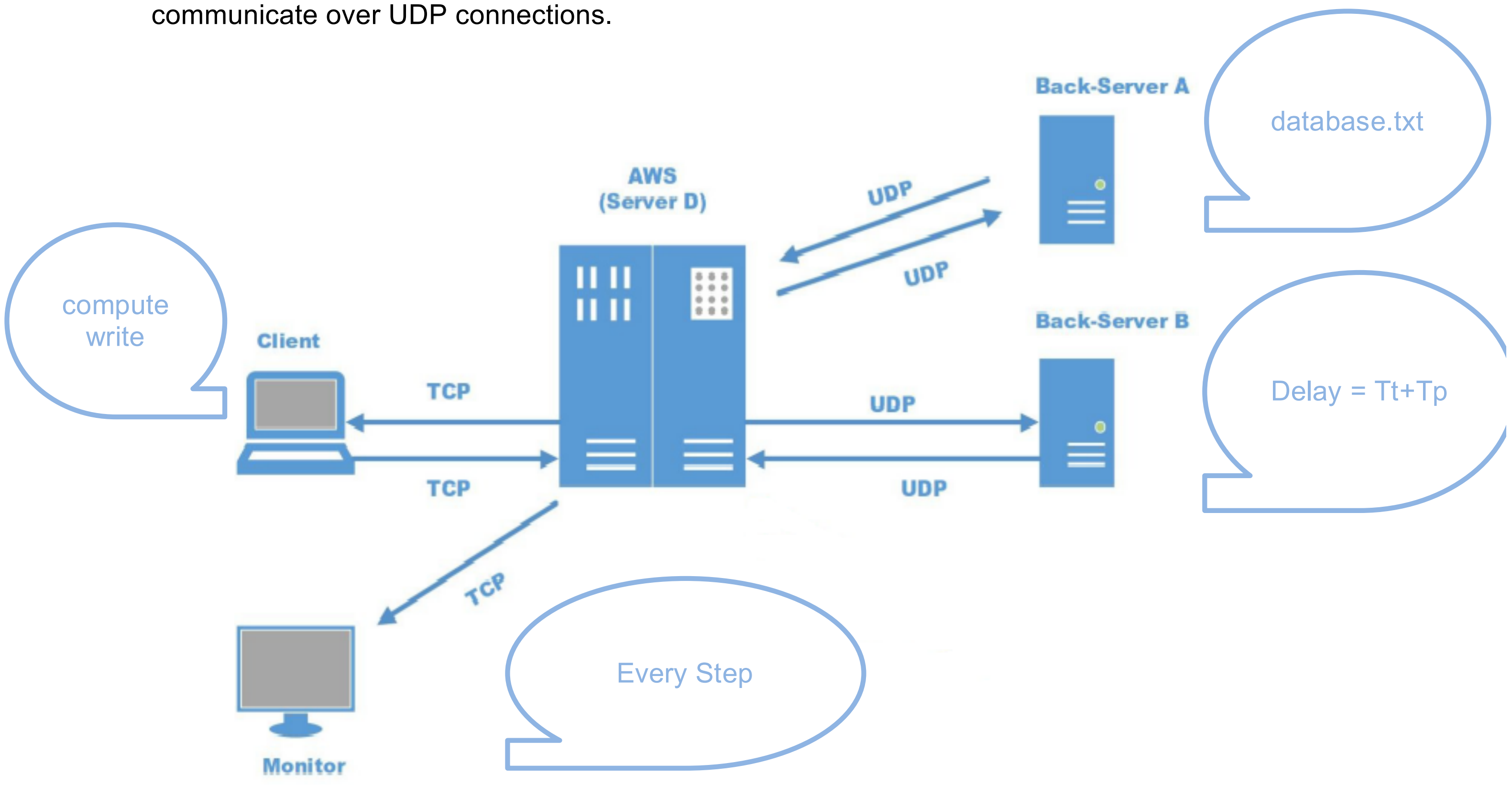

Figure 1. Illustration of the network

When the client inputs the “write” function and according parameters, the AWS server will send the information to both serverA and monitor, and serverA will write the information to its database. If the writing process is successful, the AWS server will send the success message to the monitor and client. The link ID of the given link is automatically added 1 to the last link in the serverA’s database (Therefore, client does not need to input the link ID).

