EE450 Socket Programming Project, Spring 2017
Due Date : Friday Apr 7th, 2017 11:59 PM (Midnight)

(The deadline is the same for all on-campus and DEN off-campus students)

Hard Deadline (Strictly enforced)

1. The objective of this assignment is to familiarize you with UNIX socket programming.
2. This assignment is worth **10%** of your overall grade in this course.
3. **It is an individual assignment and no collaborations are allowed.**
4. **Please do not look at your friend's screen while writing your code. Write your code independently.**
5. **Any cheating will result in an automatic F for the course (not just in the project).**
6. If you have any doubts/questions, post your questions on Piazza (as anonymous if you feel uncomfortable).
7. **You must discuss all project related issues on Piazza**.
8. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points (you need to post as yourself, not anonymous!).
9. **Project can and will only be graded on the VM posted on Piazza (Ubuntu 16.04). https://piazza.com/class/ixqnp4xlv715yy?cid=8**
10. **Submissions WITHOUT README OR Makefiles WILL NOT BE GRADED**.
11. All submissions will be checked for plagiarism using MOSS.
12. **In addition to Beej PDF on D2L, feel free to watch these two videos for the project:**
    a. **https://www.youtube.com/watch?v=eVYsIolL2gE**
    b. **https://www.youtube.com/watch?v=Emuw71lozdA**

Problem Statement:

In this project you will implement a simple model of computational offloading where a single client offloads some computation to a server (edge server) which in turn distributes the load over 2 backend servers. The server facing the client then collects the results from the backend and communicates the same to the client in the required format. As a result, the system is comprised of three different parts:

1. Client: submits jobs to Google Compute Engine for processing.
2. Edge Server: Communicates with the client, receives the job, dispatches the job to back end servers, receives their responses, builds the final output and sends the result back to client.
3. Backend Servers: They perform specific computations they are assigned to. In our case, one server performs **bitwise** "and" and the other performs bitwise "or" operations.

The servers together constitute "Google Compute Engine" (edge and backend servers, in total 3 servers). The client and the edge server communicate over a TCP connection while the communication between the edge server and the Back-Servers is connectionless and over UDP. This setup is illustrated in Figure 1.

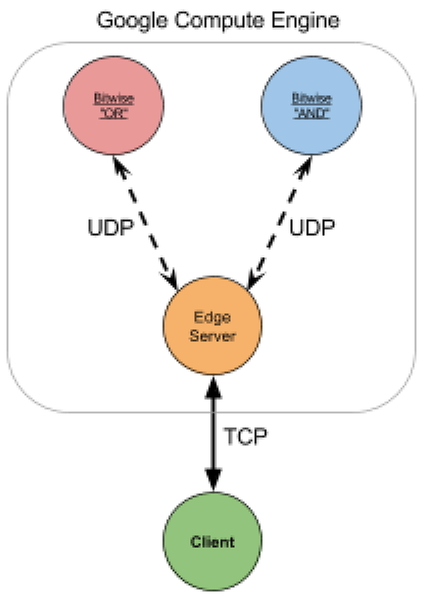

Figure 1. Problem Setup for Socket Programming Project

Input Files Used:

The files specified below will be used as **inputs** in your programs in order to **dynamically** configure the state of the system.

The contents of the files must **NOT** be "**hardcoded**" in your source code, because during grading, the input files will be different, but the formats of the files will remain the same.

If you are working in an environment other than UNIX, **pay particular attention to line endings or newlines**. For this project, it is assumed that all files follow the **UNIX line ending convention**. This is particularly important while handling the input file(s). See the articles here and here for more information.

Input file is an ASCII file that contains three columns of strings all in lowercase. Each row specifies three elements: an operation and two operands. The first column is the operation in string format which is either "and" or "or". The other columns specify two **strings of zeros and ones** representing two binary numbers. This file includes at most 100 rows and each binary number is at most 10 digits ("0"s or "1"s). This file will always reside in the same directory as the client. Each row shall be processed independently.

Source Files:

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

1. Edge Server: You must name your code file: **edge.c** or **edge.cc** or **edge.cpp** (all small letters). Also you must call the corresponding header file (if you have one; it is not mandatory) **edge.h** (all small letters).
2. Back-Server "AND" and "OR": You must use one of these names for this piece of code: **server_#.c** or **server_#.cc** or **server_#.cpp**. Also you must call the corresponding header file (if you have one; it is not mandatory) **server_#.h**. The "#" character must be replaced by the server identifier (i.e. 'or' or 'and'), depending on the server it corresponds to (such as server_or.cpp and server_and.cpp).
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).

