CS 213, Spring 2000
Lab Assignment L5
Assigned: April 13
Due: May 1, 11:59PM
(Mini-milestone check due April 24, 11:59PM)

Jun Gao (jungao@cs.cmu.edu) is the lead person for this assignment.

Introduction

In this assignment, you will implement the client side of a chat system. The chat system consists of two distributed components: chat server and chat client, which may run on different hosts in the network. We provide you the server side of the system. The chat client you will build must support a user interface to allow users of the chat client to type in control commands and chat messages. The chat client must also be able to display received chat messages. The chat server conducts a chat session in *chat rooms*. At any given time there may be multiple chat clients in a chat session. The chat server is responsible for managing all the chat clients in the session and distributing chat messages. A chat client starts by registering with the chat server to join the chat session. After joining the session, a chat client can allow the user to switch to one of the chat rooms and send and receive chat messages. The communication between the client and server includes two types of messages: *control* messages and *chat* messages. The control messages will allow a chat client to perform tasks such as joining and leaving the chat session, creating a chat room, switching to a certain chat room, etc. There are two types of chat messages: *public* and *private* chat messages. To send a public chat message, a chat client must be present in one of the chat rooms. The chat client sends all its public chat messages to the chat server, and it is the chat server's job to relay these messages to all the chat clients within that room. On the other hand, a chat client sends private chat messages directly to another registered chat client. The private message sender and receiver are not required to be in the same chat room, actually they do not need to be even in any chat room.

The primary goal of this assignment is for you learn two important Internet transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) and specifically how to use Berkeley Socket API to write applications on top of these protocols. In addition, since the messages exchanged between client and server use a simple protocol that we specified, by doing the assignment, you will learn how to work with and implement a network protocol, particularly how to build network packets and how to interpret packets according to the protocol specification. One of the network programming difficulty comes from the fact that you have to make distributed entities (client/server) residing at different locations in the network to work together. This makes coding and debugging harder. The network programming skills you will pick up from this assignment would certainly be a valuable asset to your programming repertoire.

System Overview

In the chat system, we use TCP for the *control* messages between chat client and server, we use UDP for the *public chat* messages between chat client and chat server, and we use TCP for the *private chat* messages between chat clients.

The rationale behind this design is from the basic properties of the two protocols. TCP is a connection-oriented reliable protocol and states related to each TCP connection need to be maintained at both client and server side. To allow the chat server scale (accomodate many clients), we use TCP only for mission critical and rare actions, e.g., control messages. Since we view chat messages as non-critical and there will be many of them, we use a lighter weight protocol, UDP, for them. UDP is connection-less and there are no connection related states need to be kept. However UDP is unreliable, which means users' packets can get lost inside the network. Packet loss is considered rare in the our LAN environment.

We now summarize the network related components needed at the chat server side and the chat client side. For the chat server, we implemented a TCP server and a UDP server. The TCP server is responsible for receiving TCP control messsages from clients and sending clients replies. The UDP server is responsible for receiving chat messages from chat clients and sending chat messages to the clients.

For the chat client, you need to implement the following:

TCP-based client connects to chat TCP server, sends and receives control messages.

UDP-based client/server sends public chat messages, receives public chat messages.

TCP-based private server receives private chat messages.

One TCP connection is established for each round of control message exchange between the chat client and the chat server. On the chat server side, after a control message is received and a reply is sent, the connection is immediately terminated. We do not want to maintain this connection for the duration of the chat client because once a client joined the session, it may only send chat messages and never send a control message until it is ready to quit the session. To keep the TCP connection around is a waste of resource and will not allow the chat server to scale.

A public chat message is sent from chat clients to the chat server via UDP, and the chat server will then relay the message to all the clients in the appropriate chat room (including the sender client) also via UDP.

To send a private chat message, a chat client has to first send a control message to the chat server to retrieve certain information about the intended receiver client. The information includes the IP address and TCP port number of the receiver client. Once the information is received, the sender client will connect to the receiver client directly and send a private chat message. This connection is also only good for one message and must be shut down immediately after the message is sent.

