EE450 Socket Programming Project

Spring 2021

Due Date:

11:59pm April 23, 2021

(Hard Deadline, Strictly Enforced)

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

OBJECTIVE

The objective of this assignment is to familiarize you with UNIX socket programming. This assignment is worth **15%** of your overall grade in this course. **It is an individual assignment and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).**

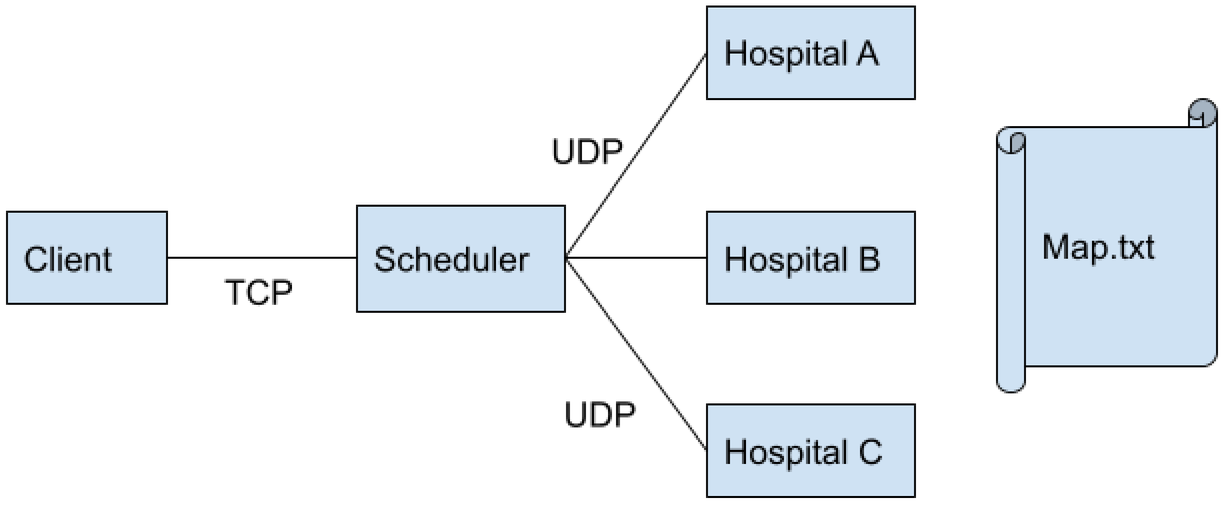
If you have any doubts/questions, post your questions on Piazza. **You must discuss all project related issues on Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points to the project.

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

PROBLEM STATEMENT

Last year, a terrifying storm named COVID-19 swept the world, affecting tons of families, businesses, and countries. It even changed the way we live, travel and socialize. Masks are now our daily outfits, and nucleic acid test results have become a certificate for accessing to the public places, especially to public transportation, such as taking airplanes. Taking the nucleic acid test is one of the weekly routines for schools' faculty and students. How to reasonably allocate medical resources is a challenge we aim at addressing here.

In this project, you will implement a simple application to generate customized resource allocation based on user queries. Specifically, consider that USC students in different locations in LA want to schedule appointments for COVID-19 testing. They would send their queries to the USC health center (i.e., scheduler) and receive the location of an available hospital as the reply from the same scheduler. Nowadays, the medical network is so large that it is impossible to store all the information in a single machine. So we consider a distributed system design where the scheduler is further connected to many (in our case, three) hospitals. Each hospital stores its own *capacity*, *availability* and the *map of LA*. Since users and hospitals are all in LA, they will share the same map. i.e., hospitals A, B and C store the same map. Therefore, once the scheduler receives a client query, it decodes the query and further sends a request to the appropriate hospitals. Each of the hospitals will calculate a *score* according to its own availability and the distance to the client. Then, each hospital will reply to the scheduler. Finally, the scheduler will assign the client to the hospital with the highest score, and inform the client and the hospitals of the decision.



The detailed operations to be performed by all the parties are described with the help of Figure 1. There are in total 5 communication end-points:

- Client: representing a student, who can locate anywhere in LA.
- Scheduler: responsible for interacting with the client and the hospitals
- Hospital A, Hospital B and Hospital C: responsible for calculating the matching scores based on the query
    - For simplicity, the map of LA is stored as a single text file, and will be read by all hospitals.

