CSE 489/589
Programming Assignment 1
Text Chat Application

1. Objectives

Develop the client and server components of a text chat application, consisting of one chat server and multiple chat clients over TCP connections.

2. Getting Started

2.1 Socket Programming
Beej Socket Guide: http://beej.us/guide/bgnet
2.2 Install the PA1 template
Read the document at https://goo.gl/L2kgb5 in full and install the template.
You should complete this step before reading further.
It is mandatory to use this template.

3. Implementation

3.1 Programming environment
You will write C (or C++) code that compiles under the GCC (GNU Compiler Collection) environment. Furthermore, you should ensure that your code compiles and operates correctly on 5 dedicated servers. Read the document at https://goo.gl/L2kgb5 for details on the 5 servers. Your code should successfully compile using the version of gcc (for C code) or g++ (for C++ code) found on the 5 dedicated servers and should function correctly when executed.

NOTE: You are NOT allowed to use any external libraries for the socket programming part. You can however use external modules for other parts of the assignment (like maintaining a linked list). If you are not sure whether you are allowed to use an external library or not, consult with the course staff. Further, your implementation should NOT invoke any external binaries (e.g., ifconfig, nslookup, etc.).

3.2 Sockets
- Use **TCP Sockets** only for your implementation.
- Use the **select() system call** only for handling multiple socket connections. Do not use multi-threading or fork-exec.

3.3 Running your program
Your program will take 2 command line parameters:
1. The first parameter (s/c) indicates whether your program instance should run as a server or a client.
2. The second parameter (number) is the port number on which your process will listen for incoming connections. In the rest of the document, this port is referred to as the **listening port**.

E.g., if your executable is named chat_app:
- To run as a server listening on port 4322

./chat_app s 4322
- To run as a client listening on port 4322

./chat_app c 4322

3.4 Dedicated Servers
For the purpose of this assignment, you should only use (for development and/or testing) the directory the directory created for you on each of the 5 dedicated servers. Change the access permission to this directory so that only you are allowed to access the contents of the directory. This is to prevent others from getting access to your code.

4. Output Format

We will use automated tests to grade this assignment. The grader, among other things, will also look at the output generated by your program. Towards this end, **ALL the *required* output** (as described in section 5) generated by your program needs to be written to **BOTH stdout and to a specific logfile**. Later sections provide the exact format strings to be used for output, **which need to be strictly followed.**

4.1 Print and LOG

We have already provided a convenience function for this purpose in the template (see src/logger.c and include/logger.h), which writes both to stdout and to the logfile. **You should use ONLY this function, for all output described in this assignment**. On the other hand, if you want to output something more to stdout (for debugging etc.) than what is described, **do NOT** use this function and rather use native C/C++ function calls. **Any extra output in the log file will cause the test cases to fail.**

To use the function, you will need to have the following statement at the top of your .c/.cpp source file(s) where you want to use this function:

#include "../include/logger.h"

The function is designed to behave almost exactly as printf. You can use the function as:

cse4589_print_and_log(char* format, ...)

Read the comments above the function definition contained in the src/logger.c file, for more information on the arguments and return value.

5. Detailed Description

The chat application follows a typical client-server model, whereby we will have one server instance and two or more client instances. Given that we will be testing on the five dedicated servers listed before, you can assume that at most four clients will be online at any given time.
The clients, when launched, log in to the server, identify themselves, and obtain the list of other clients that are connected to the server. Clients can either send a unicast message to any one of the other clients or a broadcast a message to all the other clients.
Note that the clients maintain an active connection only with the server and not with any other clients. Consequently, **all messages exchanged between the clients must flow through the server**. Clients never exchange messages directly with each other.
The server exists to facilitate the exchange of messages between the clients. The server can exchange control messages with the clients. Among other things, it maintains a list of all clients that are connected to it, and their related information (IP address, port number, etc.). Further, **the server stores/buffers any messages destined to clients that are not logged-in at the time of the receipt of the message at the server from the sender, to be delivered at a later time when the client logs in to the server**. You do NOT need to buffer messages for EXITed clients or from BLOCKed clients (see section 5.6). You can assume that the total number of buffered messages will not exceed 100.

Your code, when compiled, will produce a single executable file. Depending on what arguments are passed to this executable (see section 3.3), a client or a server instance should be started.

5.1 Network and SHELL Dual Functionality

When launched (either as server or client), your application should work like a UNIX shell accepting specific commands (described below), **in addition** to performing network operations required for the chat application to work. You will need to use the ***select()*** system call which will allow you to provide a user interface and perform network functions **at the same time** (simultaneously).

5.2 SHELL Commands

Your application should accept commands only when they are inputted:

- In **UPPER CASE**.
- Having exactly the same syntax as described below.

5.3. SHELL Command Output [IMPORTANT]

The **first line of output** of all the commands should declare whether it was successfully executed or it failed.