Makefile:

Makefile should support following functions. TAs will first compile all codes using make all. They will then open 4 different terminal windows. On 3 terminals they will start servers AND, OR, and Edge using commands make server_and, make server_or, and make edge. On the fourth terminal they will start the client such as ./client job.txt. The terminals should display the messages shown in table 4, 5, 6, 7.

make all	**Compiles** all your files and creates executables
make server_or	**Runs** server_or server
make server_and	**Runs** server_and server
make edge	**Runs** edge server
./client `<input_file_name>`	Starts the client with job file

Phase1: (25 points)

All three server programs (Edge, Backend-Server 'or' and Backend-Server 'and') boot up in this phase. While booting up, the servers **must** display a boot message on the terminal. The format of the boot message for each server is given in the onscreen messages tables 4, 5, 6, 7. As the boot message indicates (look at the table of messages down below), each server must listen on the appropriate port for incoming packets/connections.

Once the server programs have booted up, the client program is run. The client displays a boot message as indicated in the onscreen messages table. Note that the client code takes an input argument from the command line, that specifies the computation that is to be run. The format for running the client code is

```
./client <input_file_name>
```

where `<input_file_name>` is the name of the input file that contains operations and operands as explained above. For instance, to open a job.txt file right besides the client binary use:

```
./client job.txt
```

After booting up, the client establishes a TCP connection with the edge server. After successfully establishing the connection, the client reads all lines from the input file and proceeds to send them to edge server over our TCP connection. After successfully sending the lines, the client should print the number of lines sent to edge server. This ends Phase 1 and we now proceed to Phase 2.

Example of job.txt:

and,10111,101
or,10,1011
and,0,101

Every line ends with a '\n', the last line might or might not end with '\n'

Phase 2: (40 points)

After receiving data from the client, the edge server uses UDP to communicate with two backend servers and relays the data to corresponding backend server. For example, “and,101,1001” should **only** be relayed to the backend server **‘and’**, and the backend server **‘or’** should not receive this line. The operation type are randomly distributed in the file, so you need to make a decision about the destination of the data. The data sent to the backend server is not required to be the same as the original data. You can add/delete some information if you need (like a key to sort the lines later, such as line #). But if you don’t need to do that, it’s totally fine. You're free to implement your code in any way as long as it works and conforms to the rules.

Once the backend server receives a line of data, it should complete the operation on the binary numbers. We will only have two types of operations: and / or. They are all basic binary operations. For example, 0 and 1 = 0, 0 or 1 =1. The binary number should be calculated bit by bit and the result of bit operation is shown in the following Table. The length of the binary numbers (operands) is not guaranteed to be the same. They may have different lengths. You need to **pad 0 before** the shorter one to make it as long as the longer one for the operations (or you can do it in other ways as you wish). For example, **1010 and 110** should be calculated as **1010 and 0110 = 0010**.

Backend server ‘and’ only makes ‘and’ operations while backend server ‘or’ only deals with ‘or’ operations. These two backend servers are totally independent and there is no communication between them.

After calculation, the backend server need to print an equation of the result to the screen. Please don’t print those padded 0s. The leftmost bit of every binary number must be 1, **except when it is all 0s**. Please see Phase 3 for subsequent actions and example outputs.

Table 2. Results for bit operations

Result for bit operation ‘and’		
and	0	1
0	0	0
1	0	1

Result for bit operation ‘or’		
or	0	1
0	0	1
1	1	1

<u>Phase 3:</u> (25 points)

At the end of Phase 2, two backend-servers should have their computation results ready. Those results should be sent to the edge server using UDP. When the edge server receives the computation result, it needs to print the result on the screen and forward the result to the client using TCP. **The order of lines printed on the edge server's screen can be any regardless of the order of input file.**

When the client receives the results, it prints results on the screen line by line. **The order of lines printed on client's screen should stick to the order of input file.**

For example, if the input file job.txt is:

and,10111,101
or,10,1011
and,0,101

then the output order of final result on client's screen should be:

101
1011
0

Once all lines are printed on client's screen in order, we are done!

Ports and Message Formats:

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

Table 3. Static and Dynamic assignments for TCP and UDP ports.