Protocol Specification

TCP-based control message protocol

This protocol is used between the chat client and the chat server to exchange control messges. Each message exchanged has a common header (two 4-byte words) and may contain extra data in the data area. The data area in a packet is known as the *payload* in the networking lingo. A control message looks like this:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   msg_type                      |      member_id                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   msg_len                       |      reserved                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   msg_data ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                                                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The header is defined by the following structure in `defs.h`:

```
struct control_msghdr {
    u_int16_t msg_type;
    u_int16_t member_id;
    u_int16_t msg_len;
    u_int16_t reserved;
    caddr_t   msgdata[0];
};
```

`msg_type` represents the type of this message. We will explain each of the defined message types next.

`member_id` is the id of the member that sends the message. An id is issued to each client after they successfuly registered. For each message (except REGISTER_REQUEST) sent from the client to the server, a valid id must be included. The `member_id` in a message from the server to a client is the `member_id` of the intended client.

`msg_len` denotes the length (in bytes) of the message including both the header and data.

`reserved` field is not currently used.

`msg_data` is a place holder pointing to the data area of the message. (`caddr_t` is defined as `char *`). The specific meaning of the data area depends on the message type.

The following message types are defined in `defs.h`

1. REGISTER_REQUEST

   This is the first message sent from client to server after the client starts. Since at this point, the chat client does not have a valid member_id, this field will be ignored by the server. The data area of the message will contain information specific to this client: the UDP port that the client uses to receive public chat messages, the TCP port that client uses to receive private chat messages, and the member name that the client would like to use. This part is defined by another structure in `defs.h`:

   ```
   struct register_msgdata {
       u_int16_t udp_port;
       u_int16_t tcp_port;
       caddr_t member_name[0];
   };
   ```

   The port numbers in this structure must be in network representation (Big Endian).

   We now give a code snippet of how a REGISTER_REQUEST message might be built:

```
char buf[1024];
/* allocate a block of memory to hold the message */

struct control_msghdr *cmh;
/* common header pointer */

struct register_msgdata *rdata;
/*register data area pointer */

char *tmp;

bzero(buf, 1024);
/* initialize the block to all 0 */

cmh = (struct control_msghdr *)buf;
/* by casting, cmh points to the beginning of the memory block, and header
field values can then be directly assigned */

cmh->msg_type = REGISTER_REQUEST;
/* message type */

tmp = (char *)cmh->msgdata;

rdata = (struct register_msgdata *)tmp;
/* rdata points to the data area */

rdata->udp_port = htons(7777);
/* udp port */

rdata->tcp_port = htons(8888);
/* tcp port */

strcpy((char *)rdata->member_name, "acc1");
/* member name */

cmh->msg_len = sizeof(struct control_msghdr) +
               sizeof(struct register_msgdata) +
               strlen("acc1");
/* message length */

write(socket_fd, buf, cmh->msg_len);
/* send the message */
```

When the server receives this message, it will decide whether to accept the client or not and reply with one of the following two messages.

2. REGISTER_SUCC

   This message is sent from server to the client as a successful response to the previous message. The `member_id` field will contain the id that the server generated for this client. Every client will receive a different id, which will be used for all the later control messages sent from client to server. There is no extra data in this message.

3. REGISTER_FAIL

This message is sent from the server to the client as a failed response to the message REGISTER_REQUEST. The `member_id` field will not contain valid information. The `msgdata` area will contain an ASCII string that explains the reason for failure. For example, it could be that the memeber name a client wants to use has already been used by others. The client may choose to print this info out.

4. ROOM_LIST_REQUEST

   This message is sent from client to server to request a list of current room names in the session. There should be no extra data in this message.

5. ROOM_LIST_SUCC

   Succeeded request reply to ROOM_LIST_REQUEST from server to client. The data area contains an ASCII string list of the room names and number of memebers in each of the room. The list is in the following format:

   [room1 (n1)] [room2 (n2)] ...

6. ROOM_LIST_FAIL

   Failed request reply to ROOM_LIST_REQUEST from server to client. The data area contains an ASCII string of the reason for failure.

7. MEMBER_LIST_REQUEST

   This message is sent from client to server to request a list of current members in a particular room. The `msgdata` must include the room's name.

8. MEMBER_LIST_SUCC

   Succeeded request reply to MEMBER_LIST_REQUEST from server to client. The data area contains an ASCII string list of the member names in the requested room. It has the following format:

   (memeber1) (member2) ...

9. MEMBER_LIST_FAIL

   Failed request reply to MEMBER_LIST_REQUEST from server to client. The data area contains an ASCII string of the reason for failure.