When the client inputs the “compute” function, the AWS server will send the information to both serverA and monitor, and serverA will search for the information of the designated link (identified by Link ID) over its database. After receiving replies from serverA, the AWS server then sends the link information (bandwidth, link length, propagation velocity, noise power) with client inputs file size, signal power) to the computing server. Once the AWS server receives the computed results (which will be end-to-end delay, propagation delay, and transmission delay) from server B, it finally sends the end-to-end delay 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).

Table 1. A Toy Database Example

Link ID	Bandwidth (MHz)	Length (km)	Velocity (km/s)	Noise Power (dBm)

1	25	3	200000	-90
2	50	0.5	80000	-82
3	100	10.7	123456	-104
4	75	5.6	246810	-93
5	25	3	200000	10
⋮	⋮	⋮	⋮	⋮

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 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 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), 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 1: (15 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 message on the terminal. The format of the boot message for each server is given in the onscreen message 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 running. 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 write <BW> <LENGTH> <VELOCITY> <NOISEPOWER>
./client compute <LINK_ID> <SIZE> <SIGNALPOWER>
```

(Between two inputs, there should be a space)

where all the variables are not always Integers (except for LINK_ID).

An example of “write” command, the client wants to add the info of a link (BW: 25MHz, Link Length: 3km, Velocity: 200000km/s, Noise Power: -90dBm) to database. After the “write” command there will be one more line in the database file containing these values, with link id incremented.

```
./ client write 25 3 200000 -90
```

An example of “compute” command, the client wants to calculate the end-to-end delay of No.3 link, and the file size is 10K bits and power of signal is -30 dBm.

