EE450 Socket Programming Project

Part 2

Fall 2024

Due Date:

Wednesday, November 6, 2024, 11:59PM

(Hard Deadline, Strictly Enforced)

OBJECTIVE

The objective of project 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 email the TA your questions, come by TA's office hours, or ask during the weekly discussion session. **You can ask TA any question about the content of the project, but TA has the right to reject your request for debugging.**

PROBLEM STATEMENT

Consider a Student Dormitory Reservation System for managing student housing efficiently. It helps organize room bookings, makes finding and booking rooms easier, and gives useful information for making decisions. Due to the university's multiple campuses, student reservations are processed by several Campus Servers, with each Campus server in charge of multiple departments.

In this part of the project, you will implement socket programming between servers using UDP. There are three Campus servers, A, B, and C, which communicate with the main server through UDP. Each Campus server stores a data file, dataA.txt for server A, dataB.txt for server B, dataC.txt for server C. The data file contains Department names and dormitory information. Main server first asks Campus servers which Department they are responsible for and retrieves a list of Departments from each Campus server. Main server will send a request only to its responsible Campus server which contains the requested Department. The Campus server then replies to the main server with the requested information.

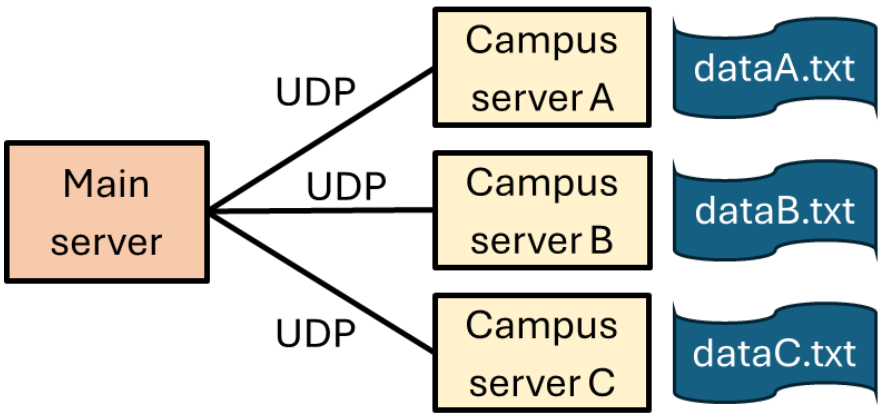

Figure 1: Student Dormitory Reservation System - Part 2

The detailed operations to be performed by all the parties are described with the help of Figure 1. There are in total 4 communication endpoints, which are run in 4 individual terminal windows:

- Main server: decides which Campus server a request should be sent to.
- Campus servers A, B, and C: store data, process, and send results.

The full process can be roughly divided into three phases, and their communication and computation steps are as follows:

Bootup

1. [Computation]: Campus servers A, B, and C read dataA.txt, dataB.txt, and dataC.txt respectively, store the department names and dormitory information, and count how many available rooms for each type are there in the campus.
    - Campus servers only need to read the text files once. When Campus servers are handling queries, they will refer to the data structures, not the text files.
    - For simplicity, there is no overlap of departments among dataA.txt, dataB.txt, and dataC.txt.
2. [Communication]: After step 1, Main server asks each of the Campus servers which departments they are responsible for. Campus servers reply with a list of departments to the main server.
3. [Computation]: Main server will construct a data structure to book-keep such information from step 2. When the queries come, the main server can send a request to the responsible Campus server. Main server then asks the user to input a department name.

Query

1. [Computation]: Main server searches in the database and decides which Campus server(s) should handle the requested department name.
2. [Communication]: Main server sends the department name and the query type of dormitory to the responsible Campus server.

Reply

1. [Computation]: Campus server receives the request, searches in the database, and finds how many available rooms for a certain type in the campus dormitory. The types include single-room, double-room, and triple-room dorms.
2. [Communication]: Campus server prepares a reply message with the results and sends it to the Main server.
3. [Communication]: Main server receives the message from a Campus server and displays the results of available rooms and their building IDs. Main server should keep active for further inputted queries, until the program is manually killed.

Data file

The format of the data file (dataA.txt, dataB.txt, dataC.txt) is defined as follows.

```
<Department Name 1>,<Department Name 2>,<Department Name 3>,...
<Type>,<Building ID>,<Availability>,<Price>
<Type>,<Building ID>,<Availability>,<Price>
<Type>,<Building ID>,<Availability>,<Price>
...
```

Part of the example dataA.txt:

```
ECE,CS,Physics,Accounting,Business
S,101,5,1200
S,202,10,1500
D,101,2,1000
D,321,1,900
T,101,4,800
T,202,0,750
```

Assumptions on the data file:

1. The first line is a list of department names that this Campus server is responsible for. The following lines are the information of dormitories.
2. Department names are letters. The length of a department name can vary from 1 letter to at most 20 letters. It may contain both capital and lowercase letters but does not contain any white spaces.
3. There are three types of dormitory rooms: Single (S), Double (D), and Triple (T).
4. There are multiple buildings denoted in Building IDs. The Building IDs are non-negative integer numbers less than 20 digits.
5. A certain building may contain various types of dormitory rooms.
6. The availability means a personal slot. For instance, availability=1 in a double-room dorm means 1 slot rather than 2.
7. There is no additional empty line(s) at the beginning or the end of the file. The whole data file does not contain any empty lines. A data file is not empty.
8. There is no overlap of department names among all Campus servers.