- If the command was successful, use the following format string:

("[%s:SUCCESS]\n", command_str) //where command_str is the command inputted without its arguments

- If the command failed with error, use the following format string:

("[%s:ERROR]\n", command_str) //where command_str is the command inputted without its arguments

For all such required output, you should only use the special print/log function described in section 4. See section 5.4 (IP command) for an example code snippet.

If the command is successful, it should immediately be followed by its real output (if any) as described in the sections below. **Extra output lines in the log file between the success message and output will cause test cases to fail.**

The **last line of the output** of all the commands (whether success or failure) should use the following format string:

("[%s:END]\n", command_str) //where command_str is the command inputted without its arguments

For events, printing format/requirements will be the same as for commands. Each of the event's description tells the value of the command_str you should use for it.

5.4 Server/Client SHELL Command Description

This set of commands should work irrespective of whether the application is started as a server or a client.

- *AUTHOR*

Print a statement using the following format string:

("I, %s, have read and understood the course academic integrity policy.\n", your_ubit_name)

Your submission will not be graded if the AUTHOR command fails to work.

- *IP*

Print the IP address of this process. Note that this should not be the localhost address (127.0.0.1), but the external IP address. Use the following format string:

("IP:%s\n", ip_addr) //where ip_addr is a null-terminated char array storing IP

Example Code Snippet

To generate the required output for this command, you would need the following lines in your code:

```
//Successful
cse4589_print_and_log("[%s:SUCCESS]\n", command_str);
cse4589_print_and_log("IP:%s\n", ip_addr);
cse4589_print_and_log("[%s:END]\n", command_str);

//Error
cse4589_print_and_log("[%s:ERROR]\n", command_str);
cse4589_print_and_log("[%s:END]\n", command_str);
```

Here, command_str and ip_str are char arrays containing **"IP"** and some valid IP address like **"xxx.xx.xx.xx"**, respectively. Any extra output in between the SUCCESS/ERROR and END output will cause the test cases to fail.

- *PORT*

Print the port number this process is listening on. Use the following format string:

("PORT:%d\n", port)

- *LIST*

Display a numbered list of all the currently logged-in clients. The output should display the hostname, IP address, and the listening port numbers, **sorted by their listening port numbers, in increasing order**. E.g.,

1	stones.cse.buffalo.edu	128.205.36.46	4545
2	embankment.cse.buffalo.edu	128.205.36.35	5000
3	highgate.cse.buffalo.edu	128.205.36.33	5499
4	euston.cse.buffalo.edu	128.205.36.34	5701

Use the following format string:

/*The following printf will print out one host. Repeat this printf statement to
 * print all hosts
 * list_id: integer item number
 * hostname: null-terminated char array containing fully qualified hostname
 * ip_addr: null-terminated char array storing IP
 * port_num: integer storing listening port num */
("%-5d%-35s%-20s%-8d\n", list_id, hostname, ip_addr, port_num)

Notes:

➔ LIST output should contain all the currently logged-in clients, **including the client that executed the command**.

➔ The server should NOT be included in the output.

If you do not implement the LIST command correctly, most automated tests for other commands will fail.

5.5 Server SHELL Command/Event Description
This set of commands should work only when the application is started as a server.

- *STATISTICS*

Display a numbered list of all the clients that haveever logged-in to the server (but have never executed the EXIT command) and statistics about each one. The output should display the hostname, #messages-sent, #messages-received, and the current status: logged-in/logged-out depending on whether the client is currently logged-in or not, **sorted by their listening port numbers, in increasing order**. E.g.,

1	stones.cse.buffalo.edu	4	0	logged-in
2	embankment.cse.buffalo.edu	3	67	logged-out
3	highgate.cse.buffalo.edu	7	14	logged-in
4	euston.cse.buffalo.edu	11	23	logged-in

Use the following format string:

/*The following printf will print out one host. Repeat this printf statement to
 * print all hosts
 * list_id: integer item number
 * hostname: null-terminated char array containing fully qualified hostname
 * num_msg_sent: integer number of messages sent by the client
 * num_msg_rcv: integer number of messages received by the client */
 * status: null-terminated char array containing logged-in or logged-out
("%-5d%-35s%-8d%-8d%-8s\n", list_id, hostname, num_msg_sent, num_msg_rcv, status)

- *BLOCKED <client-ip>*

Display a numbered list of all the clients (see BLOCK command in section 5.6) blocked by the client with ip address: <*client-ip*>. The output should display the hostname, IP address, and the listening port numbers, **sorted by their listening port numbers, in increasing order**. The output format should be identical to that of the LIST command.

Exceptions to be handled

➔ Invalid IP address

➔ Valid but incorrect/non-existent IP address

- [EVENT]: Message Relayed

All messages exchanged between clients pass through (are relayed by) the server. In the event of relay of a message <*msg*> from a client with ip address: <*from-client-ip*> addressed to another client with ip address: <*to-client-ip*>, print/log the message using the following format string:

("msg from:%s, to:%s\n[msg]:%s\n", from-client-ip, to-client-ip, msg)

In case of a broadcast message, <*to-client-ip*> will be 255.255.255.255
For the purposes of printing/logging, use command_str: **RELAYED**

5.6 Client SHELL Command/Event Description
This set of commands should work only when the application is started as a client.

