EE450 Socket Programming Project

Part2

Fall 2021

Due Date:

Sunday, November 7, 2021 11:59PM

(Hard Deadline, Strictly Enforced)

OBJECTIVE

The objective of project is to familiarize you with UNIX socket programming. This assignment is worth **4%** of your overall grade in this course. **It is an individual assignment and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).** If you have any doubts/questions, post your questions on Blackboard -> Discussion Board, email TA your questions or come by TA's office hours. **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 socket programming between servers using UDP. There are two backend servers, A and B, which communicate with the main server through UDP. Because the storage space and computation resources of the main server are limited, the required information is stored in two backend servers. Each backend server stores a data file, dataA.txt for server A and dataB.txt for server B, which contains state and city names. Main server first asks backend servers which states they are responsible for and retrieve a list of cities from each backend server. Main server will send a request only to its responsible backend server which contains the requested state. The backend server then replies to the main server with the requested information.

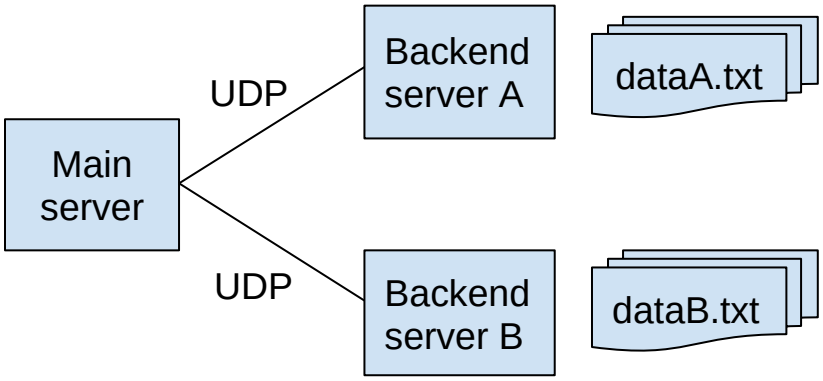

Figure 1

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:

- Main server: decide which backend server should a request be sent.
- Backend server 1 and Backend server 2: stores data and compute the result

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

Bootup

1. [Computation]: Backend server A and B read dataA.txt and dataB.txt respectively, store the state names and their city names, and count how many **distinct** cities there are in a state.
    - Backend servers only need to read the text files once. When Backend servers are handling queries, they will refer to the data structures, not the text files.
    - For simplicity, there is no overlap of states between dataA.txt and dataB.txt.
2. [Communication]: after step 1, Main server asks each of Backend servers which states they are responsible for. Backend servers reply with a list of states 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 Backend server. Main server then asks the user to input a state name.

Query

1. [Computation]: Main server searches in the database and decides which backend server(s) should handle the requested state name.
2. [Communication]: Main server sends the state name to the responsible Backend server.

Reply

1. [Computation]: Backend server receives the state name, searches in the database, and finds how many distinct cities belong to the state and what those cities are.
2. [Communication]: Backend server prepares a reply message with the results and sends it to the Main server.
3. [Communication]: Main server receive the message from Backend server and display the results. Main server should keep active for further inputted queries, until the program is manually killed.

The format of dataA.txt or dataB.txt is defined as follows.

```
<State Name 1>
<city name 1>,< city name 2>, … ,<city name k>
<State Name 2>
<city name 1>,<city name 2>,…,<city name k>
...
```

Let's consider an example:

Example dataA.txt:

```
Ohio
Cleveland,Columbus,Dayton,Toledo
Indiana
Indianapolis,Bloomington,Evansville
```

Example dataB.txt:

```
California
Los Angeles,Sacramento
```

Assumptions on the data file:

1. State names are letters. The length of a state name can vary from 1 letter to at most 20 letters. It may contain both capital and lowercase letters and CAN contain white spaces. State names "Abc" and "abc" are different.
2. City names can contain letters, white spaces, and numbers.
3. There are at most 20 state names in total.
4. 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 files is not empty.
5. There is no overlap of state names among two backend servers.
6. For a given state name, there may be repeated city names.
7. A state name is associated with at least one city name, and at most 100 city name.