Example dataA.txt, dataB.txt, and dataC.txt are provided for you as reference. Other data files 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:

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

2. Campus Server: You must name your code file: **server#.c** or **server#.cc** or **server#.cpp** (all small letters except for #). Also, you must name the corresponding header file (if you have one; it is not mandatory) **server#.h** (all small letters, except for #). The "#" character must be replaced by the server identifier (i.e. A,B, C), e.g., **serverA.c.**

   Each Campus server should have its own file!!!

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

DETAILED EXPLANATION

Phase 1 -- Bootup

All server programs (Main server and Campus servers) boot up in this phase. While booting up, the servers must display a bootup message on the terminal. The format of the bootup message for each server is given in the onscreen message tables at the end of the document. As the bootup message indicates, each server must listen to the appropriate port for incoming packets.

Campus servers should boot up first and display a bootup message. A Campus server then needs to read the text file, store the department names and dormitory information, and count the availability for each type of rooms. Taking the above example data file as an example: the availability of single-room dormitories is 15; double-room dormitories is 3, triple-room dormitories is 4. There are many ways to store the dormitory information, such as dictionary, array, vector, etc. You need to decide which data structure to use based on the requirement of the problem. You can use **any** data structure if it can give you correct results.

Main server then boots up after Campus servers are running. Once the Main server programs have booted up, it should display a bootup message as indicated in the onscreen messages table. Note that the Main server code takes no input argument from the command line.

Once all three programs are booting up, Main server will ask Campus servers so that Main server knows which Campus server is responsible for which departments. Campus servers will reply with

a list of departments to Main server. The communication between Main server and Campus servers is using **UDP**. Main server may store the received information using an unordered map:

```
std::unordered_map<std::string, int> department_campus_mapping;
department_campus_mapping["ECE"] = 0;
department_campus_mapping["CS"] = 0;
department_campus_mapping["Art"] = 1;
```

Above lines indicate that "ECE" and "CS" stored in dataA.txt in Campus server A (represented by value 0) and "Art" is stored in dataB.txt in Campus server B (represented by value 1). Again, you could use **any** data structure or format to store the information.

After obtaining the correspondence between Campus servers and departments, **Main server** should display messages to ask the user to enter a query department name and the query type of dormitory (e.g., implement using std::cin). These two inputs are two steps (showing the second line after the input of the first line):

```
Enter department name:
Enter dormitory type (S/D/T):
```

Each of the servers has its unique port number specified in "PORT NUMBER ALLOCATION" section with the source and destination IP address as localhost/127.0.0.1. Main server and Campus 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 if 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 exact format when printing the on-screen messages.

Phase 2 -- Query

Main server gets the inputted department name and searches in the database. If the department name or the dormitory type is not found, the main server will print out a message (see the "On Screen Messages" section) and return to standby. If the department name and dormitory type is found to be in a Campus sever, Main server sends the department name and the query type of dormitory to the corresponding Campus server. For example, for the inputted department name "ECE", the main server should send it only to Campus server A, but not to Campus server B or C.

Phase 3 – Reply

The responsible Campus server receives the query of a certain type of dormitory from the Main server. It then searches from its data structure containing the dormitory information and prepares a message containing the availability of the query type, then sends it to the Main server using UDP. The Campus server prints out the results, for example:

```
Server A found 15 available rooms in S type dormitories. Their
Building IDs are: 101, 202.
Server A has sent the results to Main Server
```

When the Main server receives the result, it displays the result as follows:

```
...

There are 15 available rooms in S type dormitories. Their
Building IDs are: 101, 202.

-----Start a new query-----
Enter department name:
```

See the ON SCREEN MESSAGES table for full example output.

PORT NUMBER ALLOCATION

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

Table 1. Assignments for UDP ports

Process	**Static Ports**
Campus Server A	UDP: 30xxx
Campus Server B	UDP: 31xxx
Campus Server C	UDP: 32xxx
Main Server	UDP(with servers): 33xxx

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 Campus Server (A), etc. Port number of all processes print port number of their own.

ON SCREEN MESSAGES

Table 2. Campus Server A/B/C on-screen messages

Event	**On-screen Messages**
Booting up (Only while starting):	Server A/B/C is up and running using UDP on port <server A/B/C port number>
Sending the department list that contains in "dataA/B/C.txt" to Main Server:	Server A/B/C has sent a department list to Main Server
Upon receiving the input query:	Server A/B/C has received a request for department <Department Name> dormitory type <S/D/T>
Upon finding the department and the student IDs, sending result(s) back to Main Server:	Server A/B/C found totally <num> available rooms for < S/D/T> type dormitory in Building: <Building ID1>, < Building ID2>...
	Server A/B/C has sent the results to Main Server

NOTE: **A/B/C** means A or B or C, print the corresponding index in a terminal.