- *LOGIN <server-ip> <server-port>*

This command is used by a client to login to the server located at ip address: <*server-ip*> listening on port: <*server-port*>. The LOGIN command takes 2 arguments. The first argument is the IP address of the server and the second argument is the listening port of the server.

On successful registration, the server responds with:

1. List of all currently logged-in clients. The client should store this list for later display and use.

2. All the stored/buffered messages for this client in the order they were received at the server. Each of these messages will trigger an *[EVENT]: Message Received*, described at the end of this section.

A client should not accept any other command, except LOGIN, EXIT, IP, PORT, and AUTHOR, or receive packets, unless it is successfully logged-in to the server.

Notes

➔ You should NOT print the list of clients received as part of the output of the LOGIN command.

➔ The LOGIN SUCCESS/ERROR message should be printed after all the event related output (triggered by the buffered messages).

Exceptions to be handled

➔ Invalid IP address/port number (e.g., 127.abc is an invalid IP; 43f is an invalid port). You can assume that a valid IP/port will always be the actual IP/listening port of the server.

- *REFRESH*

Get an updated list of currently logged-in clients from the server.

- *SEND <client-ip> <msg>*

Send message: <*msg*> to client with ip address: <*client-ip*>. <msg> can have a maximum length of 256 bytes and will consist of valid ASCII characters.

Exceptions to be handled

➔ Invalid IP address.

➔ Valid IP address which does not exist in the local copy of the list of logged-in clients (This list may be outdated. Do not update it as a result of this check).

- *BROADCAST <msg>*

Send message: <*msg*> to all logged-in clients. <msg> can have a maximum length of 256 bytes and will consist of valid ASCII characters.

This should be a server-assisted broadcast. The sending client should send only one message to the server, indicating it is a broadcast. The server then forwards/relays this message to all the currently logged-in clients and stores/buffers the message for the others.

Notes

➔ The client that executes BROADCAST should not receive the same message back.

- *BLOCK <client-ip>*

Block all incoming messages (unicast and broadcast) from the client with IP address: <*client-ip*>. The client implementation should notify the server about this blocking. The server should not relay or store/buffer any messages from a blocked sender destined for the blocking client. The blocked sender, however, will be unaware about this blocking and should execute the SEND command without any error.

Exceptions to be handled

➔ Invalid IP address.

➔ Valid IP address which does not exist in the local copy of the list of logged-in clients (This list may be outdated. Do not update it as a result of this check).

➔ Client with IP address: <*client-ip*> is already blocked.

- *UNBLOCK <client-ip>*

Unblock a previously blocked client with IP address: <*client-ip*>. The client implementation should notify the server about the unblocking.

Exceptions to be handled

➔ Invalid IP address.

➔ Valid IP address which does not exist in the local copy of the list of logged-in clients (This list may be outdated. Do not update it as a result of this check).

➔ Client with IP address: <*client-ip*> is not blocked.

- *LOGOUT*

Logout from the server. However, your application should not exit and continue to accept LOGIN, EXIT, IP, PORT, and AUTHOR commands. In general, on LOGOUT all state related to this client is maintained on both the client and the server.

Notes

➔ LOGOUT does NOT reset the statistic counters (see STATISTICS command in section 5.5).

➔ LOGOUT does NOT unblock any clients blocked by this client.

➔ LOGOUT does NOT change the blocked/unblocked status of this client on the server.

- *EXIT*

Logout from the server (if logged-in) and terminate the application with exit code 0. This should delete all the state for this client on the server. You can assume that an EXITed client will never start again.

- [EVENT]: Message Received

In the event of receipt of a message <*msg*> from a client with ip address: <*client-ip*>, print/log the message using the following format string:

("msg from:%s\n[msg]:%s\n", client-ip, msg)

Note that <*client-ip*> here is the IP address of the original sender, not of the relaying server. For the purposes of printing/logging, use command_str: **RECEIVED**

5.7 BONUS: Peer-to-peer (P2P) file transfer

Implement additional functionality to allow clients to send/receive files. Here, however, **the transfer will take place directly between two clients and will not involve the server**. For this, your implementation should establish a TCP connection between the two clients involved in a file transfer. The implementation does not need to handle any broadcast file transfers. All transfers will take place between a pair of clients.

To send a file <*file*> residing in the same folder as the executable to a client with ip address: <*client-ip*>, a client would execute the following command:

SENDFILE <client-ip> <file>

The receiving client should store the file in the same folder as the executable, with the same name.

Your implementation should be able to transfer both text and binary files. You can assume the maximum file size to be 10 MB.

6. Grading and Submission

The grading will be done using automated tests. Any deviation from the output format/syntax described in previous sections will cause the tests to fail. For a detailed breakup of points associated with each command/functions, see https://goo.gl/UAVWgY

For packaging and submission, see the section **Packaging and Submission** in https://goo.gl/L2kgb5