EE450 Socket Programming Project

Part 1

Fall 2021

Due Date:

Sunday, October 17, 2021 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 TAs any question about the content of the project, but TAs have the right to reject your request for debugging.**

PROBLEM STATEMENT

In this part of the project, you will implement client-server socket programming using TCP. Clients would like to ask Main server which state a city is located in. A client sends a city name to Main server and Main server will search in its database and reply to the client with a state name.

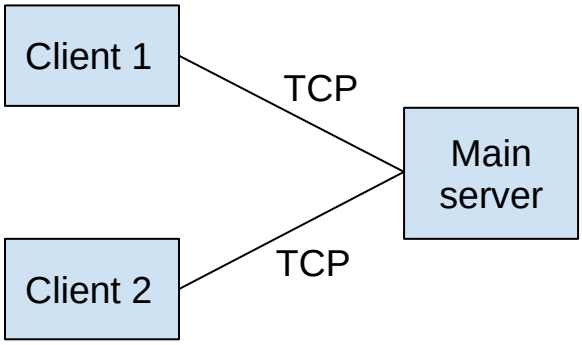

Figure 1

The detailed operations to be performed by all the parties are described with the help of Figure 1. There are in total 3 communication endpoints, which are run in 3 individual terminal windows:

- Client 1 and Client 2: represent two different users, send queries to main server
- Main server: store information, search, send responses to clients

You are highly encouraged to use Beej's Guide to Network Programming to complete this assignment. You can use code from Beej's Guide as a starting point (remember to mention any code you take directly from other sources like Beej's Guide in the README and your comments).

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

Bootup

1. [Computation]: Main server read the file list.txt and store the information.
2. [Communication]: Main server process wait for client processes to connect

3. [Computation]: Two clients are run and ask the user to input a city name.

Query

1. [Communication]: Each client then establishes a TCP connection to the Main server and sends its queries (the city name) to the Main server.
    - A client can terminate itself only after it receives a reply from the server (in the Reply phase).
    - Main server may be connected to both clients at the same time.
2. [Computation]: Once the Main server receives the queries, it decodes the queries and searches in the list with the received country name, obtaining the corresponding backend server ID.

Reply

1. [Communication]: Main server prepares a reply message and sends the result to the appropriate client.
2. [Communication]: Clients receive the reply message from Main server and display it. Clients should keep active for further inputted queries, until the program is manually killed (Ctrl-C).

The format of list.txt is as follows.

```
<State Name>
<City Name>,<City Name>,<City Name>
<State Name>
<City Name>,<City Name>,<City Name>
<State Name>
<City Name>,<City Name>,<City Name>
…
```

Example list.txt:

```
California
Los Angeles,San Diego
Illinois
Chicago,Peoria,Springfield
Texas
Austin,Dallas,Houston
…
```

Assumptions on the list.txt file:

1. City and state names are letters. The length of a city name can vary from 1 letter to at most 20 letters. States and cities may contain both capital and lowercase letters and can contain white spaces.
2. Cities associated to the same state will be on the same line and delimited by commas.
3. There is no additional empty line(s) at the beginning or the end of the file. That is, the whole list.txt do not contain any empty lines.
4. For simplicity, there is no overlap of city names among different states.
5. For a given state, there may be repeated city names.
6. list.txt will not be empty.
7. A state server will have at least one city name, and at most 100 city names.

An example list.txt is provided for you as a reference and for testing. A different list.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:

1. servermain: 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. 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). **There should be only one client file!!!**

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

DETAILED EXPLANATION

Phase 1 -- Bootup

Main server program first boots 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 Main server is given in the on-screen message table at the end of the document. As the boot up message indicates, Main server must listen on the appropriate port for incoming packets/connections.

As described in the previous section, the main server needs to read the text file and store the information. There are many ways to store the information, such as dictionary, array, vector, etc. You need to decide which format to use based on the requirement of the problem. You can use **any** format if it can give you correct results.

Once the server programs have booted up, two client programs run. Each client displays a boot up message as indicated in the onscreen messages table. Note that the client code takes no input argument from the command line. The format for running the client code is:

```
./client
```

After running it, it should display messages to ask the user to enter a query country name (e.g., implement using std::cin):

```
./client
Client is up and running.
Enter City Name:
```

For example, if the client 1 is booted up and asks for a state for city Dallas, then the terminal displays like this after user types “Dallas” as the input:

```
./client
Client is up and running.
Enter City Name: Dallas
```

The main server has its unique port number specified in “PORT NUMBER ALLOCATION” section with the source and destination IP address as localhost/127.0.0.1. Clients use dynamic ports.

Clients and Main server 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

After booting up, Clients establish TCP connections with Main server. After successfully establishing the connection, Clients send the input city name to Main server. Once this is sent, Clients should print a message in a specific format. Repeat the same steps for Client 2.

