1. Overview

In this assignment, you will implement a chat client. The client will be written according to a protocol that we specify and will interact with a chat server that we have written and provided. Communication between the client and server includes actual chat messages as well as control messages. Chat messages are exchanged using the User Datagram Protocol (UDP), while control messages are exchanged using the Transmission Control Protocol (TCP). In addition, you will implement a TCP-based information server associated with the chat client that serves information regarding the chat client.

The objectives of this lab are to:

- Work with and implement a simple protocol.
- Learn to use the Berkeley Socket API to a network stack.
- Develop the C coding tricks and practices needed for successful network programming.

2. System Architecture and Operation

Figure 1 summarizes the architecture of the chat system, indicating parts that you will implement and parts that we will provide.

The server conducts chat sessions in chat rooms. The names of chat rooms are read in from a configuration file when the server begins executing. They remain unchanged throughout the lifetime of the server. Every client is present in one of the rooms at any time and may switch to a different chat room at any time.

Each client initiates a chat session by providing the following information to the server: (i) nickname that client wishes to use; (ii) name of the host (e.g., bass.cmcl.cs.cmu.edu) on which the client is running; (iii) UDP port of client where chat messages must be sent; and (iv) TCP port on which information server is listening for new connections. The server responds by either granting or refusing the client permission
to join. Once a client is registered with the chat server, it may send control messages that ask the server to perform the following functions:

- List users currently logged on.
- List names of chat rooms.
- Switch to a given chat room.
- Quit the chat session.

A TCP connection must be established for every control message that the client sends, and each connection must be used for exactly one control message. When the server receives a control message, it sends a reply and immediately shuts down its end of the connection.

Chat messages are sent from the client to the server via UDP, and then relayed via UDP to all clients present in the room of the sending client (including the sender).

Every client has some information associated with it that it chooses to make available to other clients, e.g., finger information stored in an ASCII file. A client $A$ that wishes to access the information associated with client $B$ contacts the chat server to get the TCP port of client $B$’s information server, and then contacts $B$’s information server for the relevant information.

**Rationale**

We use UDP for chat messages for two reasons. First, TCP implies maintaining per-connection state at the chat server, which could affect server scalability. Second, we are operating in a LAN environment where packet-loss and congestion are rare.

We use TCP for control messages because these are more critical and exchanged less frequently. A connection lasts only for the duration of a single control message, rather than throughout the lifetime of
the client. (Earlier versions of the HTTP protocol use a similar strategy where the server processes a client request and shuts down the connection immediately.)

We use an information server on the client side rather than storing the information in the chat server because it allows the client to make arbitrary chunks of information available, and because it could allow clients to be selective about who gets access to the information.

3. Description of Messages Exchanged

Messages exchanged between entities in the chat system are composed entirely of ASCII characters.

TCP control messages between chat server and chat client

All exchanged messages consist of a list of fields separated from each other by spaces and terminated with a newline character (’\n’). A newline should not occur anywhere inside the message. All fields (including integer fields) are represented as text. For example, 15213 is represented as the string “15213”.

A client may send the following TCP control messages to the server.

- JOIN_REQUEST <membername> <hostname> <udpPort> <tcpPort> \n  <membername> is the nickname the member wishes to use.
  <hostname> is the name of the machine on which the client is running.
  <udpPort> is the UDP port of the client where chat messages must be sent.
  <tcpPort> is the TCP port of the information server where information requests must be sent.
  The client uses this message to log on to the server (i.e., initiate a chat session). The client must send this message first, before sending any other messages.

- SWITCH_ROOM_REQUEST <memberID> <roomName> \n  <memberID> is the ID assigned to the member. (Refer to JOIN_RESPONSE below)
  <roomName> is the name of the chat room the member wishes to switch to.

- MEMBER_LIST_REQUEST \n  This message asks for a list of all members logged on to the chat session.

- ROOM_LIST_REQUEST \n  This message asks for a list of names of all chat rooms.

- MEMBER_INFO_REQUEST <remoteMemberName> \n  <remoteMemberName> is the nickname of the member whose information the client wants.

- QUIT_MSG <memberID> \n  <memberID> is the ID assigned to the member.
  This message must be sent when the client terminates the chat session.