```
./client compute 3 10000 -30
```

Note that $P(\mathrm{dBm}) = 10\log_{10}\left(\frac{1000P(\mathrm{Watt})}{1\mathrm{Watt}}\right)$, where P stands for power.

After booting up, the client and monitor establish TCP connections with AWS. After successfully establishing the connection, the client first sends the `<input>` (compute `<LINK_ID>`, `<SIZE>`, `<POWER> OR write <BW> <LENGTH> <VELOCITY> <NOISEPOWER>`) to AWS. Once this is sent, the client should print a message in the format given in the Table 8. This ends Phase 1 and we now proceed to Phase 2.

Phase 2: (20 points+40 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 the back-servers happens over UDP. The port numbers used by back-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, the {write} operation will be implemented. In Phase 2B, the {search} and {compute} operation will be implemented (see Table. 2).

Phase 2A: (20 points)

Phase 2A is initiated when the AWS receive <write> command and corresponding data from clients. The AWS will forward the request from clients to back-end server A. Back-end server A is the storage server. It possesses a database file in which information of links are stored. After the back-end server A receives the request, it will perform {write} operation (see Table. 2) by adding the data to the database and generating a new `<LINK_ID>.` Once finished, the acknowledgement will be sent back to AWS. Note that all messages required to be printed for the AWS server and the storage server A can be found in the format given in the Table 6 and 4, respectively.

The new <LINK_ID> should be <largest existed link ID + 1>. All existed data will be stored in ascending order of <LINK_ID> so the biggest existed link is the last one. For example, the largest existed link ID is <7> then the new generated <LINK_ID> should be <8>. The new data will then be added at the end of database.

Phase 2B: (40 points)

Phase 2B starts when the AWS receives <compute> command and corresponding data from clients. The AWS server first looks up and requests the link information of the client-specified link `<LINK_ID>` from back-end storage server A. After receiving `<LINK_ID>` from the AWS server, the storage server A performs the operation `{search}` (see Table. 2) on its database for an exact match of the given `<LINK_ID>` in the Link ID attribute. If the requested link is found, the storage server replies the AWS server a message `<m>=1` with link information; otherwise, the storage server replies the AWS server a message `<m>=0`. The message `<m>` taking value either '1' or '0' indicates "requested link found" or "requested link not found", respectively. Note that all messages required to be printed for the AWS server and the storage servers A can be found in the format given in the Table 6 and 4, respectively.

Once the AWS server receives replies from the storage server, the AWS server checks the replied messages. If the messages indicate "requested link not found", the AWS server sends a result indicating "no match found" to both the monitor and the client. If the message indicates "requested link found", the AWS server sends the `<input>` with the received link information to the back-end server B. The back-end server B is a computing server. It performs the operation `{compute}` (see Table. 2) based on the data sent by the AWS server. When the operation is done, the back-end server B will reply all the computed results to the AWS server. All messages required to be printed for the AWS server and the computing server B can be found in the format given in the Table 6 and 5, respectively. This ends Phase 2B and we now proceed to Phase 3.

Table 2. Server Operations	
`write`	In this function, you write the information to the database, generate a new <LINK_ID> and return the acknowledgement with the <LINK_ID>.

`search`	In this function, you search for an exact match of the given `<LINK_ID>` in the database. If a match is found, return the link information associated with `<LINK_ID>`.
`compute`	In this function, you compute the transmission delay (in ms), the propagation delay (in ms), and the end-to-end delay (in ms) of link `<LINK_ID>` based on `<input>` and the link information associated with `<LINK_ID>`. For display, the

Phase 3: (15 points)

At the end of Phase 2A, backend-server A should have finished the write operation. The acknowledgment should be sent to the AWS using UDP. When the AWS receives the acknowledgement, it needs to forward the acknowledgment to the client and monitor using TCP. The format on how to display the results is explained in the message table and example outputs below.

At the end of Phase 2B, backend-server B should have the computation results ready. Those results should be sent to the AWS using UDP. When the AWS receives the computation results, it needs to forward the result “end-to-end delay” to the client and all results (all three delays) to the monitor using TCP. The format on how to display the final result is explained in the message table and example outputs below. Please note that there can be a no match result instead of the regular. Make sure you round the results of three delay time to the 2nd decimal place for display. Round the result after summing *T_trans* and *T_prop.* Do not sum rounded *T_trans* and rounded *T_prop* as your total delay.

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 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
AWS	-	1 UDP, 23000+xxx 1 TCP with client, 24000+xxx 1 TCP with monitor, 25000+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 Server A is up and running using UDP on port <port number>.”
For write, upon receiving the input:	“The Server A received input for writing”
For write, after finishing writing and sending result to AWS:	“The Server A wrote link <LINK_ID> to database”
For compute, upon receiving the input:	“The Server A received input <LINK_ID> for computing”
For compute, after sending the search results to the AWS server:	“The Server A finished sending the search result to AWS” OR “Link ID not found”

ON SCREEN MESSAGES:
Table 5. Backend-Server B on screen messages

Event	On Screen Message (inside quotes)
Booting up (Only while starting):	“The Server B is up and running using UDP on port <port number>.”
Upon receiving the input and link information:	“The Server B received link information: link <LINK_ID>, file size <SIZE>, and signal power <POWER>”
After calculating:	“The Server B finished the calculation for link <LINK_ID>”
After sending the results to the AWS server:	“The Server B finished sending the output to AWS”

ON SCREEN MESSAGES: **Table 6. AWS on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting up (only while starting):	"The AWS is up and running."
Upon Receiving the input from the client:	"The AWS received operation <FUNCTION> from the client using TCP over port <port number>" <FUNCTION = write, compute>
For both functions, after sending the input to the monitor:	"The AWS sent operation <FUNCTION> and arguments to the monitor using TCP over port <port number>"
For both functions, after sending the input to server A:	"The AWS sent operation <FUNCTION> to Backend-Server A using UDP over port <port number>"
For write, after receiving result from server A:	"The AWS received response from Backend-Server A for writing using UDP over port <port number>"
For compute, after receiving result from server A:	"The AWS received link information from Backend-Server A using UDP over port <port number>" OR "Link ID not found"
After querying backend-server B:	"The AWS sent link ID=<LINK_ID>, size=<SIZE>, power=<POWER>, and link information to Backend-Server B using UDP over port <port number>"
After receiving result from backend-server B:	"The AWS received outputs from Backend-Server B using UDP over port <port number>"
For both function, after sending the result to the client	"The AWS sent result to client for operation <FUNCTION> using TCP over port <port number>"
For write, after sending the result to the monitor:	"The AWS sent write response to the monitor using TCP over port <port number>"
For compute after sending result to monitor:	"The AWS sent compute results to the monitor using TCP over port <port number>"

ON SCREEN MESSAGES: **Table 7. Client on screen messages**	
Event	**On Screen Message (inside quotes)**

Booting Up:	“The client is up and running”
For write, after sending the input to AWS:	“The client sent write operation to AWS”
For compute, after sending the input to AWS:	"The client sent ID=<LINK_ID>, size=<SIZE>, and power=<POWER> to AWS”
For write, after receiving the result from AWS:	“The write operation has been completed successfully”
For compute, after receiving the result from AWS:	“The delay for link <LINK_ID> is <DELAY>ms” OR “Link ID not found”

ON SCREEN MESSAGES:
Table 8. Monitor on screen messages

Event	On Screen Message (inside quotes)
Booting Up:	“The monitor is up and running.”
For write, after receiving the input from AWS:	“The monitor received BW = <BANDWIDTH>, L = <LENGTH>, V = <VELOCITY> and P = <NOISE POWER> from the AWS”
For compute, after receiving the input from AWS:	“The monitor received link ID=<LINK_ID>, size=<SIZE>, and power=<POWER> from the AWS”
For write, after receiving the result from AWS:	“The write operation has been completed successfully”
For compute, after receiving the result from AWS:	“The result for link <LINK_ID>: Tt = <Transmission Time>ms, Tp = <Propagation Time>ms, Delay = <Delay>ms” OR “Link ID not found”

Example Output to Illustrate Output Formatting:

Please note that the number (including port number) in the example output is for display only, not a real result. If this example has difference with the output table, please follow the table.

Backend-Server A Terminal:

```
The Server A is up and running using UDP on port <21319>.
The Server A received input for writing
The Server A wrote link <10> to database
The Server A received input <5> for computing
The Server A finished sending the search result to AWS
```

Backend-Server B Terminal:

```
The Server B is up and running using UDP on port <22319>
The Server B received link information: link <5>, file size <1000>, and signal power <10>
The Server B finished the calculation for link <5>
The Server B finished sending the output to AWS
```

AWS Terminal:

```
The AWS is up and running
The AWS received operation <write> from the client using TCP over port <24319>
The AWS sent operation <write> and arguments to the monitor using TCP over port
<25319>
The AWS sent operation <write> to Backend-Server A using UDP over port <23319>
The AWS received response from Backend-Server A for writing using UDP over
port <23319>
The AWS sent write response to the monitor using TCP over port <25319>
The AWS sent result to client for operation <write> using TCP over port <24319>
The AWS received operation <compute> from the client using TCP over port
<24319>
The AWS sent operation <compute> and arguments to the monitor using TCP over
port <25319>
The AWS sent operation <compute> to Backend-Server A using UDP over port
<23319>
The AWS received link information from Backend-Server A using UDP over port
<23319>
The AWS sent link ID = <5>, size = <1000>, power = <10>, and link information to
Backend-Server B using UDP over port <23319>
The AWS received outputs from Backend-Server B using UDP over port < 23319 >
The AWS sent result to client for operation <compute> using TCP over port <24319>
The AWS sent compute results to the monitor using TCP over port <25319 >
```

Client Terminal:

```
The client is up and running
The client sent write operation to AWS
The write operation has been completed successfully
The client sent link ID=<5>, size=<1000>, and power=<10> to AWS
The delay for link <5> is <0.42>ms
```

Monitor Terminal:

```
The monitor is up and running
The monitor received BW = <25>, L = <3>, V = <200000> and P = <10> from the AWS
The write operation has been completed successfully
The monitor received input=<5>, size=<10000>, and power=<10> from the AWS
The result for link <5>:
Tt = <0.40>ms
Tp = <0.015>ms
Delay = <0.42>ms
```

Assumptions:

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

2. Your program should be able to create the .txt file on backend-server (A) upon booting up; when grading your work, it's possible that we enter multiple "write" commands before a "compute", vice versa.

3. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and mention them all in your README file.

4. You are allowed to use 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.

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 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 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 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.
   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 BE SUBJECT TO A SERIOUS PENALTY

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 aws`	**Run**s AWS
`make monitor`	**Run**s monitor
`./client write <BW> <LENGTH> <VELOCITY> <NOISEPOWER>`	**Write** data to database
`./client compute <LINK_ID> <SIZE> <SIGNALPOWER>`	**Compute** delay

TAs will first compile all codes using `make all`. They will then open 5 different terminal windows. On 4 terminals they will start servers A, B and AWS using commands `make serverA`, `make serverB`, `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 5th terminal they will start the client as "`./client write <BW> <LENGTH> <VELOCITY> <NOISEPOWER>`". And then they will input `"./client compute <LINK_ID> <SIZE> <SIGNALPOWER>"` in the client's terminal. 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.

   c.
   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 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 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.

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.