EE450 Socket Programming Project, Spring 2023

Due Date : Sunday, Apr 23, 11:59PM (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. **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 D2L. **You must discuss all project related issues on the Piazza discussion forum**. We will give those who actively help others out by answering questions on the Piazza discussion forum up to 10 bonus points.

Problem Statement:

Finding a time that works for everyone to meet is often a headache issue, especially for a group of people. People often need to exchange availability through back-and-forth emails and allow for several days to finally schedule a meeting. With the help of a meeting scheduler that gives a power to book meetings with users efficiently, users can save hours of time on unnecessary emailing and focus on more important things. There exist some survey tools online to help a group find the best meeting time by manually filling a shared table with their availability. It could however be more productive if the scheduler can maintain the availability of all users and schedule the meeting at the time that works for everyone accordingly.

In this project, we are going to develop a system to fasten the meeting scheduling process. The system receives a list of names involved in the meeting and returns the time intervals that work for every participant based on their availability. To maintain the privacy of everyone's daily schedule, the availability of all users is stored on backend servers and cannot be accessed by any other servers or clients. Due to a large number of users, more than one backend servers are deployed to store their availability. A main server is then needed to manage which backend server a user's information is stored in. The main server also plays a role of handling requests from clients.

When a user wants to schedule a meeting among a group of people, the user will start a client program, enter all names involved in the meeting and request the main server for the time intervals that works for every participant. Once the main server receives the request, it decides which backend server the participants' availability is stored in and sends a request to the responsible backend server for the time intervals that works for all participants belonging to this backend server. Once the backend server receives the request from the main server, it searches in the database to get all requested users' availability, runs an algorithm to find the intersection

among them and sends the result back to the main server. The main server receives the time slots from different backend servers, runs an algorithm to get the final time slots that works for all participants, and sends the result back to the client. The scheduler then can decide the final meeting time according to the available time recommendations and schedule it in the involved users' calendars.

Without the loss of generality, we assume there are one client, one main server and two backend servers in our project, as indicated in Figure 1. The full process can be roughly divided into four phases: **Boot-up**, **Forward**, **Schedule**, **Reply** (read details in "Application Workflow Phase Description" section).

- Client: used to access the meeting scheduling system.
- Main server (serverM): coordinate with the backend servers.
- Backend server (serverA and serverB): store the availability of all users and get the time slots that work for all meeting participants once receiving requests.