Process	**Dynamic Ports**	**Static Ports**
Backend-Server OR	-	1 UDP, 21000+xxx (last three digits of your USC ID)
Backend-Server AND	-	1 UDP, 22000+xxx (last three digits of your USC ID)
Edge-Server	-	1 TCP, 23000+xxx (last three digits of your USC ID) 1 UDP, 24000+xxx (last three digits of your USC ID)
Client	1 TCP	-

NOTE: 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.**

ON SCREEN MESSAGES:

Table 4. Backend-Server "OR" on screen messages

Event	**On Screen Message (inside quotes)**
Booting Up:	"The Server OR is up and running using UDP on port <port number>." "The Server OR start receiving lines from the edge server for OR computation. The computation results are:"
Upon receiving every line for "OR computation":	"<binary string 1> or <binary string 2> = <"OR" computation result>"
…	**…**
After receiving all lines and finishing all "OR computation":	"The Server OR has successfully received <count> lines from the edge server and finished all OR computations.
After sending all computation results to the edge server:	"The Server OR has successfully finished sending all computation results to the edge server."

ON SCREEN MESSAGES: **Table 5. Backend-Server “AND” on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The Server AND is up and running using UDP on port <port number>.” “The Server AND start receiving lines from the edge server for AND computation. The computation results are:”
Upon receiving one line for “AND computation”:	“<binary string 1> and <binary string 2> = <”AND” computation result>”
…	…
After receiving all lines and finishing all “AND computation”:	“The Server AND has successfully received <count> lines from the edge server and finished all AND computations
After sending all computation results to the edge server:	“The Server AND has successfully finished sending all computation results to the edge server.”

ON SCREEN MESSAGES: **Table 6. Edge Server on screen messages**	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The edge server is up and running.”
After receiving all lines from the client:	“The edge server has received <count> lines from the client using TCP over port <port number>.”
After sending all “OR computation” lines to the Backend-Server OR	“The edge has successfully sent <count> lines to Backend-Server OR.”
After sending all “AND computation” lines to the Backend-Server AND	“The edge has successfully sent <count> lines to Backend-Server AND.”
Start receiving the computation results from “Backend-Server OR” and “Backend-Server AND” using UDP	“The edge server start receiving the computation results from Backend-Server OR and Backend-Server AND using UDP over port <port number>.” “The computation results are:”

Upon receiving one computation result from “Backend-Server OR” or “Backend-Server AND”	“<binary string 1> or <binary string 2> = <OR computation result>” or (depending on operation print the above or below line) “<binary string 1> and <binary string 2> = <AND computation result>”
…	…
After receiving all computation results from “Backend-Server OR” and “Backend-Server AND”	“The edge server has successfully finished receiving all computation results from Backend-Server OR and Backend-Server And.”
After sending all computation results to the client	“The edge server has successfully finished sending all computation results to the client.”

ON SCREEN MESSAGES:
Table 7. Client on screen messages

Event	**On Screen Message (inside quotes)**
Booting Up:	“The client is up and running.”
After sending all lines to the edge server:	"The client has successfully finished sending <count> lines to the edge server.”
After receiving all computation results from the edge server	“The client has successfully finished receiving all computation results from the edge server.”
After sorting the computation result	“The final computation result are: <computation result> ...

Example job.txt:

```
and,10111,101
or,10,1011
or,11,10001
and,1001,111
```

Example Output:

Backend-Server OR Terminal:

```
The Server OR is up and running using UDP on port 21319.
The Server OR start receiving lines from the edge server for OR computation. The computation results are:
10 or 1011 = 1011
11 or 10001 = 10011
The Server OR has successfully received 2 lines from the edge server and finished all OR computations.
The Server OR has successfully finished sending all computation results to the edge server
```

Backend-Server AND Terminal:

```
The Server AND is up and running using UDP on port 22319.
The Server AND start receiving lines from the edge server for AND computation. The computation results are:
10111 and 101 = 101
1001 and 111 = 1
The Server AND has successfully received 2 lines from the edge server and finished all AND computations.
The Server AND has successfully finished sending all computation results to the edge server
```

Edge server Terminal:

```
The edge server is up and running.
The edge server has received 4 lines from the client using TCP over port 23319
The edge server has successfully sent 2 lines to Backend-Server OR.
The edge server has successfully sent 2 lines to Backend-Server AND.
The edge server start receiving the computation results from Backend-Server OR and Backend-Server AND using UDP port 24319.
The computation results are:
10 or 1011 = 1011
11 or 10001 = 10011
10111 and 101 = 101
1001 and 111 = 1
The edge server has successfully finished receiving all computation results from the Backend-Server OR and Backend-Server AND.
The edge server has successfully finished sending all computation results to the client.
```

Client Terminal:

```
The client is up and running.
The client has successfully finished sending 4 lines to the edge server.
The client has successfully finished receiving all computation results from the edge
server.
The final computation results are:
101
1011
10011
1
```

Assumptions:

1. It is recommended to start the processes in this order: **backend-server (AND), backend-server (OR), Edge server, Client.**

2. 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.