A example dataA.txt and dataB.txt is provided for you as a reference. Other dataA.txt and dataB.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. 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. Backend 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 or B), e.g., **serverA.c.**

   Each backend 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 (10 points) -- Bootup

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

Backend servers should boot up first and display a boot up message. A backend server then needs to read the text file, store the state and city names, and count how many **distinct** cities there are in a state. For example, the number of distinct cities in state "Indiana" is 3. There are many ways to store the 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 Backend servers are running. Once the Main server programs have booted up, it should display a boot up 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 Backend servers so that Main server knows which Backend server is responsible for which states. Backend servers will reply with a list of states to Main server. The communication between Main server and Backend servers is using UDP. Main server may store the received information using an unordered_map:

```
std::unordered_map<std::string, int> state_backend_mapping;
state_backend_mapping["Indiana"] = 0;
state_backend_mapping["California"] = 1;
state_backend_mapping["Ohio"] = 0;
```

Above lines indicate that “Indiana” and “Ohio” stored in dataA.txt in Backend server A (represented by value 0) and “California” is stored in dataB.txt in Backend server B (represented by value 1). Again, you could use **any** data structure or format to store the information.

After obtaining the correspondence between Backend servers and states, main server should display messages to ask the user to enter a query state name (e.g., implement using std::cin):

```
Enter state name:
```

Each of three 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, Backend server A and Backend server B 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 (40 points) -- Query

Main server gets the inputted state name and searches in the database. If the state name is not found, the main server will print out a message (see the “On Screen Messages” section) and return to standby. If the state name is found to be in a Backend sever, Main server sends the state name to the corresponding Backend server. For example, for the inputted state name “California”, the main server should send it only to Backend server B, but not to Backend server A.

Phase 3 (10 points) -- Reply

The responsible Backend server receives the query from the Main server. It searches the state name in the database and finds the cities that are associated with the state name. It then prepares a message containing the state names and sends it to the Main server using UDP.

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

```
...

There are 3 distinct cities in Indiana. Their names are
Indianapolis, Bloomington, Evansville.


-----Start a new query-----
Enter state 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**
Backend Server A	UDP: 30xxx
Backend Server B	UDP: 31xxx
Main Server	UDP(with servers): 32xxx

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

ON SCREEN MESSAGES

Table 2. Backend Server A on-screen messages

Event	On-screen Messages
Booting up (Only while starting):	Server A is up and running using UDP on port <server A port number>
Sending the state list that contains in “dataA.txt” to Main Server:	Server A has sent a state list to Main Server
Upon receiving the input query:	Server A has received a request for <State Name>
Upon finding the state and the city names, sending result(s) back to Main Server:	Server A found <num> distinct cities for <State Name>: <city name1>, <city name2>...
	Server A has sent the results to Main Server

Table 3. Backend Server B on-screen messages

Event	On-screen Messages
Booting up (Only while starting):	Server B is up and running using UDP on port <server B port number>
Sending the state list that contains in “dataB.txt” to Main Server:	Server B has sent a state list to Main Server
Upon receiving the input query:	Server B has received a request for <State Name>
Upon finding the state and the city names, sending result(s) back to Main Server:	Server B found <num> distinct cities for <State Name>: <city name1>, <city name2>...
	Server B has sent the results to Main Server

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 state lists from server A:	Main server has received the state list from server A using UDP over port <Main server UDP port number>
Upon receiving the state lists from server B:	Main server has received the state list from server B using UDP over port <Main server UDP port number>
List the results of which states serverA/B is responsible for:	Server A <State Name 1> <State Name 2> Server B <State Name 3>
Ask the user to input the query:	Enter state name:
If the input state name cannot be found:	<State Name> does not show up in server A&B
If the input state name can be found, decide which server contains related information about the state name and send a request to serverA/B	<State Name> shows up in server <A or B>
	The Main Server has sent request for <State Name> to server <A or B> using UDP over port <Main server UDP port number>
Main Server receives the searching results from serverA/B, displays it and starts a new query:	The Main server has received searching result(s) of <State Name> from server<A or B>
	There are <num> distinct cities in <State Name>: <city name1>, <city name2>... -----Start a new query----- Enter state name:

ASSUMPTIONS

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

2. The dataA.txt and dataB.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 backend serversr 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 Platform and Environment

1. All your submitted code **MUST** work well on the provided virtual machine Ubuntu.

2. All submissions will only be graded on the provided Ubuntu. TAs won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code 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

may 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
- Briefly summarize what you have done in the assignment. (Please do not repeat the project description).
- List all your code files and briefly summarize their fulfilled functionalities. (Please do not repeat the project description).
- The format of all the messages exchanged between servers.
- 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 Main Server	make servermain
Run Server A	./serverA
Run Server B	./serverB
Run Main Server	./servermain

TAs will first compile all codes using **make all**. We will then open 3 different terminal windows, on which start servers A, B and Main Server using commands **./serverA**, **./serverB**, 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!

1. 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.

2. 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.

3. 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.

4. 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!

5. 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.

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 (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.

4. Plagiarism will not be tolerated and will result in an "F" in the course.