The full process can be roughly divided into four phases (see also “DETAILED EXPLANATION” section), the communication and computation steps are as follows:

Boot-up

1. [**Computation**]: Hospital A, B and C read the file map.txt, and construct a list of “graphs” (see “DETAILED EXPLANATION” for description of suggested data structures for graphs).
    - Assume a “static” map where contents in map.txt do not change throughout the entire process.
    - Hospitals only need to read map.txt once. When Hospitals are handling client queries (in “Scoring” phase), they will refer to the internally represented graphs, not the text files.
    - There is only one file map.txt.
2. [**Communication**]: Scheduler will establish the connection to each hospital, and obtain the initial availability of the hospitals.
3. [**Computation**]: Scheduler will construct a data structure to book-keep the availability from step 2. When the client queries come, the Scheduler can send a request to the available Hospitals.

Forward

1. [**Communication**]: The client sends its query to the Scheduler.
    - The client can terminate itself only after it receives a reply (in the “Reply” phase).
2. [**Computation**]: The scheduler decodes the query and decides which hospital(s) should handle the queries.

Scoring

1. [**Communication**] Scheduler constructs a message based on client query, and sends such a message to those available Hospitals.

2. [**Computation**]: Each Hospital performs some operations on the graph / map for distance calculation (you can choose any shortest path algorithm you like). You need to combine the hospital-client distance and hospital availability to get the final score (see "DETAILED EXPLANATION" for the algorithm of computing the final score).
3. [**Communication**]: Each Hospital sends the score to the Scheduler.

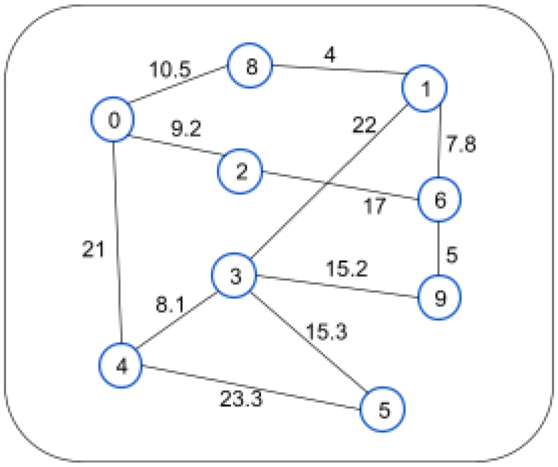
Reply

1. [**Computation**]: Scheduler decodes the messages from Hospitals and then decides which hospital the client should go to.
2. [**Communication**]: Scheduler prepares a reply message and sends it to the Client, and also sends a message to the selected Hospital to let it update its occupancy / availability.
3. [**Communication**]: Client receives the decision from Scheduler, and then terminates itself.
4. [**Computation**]: Selected Hopital updates its availability.

The format of map.txt is defined as follows.

```
<Vertex index for one end of the edge> <Vertex index for the
other end> <Distance between the two vertices>
... (Specification for other edges)
...
```

Let's consider an example:



LA

Let’s say there are nine locations. The distance between two locations is denoted as the edge weight. One valid map.txt may look like this:

```
3 1 22
0 8 10.5
1 3 22
3 5 15.3
0 4  21
1 8 4
4 5 23.3
1 6 7.8
3  4 8.1
2 6 17
1 8  4
0 2 9.2
3 9 15.2
6 9 5
```

Note that the graph is **undirected**, meaning that if there is an edge from u to v with weight d, then there must be an edge from v to u with the same weight d. However, in map.txt, it is possible that only one of the “u v d” and “v u d” rows will appear. For example, for the edge connecting nodes 3 and 1, there are two rows “3 1 22” and “1 3 22”. For the edge connecting nodes 3 and 5, there is only one row “3 5 15.3”. In other words, if we add a row “5 3 15.3” to the above map.txt, the graph remains the same.