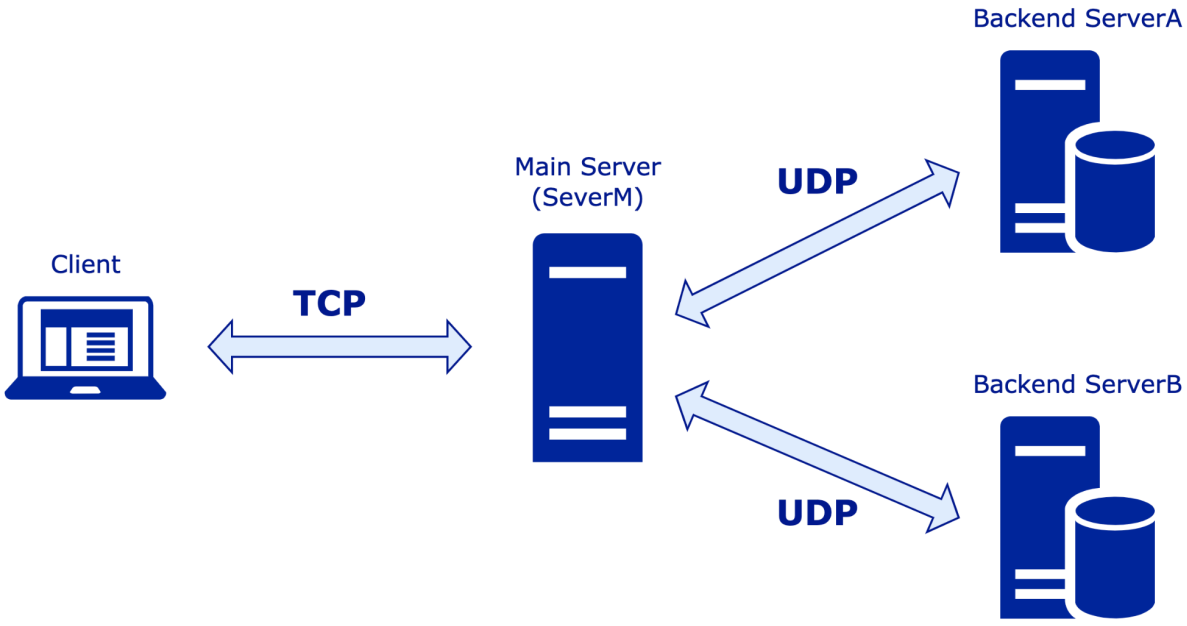


Figure 1: Illustration of the system

Source Code Files

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

1. ServerM (Main Server): You must name your code file: **serverM.c** or **serverM.cc** or **serverM.cpp** (all small letters except 'M'). Also you must include the

corresponding header file (if you have one; it is not mandatory) serverM.h (all small letters except 'M').

2. Backend-Servers 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 include the corresponding header file (if you have one; it is not mandatory). **server#.h** (all small letters, except for #). The "#" character must be replaced by the server identifier (i.e. A or B), depending on the server it corresponds to. (e.g., serverA.cpp & serverB.cpp)

   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).

Input Files:

There are two input files provided, **a.txt** for backend server A and **b.txt** for backend server B. Each file is accessible by its corresponding backend server only. The main server or client should not be able to access either of the files. Two files share the same format as follows.

The file contains a list of time availability for a group of individuals identified by their usernames. The format of the time availability information stored in the file for a specific user is provided below.

```
saylor;[[5,10],[11,16]]
```

The format of each line will always be `username;time_availability`, ending with a newline character. Username and time availability are always separated by a semicolon. The username contains only small letter alphabets with a maximum length of 20. The usernames present in both the files are completely unique.

The time availability consists of a list of time intervals in the format:

```
[[t1_start,t1_end],[t2_start,t2_end]... [t10_start,t10_end]]
```

Each time interval contains a start and an end time which are always non-negative

integers. The start and end times are separated by a comma, and the time intervals are also separated by a comma. The end time will always be larger than the start time, i.e., `t[i]_start < t[i]_end`, and the start time of the next time interval will always be larger than the end time of the current time interval, i.e., `t[i]_end < t[i+1]_start`. Each username can contain a maximum of 10 time intervals and a minimum of 1 time interval. If a user has only 1 time interval then the format would be:

```
Username;[[t1_start,t1_end]]
```

The minimum start time for each username has to be at least 0 and the maximum end time of each username can not exceed 100. There can be spaces at any point in the line (except inside a username), you have to remove the extra spaces in your preprocessing. Please find the following examples to see what is a valid or invalid line in the input files:

```
saylor;[[0,100]]
```
This is a valid line in the input files.

```
saylor;[[1,2],[3,4],[5,6],[7,8],[9,10],[11,12],[13,14],[15,16],[17,18],[19,20]]
```
This is a valid line in the input files.

```
saylor;[[-1,10.5]]
```
This is an invalid line because only non-negative integer times are allowed. Such a line will not exist in the input files.

```
sayl!or;[[a,10]]
```
This is an invalid line because usernames should have only small letters and no special characters, other than the ones specified. Time intervals will have digits only. Such a line will not exist in the input files.

```
Saylor;[[1,101]]
```
This is an invalid line because usernames should only have small letters. Such a line will not exist in the input files.

```
say lor;[[1,10],[11,12]]
```
This is an invalid line because usernames should have small letters only and no spaces occur between the letters. Such a line will not exist in the input files.

```
saylor            ;[[1   ,10]  ,[11,12] ]
```
This is a valid line. You should consider “saylor” as the username (ignoring all the whitespaces) and `t1_start = 1` and `t1_end = 10`.