Table 4. Main Server on-screen messages

Event	**On-screen Messages**
Booting up(only while starting):	Main server is up and running.
Upon receiving the department lists from Campus server A/B/C:	Main server has received the department list from Campus server A/B/C using UDP over port <Main server UDP port number>
List the results of which department Campus server A/B/C is responsible for:	Server A <Department Name 1> , <Department Name 2>… Server B <Department Name 3>,… Server C <Department Name 4>,…
Ask the user to input the department:	Enter department name:
If the input department name cannot be found:	<Department Name> does not show up in Campus servers
Ask the user to input the query:	Enter dormitory type (S/D/T):
If the dormitory type cannot be found:	<Dormitory Type> does not show up in Campus servers
If the input department name can be found, decide which server contains related information about the department name and send a request to Campus server A/B/C	<Department Name> shows up in server <A/B/C >
	The Main Server has sent query for <Department Name> department and <S/D/T> type dormitory to server <A/B/C> using UDP over port <Main server UDP port number>
Main Server receives the searching results from Campus server A/B/C, displays it and starts a new query:	The Main server has received searching result(s) of <S/D/T> type dormitory from Campus server <A/B/C>
	There are totally <num> available rooms for < S/D/T> type dormitory in Building: <Building ID1>, < Building ID2>... -----Start a new query----- Enter department name:

ASSUMPTIONS

1. You must start the processes in this order: **Campus-server (A), Campus-server (B), Campus-server (C), Main-server.**

2. The dataA.txt, dataB.txt, dataC.txt files are created 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 them all in your README file.

4. You can use code snippets from Beej's socket programming tutorial (Beej's guide to network programming) in your project. However, you need to mark the copied part in your code.

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 port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you must 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 developer
```

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

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

6. You may use the following command to examine your port numbers:

```
lsof -i -P -n | grep UDP
```

REQUIREMENTS

1. The host name must be hard coded as **localhost (127.0.0.1)** in all codes.

2. Your main server program should keep running and ask to enter a new request after displaying the previous result, until the TAs manually terminate it by Ctrl+C. The serversrs

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.

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

4. You are not allowed to pass any parameter or value or string or character as a command-line argument.

5. 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 the extra messages before you submit your project.

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

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

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

Do NOT forget the mandatory naming conventions mentioned before!

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

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

Submission Rules

Along with your code files, include a **README file and a Makefile**. **Submissions without README and Makefile will be subject to a serious penalty.**

In the README file write:

- Your **Full Name** as given in the class list.
- Your Student ID.
- Your platform (Ubuntu version).
- 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.)

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
Compile Server A	make serverA
Compile Server B	make serverB
Compile Server C	make serverC
Compile Main Server	make servermain
Run Server A	./serverA
Run Server B	./serverB
Run Server C	./serverC
Run Main Server	./servermain

TAs will first compile all codes using **make all**. We will then open 4 different terminal windows, on which start servers A, B, C and Main Server using commands **./serverA**, **./serverB**, **./serverC** and **./servermain**. **Remember that servers should always be on once started.** TAs will check the outputs for multiple queries. The terminals should display the messages specified in the tables.

1. Compress all your files including the README file into a single "tar ball" and call it: ee450_yourUSCusername.tar.gz (all small letters) e.g. an example filename would be ee450_nanantha.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 such as data files!!! Only include the required source code files, Makefile and the README file. Now run the following commands:

   ```
   tar cvf ee450_yourUSCusername.tar *
   gzip ee450_yourUSCusername.tar
   ```

   Now, you will find a file named "ee450_yourUSCusername.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.tar.gz” to Blackboard -> Assignments. After the file is submitted, you must click on the “submit” button to actually submit it. If you do not click on “submit”, the file will not be submitted.

4. Blackboard 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. Please consider all kinds of possible technical issues and do expect a huge traffic on the Blackboard website very close to the deadline which may render your submission or even access to Blackboard unsuccessful.

6. 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!

7. After submitting, 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.

8. You have sufficient 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.

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 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 10 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 15 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 you add subfolders or compress files in the wrong way, you will lose 2 points each.

9. If your data file path is not the same as the code files, you will lose 5 points.

10. Do not submit datafile (three .txt files) used for test, otherwise, you will lose 10 points.

11. If your code does not correctly assign the TCP port numbers (in any phase), you will lose 10 points each.

12. Detailed points assignments for each functionality will be posted after finishing grading.

13. The minimum grade for an on-time submitted project is 10 out of 100, the submission includes a working Makefile and a README.

14. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend plenty of time on this project and it doesn't even compile, you will receive only 10 out of 100.

15. 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 project description, we will follow the project description.

FINAL WORDS

1. Start on this project early. Hard deadline is strictly enforced. No grace periods. No grace days. No exceptions.

2. In view of what is a recurring complaint near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is *the provided* ***Ubuntu.*** 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 Blackboard for additional requirements and latest updates about the project guidelines. Any project changes announced on Blackboard 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.