More assumptions on the file map.txt:

1. We consider undirected connections between any pair of locations. See above.
2. The graph is connected. *i.e.*, for *any* two nodes u and v in the graph, there must be a path connecting u and v.
3. Each edge will have at least one line in the text file.
4. Locations are represented by non-negative integer numbers between 0 to 2^31 - 1.
    - This ensures that you can always use int32 to store the locations.
    - When Hospitals construct internal representations of the graph, they may re-index the locations. For example, a node 2147483647 may be re-indexed to 10. The re-indexing may improve execution time for certain graph data structures, but this re-indexing step is not mandatory.
5. Locations indexes may not be consecutive. *i.e.*, if a city contains N locations, their location# do not need to be 0, 1, 2, …, N-1. See the example above, there is no number 7.
6. Location index may not start from 0.

7. The LA city will have at least 3 locations, and at most 100 locations.
8. The whole map.txt does not contain any empty lines.
9. There may be *duplicate* lines representing the same edge. For example, see “3 1 22” and “1 3 22”
10. The location indices in map.txt are separated by white space(s). That is, there will be at least one white space between two locations, but there *can also be multiple spaces*. For example, see “1 8 4” and “1 8 4”.
11. The location indices in the text are not sorted.
12. The distance between two locations will be float type with the range [1.00, 10000.00)

Sample map.txt will be provided for you as reference. They will be posted on DEN and Piazza. Other map.txt will be used for grading, so you are advised to prepare your own files for testing purposes.

Source Code Files

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

1. Scheduler: You must name your code file: **scheduler.c** or **scheduler.cc** or **scheduler.cpp** (all small letters). Also you must name the corresponding header file (if you have one; it is not mandatory) **scheduler.h** or **scheduler.hpp** (all small letters).

2. Hospital A, B and C: You must use one of these names for this piece of code: **hospital#.c** or **hospital#.cc** or **hospital#.cpp** (all small letters except for #). Also you must name the corresponding header file (if you have one; it is not mandatory) **hospital#.h** or **hospital#.hpp** (all small letters, except for #). The “#” character must be replaced by the server identifier (i.e. A, B or C), e.g., hospitalA.c.

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

Note: Your compilation should generate separate executable files for each of the components listed above.

<u>DETAILED EXPLANATION</u>

Phase 1 Boot-up

All server programs (Scheduler, Hospital A & B & C) 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 in the late section of the document. As the boot up message indicates, each server must listen on the appropriate port for incoming packets/connections.

Scheduler: The scheduler is booted-up first. It simply waits for the messages from the Hospitals once the hospitals finish the initialization.

Hospitals: Each of the three hospitals perform the following two operations during boot-up:

1. Read in map.txt and construct the graph.
2. Initialize their capacity and occupancy and inform the scheduler.

<u>Step 1</u>: Each Hospital server needs to convert the map.txt into a graph. There are many ways to represent a graph. For example, adjacency matrix, adjacency list or Compressed Sparse Row (CSR) format. You need to decide which format to use based on the requirement of the problem. You can use any format as long as you generate the correct allocation.

For example, suppose the Hospital server re-indexed the LA map shown in the PROBLEM STATEMENT as

Original location #	**Re-indexed location #**
0	0
1	1
2	2
3	3
4	4
5	5
6	6

8	7
9	8

Then the adjacency matrix for the graph will be:

$$\begin{bmatrix} 0 & 0 & 9.2 & 0 & 21 & 0 & 0 & 10.5 & 0 \\ 0 & 0 & 0 & 22 & 0 & 0 & 7.8 & 4 & 0 \\ 9.2 & 0 & 0 & 0 & 0 & 0 & 17 & 0 & 0 \\ 0 & 22 & 0 & 0 & 8.1 & 15.3 & 0 & 0 & 15.2 \\ 21 & 0 & 0 & 8.1 & 0 & 23.3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 15.3 & 23.3 & 0 & 0 & 0 & 0 \\ 0 & 7.8 & 17 & 0 & 0 & 0 & 0 & 0 & 5 \\ 10.5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 15.2 & 0 & 0 & 5 & 0 & 0 \end{bmatrix}$$

where the element at the i-th row and j-th column will be the distance between the (re-indexed) location i and location j; zero of a non-diagonal element means there is no direct path between these two locations.