`saylor;[[1,1]]`

This is an invalid line because end time should always be larger than start time. Such a line will not exist in the input files.

`saylor;[[1,2],[2,3]]`

This is an invalid line because the start time of the second interval should be greater than the end time of the first interval. Such a line will not exist in the input files.

Example a.txt and b.txt files are provided along with the project document for you to test your code on, these files contain only 10 lines, but the final grading will be performed on files which can contain up to 200 lines in each file. Your code should be able to handle up to 200 lines in both files. You are encouraged to design more cases including edge cases to test your program.

Application Workflow Phase Description:

Phase 1: Boot-up

Please refer to the “Process Flow” section to start your programs in order of the main server, server A, server B and Client. Your programs must start in this order. Each of the servers and the client have boot-up messages which have to be printed on screen, please refer to the on-screen messages section for further information.

When two backend servers (server A and server B) are up and running, each backend server should read the corresponding input file and store the information in a certain data structure. You can choose any data structure that accommodates the needs. After storing all the data, server A and server B should then send all usernames they have to the main server via UDP over the port mentioned in the PORT NUMBER ALLOCATION section. Since the usernames are unique the main server will maintain a list of usernames corresponding to each backend server. In the following phases you have to make sure that the correct backend server is being contacted by the main server for corresponding usernames. You should print correct on screen messages onto the screen for the main server and the backend servers indicating the success of these operations as described in the “ON-SCREEN MESSAGES" section.

After the servers are booted up and the required usernames are transferred from the backend servers to the main server, the client will be started. Once the client boots up and the initial boot

up messages are printed, the client waits for the usernames to be taken as inputs from the user, The user can enter up to 10 usernames (all of which are separated by a single space).

After booting up. Your client should first show a prompt:

```
Please enter the usernames to check schedule availability:
```

Assuming one entered:

```
theodore callie
```

You should store the above two usernames (`theodore` and `callie`). Once you have the usernames stored in your program, you can consider phase 1 of the project to be completed. You can proceed to phase 2.

Phase 2: Forward

Once usernames involved in the meeting are entered in the client program, the client forms a message with these names and sends it to the main server over the established TCP connection. You can use any format for the message as long as the main server can receive the correct and intact list of usernames and can extract the information properly for the next step.

The main server first examines if all usernames exist in the database of backend servers according to the username list received from two backend servers in Phase 1. For the usernames that do not exist in the username list, it replies the client with a message saying that which usernames do not exist. For those that do exist, the main server splits them into two sublists based on where the user information is located and forwards those names to the corresponding backend server through UDP connection. Each of these servers have its unique port number specified in the “PORT NUMBER ALLOCATION” section with the source and destination IP address as “localhost” or “127.0.0.1”.

Client, main server and two backend servers 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 does 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: Schedule

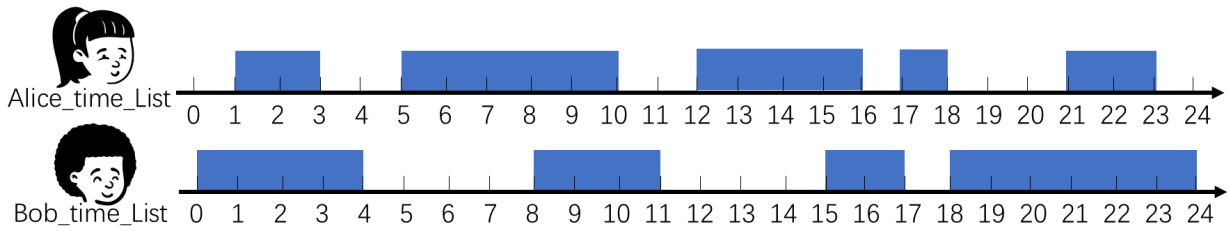
Once the backend servers receive the usernames, in order to help each participant schedule a common available time, in this phase, you will need to implement an algorithm to find the intersection of time availability of multiple users.