The chat server will send the following TCP control message responses to the client:
• JOIN_RESPONSE_OK <memberID> \n
JOIN_RESPONSE_REFUSE \n
<memberID> is a randomly assigned ID that is used to identify this client in subsequent messages. The server can deny the client permission to enter the system either because the server limits the total number of chat session, or because a member with that nickname is already logged in. If a server grants permission to the client, then it returns a randomly chosen memberID that identifies the client in subsequent exchanges. This ID makes it more difficult for another client to spoof messages under the nickname of the original client.

• SWITCH_ROOM_RESPONSE_OK \n
SWITCH_ROOM_RESPONSE_REFUSE \n
The server can deny the client permission to switch to a different room, either because there are too many members already present in the room, or because no room exists by that name.

• MEMBER_LIST_RESPONSE <space-delimited list of member nicknames> \n
• ROOM_LIST_RESPONSE <space-delimited list of rooms> \n
• MEMBER_INFO_RESPONSE_INFO_FOLLOWS <hostName> <TCPPort> \n
MEMBER_INFO_RESPONSE_UNKNOWN_MEMBER \n
<hostName> is the name of the machine on which the remote client is executing. <TCPPort> is the TCP port of the information server of the remote client.

If no member exists with the requested name (in MEMBER_INFO_REQUEST), then the server replies with a UNKNOWN_MEMBER response. Otherwise, the server returns the hostname on which the remote client is executing, and the TCP port of the information server of the remote client.

**UDP chat messages between chat server and chat client**

All exchanged messages consist of a list of fields separated from each other by spaces and terminated with a “\n”. A “\n” should not occur anywhere inside the message.

This is the format of a UDP chat message from the client to the server:

• <memberID> <message> \n
  <memberID> is the ID assigned to the member.
  <message> is the message to be sent. It should not contain any newline characters and the message must not exceed a given length (specified in the header files given to you).

This is the format of a UDP chat message from the server to the client:

• <fromMemberName> <message> \n
  <fromMemberName> is the nickname of the member who originated the chat message.

**TCP messages between information server and information client**

This is the format of a TCP message from the information server to the client:
<message length>  <message>

<message length> is length in bytes (expressed as a sequence of ASCII characters) of <message>, where <message> is an arbitrary sequence of ASCII characters. In contrast to earlier messages, <message> may contain newline characters need not be terminated with a newline.

The information client does not send any messages to the server.

4. User Interface

The focus of this Lab is on programming with sockets and not on fancy user interfaces. We are thus interested in a minimal interface that does a reasonable job. Here a few requirements that we lay down.

- The user should only need to execute a single binary, chatclient, along with commandline arguments (described later). The user must not be required to physically execute any other binary. However, chatclient may choose to call other binaries from inside it.

- The client must employ two windows: A user input window where the user types commands and chat messages, and a chat window that displays chat messages (including the chat messages from this client).

A simple way of generating a second window from inside a C program is to exec an xterm using the -e option. For example, xterm -e mybinary’ launches a new xterm and begins executing mybinary inside it. Check the xterm man pages for details. You are free to choose any other technique for the generation of windows: X libraries, Motif, the curses library, Tcl/Tk, etc. However, please do not spend too much time on it. We would rather you concentrate on the systems programming aspects. Also, bear in mind that we may not be able to help you too much here.

- The user input window displays a prompt such as “>”. The user can either type in a message or a command. Commands begin with a “!” (bang) character; all other user input must be interpreted as messages. The list of commands that you must support, and actions that must be taken are summarized in Table 1.

- The executable chatclient must support the following command-line arguments:
  -m nickname that user wishes to use.
  -i file with member information used by the information server.
  -h hostname of chat server.
  -t TCP port of chat server.
  -u UDP port of chat server.

In addition, you may use other arguments to support your individual implementation.

