CMPT 300 D100
Assignment 3
Our new chatbot – Lets-talk
Marks: 100 marks

Notes:

1. **Failure to follow the instructions and rules may lead to failed test cases and/or a final grade of 0.**
2. You can do this assignment individually or in a team of two. If you are doing it in a group, only one submission per group is required. For example:

   If Person A and B are in one group, only one person has to accept the assignment link and do the submission.
   Person A: upload all files (.txt, .c), group.txt.
   Person B: no submission required.

   For group assignment, submit a "***group.txt***" indicating the Github ids of each member. For example: if I am working with John in a group then group.txt will be as below:

   ```
   hazra.imran
   Johndoe
   ```

- You may submit multiple times until the deadline. Grade penalties will be imposed for late submissions (see the course outline for the details).
- Always plan before coding.
- All the codes in this lab must be done using C language only. No other languages should be used.
- Use function-level and inline comments throughout your code. We will not be specifically grading documentation. However, remember that you will not be able to comment on your code unless sufficiently documented. Take the time to document your code as you develop it properly.
- We will carefully analyze the code submitted to look for plagiarism signs, so please do not do it! If you are unsure about what is allowed, please talk to an instructor or a TA.

Coding Rules

- You have to follow the file name as specified in the instructions.
- **Makefile**: Makefile provides the following functionality:
  - **all**: compiles your program (this is the default behavior), producing an executable file named the same as the C file.
  - **clean**: deletes the executable file and any intermediate files (.o, specifically)
  - You will receive 0 if your makefile fails.
    - Check your build to ensure that there are no errors.
    - Visit TA's programming office hours to get help.

Lets-talk

Background

For this assignment, we are going to create a simple "chat"-like facility that enables a user at one terminal to communicate with a user at another terminal. We are not expecting a pretty interface but it should be functional.

For this assignment, you will be using two main LINUX concepts.

1. Threads - creating a threads list and assigning tasks to them.
2. UDP - Check the following man pages for help with UDP:
   - socket
   - bind
   - sendto
   - recvfrom
   - getaddrinfo

There are a couple of good web pages that we will point you to:

- Socket programming: http://beej.us/guide/bgnet/
- Socket programming: https://habibiefaried.medium.com/cnet-01-basic-c-socket-programming-a818b0da5ae0
- Pthreads documentation: https://computing.llnl.gov/tutorials/pthreads/
- Pthreads: https://medium.com/@jithmisha/a-brief-intro-to-shared-memory-programming-with-posix-threads-a663b590e38c

This assignment will be done using Pthreads, a kernel-level thread implementation for LINUX. Pthreads allows you to create any number of threads inside one UNIX process. All threads running in the same UNIX process share memory (which means pointers valid for one thread are valid in another thread which is not possible between processes) and also have access to semaphores (mutexes) and the ability to use conditional signal/wait to synchronize their actions in relation to each other. UNIX itself also allows you to create multiple processes. Communication between UNIX processes can be done using something called "datagram sockets" which use a protocol called UDP (user datagram protocol).

Assignment requirements

In this assignment, you will be dealing with processes/threads on two levels. You will have two LINUX processes (might not be on the same machine). Each one is started by one of the people who want to talk (by executing lets-talk).

Required threads (in each of the processes):