In the case of one participant, the backend server can directly send the time availability (i.e., a list of time intervals) back to the main server.

In the case of two participants, for example Alice and Bob, the backend server should first search in its database and find the time availability of them. The time availability of Alice and Bob are two lists of time intervals as stated in the "Input Files" section, denoted as

```
Alice_Time_List=[[t1_start,t1_end],[t2_start,t2_end],...]
Bob_Time_List=[[t1_start,t1_end],[t2_start,t2_end],...]
```

An algorithm should be developed with `Alice_Time_List` and `Bob_Time_List` as the input and output the intersection of time intervals of two lists. An illustration of time intervals and some example inputs and outputs are given in Figure 2, in which the intersection between time intervals `[5,10]` and `[8,11]` which is `[8,10]` are considered as a valid intersection (and the two time lists can have more than one valid intersections). However, the intersection between `[15,17]` and `[17,18]` which is `[17,17]` is **NOT** an intersection in our scenario because the definition of time intervals requires `t[i]_start < t[i]_end` and `t[i]_end < t[i+1]_start` as stated in the "Input Files" section.



Example 1:
Input:
```
Alice_time_List  = [[1,3], [5,10], [12,16], [17,18], [21,23]],
Bob_time_List = [[0,4], [8,11], [15,17], [18,24]]
```
Output:
```
[[1,3],  [8,10], [15,16], [21,23]]
```

Example 2 (no intersection):
Input:
```
Alice_time_List  = [[1,2], [7,10]],
Bob_time_List = [[3,5], [15,20]]
```
Output:
```
[]
```

Example 3 (no intersection):
Input:
```
Alice_time_List  = [[1,7], [9,10]],
Bob_time_List = []
```
Output:
```
[]
```

Figure 2: Illustration of time intervals and example inputs and outputs

In the case of more than two participants, you may run the algorithm for two participants iteratively, and each time find the intersection between the previously found intersection result and the time list of a new participant. You can also develop other algorithms that can directly work for multiple users. An example of the whole process is given as follows. A backend server receives the request for the time availability among Alice, Bob and Amy. The backend server first finds in its database the time availability of them obtaining

```
Alice_Time_List=[[1,10],[11,12]]
Bob_Time_List=[[5,9],[11,15]]
Amy_Time_List=[[4,12]]
```

An algorithm is then runned to output the intersection result `[[5,9],[11,12]]`.

At the end of this phase, a backend server should have the intersection result among all participants, which is a list of time intervals, and be ready to send it to the main server. If no time interval is found, the result should be `[]` as shown in figure 2. The backend servers are required to print out on-screen messages after executing each action as described in the "ON-SCREEN MESSAGES" section.

Phase 4: Reply

During this phase, the backend servers send the intersection result among all participants to the main server via UDP. Once the main server receives the results, the main server runs the algorithm for two participants developed in Phase 3 to identify the intersection between two intersection results received from two backend servers. For example, the main server receives the following intersection results which are two lists of time intervals from two backends servers

```
result_1= [[6,7],[10,12],[14,15]]
result_2= [[3,8],[9,15]]
```

Then the main server should feed these time intervals to your algorithm and have the result `[[6,7],[10,12],[14,15]]` ready. Then the main server sends the intersection result which is a list of time intervals back to the client via TCP.

When the client receives the result, it will print out the result and the prompt messages for a new request as follows:

```
Time intervals <[time intervals]> works for <username1, username2, …>.
```

```
-----Start a new request-----
Please enter the usernames to check schedule availability:
```

All servers and the client are required to print out on-screen messages after executing each action as described in the “ON-SCREEN MESSAGES" section.