10. SWITCH_ROOM_REQUEST

    This message is sent from client to server to switch to a room. The `msgdata` must include the room's name.

11. SWITCH_ROOM_SUCC

    Succeeded request reply to SWITCH_ROOM_REQUEST from server to client. There is no additional message data in this message.

12. SWITCH_ROOM_FAIL

    Failed request reply to SWITCH_ROOM_REQUEST from server to client. The data area contains an ASCII string of the reason for failure.

13. CREATE_ROOM_REQUEST

    This message is sent from client to server to create a room. The `msgdata` must include the room's name.

14. CREATE_ROOM_SUCC

    Succeeded request reply to CREATE_ROOM_REQUEST from server to client. There is no additional message data in this message.

15. CREATE_ROOM_FAIL

    Failed request reply to CREATE_ROOM_REQUEST from server to client. The data area contains an ASCII string of the reason for failure.

16. MEMBER_INFO_REQUEST

    This message is sent from client to server to request a particular member's private TCP server information. The `msgdata` must include the member's name.

17. MEMBER_INFO_SUCC

    Succeeded request reply to MEMBER_INFO_REQUEST from server to client. The data area contains the requested member info. Member info is defined by:

    ```
    struct member_info {
        struct in_addr sin_addr;
        u_int16_t sin_port;
    };
    ```

    The IP address value and port number in this structure must be in network representation.

18. MEMBER_INFO_FAIL

    Failed request reply to MEMBER_INFO_REQUEST from server to client. The data area contains an ASCII string of the reason for failure.

19. QUIT_REQUEST

    Client sends this message to server to leave the session. No additional data is needed.

UDP-based public chat message format

Public chat messages are sent from a client to the server. The server then relays messages to all the clients in the sender's room. The format for each message is defined as follows:

```
struct chat_msghdr {
    union {
        char member_name[MAX_MEMBER_NAME_LEN];
        u_int16_t member_id;
    }sender;
    u_int16_t msg_len;
    caddr_t msgdata[0];
};
```

When a client sends a chat message to the server, it must include its member_id in the union field, and the `msgdata` field would contain the actual message. `msg_len` is the total length (in bytes) of the message (header and the data) The chat message relayed by the server to the clients would have the union field intialized to the member name of the original sender, so that when a client receives this chat message, it will know where it is from.

TCP-based private chat message format

A private chat message is sent from a client directly to another client. The message format is also defined by `struct chat_msghdr`, however, in this case, the sender puts its own member name instead of its member_id in the union field.

User interface

The focus of this assignment is on network programming and not on fancy user interface design. We are only interested in a minimal design of an interface that does a reasonable job. You are, however, free for embolish it once you have got it to work. The requirements for the chat client are as follows:

- To start the client must execute a single binary, `chatclient`, along with a set of command line arguments. However, within `chatclient` it may choose to start other binaries. The executable `chatclient` must support at least the following command line arguements:

  `-h` host name of the chat server

  `-t` TCP port of the chat server

  `-u` UDP port of the chat server

  `-n` member name the client chooses to use

- The chat client must employ the following windows:

  1. A user input window. This window will display a prompt such as ">" similar to a UNIX shell. The user can either type in a control command or a chat message. A line starts with "!" will be interpretated as a command, and all other input will be considered as chat messages.
  2. A public chat message display window. This window will display chat messages received from the chat server (including the messages sent by this user). For each message displayed, the sender's member name also needs to be shown along with the message. A simple way of generating a second window from inside a C program is to `exec` and `xterm` using the `-e` option. For example, `xterm -e mybinary` launches a new xterm and starts executing `mybinary` inside it. Do `man xterm` for more details on this. You are free to choose other techniques for this purpose, for example, X lib, Motif, Tcl/Tk, etc, but bear in mind a minimal set up would suffice the requirement.
  3. Displaying private chat messages. For this, you also have several choices. The first choice, which we strongly recommend, is similar to a zephyr message window. When a client receives a private message, a window with the message is popped up and can then be clicked away. In our reference implementation, we use `xmessage`, which is a convenient tool to display arbitrary ASCII strings or files, in a seperate window with buttons. The window can then be killed by clicking on the buttons. Another choice would be to display the private message in the window that is used to display public chat messages, but if you choose this way, you must have a way to show the private messages differently, e.g., prefix it with "private". The other choice would be to start yet another window just for the purpose of displaying private messages. Showing private messages in the control window will not receive full credit.