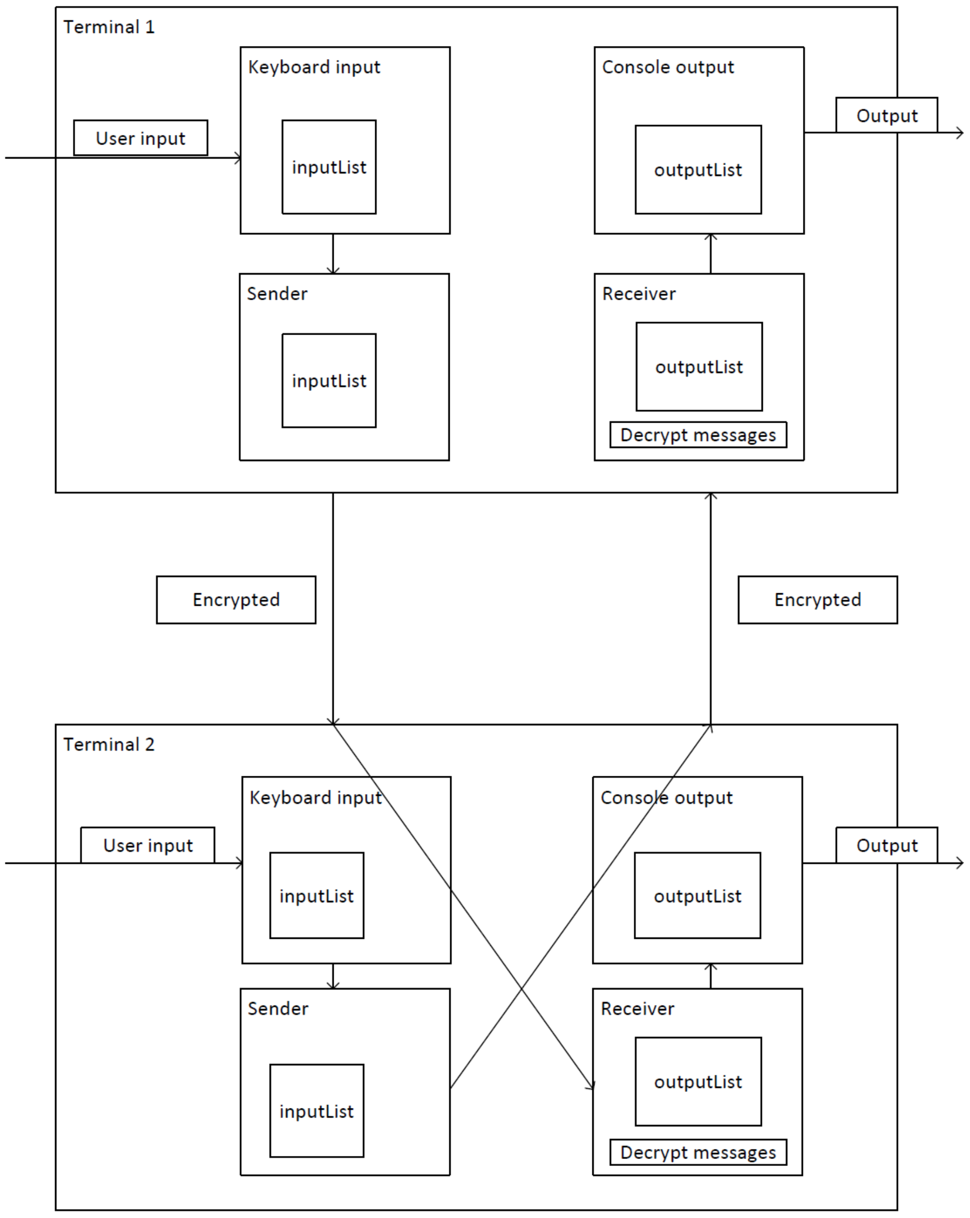
1. One of the threads awaits input from the keyboard.
2. The other thread awaits a UDP datagram from another process.
3. There will also be a thread that prints messages (sent and received) to the screen.
4. Finally, a thread that sends data to the remote UNIX process over the network using UDP (use **localhost** IP(127.0.0.1) while testing it on the same machine with two terminals).

All four threads will share access to a list ADT..

- The **keyboard input** thread, on receipt of the input, adds the input to the list of messages that need to be sent to the remote lets-talk client.
- The UDP **sender thread** will take each message off this list and send it over the network to the remote client.
- The UDP **receiver thread**, on receipt of input from the remote lets-talk client, will put the message onto the list of messages that need to be printed to the local screen.
- The **console output thread** will take each message off this list and output it to the screen.

Below is the diagram to show the workflow.

Joinable or detached? Probably detatched
why? Each thread has a unique task



What should the end product look like?
You will be left with **one** executable file (lets-talk) after compiling the assignment. It should take 3 parameters to start.

1. The IP address of a remote/local client.
2. Port number on which remote/local process is running.
3. Port number your process will listen on for incoming UDP packets.

Say that Alice and Bob want to talk. Alice is on machine "machine1" and will use port number 6060. Bob is on machine "machine2" and will use port number 6001.