Extra Credits: (10 points extra, not mandatory)

After showing the meeting time recommendations (i.e., the intersection of all time intervals), a user can decide the final meeting time and schedule it in the involved users’ calendars. In this extra credit part, you should develop a functionality to schedule a meeting in the participants’ calendar. This phase happens after the previous four phases but before starting a new request.

The program should ask the user to enter the meeting time by printing the following prompt:

```
Please enter the final meeting time to register an meeting:
```

The user should enter an interval `[t_start, t_end]` chosen from the recommendations, such as `[1, 2]`.

This entered time interval must be one of the intervals in the recommendations. For example, the recommendations are `[[1,3],[8,10],[15,16],[21,23]]`, then the acceptable meeting times are `[1,2]` or `[2,3]` or `[1,3]`, etc.. But `[2,4]` or `[3,5]` are not acceptable. Your program should check if the entered time period is valid or not. If it is not valid, you should print a message and ask the user to enter again:

```
Time interval <[t_start, t_end]> is not valid. Please enter again:
```

If it is valid, the client should pass the interval to the main server and the main server passes it to the corresponding backend server based on the entered username in Phase 1. The backend servers then remove this time interval from the involved users time availability list, indicating that this time slot is occupied by a meeting. For example, the original availability of a user is `[[1,5],[7,8]]`. After registering `[1,2]` as a meeting time, the availability in the database becomes `[[2,5],[7,8]]`. The backend server should print the on-screen messages showing the updates:

```
Register a meeting at <time slot> and update the availability for the following users:
<username 1>: updated from <original time availability list> to <updated time availability list>
```

```
<username 2>: updated from <original time availability list> to <updated time availability list>
…
```

For example:

```
Register a meeting at [1,2] and update the availability for the following users:
Alice: updated from [[1,2],[7,10]] to [[7,10]]
Bob: updated from [[1,5],[7,8]] to [[2,5],[7,8]]
```

After updating, the backend server sends a message to the main server indicating that the update is finished. And the main server will notify the client so that the client can start a new request.

NOTE: The extra points will be added to the full 100 points. The maximum possible points for this socket programming project is 110 points.

Process Flow/ Sequence of Operations:

- Your project grader will start the servers in this sequence: ServerM, ServerA, ServerB and client in 4 different terminals.
- Once all the ends are started, the servers and clients should be continuously running unless stopped manually by the grader or meet certain conditions as mentioned before.

Required Port Number Allocation

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

Note: Major points will be lost if the port allocation is not as per the below description.

Table 3. Static and Dynamic assignments for TCP and UDP ports.		
Process	**Dynamic Ports**	**Static Ports**
serverA	-	1 UDP, 21000+xxx
serverB	-	1 UDP, 22000+xxx
serverM	-	1 UDP, 23000+xxx 1 TCP, 24000+xxx
Client	1 TCP	<Dynamic Port assignment>

NOTE: xxx is the last 3 digits of your USC ID. For example, if the last 3 digits of your USC ID are "319", you should use the port: **21000+319 = 21319** for the Backend-Server (A). **It is NOT going to be 21000319.** Note that the serverM has only one UDP port. The same port is used to connect to both of the backend servers

ON SCREEN MESSAGES: Table 4. Backend Server on screen messages (For both A and B Backend Servers)	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	Server <A or B> is up and running using UDP on port <port number>.
After sending the list of usernames to the main server:	Server <A or B> finished sending a list of usernames to Main Server.
After receiving the usernames from the Main Server:	Server <A or B> received the usernames from Main Server using UDP over port <port number>.
After running the algorithm to find the intersections of time availability among all participants:	Found the intersection result: <[[t1_start, t1_end], [t2_start, t2_end], …]> for <username1, username2, …>.
After sending the results to the main server:	Server <A or B> finished sending the response to Main Server.