Step 2: Each Hospital initializes its capacity and occupancy from command line arguments.

```
./hospitalA <location A> <total capacity A> <initial occupancy A>
```

For example, Hospital A at location 2 with total capacity 10 and initial occupancy 8:

```
./hospitalA 2 10 8
```

After each hospital finishes its own initialization, it sends the total capacity and initial occupancy to the Scheduler via UDP.

On-screen messages should be printed to indicate the status of the two steps.

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

```
./client <Location of the client>
```

For example, if a client at location 3 (the index is the **original** index specified in map.txt) wants to schedule a hospital appointment, then the command should be:

```
./client 1234 3
```

After booting up, the client establishes TCP connections with the Scheduler. Then the client sends the location index to the Scheduler. Finally, the client should print a message in a specific format.

This ends Phase 1.

Phase 2 Forward

In the previous phase, the client sends the query parameters to the Scheduler over TCP socket connection. In phase 2, the Scheduler will query Hospital A, B, and C for assigning the client to an appropriate hospital, considering both the availability and the traveling distance.

The socket connection between the Scheduler and Hospital A, B and C are established over UDP. Each of these Hospitals and the Scheduler have its unique port number specified in the "Port Number Allocation" section with the source and destination IP address as localhost/127.0.0.1/::1.

Specifically, the scheduler needs to send the client location to Hospital A, B and C through UDP. Note that because the Scheduler has received the initial capacity and occupancy, it will not send to the hospital that is already fully occupied. In other words, the Scheduler will only send messages to those hospitals that have availability.

Hospital A, B and C are required to print out on screen messages after executing each action as described in the "On Screen Messages" section. These messages will help with grading in the event that the process did not execute successfully. Missing some of the on screen messages might result in misinterpretation that your process failed to complete. Please follow the format when printing the on screen messages.

Phase 3 Scoring

Phase 3 starts when the Hospital A, B and C have received the client request and the location information. The hospitals will calculate a score according to the distance and availability. Distance refers to the shortest path in the given map between the client and a hospital, any shortest path algorithm is acceptable as long as you are showing the correct results. Availability describes how free a hospital is. Each hospital will be provided with a total capacity and an initial occupation (via command line arguments), through which we can acquire the availability.

Availability is represented by a floating point value between 0-1. Zero means the hospital is extremely busy and there is no spare resource to treat a new client, while one means the hospital is not busy at all and totally available. The score is defined as below using these two factors:

```
d = shortest distance

a = availability = (capacity-occupation)/capacity

score = 1/(d*(1.1-a))
```

For example, for a client at location 1, through Dijkstra's we find the shortest path to Hospital A is 7, to Hospital B is 10 and to Hospital C is 20. The availability of Hospital A, B, C are 0.2, 0.3 and 0.1, respectively. From the setting, we know that Hospital A is closer but busier than Hospital B. Hospital C is both farther away and less available Using the above definition to compute the scores for each hospital, the score of Hospital A is 0.159, which is larger than the score of Hospital B 0.125. As a result, the Scheduler should assign the client to Hospital A.

Hospital A:

```
Capacity = 10

Occupation = 8

a = (10-8)/10=0.2

d = 7

score = 1/(7*(1.1-0.2)) = 0.159
```

Hospital B:

```
Capacity = 20

Occupation = 14

a = (20-14)/20=0.3
```

```
d = 10

a = 0.3

score = 1/(10*(1.1-0.3))= 0.125
```

You can decide your own algorithm to find the shortest path between two nodes, e.g. Bellman-ford, Dijkstra's, etc. Let's consider an example where the client is at location L. The following are all the possible cases:

1. L is in the map:

    a. L is not the same as hospital location: the distance is the value of shortest path;

    b. L is the same as hospital location (d=0): the distance is `None`;

2. L is not in the map: the distance is `None`.

Concerning the availability a, consider the following cases:

1. `a>=0` and `a<=1`: use the availability `= a`;
2. `a<0` or `a>1`: availability `= None`;

Concerning the final score:

1. If either availability = `None` or distance = `None`, then the score = `None`
2. Otherwise, compute the score using the defined equation.