- The control commands that a user can type:

  – !r This will cause the chat client to retrieve the list of rooms in the session.

- !c <room_name> This will cause the chat client try to create a room with <room_name> on the server.
- !m <room_name> This will cause the client to retrieve the list of member names in <room_name>.
- !s <room_name> This will cause the client to try to switch to <room_name>.
- !p <member_name> This command is followed by one or more lines of private chat messages. The message is terminated by a "." on separate line. When the client sees this command, it will first issue a MEMBER_INFO_REQUEST to retrieve the intended member's information from the server. When this succeeds, the client then makes a TCP connection to the intended receiver and sends the chat message over.
- !q
  This will cause the client to quit from the session. A QUIT_REQUEST message will be sent to the server and no corresponding reply message is sent from the server to the client.

Users should be given warnings on inputing non-conformant commands. White spaces (space, carriage return, tab, etc.) and non displayable characters are not allowed in <room_name> and <member_name>.

- Chat message that a user types: Anything that does not start with a "!" is a chat message. A chat message can contain arbitrary ASCII charaters and the message may contain multiple lines. In the control window, a user terminates a chat message with "." at the beginning of a seperate line. The "." is not part of the message.
- Dislaying private/public messages should not interfere with the user input window and should not block the main process. For example, if `xmessage` was used to show a private message, the user should not be required to kill the message window before he/she can continue typing in the input window. However, when the chatclient quits (the user types !q), all associating windows to chatclient must be destroyed and the resources allocated to them (e.g., TCP, UDP socket) must be released.
- You have the full freedom to add any additonal cool features to your client implementation.

Snapshot of a chat session

We present a snapshot of a chat session below to give you an idea how a working system may look like. The chat server is running on machine bass.cmcl.cs.cmu.edu and waiting on TCP port 4000 and UDP port 4000.

```
[jungao@chum]$ ./chatclient -h bass.cmcl.cs.cmu.edu  -t 4000 -u 4000 -n wpo
[wpo]> !r
[golf (0)] [basketball (0)] [soccer (0)] [hockey (0)]
[wpo] !s golf
switch to room: golf
[wpo]> !r
[golf (1)] [basketball (0)] [soccer (0)] [hockey (0)]
```

A user starts chatclient on machine chum with nickname wpo. He finds there are four rooms in the session (apparently whoever created the rooms is a crazy sports fan) and he happily switches to his favorite room.

Later another user starts chatclient on machine char with nickname mic.

```
[jungao@char]$ ./chatclient -h bass -t 4000 -u 4000 -n mic
[mic]> !s golf
switch to room:golf
[mic]> !r
[golf (2)] [basketball (0)] [soccer (0)] [hockey (0)]
[mic]> !m golf
members:(wpo) (mic)
```

Everybody likes golf these days. Then on machine char,

```
[mic]> did you see the masters last weekend?
.
```

On both mic and wpo's chat message displaying window:

```
mic::
did you see the masters last weekend?
```

In response,

```
[wpo]> yeah, vijay won.
Tiger made a charge the last two days
but it was just vijay's day
.
```

This message is shown in both mic and wpo's chat message displaying window:

```
mic::
did you see the masters last weekend?
wpo::
yeah, vijay won.
Tiger made a charge the last two days
but it was just vijay's day
```

Now wpo feels the desire to share his dark secret with his new golf friend:

```
[wpo]> !p mic
i am taking an Intro to golf class this semester,
it's a secret and my advisor does not know :)
.
```

This message is popped up on mic's screen. wpo then heard a knock on his office door, oops, guess what, his advisor stopped by checking his progress on the Project Folg. wpo calmly quits the session (instead of hitting Ctrl-C) and switches to a console that has about 29,578 lines of Project Folg code.

```
[wpo]> !q
```

mic then sadly finds out that his friend is gone:

```
[mic]> !m golf
members:(mic)
```

Implementation Stages

Since this is a rather large project, we recommend you to use the following guideline to ensure progress. The strategy of developing everything all at once and debugging in the end will most likely cause you more sleepless nights. Based on the structure of the system, we divide the project into three stages. Before you move on to a next stage, you want to make sure that the code you write for the current stage is fully tested and free of bugs. However, Stage 2 and 3 are not completely built on top of Stage 1, which means the implementation of later stages may cause you to do minor fixes in an earlier stage.