ON SCREEN MESSAGES: Table 7. Main Server on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (only while starting):	Main Server is up and running.
After receiving the username list from server A or B:	Main Server received the username list from server<A or B> using UDP over port <port number>.
After receiving the request from client:	Main Server received the request from client using TCP over port <port number>.
If some usernames do not exist in the username list:	<username1, username2, …> do not exist. Send a reply to the client.
If the usernames exist in the username list, the main server sends the corresponding usernames to the responsible backend server:	Found <username1, username2, …> located at Server <A or B>. Send to Server <A or B>.
After receiving the intersection result from the Main server:	Main Server received from server <A or B> the intersection result using UDP over port <port number>: <[[t1_start, t1_end], [t2_start, t2_end], …]>.
Upon finishing finding the intersection between the results from server A and B:	Found the intersection between the results from server A and B: <[[t1_start, t1_end], [t2_start, t2_end], …]>.
After sending the final result (i.e., meeting time recommendations) to the client:	Main Server sent the result to the client.

ON SCREEN MESSAGES: Table 8. Client on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up:	Client is up and running.
Asking user to enter username:	Please enter the usernames to check schedule availability: <username1> <username2> … <username10>
After sending the usernames to the main server:	Client finished sending the usernames to Main Server.
After receiving the reply from main server if some usernames do not exist:	Client received the reply from Main Server using TCP over port <port number>: <username1, username2, …> do not exist.
After receiving the final result (i.e., meeting time recommendations) from main server:	Client received the reply from Main Server using TCP over port <port number>: Time intervals <[[t1_start, t1_end], [t2_start, t2_end], …]> works for <username1, username2, …>.
Upon finishing all operations and starting a new request:	-----Start a new request----- Please enter the usernames to check schedule availability:

Submission files and folder structure:
(Additionally, refer #2 of submission rules for more details)

Your submission should have the following folder structure and the files (the examples are of .cpp, but it can be .c files as well):

- **ee450_lastname_firstname_uscusername.tar.gz**
  - **ee450_lastname_firstname_uscusername**
    - **client.cpp**
    - **serverM.cpp**
    - **serverA.cpp**
    - **serverB.cpp**
    - **Makefile**
    - **readme.txt (or) readme.md**
    - **<Any additional header files>**

The grader will extract the tar.gz file, and will place all the input data files in the same directory as your source files. The executable files should also be generated in the same directory as your source files. So, after testing your code, the folder structure should look something like this:

- **ee450_lastname_firstname_uscusername**
  - **client.cpp**
  - **serverM.cpp**
  - **serverA.cpp**
  - **serverB.cpp**
  - **Makefile**
  - **readme.txt (or) readme.md**
  - **client**
  - **serverM**
  - **serverA**
  - **serverB**
  - **a.txt**
  - **b.txt**
  - **<Any additional header files>**

Note that in the above example, the input data files (ee.txt, cs.txt and cred.txt) will be manually placed by the grader, while the 'make all' command should generate the executable files.

Example Output to Illustrate Output Formatting:

Backend-ServerA Terminal:
ServerA is up and running using UDP on port 21319.
ServerA finished sending a list of usernames to Main Server.
Server A received the usernames from Main Server using UDP over port 21319.
Found the intersection result: [[1,3],[12,18]] for eliana, alice.
Server A finished sending the response to Main Server.

Backend-ServerB Terminal:
ServerB is up and running using UDP on port 22319.
ServerB finished sending a list of usernames to Main Server.
Server B received the usernames from Main Server using UDP over port 22319.
Found the intersection result: [[2,5],[17, 20]] for theodore, bob, amy.
Server B finished sending the response to Main Server.