After being assigned to a hospital, the hospital occupation needs to increase by 1, the availability and the final score will also need to be updated. For the example, Hospital A is assigned to a new client, the occupation will be increased to 9, the information of Hospital A for the same client location will be updated as (you only need to print the new occupation and new availability. See the tables for On-screen messages):

Hospital A:

```
Capacity = 10

Occupation= 9
```

```
a = (10-9)/10=0.1

d = 7

score = 1/(7*(1.1-0.1)) = 0.143
```

The score is decreased due to a lower availability.

Phase 4 Reply

At the end of Phase 3, the scores for all hospitals should be prepared and sent back to the Scheduler using UDP. When the Scheduler receives the score, it needs to forward the decision of assignment to the corresponding Client using TCP.

Note if there is a score tie for both hospitals, the Scheduler should assign a client to a closer hospital with smaller distance. To achieve this feature, you may need to send both the final score and the distance value to the Scheduler.

The rule is as follows:

1. Distance A OR Distance B OR Distance C == None:
    a. Reply None, because this means the location information is illegal
2. If Distances are legal:
    a. Score A/B/C are not all the same, reply the hospital with the highest score
    b. Tie for highest scores, reply the hospital with the shortest distance
    c. Any hospital with Score == None will not participate in comparison.
    d. If all scores == None, reply None.

You should decide on your own what information is required to be sent from the Scheduler to the hospitals. Similarly, it is your own choice what format/data structure to encode the communication. We will grade based on your print-out message only.

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

DOWNLOAD SAMPLES

Samples of map.txt for this project will be available online for download. The download link will be posted on DEN and Piazza soon.

PORT NUMBER ALLOCATION

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

Table 1. Static and Dynamic assignments for TCP and UDP ports

Process	**Dynamic Ports**	**Static Ports**
Hospital A		UDP: 30xxx
Hospital B		UDP: 31xxx
Hospital C		UDP: 32xxx
Scheduler		UDP(with hospital): 33xxx TCP(with client): 34xxx
Client	TCP	

NOTE: xxx is the last 3 digits of your USC ID. For example, if the last 3 digits of your USC ID are "319", you should use the port: **30319** for the Hospital (A), etc.

Port number of all processes print port number of their own

ON SCREEN MESSAGES

Table 2. Hospital A on screen messages

Event	**On Screen Message**
Booting up (Only while starting):	Hospital A is up and running using UDP on port `<Hospital A port number>`. Hospital A has total capacity `<total capacity A>` and initial occupancy `<initial occupancy A>`.
For location finding, upon receiving the input query:	Hospital A has received input from client at location `<Source Vertex Index>`
For location finding, no location found	Hospital A does not have the location `<Source Vertex Index>` in map
For location finding, no location found after sending to the Scheduler:	Hospital A has sent " location not found" to the Scheduler
For calculating the availability	Hospital A has capacity = `<capacity>`, occupation= `<occupation>`, availability = `<availability>`
For finding the shortest path	Hospital A has found the shortest path to client, distance = `<distance>`
For calculating the final score	Hospital A has the score = `<score>`
For graph finding, after sending to the Scheduler:	Hospital A has sent score = `<score>` and distance = `<distance>` to the Scheduler
For receiving the result from the Scheduler	Hospital A has been assigned to a client, occupation is updated to `<new occupation>`, availability is updated to `<new availability>`

Note that if any of the results are defined as "`None`", then just show "`None`". For example, if the distance = 0, which is not a legal value and is defined to be "`None`" in Phase 3, you should show on screen that "distance = None".

For on-screen messages of Hospitals B and C, replace "A" with "B" or "C" in Table 2.