4.1. Snapshot of a client session

We present a snapshot of the client session below. This only shows the user-input window and does not show the chat-message window. The message “Baseball...” is the only chat message (all others are preceded
<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>![r]</td>
<td>List names of all chat rooms. The client sends a ROOM_LIST_REQUEST to the server and lists all the room names provided in the ROOM_LIST_RESPONSE the server sends.</td>
</tr>
<tr>
<td>![m]</td>
<td>List names of all members. The client sends a MEMBER_LIST_REQUEST to the server and lists all the member names provided in the MEMBER_LIST_RESPONSE the server sends.</td>
</tr>
<tr>
<td>![s] &lt;roomname&gt;</td>
<td>Switch to a new room indicated by &lt;roomname&gt;. The client sends a SWITCH_ROOM_REQUEST to the server and waits for a SWITCH_ROOM_RESPONSE. If permission to switch is granted, a “Switch to &lt;roomname&gt; successful” message must be printed. Otherwise, a “Switch to &lt;roomname&gt; failed” message must be printed.</td>
</tr>
<tr>
<td>![i] &lt;membername&gt;</td>
<td>Get information about member &lt;membername&gt;. This is a two-step process. First, the client sends a MEMBER_INFO_REQUEST to the server and waits for a MEMBER_INFO_RESPONSE. If the server replies with an UNKNOWN_MEMBER, a “Member not known” message must be printed. Otherwise, the server provides the hostname and TCP port of the information server of &lt;membername&gt;. The second step involves the client contacting the information server of &lt;membername&gt;, gathering the required information and displaying it on the screen. A member may not request its own information.</td>
</tr>
<tr>
<td>![l]</td>
<td>Print the number of requests that the information server has received.</td>
</tr>
<tr>
<td>![q]</td>
<td>Quit. The client sends a QUIT_MSG to the server and exits.</td>
</tr>
</tbody>
</table>

Table 1: Commands that your client must support

by the ![], and is relayed to all members present in the room “cricket”. Note that the “-f” option is required by our particular implementation of chatclient, and you need not support it. The information server has received and serviced 2 requests during the session.

[sanjay@char]$ chatClient -m student -h chum -i info -t 4000 -u 4000 -f /tmp/fifo
>![r]
CRICKET SOCCER
>![m]
STUDENT SANJAY
>![i] sanjay
Dr. Evil (Randy/Dave) hired me to give you all sleepless nights struggling with this assignment. Muahahahahahaha! And yet, wouldn’t you agree this was the most fun lab of them all?
>![s] cricket
Switch to cricket done
>Baseball sucks, cricket rules. Football sucks, soccer rocks.
>![l]
The information server has received 2 requests so far
>![q]
>
[sanjay@char]$
5. Simplifying Assumptions

Here are assumptions you may make to simplify the design of your chat client.

- The duration of each information request is short enough that a simple iterative information server suffices. There is no need to develop a concurrent server. (An iterative server is one that sequentially processes every client, as opposed to a concurrent server that processes several clients in parallel). Server design might get pretty complex in general - for this assignment we are only interested in a very simple server.

- Users may request at most one command at a time. A second user command (or message) will block till the first command is completed.

- The duration of actions taken when each of the following I/O events occur — (i) a user command is issued; (ii) a UDP message is received; and (iii) an information request is received — is so small that it is acceptable to have the client block for this duration of time. However, note that any of the events may by themselves occur independent of each other and your design must take concurrent arrival of events into account.

- All information served by the information server is printable ASCII text and has no binary characters.

6. Resources, References and Useful Tips

In this assignment, you will be using a large number of system calls that you have probably not used previously. We plan to discuss important system calls, programming techniques and example code during the next few lectures and recitations. Please plan to attend. We will make server code available to you. You can use this as a starting point in designing client code. While we will try to provide as much information as possible, realize that it cannot be complete. Here are some sources of information that you could use:

- The class Web page will have pointers to tutorials on network programming.

- Man pages for detailed descriptions of the functions, precise syntax, return values etc. Further, function calls that you use might require special include files - the man pages will tell you which files you must include.

- An excellent reference that I personally find useful is: Unix Network Programming : Networking APIs: Sockets and XTI (Volume 1) by W. Richard Stevens.

**Useful system calls:** We list below system calls that you might expect to use. While this is not an exhaustive list, if you find yourself using many calls outside this list, you might want to check with your TA as to whether you are on the right track. (Also, keep a watch on the webpage. We’ll keep adding to the list as and when we realize we omitted to mention a call.) Further, depending on your particular design, you may not require to use some of these calls.