Main server then receives requests from two Clients. If the city name is not found, the Main server will print out a message (see the "On Screen Messages" section) and return to standby.

For a server to receive requests from several clients at the same time, the function ***fork()*** should be used for the creation of a new process. *fork()* function is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the *fork()* call (***parent process***).

For a TCP server, when an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using *accept()*. After the connection with the client is successfully established, the *accept()* function returns a non-zero descriptor for a socket called the *child socket*. The server can then fork off a process using *fork()* function to handle connection on the new socket and go back to wait on the original socket. Note that the socket that was originally created, that is the parent socket, is going to be used only to listen to the client requests, and it is not going to be used for computation or communication between client and Main server. Child sockets that are created for a parent socket have the identical well-known port number and IP address at the server side, but each child socket is created for a specific client. Through using the child socket with the help of *fork()*, the server can handle the two clients without closing any one of the connections.

Once the Main server receives the queries, it decodes the queries and searches in the list with the received city name, finding the corresponding state ID.

Phase 3 -- Reply

At the end of Phase 2, the Main server should have the result ready. The result is the state name that the city is associated with. The result should be sent back to the corresponding client using

TCP. The client will print out the city name and then print out the messages for a new request as follows:

```
...

City Dallas is located in state Texas.

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

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

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**
Main Server		TCP(with client): 33xxx
Client 1	TCP	
Client 2	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: **33319** for the Main Server, etc. Port number of all processes print port number of their own.

ON SCREEN MESSAGES

Table 2. Main Server on-screen messages

Event	**On-screen Messages**
Booting up (only while starting):	Main server is up and running.
Upon reading the state lists:	Main server has read the state list from list.txt.
Print the results of which city is located in which country: **(Repeated cities should be printed only once)**	<State Name 1>: <City Name 1> <City Name 2> <State Name 2>: <City Name 3> …
Upon receiving the input from the client:	Main server has received the request on city <City Name> from client <client ID> using TCP over port <Dynamic TCP port number for Client>
If the input city name could not be found, send the error message to the client:	<City Name> does not show up in states < State Name 1, State Name 2, …> **(Print all state names)**
	The Main Server has sent "<City Name>: Not found" to client<client ID> using TCP over port < Dynamic TCP port number for Client >
If the input city name could be found, decide which state contains the input city name and send appropriate message to client:	<City Name> is associated with state < State Name>
	Main Server has sent searching result to client <client ID> using TCP over port <Main Server TCP port number>

Table 3. Client 1 or Client 2 on-screen messages

Event	**On-screen Messages**
Booting up(only while starting)	Client is up and running.
	Enter City Name:
After sending city name to Main Server:	Client has sent city <City Name> to Main Server using TCP.
If input city cannot be found:	<City Name> not found.
If input city can be found:	Client has received results from Main Server: <City Name> is associated with state <State Name>.
After the last query ends:	-----Start a new query----- Enter City Name:

ASSUMPTIONS

1. You must start the processes in this order: **Main-server, Client 1, and Client 2.**

2. list.txt is 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 guide to network programming in your project. However, you need to mark the copied part in your code and mention in README.

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 TCP 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>
```

REQUIREMENTS

1. **Do not hardcode the TCP port numbers that are to be obtained dynamically**. Refer to Table 1 to see which ports are statically defined and which ones are dynamically assigned. You can 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. Your client 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 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.

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 the extra messages before you submit your project.

7. Using fork() to create a child process when a new TCP connection is accepted is mandatory and everyone should support it. This is useful when different clients are trying to connect to the same server simultaneously.

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.

2. All submissions will only be graded on the provided Ubuntu. TA 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 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
- 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 Main Server	make servermain
Compile Client	make client
Run Main Server	./servermain
Run client 1	./client
Run client 2	./client

TA will first compile all codes using *make all*. TA will then open 3 different terminal windows. On one terminal, start Main Server using commands *./servermain*. **Remember that main server should always be on once started.** On the other two terminals, start the client as *./client*. TA will check the outputs for multiple queries. The terminals should display the messages specified above.

Please follow these rules to submit your assignment:

1. Compress all your files including the README file into a single "tar ball" and call it: ee450_yourUSCusername.tar.gz (all small letters). 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.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, for example data file list.txt. 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. Please DO NOT wait till the last 5 minutes to upload and submit because some technical issues might happen.

6. 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.

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 Main Server. But the result is not received by the Main server (no on-screen message) because Main server 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 TCP sockets.

2. Inline comments in your code. This is important as this will make your code more readable and help in understanding what you have done. Good comments will improve your chances of receiving partial credit if your code is not functioning correctly.

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 at most 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 at most 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 if 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 (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 Blackboard (Discussion Board & Announcement) regularly 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.