Main Server Terminal:
Main Server is up and running.
Main Server received the username list from server A using UDP over port 23319.
Main Server received the username list from server B using UDP over port 23319.
Main Server received the request from the client using TCP over port 24319.
luis do not exist. Send a reply to the client.
Found eliana, alice located at Server A. Send to Server A.
Found theodore, bob, amy located at Server B. Send to Server B.
Main Server received from server A the intersection result using UDP over port 23319:
[[1,3],[12,18]].
Main Server received from server B the intersection result using UDP over port 23319:
[[2,5],[17, 20]].
Found the intersection between the results from server A and B:
[[2,3],[17, 18]].
Main Server sent the result to the client.

Client Terminal:
Client is up and running.
Please enter the usernames to check schedule availability:
eliana theodore luis bob alice amy
Client finished sending the usernames to Main Server.

Client received the reply from the Main Server using TCP over port <port number>:
luis do not exist.
Client received the reply from the Main Server using TCP over port <port number>:
Time intervals [[2,3],[17, 18]] works for eliana, theodore, bob, alice, amy.
-----Start a new request-----
Please enter the usernames to check schedule availability:

(***Note***: you should replace <port number> with the TCP port number dynamically assigned by the system.)

The following is only for the **extra credits** part. Only key messages are given here, for other messages you can follow a similar format as in previous phases.

Client Terminal:
Time intervals [[2,3],[17, 18]] works for eliana, theodore, bob, alice, amy.
Please enter the final meeting time to register an meeting:
[3,4]
Time interval [3,4] is not valid. Please enter again:
[17,18]
Sent the request to register [17,18] as the meeting time for eliana, theodore, bob, alice, amy.
…
Received the notification that registration has finished.
-----Start a new request-----
Please enter the usernames to check schedule availability:

Backend-ServerA Terminal:
Register a meeting at [17,18] and update the availability for the following users:
eliana: updated from [[1,5],[12,18]] to [[1,5],[12,17]]
alice: updated from [[1,3],[10,24]] to [[1,3],[10,17],[18,24]]
Notified Main Server that registration has finished.

Backend-ServerB Terminal:
Register a meeting at [17,18] and update the availability for the following users:
theodore: updated from [[1,5],[17,20]] to [[1,5],[18,20]]
bob: updated from [[2,7],[16,21]] to [[2,7],[16,17],[18,21]]
amy: updated from [[2,8],[15,20]] to [[2,8],[15,17][18,20]]
Notified Main Server that registration has finished.

Assumptions:

1. You have to start the processes in this order: **ServerM, ServerA, ServerB and client.** 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.
2. 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 cite the copied part in your code. Any signs of academic dishonesty will be taken very seriously.**
3. 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**. 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.

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" or "127.0.0.1"** in all codes.

3. Your client, the backend servers and the main 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.
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 what is already described in the project document.
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/Graders won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code works well on the provided Ubuntu. **"It works well on my machine" is not an excuse**.
3. Your submission MUST have a Makefile. Please follow the requirements in the following "Submission Rules" section.

Programming languages and compilers:

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

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

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

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

You can use a unix text editor like emacs to type your code and then use compilers such as g++ (for C++) and gcc (for C) that are already installed on Ubuntu to compile your code. You must use the following commands and switches to compile yourfile.c or yourfile.cpp. It will make an executable by the name of "yourfileoutput".

gcc -o yourfileoutput yourfile.c
g++ -o yourfileoutput yourfile.cpp

Do NOT forget the mandatory naming conventions mentioned before!