3. 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 mark the copied part in your code.

4. 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** (from past logins or previous runs of code that are still not terminated and hold the port busy). 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. If you have zombie processes you can kill them using unix commands: **kill -9 <pid>** or **killall <proc name>**

Requirements:

1. Do not hardcode the TCP or UDP 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.

2. The host name must be hardcoded as localhost (127.0.0.1) in all pieces of code.

3. The user will terminate backend servers and edge server at the end by pressing ctrl-C. Do not terminate any process on its own.

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 while running the client in Phase 1 to read the input file.

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.

Programming platform and environment:

1. All your codes **must** run (work) on the provided Ubuntu VM: https://piazza.com/class/ixqnp4xlv715yy?cid=8
2. You can write your code anywhere as long as you test it on the VM. **Project can and will only be graded on the VM posted on Piazza (Ubuntu 16.04).**
3. No MS-Windows programs will be accepted.
4. You MUST have a Makefile.
5. Your code MUST compile using gcc/g++ compiler and Make command.

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 (or use sublime) and then use compilers such as g++ (for C++) and gcc (for C) that are already installed on Unix 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 (10 points):

1. Along with your code files, include a **README file & a Makefile**. In the README file write
   a. Your **Full Name** as given in the class list
   b. Your Student ID
   c. What you have done in the assignment
   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. What the TA should do to run your programs. (Any specific order of events should be mentioned.)
   f. The format of all the messages exchanged.
   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.)

Submissions WITHOUT README OR Makefiles WILL NOT BE GRADED.

2. Compress all your files including the README file into a single "tar ball" and call it: **ee450_yourUSCusername_session#.tar.gz** (all small letters) e.g. my filename would be **ee450_xiaohanw_session1.tar.gz**. Please make sure that your name matches the one in the class list.

3. Upload "ee450_yourUSCusername_session#.tar.gz" to the Digital Dropbox (available under Tools) on D2L. 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.

4. D2L will keep a history of all your submissions if you submit multiple times. We will grade your latest submission.

5. You will receive a confirmation email from D2L to inform you whether your project is received successfully, so please do check your emails well before the deadline to make sure your attempt at submission is successful.

6. By the announced deadline all Students must have already successfully submitted their projects and received a confirmation email.

7. 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.

8. Please DO NOT wait till the last 5 minutes to upload and submit your project because if some technical issue occurs you may miss the deadline.

9. **After submission on D2L, confirm your own submission by downloading whatever you submitted and compiling it on your machine. If the outcome is not what you expected, resubmit. Make sure that the file you uploaded is not corrupt by verifying its checksum. You can do that using "`gunzip -t`" command on your local machine when you download the file you uploaded to D2L.**

10. **You have plenty of time to work on this project and submit it in time hence there is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a zero for the project.**

Grading Criteria:

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. If your submitted codes, do not even compile, you will receive 5 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 10 out of 100.

7. If your codes compile but when executed only perform phase1 correctly, you will receive 35 out of 100.

8. If your code compiles and performs all tasks up to the end of 2 phases correctly and error-free, and your README file conforms to the requirements mentioned before, you will receive 75 out of 100.

9. If your code compiles and performs all tasks of all 3 phases correctly and error-free, and your README file and Makefile conforms to the requirements mentioned before, you will receive 100 out of 100.

10. 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.)

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

12. You will lose 5 points for each error or a task that is not done correctly.

13. The minimum grade for an on-time submitted project is 10 out of 100, assuming there are no compilation errors and the submission includes a working Makefile and a README.

14. There are **no points** for the effort or the time you spend working on the project or reading the tutorial. If you spend more than 1 month on this project and it doesn't even compile, you will receive only 5 out of 100.

15. **You must discuss all project related issues on Piazza**. We will give those who actively help others out by answering questions on Piazza up to 10 bonus points. (If you want to earn the extra credits, do remember to leave your names visible to instructors when answering questions on Piazza.)

16. The maximum points that you can receive for the project with the bonus points is 100. In other words the bonus points will only improve your grade if your grade is less than 100.

17. **Your code will not be altered in any ways for grading purposes and however it will be tested with different input files. 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.**

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 the Ubuntu VM posted on Piazza since Jan 17th (16.04). It is strongly recommended that students develop their code on this version of virtual machine. In case students wish to develop their programs on other platforms, 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 Ubuntu VM 16.04.

Academic Integrity:

All students are expected to write all their code on their own.

Copying code from friends is called **plagiarism** not **collaboration** and will result in an F for the entire course. Any libraries or pieces of code that you use and you did not write must be listed in your README file. 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.