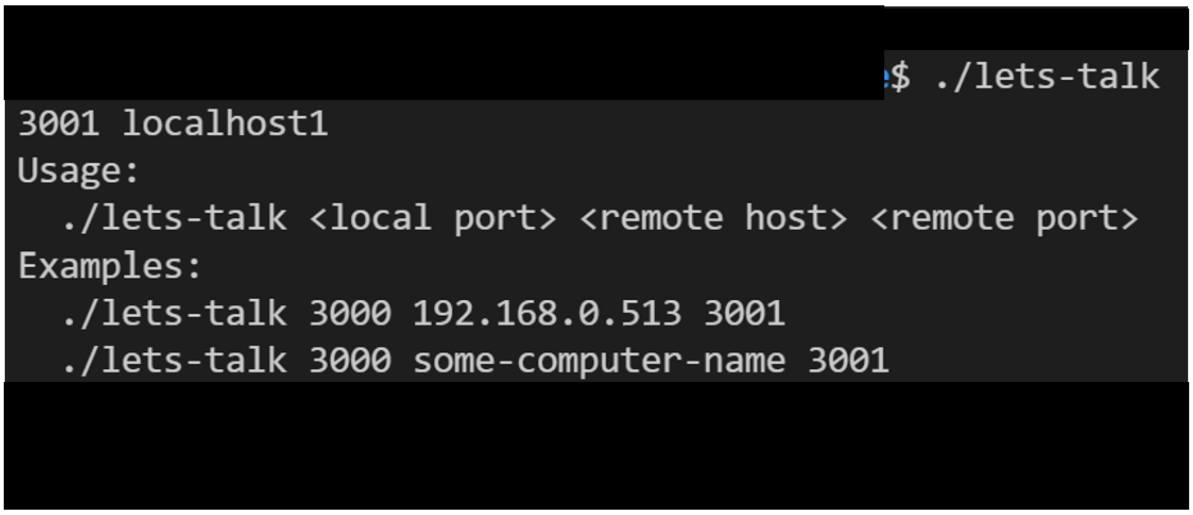
To initiate lets-talk, Alice must type:
lets-talk 6060 IP_of_machine2 6001

And Bob must type:
lets-talk 6001 IP_of_machine1 6060.

The general format is:

lets-talk [my port number] [remote/local machine IP] [remote/local port number]

If the machine IP is **incorrect** then it must print a **usage message with an example** as shown below.



```
[redacted]$ ./lets-talk
3001 localhost1
Usage:
  ./lets-talk <local port> <remote host> <remote port>
Examples:
  ./lets-talk 3000 192.168.0.513 3001
  ./lets-talk 3000 some-computer-name 3001
[redacted]
```

Below are the requirements that must be implemented.

1. Require makefile to compile code and provide a **single** executable.
    - Intermediate files should be cleaned.
    - The “make” command should generate an executable.
    - “make Valgrind” should start Valgrind for a memory leak check.

2. Use the list to communicate between threads.
    - For example, the **receiver thread** will put a message in the list and the **console output** thread removes msg from the list and prints it on the screen.
    - Use the list.c and list.h (added already in your repo) to use **mutex** so that the list should be accessible by only one thread at a time.

3. Encryption (use cipher encryption and description)
    - Use a fixed Encryption key
    - Increment each character by key before sending (take care about character size, take modulo by 256)
    - Decrement each character at the receiver side by key.

Note - In the above encryption method, it can happen that a character in the string translates to the null character in the encrypted string. That will cause the termination of the message. This happens because, in the LINUX implementation of UDP, it terminates the message sending as soon as it finds a null character or newline character.

4. Checking the ‘online’ status of another terminal
    - !status should print Online/Offline if the user on another end is online/offline.

5. **"!exit"** will quit the connection. The !exit must be implemented to ensure that program ends cleanly on both terminal. The !exit should be **case-sensitive**.
    - The program must be terminated on both the terminals if any of the user types this command

6. Require support for one lets-talk talking to itself (talk on localhost).
    - Must work on localhost (127.0.0.1)

7. Require support for copy-paste up to any number of lines of text, each line any length characters long.

8. Require support for very long lines (up to 4k characters)

9. Require no busy waits and not take up 100% CPU usage when nothing to do.
    - Avoid deadlock using mutex locks from provided list structure(list.h and list.c). Also, ensure that there is **no infinite loop** in the program.

10. Require no extra outputs when the user typing in the text (no prefix to sending or receiving messages).

11. Require checking return values on all socket functions.

12. Require calling fflush(stdout) wherever needed.

13. Require must pass Valgrind cleanly:

    * OK to have unfreed memory which comes from pthread_cancel_init
    * Everything else must be freed.
    * No uninitialized memory to syscall errors.
    * Use this if you don't know how to use Valgrind.
    * Make sure that progam exit cleanly at the time of running the Valgrind command.

