# CSCI 4273/5273: Network Systems
# Fall 2014
# Programming Assignment 1

*Due date: 09/26/2014*

**Goal:** The goal of this programming assignment is to design and implement a chat system that allows a group of users to set up private chat sessions and chat. In the process, you will learn network application programming using UDP and TCP sockets. This assignment requires familiarity with Unix system calls such as `fork()`, `execl()`, and `select()`.

**Grade:** 10% of your final grade is allocated for this assignment.

A chat system consists of a central coordinator that manages one or more chat sessions, each identified by a unique chat session name. Users can participate in a chat system as chat clients. A chat client may start a new chat session by contacting the chat coordinator at any time and specifying a unique (chat session) name. Chat clients can participate in a specific chat session by first joining that session, and then posting messages in the session and receiving messages posted in the session. Finally, a chat client may quit a chat session at any time.

Chat system consists of two separate programs: chat server and chat client. A chat server resides at a well-known network address and consists of two parts: chat coordinator and session server. Chat coordinator uses UDP for communication. For each chat session, it sets up a session server (one session sever per chat session as a separate process) and maintains a chat session directory. A session server uses TCP for communication with the chat clients. It facilitates posting and retrieving of messages with in its chat session.

Finally, a chat client runs on the user machine, and allows the user to start a chat session, participate in a chat session, and quit a chat session.

Note that the chat client and chat server are independent programs. In particular, your chat client should work (interoperate) with the chat server of your friend even though your friend may have used a different OS and a different programming language. Similarly, your friend's chat client should interoperate with your chat server. To enable this interoperability, you must implement the client and server specifications accurately.

**Chat Coordinator Specification:** A chat coordinator exports the following operations:

1. `Start` *s_name* : Start a new chat session; *s_name* is a string of up to eight characters (English alphabets).

2. `Find` *s_name* : Find the address of the session server of an existing chat session named *s_name*.

3. `Terminate` *s_name* : Terminate an existing chat session named *s_name*.

A chat coordinator starts by printing its UDP port number on the standard output, and then waits for the commands arriving at this port. On receiving a `Start` command (from a chat client), the chat coordinator first checks that there isn't an on-going chat session with the same session name. If there is one, the chat coordinator returns an error message (-1) to the client. Otherwise, it creates and binds a TCP socket, and creates a new process (session server) to manage the new chat session. It then returns a message to the client with the address of the session server (port number of the TCP socket). If there is any error, e.g. in creating a new process or a new TCP socket, the chat coordinator returns -1.

Chat coordinator also maintains a table of all on-going session names along with their session server addresses. On receiving a `Find` command (from a chat client), the chat coordinator first checks that there is an on-going chat session with the same session name. If there isn't one, the chat coordinator returns an error message (-1) to the client. Otherwise, it returns the session server address (port number) of the corresponding chat session.

Finally, on receiving a `Terminate` command (from a session server), the chat coordinator terminates the corresponding chat session. *Note that a chat session can be terminated only by its session server.*

**Session Server Specification:** A session server exports the following operations:

1. `Submit` *message_length message* : Submit a message to the chat session; *message* is a sequence of characters terminated by <CR>, *message_length* is the length (number of bytes) of the message. Maximum length of a message is 80 characters including <CR>.

2. `GetNext` : Get the next message that is not yet read by this client from the chat session.

3. `GetAll` : Get all messages that are not yet read by this client.

4. `Leave` : Leave the chat session.

A session server waits for new chat clients to connect by using the `listen()` command on the TCP socket created by the chat coordinator. For each client that connects, the session server creates a separate TCP socket (automatically created by the `accept()` command). It uses this socket to communicate with that client. It also maintains a chat history that is comprised of messages submitted to the chat session as well as the last message read by each chat client participating in the session. On receiving a `Submit` command, the session server stores the message in its chat history. On receiving a `GetNext` command, the session server returns the chat message following the last chat message read by that client and updates the identity of the last message read by this client. The format of the return message is *message_length message*. If there is no new chat message to be read, -1 is returned.

On receiving a `GetAll` command, the session server returns a sequence of messages to the client. The first return message is the number of new chat messages that will be sent one by one next. This first return message is followed by a sequence of return messages, each containing a new chat message along with the message length (as in the return message of the `GetNext` command). Finally, on receiving the `Leave` command, the session server closes the TCP connection with that client.

The session server sends the `Terminate` command to the chat coordinator and terminates the chat session if the session remains idle for one minute, i.e. the session server does not receive any commands from its clients for one minute.

**Chat Client Specification:** A chat client exports the following operations to a user:

1. `Start` *s_name* : *s_name* is character string of up to eight characters (English alphabets).
2. `Join` *s_name*.
3. `Submit` *message*.
4. `GetNext`.
5. `GetAll`.
6. `Leave`.
7. `Exit`.

A chat client starts with two command-line parameters: hostname/IP address of the chat server and port number of the chat server. For each command invoked by a user, it prompts the user for appropriate parameters. A user cannot participate in a chat session until he/she has joined that chat session. For this assignment, assume that the chat client allows a user to participate in a single chat session at any time. So, while a user may start and/or join several chat sessions, `Submit`, `GetNext`, `GetAll` and `Leave` commands always refer to the latest chat session that the user joined.

On receiving a `Start` command, the chat client sends an appropriate message (`Start`) to the chat coordinator. If there is any error, it prints an appropriate error message. Otherwise, it prints “A new chat session *s_name* has been created and you have joined this session”. Note that the user is automatically joined to the chat session that he/she starts. On receiving a `Join` command, the chat client first sends a `Find` message to the chat coordinator and then connects with the session server. On successful completion, it prints “You have joined the chat session *s_name*”. Again, in case of any error, an appropriate error message is printed.

On receiving a `Submit` command, the chat client sends an appropriate `Submit` message to the session server. On receiving a `GetNext` command, it sends an appropriate `GetNext` message to the session server and then prints the chat message received or “No new message in the chat session”. On receiving a `GetAll` command, it sends an appropriate `GetAll` message to the session server and then prints all the chat messages received one by one or “No new messages in the chat session”. On receiving a `Leave` command, it sends an appropriate `Leave` message to the session server and prints “You have left the chat session *s_name*”. In case of any error in any of these commands, it prints an appropriate error message. Finally, on receiving an `Exit` command, the chat client terminates the client program.

Read the socket handout posted on Moodle and carefully go over the `echo` program. Make sure that you understand this program thoroughly. In addition, a C program implementing a simple version of Unix `talk` utility using TCP is given in the textbook (Section 1.4.2, Page 40). Read and understand this program before starting this programming assignment.

**Assignment submission**

1. Submission deadline is Friday, September 26, midnight. No late submissions will be allowed, unless there is a valid excuse.
2. Submit a single zip file via the submission link on Moodle. Your zip file must contain chat_client.cc, chat_server.cc, chat_coordinator.cc, Makefile and README. Include any other files that are needed for compiling and running your program.
3. DO NOT include any object files in your submission.
4. In the README file, provide the following information: Your name; instructions on how to compile and run your program; current status of your program: whether it compiles or not, known bugs/limitations/unusual features, what parts of the program work, etc.; any other information that will be useful in grading your program.

Here are a few guidelines for doing well in this course:

1. Start working on homework and programming assignments as soon as they are handed out.
2. Don't code until you understand what you are doing. Design, design, design first.
3. If you don't understand something, contact the TA or the instructor or post a message on the discussion forum.
4. Enjoy.