- Basic socket related: socket, bind, connect, listen, accept, read, write, recvfrom, sendto, recv, send, close
Less common socket functions: getsockname

Byte order related: ntohs, ntohl, htons, htonl

Name resolution related: gethostbyname, uname

String to integer conversion: atoi

Select and related: select, FD_ZERO, FD_ISSET, FD_SET

Fork and related: fork, execlp

Named pipe related: mknod

Modular Development: While the system might appear a little overwhelming at first sight, the best strategy is to develop and test components one by one. It is better to have a subset of the functionality completely implemented and tested, rather than the entire system implemented and buggy. Here is a possible modular strategy for developing and testing your code - feel free to employ a different sequence of steps:

- Develop the TCP client with support for JOIN_REQUEST, JOIN_RESPONSE and QUIT_MSG.
- Add support for SWITCH_ROOM_REQUEST and SWITCH_ROOM_RESPONSE.
- Incorporate the UDP client. Check if you are able to send and receive chat messages.
- Extend your TCP client to support MEMBER_LIST_REQUEST, ROOM_LIST_RESPONSE (and their responses).
- Extend the client to add support for MEMBER_INFO_REQUEST (and response). Next ensure you are able to connect to a remote member’s information server and get required information.
- Finally, develop the information server and incorporate it to the rest of the system.

Check all return values: You’ll save yourself a lot of heartache if you check the return values of all system routines for error conditions.

Possible design: We present a very high-level outline of our client design. While this could serve as a starting point, we encourage you to think of your own design. Our code consists of two binaries. One of them is a dumb UDP client that merely receives and displays chat messages. The second binary forks and execs the UDP client, and simultaneously services user input and server requests.

Using telnet as a client: Telnet can be used as a client for a TCP server. You can use this to figure out how TCP control messages are processed by the chat server, as well as for debugging your TCP information server. Following is an example:

[sanjay@char]$ telnet chum 4000
Trying 128.2.222.197...
Connected to CHUM.CMCL.CS.CMU.EDU.
Escape character is ’”’.
JOIN_REQUEST sanjay char 1000 3000
JOIN_RESPONSE OK 1804289383
Connection closed by foreign host.
[sanjay@char]$ telnet chum 4000
Trying 128.2.222.197...
Connected to CHUM.CMCL.CS.CMU.EDU.
Escape character is ''\]''.
ROOM_LIST_REQUEST
ROOM_LIST_RESPONSE CRICKET SOCCER
Connection closed by foreign host.

[sanjay@char]$ We are distributing the server source code to allow you to examine it and play around without depending on the central server we are running. An issue is that several of you may be running multiple server copies in the same machine, but at any time at most one process can bind to a given port. We have thus written the server such that it binds to a requested port if available. However if the port is not available, then, it allows the kernel to choose a port and prints a message displaying which port it is using instead. This is an example message you may see:

[sanjay@chum]$ server -r rooms -t 4000 -u 4000
Unable to bind to UDP port 4000. Running on port 1989 instead
Unable to bind to TCP port 4000. Running on port 2070 instead

All you need to do in such a case is use the appropriate ports while executing chatclient. For example, you would execute:

chatclient -m student -h chum -i info -t 2070 -u 1989 -f /tmp/fifo1

The problem of reincarnation: Currently, once a client has registered with the server (using a JOIN_REQUEST, JOIN_RESPONSE sequence), an entry for the client is maintained with the server until the arrival of a QUIT_MSG, followed by the appropriate member id. Now, it is quite possible that you forget to send the QUIT_MSG. In such a case, you cannot reuse the same member-name as earlier when you reexecute your program, unless you send a QUIT_MSG initially. You would have to choose a fresh member-name. We are working on a strategy for dealing with this at the server (e.g., delete entries for members who have not sent chat messages in a while). Keep watching the newsgroup and web-page for an update. **UPDATE: Version 1.1 of the server has a command-line option -f <flushTime>. Entries for clients that have not sent an update message for <flushTime> minutes are deleted.**

Using named pipes on AFS: If your design involves the use of named pipes, you cannot create them on AFS. You would have to use local storage, say /tmp. Use a sensible way of naming these files such that it does not conflict with other groups. For example, you could prefix the name with your own names.