Stage 1: Implement the basic control protocol and the user input window. This includes the implemenation of a TCP client to support all the messages except the three messages related to memeber info request. You probably want to start by supporting the REGISTER_REQUEST message. The important thing is to have the first message type supported, the rest of the message types in the protocol can be done similarly. By the end of this stage, when `chatclient` starts, it joins the chat session and displays a control window, which will then allow a user to type in various control commands to request room info, member info, to create new rooms, to switch to a room and to quit. You may defer parts of full protocol implementation until later stages. Depending on the scheme you choose to design the system, some of the information you may not know at this stage, e.g., the UDP port that is required in the REGISTER_REQUEST message. You may choose to put an arbitrary port number in your REGISTER_REQUEST message for the sake of implementing this stage and later on fix it with the right port number when later stages are finished.

Stage 2: Implement the public chat messages sending, receiving and displaying. The things involved in this stage are: building a UDP client to send chat messages to the chat server; building a UDP server to receive chat messages; enhancement to the control window interface to take chat messages; mechanism to support a separate public message display window.

Stage 3: Implement the private chat messages sending, receiving and displaying. The control protocol implemented at Stage 1 needs to be enhanced to incorporate the MEMBER_INFO_REQUEST related messages. A TCP server needs to be written for receiving private messages. Some window mechanism, e.g., using `xmessage`, should be deployed to display private chat messages.

Simplifying Assumptions

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

- The duration of receiving each private message is short enough that a simple iterative TCP 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.

- A user may issue one control command at a time. The user's second command will block until the first command is processed. Since the processing of one command involves messages exchange between the chat client and the chat server over the network, the block time for the second command may be noticeable.

- 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) a private chat message is received — is considered small, so 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.

Resources

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. We will also make the chat server code available to you, which can be used as a starting point in designing the chat client.

Here are some sources of information that you could use:

- The class Web page will have pointers to tutorials on network programming.
- Man pages are your friends. They provide you detailed function specifications, such as data type of arguments, return values. Man pages also tell you the proper header files that you need to include when use a funtion. `man -k <keyword>` is particularly useful for finding information related to `<keyword>`.
- The most valueable book on network programming is by the renowned Unix expert, W. Richard Stevens (passed away last September), *Unix Network Programming : Networking APIs: Sockets and XTI (Volume 1)*. Every serious system programmer should have this book (and Stevens's other books on Unix, TCP/IP) on his/her shelf.

Useful system calls: We list below system calls that you might expect to use (Note this is not a complete list in any way).

- *Basic socket related:* socket, bind, connect, listen, accept, read, write, recvfrom, sendto, recv, send, close
- *Get protocol address associated with a socket:* getsockname, getpeername
- *Byte ordering:* ntohs, ntohl, htons, htonl
- *Byte manipulation:* bzero, bcopy
- *Name resolution related:* gethostbyname, gethostbyaddr, uname
- *Address conversion:* inet_aton, inet_addr, inet_ntoa
- *I/O multiplexing:* select, FD_ZERO, FD_ISSET, FD_SET
- *Fork and related:* fork, execlp
- *Named pipe related:* mknod

Debugging

We distribute the chat server source code to you. The purpose of this is two-folded:

1. By examing the source code, you can get a feeling of how the server is designed and implemented, and hopefully this will help you designing your chat client and writing the code.

2. You can compile the source code to get a local version of the chatserver binary. You can debug your client code by having your client interact with your local copy of chatserver. The chatserver is executed like this:

```
./chatserver -t <tcp port> -u <udp port> [-f <log file name> -s <sweep
interval(mins) -r <room file name>]
```

The parameters in square brackets are optional. The server TCP port and UDP port must be specified. Since on one machine, a given port number for each protocol can be bound to only one process, an error will be returned when multiple processes try to bind to the same port. In the server implementation, if the command line specified port has been used, we let the kernel dynamically choose a port for us from the pool of available ports. When -f is used, the server will log detailed information on each message exchanged between the server and the client. For each registered client, the server has to maintain states for this client, however, a client may die before sending an explicit QUIT_REQUEST message. In the server implementation, if -s is used, we periodically check the status of each client, if a client has been quiet (didn't send any messages to the server) for <sweep interval(mins)>, the server will remove the client from the session. If -s is not used, the server will maintain a registered client's information throughout the lifetime of the server. As aside, if -s is used, the server will remove the rooms that do not have any members for <sweep interval(mins)>. Option -r will specify a room config file which contains room names. This will allow the server to create rooms according to the file before any client joins. No rooms will be available at the server start time if this option is not set.