Table 3. Scheduler on screen messages	
Event	**On Screen Message**
Booting up (only while starting):	The Scheduler is up and running.
Upon Receiving the information of Hospital A/B/C:	The Scheduler has received information from Hospital A/B/C: total capacity is `<total capacity A/B/C>` and initial occupancy is `<initial occupancy A/B/C>`
Upon Receiving the input from the client:	The Scheduler has received client at location `<Source Vertex Index>` from the client using TCP over port `<Scheduler TCP port number>`
After sending information to Hospital A/B/C	The Scheduler has sent client location to Hospital A/B/C using UDP over port `<Scheduler UDP port number>`
After receiving results from Hospital A/B/C	The Scheduler has received map information from Hospital A/B/C, the score = `<score>` and the distance = `<distance>`
After making the assignment	The Scheduler has assigned Hospital `<A/B/C>` to the client
After sending results to client	The Scheduler has sent the result to client using TCP over port `<Scheduler TCP port number>`
After sending results to the assigned hospital A/B/C	The Scheduler has sent the result to Hospital `<A/B/C>` using UDP over port `<Scheduler UDP port number>`

Table 4. Client on screen messages	
Event	**On Screen Message**
Booting Up:	The client is up and running
After sending query to the Scheduler	The client has sent query to Scheduler using TCP: client location `<vertex index>`
After receiving output from the Scheduler	The client has received results from the Scheduler: assigned to Hospital `<A/B/C/None>`
After receiving output the Scheduler, errors	Location `<vertex index>` not found or Score = None, No assignment

ASSUMPTIONS

1. You have to start the processes in this order: **Scheduler, HospitalA, HospitalB, HospitalC, and Client.**

2. The map.txt file is stored in your project root directory before your program starts.

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

4. 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 (i.e., have a line of comment for) 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 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 (assume "developer" is your linux user name):

```
ps -aux | grep developer
```

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

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

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 hard-coded as **localhost (127.0.0.1)** in all codes.

3. The Hospital servers and the Scheduler server should keep running and be waiting for another request until the TAs terminate them by Ctrl+C. If they terminate before that, you will lose some points for it. The client should terminate itself after receiving the response.

4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.

6. All the on-screen messages must conform 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.

8. Please do remember to close the socket and tear down the connection once you are done using that socket.

Programming Platform and Environment

1. All your submitted code **MUST** work well on the provided virtual machine Ubuntu (for students with Mac of M1-chip, we will provide VM alternatives soon).

2. All submissions will only be graded on the provided Ubuntu. TAs won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code works well on the provided Ubuntu. "It works well on my machine" is not an excuse and we don't care.

3. Your submission MUST have a Makefile. Please follow the requirements in the following "Submission Rules" section.

Programming Languages and Compilers

You must use only C/C++ on UNIX as well as UNIX Socket programming commands and functions. Here are the pointers for Beej's Guide to C Programming and Network Programming (socket programming):

http://www.beej.us/guide/bgnet/

(If you are new to socket programming please do study this tutorial carefully as soon as possible and before starting the project)

http://www.beej.us/guide/bgc/

You can use 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, 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:

Compile **all** your files and creates executables	`make all`
Run hospitalA	`./hospitalA <location A> <capacity A> <initial occupancy A>`
Run hospitalB	`./hospitalB <location B> <capacity B> <initial occupancy B>`
Run hospitalC	`./hospitalC <location C> <capacity C> <initial occupancy C>`
Run scheduler	`./scheduler`
Run client	`./client <location index>`

TAs will first compile all codes using **make all**. They will then open 5 different terminal windows to execute the commands above. **Remember that hospitals should always be on once started.** TAs will check the outputs for multiple queries. The terminals should display the messages specified in the on-screen message tables.

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 the first command.

2. Do NOT include anything not required in your tar.gz file. Do NOT use subfolders. Any compressed format other than .tar.gz will NOT be graded!

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 keep a history of all your submissions. If you make multiple submissions, we will grade your latest valid submission. Submission after the deadline is considered as invalid.

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 Scheduler. But the result is not received by the client because Scheduler 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 print out the appropriate results.

4. If your submitted codes do not even compile, you will receive 5 out of 100 for the project.

5. 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.

6. 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.)

7. **If you add subfolders or compress files in the wrong way, your project won't 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. 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.

10. 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.

11. **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.

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 way for grading purposes and however it will be tested with different inputs. Your designated TA runs your project as is, according to the project description and your README file and then checks whether it works correctly or not. If your README is not consistent with the description in this file, we will follow the description here.

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.