7. Evaluation

We will evaluate your assignment through demonstrations. We will have each group visit one of the teaching staff and demonstrate the running of the client. We are working out details of this. Most likely you will have to choose from available time slots on December 2 and 3. Please watch the webpage and newsgroup for updates.

The project will primarily be graded on: (i) overall design of the system; (ii) how closely your code implements the described protocol; (iii) how much of the described functionality you are able to support,
and how correct your implementation is; and (iv) coding style. Design and style are important components in this Lab. Significant lack of documentation or elegance within your code will affect your grade.

Your grade will be based on the following:

- **Correctness and completeness (50%).** To get some credit, you must at the minimum be able to join the system, switch to a particular room and send and receive chat messages (20%). Handling all other control messages in a graceful manner earns you another 15%. Support for the Information Server accounts for a further 15%.
- **Design (25%).** Elegance, robustness of design and handling of error conditions and special cases will be considered.
- **Documentation (15%)** You should split up code into modules, make use of header files, use reasonable variable names and comment portions of your code that reflect design choices and non-intuitive ideas.
- **Demonstration (10%)** You must be able to answer questions pertaining to your design and code.

An implementation that is significantly buggy or misses important components can only earn partial points for design and documentation.

8. **Logistics**

As usual, you may work in groups of up to 2 people. The files for the chat server can be retrieved from:

/afs/cs.cmu.edu/academic/class/15213-f99/L5/L5.tar

Once you’ve copied this file into a (private) directory, run the command `tar xvf L5.tar`. Here is a brief description of the files:

- **Makefile:** Makefile for the server. You can look at this to get an idea of how to write a makefile for the client.
- **server.c:** Chat server code
- **serverInternal.h:** Internal data structures that the server uses
- **server.h:** Defines useful constants. You must include this file in your client code.
- **rooms:** A file containing list of roomnames. Feel free to edit this file.

To compile, run the command `make server`. Remember, you do not need to examine the server source. The intention is to provide you with a local copy of the server for your use, and for you to get an idea of how some of the system calls are to be used.

The server may be executed by running the following command:
server -r <roomNameFile> -t <Desired TCP Port> -u <Desired UDP Port> 
   -f <flushTime> -l <logFile>
Parameters -r, -t, -u are compulsory. -f, -l are optional.
The server deletes entries for all clients that have not sent a chat 
message for at least <flushTime> minutes.

You may also type “server” to get usage information.

8.1. Main Chat Server

We will be running a chat server on the machine bass.cmcl.cs.cmu.edu with TCP port 4000 and 
UDP Port 4000, for your amusement. (The machine and port numbers are subject to change, please watch 
announcements). We would advise you to use this only when your code is reasonably mature, and use a 
local copy of your server until then. Further, this is a new assignment, and it is possible the server has bugs. 
Please report bugs to sanjay@cs.cmu.edu. The server might not be running all the time, however your 
code development and testing should not be affected as you have access to a local copy of the server.

8.2. Handin

You must handin all the relevant client sources and header files, along with a Makefile to compile the client. 
(We will briefly talk about writing a Makefile during recitation). You must follow these steps for your 
handin:

- Copy only relevant files (client sources, client header files, Makefile) to a private directory.
- Run the command tar cvf <groupName>.<versionNum>.tar.
- Copy the tar file into the directory
  /afs/cs.cmu.edu/academic/class/15213-f99/L5/handin

Your handin must be present by the deadline on the front page, irrespective of when your actual demonstration 
is. This is to ensure we are fair to students who have an earlier scheduled demonstration. You will compile 
and run the code in this directory during the demonstration.

9. Epilogue

We realize that this handout is long, many concepts are new, and you are struggling with several course 
projects as the semester draws to an end. However, please do not feel overwhelmed. The length of the 
handout is only for clarity; this assignment is not difficult once you get warmed up and we will try to provide 
as much help as possible through lectures and recitation. Start early, and feel free to discuss difficulties with 
your teaching staff. Remember our advice on modular development.

And finally you can really have fun on this Lab with all your classmates. Think of imitating interesting 
characters, innovative chat room names - surely you need no advice from us on this! We sincerely hope 
you will learn a lot and still derive a lot of fun from the lab. We look forward to your feedback on what you 
think about this lab, and this could influence plans for future semesters.