The chatserver limits the maximum number of rooms and maximum number of members in the chat session by using pre-defined constants. To terminate chatserver, hit Ctrl-C. You should not run chatserver during the time that you are not working on your assignment.

Evaluation

We will evaluate your assignment based on a mini-milestone check and the final demonstration.

Around one week before the assignment is due, we will do a mini-milestone check of your progress. The exact date of the check will be announced soon on the bboard. For the milestone check, we basically want to see whether a your client implementation can successfully register with the main server that we are running. Each group must run their chatclient and with the first member's andrew-id as the nickname parameter and try to connect to the main server. There will be no handin for this check, and we will use our log file for grading.

You must hand in your code before the deadline for the demo. 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 May 2nd and 3rd. Please watch the webpage and newsgroup for updates.

The assignment 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 room and send and receive chat messages (20%). Handling all other control messages in a graceful manner earns you another 15%. Support for the private chat server and corresponding control messages accounts for a further 15%.
- Design (20%). Elegance, robustness of design and handling of error conditions and special cases will be considered.
- Documentation (15%) You should develop your code in a good style. You should, for example, 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, properly indent your code and use empty lines to make your code more readable.
- Mini-milestone check (5%) You must be able to successfully register with the main chatserver to earn the credits.
- 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.

Logistics

As usual, you may work in a group of up to two people.

The files for this assignment can be retrieved from:

```
/afs/cs.cmu.edu/academic/class/15213-s00/L5/L5.tar
```

Once the tar file is copied to your own directory, run command `tar xvf L5.tar`. You will get the following files:

- README: contains description of the files in this directory
- Makefile: a sample makefile to make the chatserver
- defs.h: the common header file that contains constants and data types you must use in your client code. You must include this file in your source code. You must not modify this file.
- server.h: header file for the chatserver
- server_util.c: utility functions for chatserver
- server_main.c: chatserver main function
- room.cfg: a sample room configuration file

Type `make` will generate chatserver binary. Modify the Makefile accordingly so that it can also generate your chatclient binary.

Main chat server

A chat server will be running on machine `bass.cmcl.cs.cmu.edu` with TCP port 4000 and UDP port 4000, for your amusement. The main chat server may not be running all the time, but your code development and testing should not be affected as you have a local copy of the chatserver.

Hand In

You must hand in all the files relevant to your client implementation by the deadline, irrespective of when your actual demo is. These files include the source files, header files and a Makefile. In addition, you must hand in a text file called *Read.me* to explain what parts of the assignment you have finished and what parts of the assignment you have not finished. You will compile and run your chatclient from your handin directory during the demonstration.

You must take the following steps for your handin:

- Copy all the above mentioned files to an empty directory
- Run the command

  `tar cvf chatclient.tar .`
- Hand in your `chatclient.tar` using the command `make handin NAME=yourname`, where `yourname` is the Andrew ID of one of the team members. You can submit updated versions with the command `make handin NAME=yourname VERSION=XX`, where `XX` is the version number, i.e., 2, 3, ... Only your highest numbered version submitted before the deadline will be graded.

Extra credit

You can get extra credits on addressing the following points correctly in your client implementation.

- As stated earlier, on the server side, we would periodically flush quiet clients. This may not be desirable since some client may be ”receive only”, i.e., it may want to switch to a room and stay there monitoring all the messages without sending a message. This kind of client will be purged in our current specification. To prevent this, we defined another message type MEMBER_KEEP_ALIVE. By cleverly designing the client, you can implement a ”receive only” client.
- TCP is a byte stream protocol, which means one contiguous data block from a sender to a receiver may be arbitrarily broken up into several network packets. The correct implementation of a TCP read should sit in a loop. For small packets such as the ones used in the control message protocol, most likely the data will be sent all in one IP packet, so one read is probably sufficient. However, in the case of a private chat message, it could potentially be very long and might have to be broken up into multiple IP packets. We are looking for a more robust implementation of private TCP server. (A simple TCP read implementation will get full credit though.)