Also inside your code you need to include these header files in addition to any other header file you think you may need:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h
```

Submission Rules:

- Along with your code files, include a **README file and a Makefile**.The Makefile requirements are mentioned in the section below. The README file can be in any format (Markdown or txt). The only requirement is that the TAs should be able to open the file and read it in the studentVM/the VM your project will be graded in without installing any additional software. In the README file please include:
    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, e.g., usernames are concatenated and delimited by a comma, etc.
    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). Reusing functions which are directly obtained from a source on the internet without or with few modifications is considered plagiarism (Except code from the Beej's Guide). Whenever you are referring to an online resource, make sure to only look at the source, understand it, close it and then write the code by yourself. The TAs will perform plagiarism checks on your code so make sure to follow this step rigorously for every piece of code which will be submitted.

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 clean	Removes all the executable files
./serverM	**Run**s Main server
./serverA	**Run**s Backend server A
./serverB	**Run**s Backend server B
./client	Starts the client

TAs will first compile all codes using **make all**. They will then open 4 different terminal windows. On 4 terminals they will start servers M, A and B. On the other terminal, they will start the client using **./client**. **Remember that all programs should always be on once started.** TAs will check the outputs for multiple values of input. The terminals should display the messages shown in On-screen Messages tables in this project writeup.

- Compress all your files including the README file and the Makefile into a single "tar ball" and call it: **ee450_YourLastName_YourFirstName_yourUSCusername.tar.gz** (all small letters) e.g. an example filename would be **ee450_trojan_tommy_tommyt.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 two commands:
  *tar cvf **ee450_YourLastName_YourFirstName_yourUSCusername.tar** **
  *gzip **ee450_YourLastName_YourFirstName_yourUSCusername.tar***

  Now, you will find a file named
  "ee450_YourLastName_YourFirstName_yourUSCusername.tar.gz" in the same directory.
  Please notice there is a space and a star(*) at the end of the first command.
  An example submission would be:
  First Name: John
  Last Name: Doe

USC username: jdoe (This can be found with your email address, In this case the email address would be jdoe@usc.edu)
So the submission would be:
ee450_Doe_John_jdoe.tar.gz
Any compressed format other than .tar.gz will NOT be graded!

- Upload "ee450_YourLastName_YourFirstName_yourUSCusername.tar.gz" to the Digital Dropbox on the DEN website (DEN -> EE450 -> My Tools -> Assignments -> 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.

- 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.

- 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 have not received a confirmation mail, contact your TA if it always fails.

- After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. **This is exactly what your designated TA would do, So please grade your own project from the perspective of the TA.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.

- 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.

- 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!

  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. The grading criteria are subject to change.

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, specially the communications through UDP and TCP sockets.

2. Inline comments in your code. This is important as this will help in understanding what you have done.

3. Whether your programs work as you say they would in the README file.

4. Whether your programs print out the appropriate error messages and results.

5. **Your code will only be tested on a fresh copy of the provided Virtual Machine (either studentVM (64-bit) or Ubuntu 22.04 ARM64 for M1/M2 Mac users). If your programs are not compiled or executed on these VM, you will receive only minimum points as described below. Be careful if you are going to use other environments!!! Do not update or upgrade the provided VM as well!!!**

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

7. 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.

8. The minimum points for compiled and executable codes is 15 out of 100.

9. If your code does not correctly assign the TCP or UDP port numbers (in any phase), you will lose points each.

10. We will use the same test cases to test all the programs. These test cases cover all situations including edge cases.

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 weeks 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 the Piazza Discussion Forum**. 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 D2L.)

13. The maximum points that you can receive for the project with bonus points and extra credits is 110.

14. Your code will not be altered in any way for grading purposes and however it will be tested with different inputs. Your TA/Grader 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, 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 **studentVM (64-bit) or Ubuntu 22.04 ARM64 for M1/M2 Mac users**. 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!!!

Do not post your code on Github, especially in a public repository before the deadline!!!
Double check the setting and do some testing before posting in a private repository!!!
(You can post your code after the deadline.)

Copying code from friends or from any unauthorized resources (webpages, github, etc.) is called **plagiarism** not **collaboration** and will result in an F for the entire course. Any libraries or pieces of code that you use or refer and you did not write must be listed in your README file. Students are only allowed to use the code from Beej's socket programming tutorial. Copying the code from any other resources may be considered as plagiarism. Please be careful!!! 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.