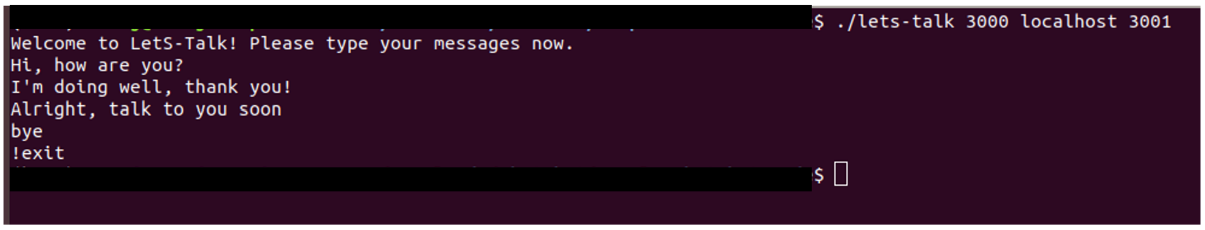
To help you to get started, below are the steps to implement the chatbot.

1. Create 4 threads as mentioned above in the main program.
2. Create two lists
    - The first list will be used by the keyboard thread and sender thread
    - The keyboard thread will get input from the keyboard and put a message in the list
    - The sender thread will get a message from the list and send it to another client
    - The second list will be used by the receiver thread and printer thread
    - The receiver thread will put messages got from another client on the list
    - The printer thread will remove messages from the list and print them on the monitor

3. The main thread will wait until any of them gives a termination signal, as soon as the main thread gets a termination signal it cancels all threads one by one.

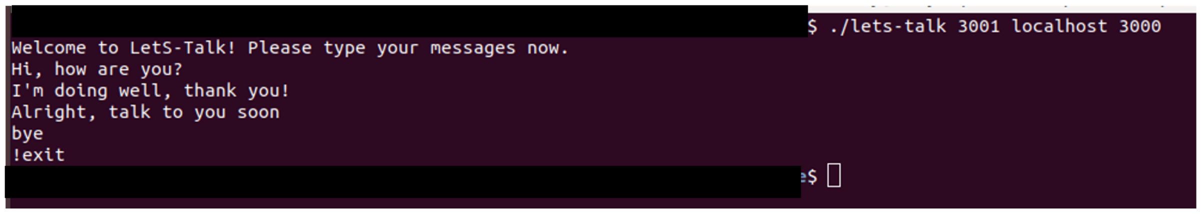
Sample output:

The user at terminal 1:



```
[redacted]$ ./lets-talk 3000 localhost 3001
Welcome to LetS-Talk! Please type your messages now.
Hi, how are you?
I'm doing well, thank you!
Alright, talk to you soon
bye
!exit
[redacted]$ 
```

The user at terminal 2:



```
[redacted]$ ./lets-talk 3001 localhost 3000
Welcome to LetS-Talk! Please type your messages now.
Hi, how are you?
I'm doing well, thank you!
Alright, talk to you soon
bye
!exit
[redacted]$ 
```

Test Cases

- Run the !exit command to validate that the connection is terminated for both users by running it on either user's side
Note: The !exit should be case sensitive
- Run the "!status" on either side to validate if the other user is online or offline
- Send messages from either terminal and validate if the message is received on the other terminal.
- The messages must be transmitted between both users only when both users are online.
- Since there is multithreading involved, you need to use mutex locks effectively. Validate to make sure that multiple threads aren't accessing the same list simultaneously.
Note: You can do this by checking mutex before accessing the list functions.

Submission Instructions:

For A3, the concrete required deliverables are:

- group.txt
- lets-talk.c
- .h files or .c files (if any) - Already added in the repo
- Makefile

Grading Criteria

Rubric (Out of 100)
Incorrect makefile: -100 (0 grade)

Correct makefile: +10
Must have four threads +10
Must use the List structure to exchange data between threads +10
Must use of mutex to protect lists from simultaneous access +10
Must not have memory leaks. Must free all lists and dynamically allocated memory before exiting. +10
Correct Encryption +10
Checking online status +5
!exit - The !exit should be case sensitive. +5
!exit - The program should terminate on both the users if any of them types this command +5
Should work on localhost(127.0.0.1) +5
Support for copy-paste up to any number of lines of text, each line any length characters long. +5
Support for very long lines (up to 4k characters) +5
Checking return values on all socket functions and calling fflush(stdout) wherever needed. +5
Must pass Valgrind cleanly +5