Operating Systems

CS 450 — Fall 2013

(large) Project 1

Adapted from Prof Kooshesh, given on Thursday, 26 September 2013

Due Dates

Phase 1: Monday, 28 Oct at 12:00 noon
Completed project: Monday, 11 Nov at 12:00 noon

Introduction

For this project, you are to write a number of related programs that cooperate to implement a multi-threaded chat application that uses Berkeley's TCP/IP sockets as a mean of communication between a chat-server and chat-clients. In what follows, we describe the details of the project in successive steps. Initially, we define the terms that we will use to describe the project. In the second part of this write-up, we will provide the details of the project. Finally, during the lecture, a number of implementation suggestions will be given.

Here is the details of the specifications for this project. For each of the two phases, you need to turn in the following.

1. A make file that supports at least two targets: server and client to build the server and the client modules, respectively.
2. A README file that explains how to run your server and your client. This includes:
   (a) the arguments that each module requires.
   (b) a list of features that do not work.
   (c) a list of features that work and a detailed description for testing those features.

For phase 2, in the README file, you also should indicate whether you have used the bounded/buffer and the reader/writer models and if so, where.

Chat-Server

A *chat-server* is a non-interactive C++ program application that facilitates the communication between *chat-clients* (see below.) There will be only one instance of the chat-server available to the clients at any given time.

Chat Server-Node

A *chat server-node* is a computer system that runs the chat-server application.

Chat-Client

A *chat-client*, which is interactive by nature, consists of a number of related C++ program applications that send messages to, and receive messages from, the chat-serve. Each chat-client has a name (client-id) and the name of chat-clients are unique within this application. The IP of the chat server-node and the port on which it listens for connections is known to all chat-clients.

Chat Client-Node

A *chat client-node* is a computer system that runs one or more chat-client applications.

Chat-Message

A *chat-message* is a string of characters, possibly including the new line characters, that a chat-client sends to a chat-server or a chat-server sends to one or more chat-clients.

Chat-Room

At each given time during the life cycle of this application, the chat-server makes one or more chat-rooms available to chat-clients. One of these chat-rooms, which we call the default chat-room, is always available as long as the chat-server is active. All other chat-rooms get created at the request of chat-clients. A non-default chat-room that has no members (whose last member leaves) gets removed from the chat-server's list of active chat-rooms.

Initially, when a new client joins the system, the server adds it to the server's default chat-room (so, at that time, the client's default chat-room is set to point to that of the server's default chat-room.) At any time, the clients can change their default chat-room to one that they select from server's existing chat-rooms.

A client, through the commands that it sends to the server, can become a member of multiple chat-rooms. In that case, all messages that get posted to any of these chat-rooms get sent to that client. On the other hand, chat messages that a client posts get sent to members of his/her default chat-room, which includes himself/herself.

Chat-Server Life Cycle

Initially, the chat-server is launched on the chat server-node. Although it is non-interactive, for the purpose of debugging, it should write detailed diagnostic messages to its standard output. After initialization, the chat-server listens on a port-number that is known to the chat-clients. As it accepts new connections from chat-clients, the server creates a new thread to service that client and immediately positions itself to listen for new connections again. The thread that the server creates provides an operating environment for the particular needs of that client. A client either terminates by notifying the server and then closing its *processes* (see below) or by the server instructing it to do so.

The chat-server terminates when a chat-administrator (a chat-client) sends a *shut* (see below) message to it. In that case, the chat-server stops accepting new connections and will ask all the existing clients to exit. Once that completes, the server terminates itself.

Chat-Client Life Cycle

After it is fully operational, a chat-client will consists of three disjointed processes that simultaneously run on the chat client-node. Feel free use a different model, such as three threads, a command thread with two child processes, or some variant thereof.

1. A *command process*. This process starts first and through negotiations with the server, chooses a name for the client (client-id.) After that point, the client will use its client-id to establish the following two services.

2. A *writer-process*. After having obtained a client-id, the user launches a second processes (using a different shell session) and uses it to establish a writer process — an interactive session that will be used to send chat-messages to the server to be posted in client's default chat-room.

3. Finally, after a writer process is active and operational, the user, using a new shell session, launches a *reader-process* — a process that receives chat-messages from the server (from the chat-rooms to which the client subscribes; initially, the server's default chat-room) and displays them on the screen.

Chat-clients terminate by terminating their processes in the same order that they are created. As mentioned above, the server could also ask a chat-client to terminate its connections.

Chat-Server Details

Initially, the server creates the default chat-room and waits for client socket connections. The server should create 4 threads to serve the default chat-room. After a message gets posted to this chat-room, one of these threads sends it to the clients that have subscribed to it. Furthermore, each new chat-room that gets created on behalf of the clients will get 4 additional threads to serve it.

When a connection is established between a client and the server, the client socket is expected to write a command, a four-byte entity, first. The server reads the client's command and takes an appropriate action. The following is a list of commands available. Note that if a command (such as *bye*) has fewer than four bytes, it should be, on the client side, padded, possibly with null-characters.

The colons that appear after the commands are not part of the command.

join: Client requests to join a chat-server. This is the first message that the command-process of the client sends after it connects to the server.

bye: The client is about to terminate (bye.)

crea: Create a new chat-room. The name of the chat-room and its description will follow.

subs: Client requests to be subscribed to an already existing chat-room. The name of the desired chat-room will follow.

unsu: Unsubscribe the client from a chat-room. The name of the chat-room will follow.

shut: Only the administrator-client sends this command. Shutdown the server.

defa: Change the client's default chat-room. The name of the desired chat-room will follow.

lscr: list chat-rooms. The server should provide a list of available chat-room to the client.

lssu: list chat-rooms that the client has subscribed to.

read: This command is issued by the client's read-process. The incoming connection should be used by the chat-server for writing messages that are directed to this client.

writ: This command is issued by the client's write-process. The incoming connection should be used by the chat-server to read client's chat-messages.

The last two commands are sent to the server by the client's read-process and write-process, respectively. All other commands are sent to the server through the client's command-process. Server's response to these commands is either a 1 or a 0 (except for *lscr* and *lssu* which will generate a list of chat-room names.) A 1 indicates success — the server was able to carry out the command that the client issued. A 0 indicates a failure. A failure message has to be interpreted by the client using the context in which it is received. For example, if a client attempts to subscribe to a chat-room by issuing *subs* followed by the name of the desired chat-room and gets back a failure message, it would mean that the server does not have a chat-room with that name. On the other hand, if the client sends crea followed by the name of a new chat-room to be created and receives a failure, it means that a chat-room with that name already exists.

The only other command that a server sends to the clients is a shutdown message. This message is sent to the client through the client's read-process. This message contains the 10 characters in the string: *shutdown* including the asterisks.

Chat-Client Details

As indicated above, a client may consist of three disjointed processes. This design decision is intended to simplify the task of writing this project specifically in light of the fact that we are writing a text-based chat system: whether it does or not depends on your point of view: feel free to change it. What follows assumes this structure, but does not require it. Within the context of the chat-server, a chat-client is represented by three threads, each connected to its corresponding client process. These threads share a data-structure (an instance of a user-defined data-type) to coordinate with one another to read messages from and write messages to chat-rooms.

The process of creating a new client begins by the user, at a shell prompt, launching the client's command-process and providing it with the IP of the server and port-number where it listens for connections. After having started, the command-process sends the *join* command to the server followed by a string of characters to be used by the server as the client-id of the server. The command-process should prompt the user for this name, read it, and pass it along to the server. The command-process should repeat these steps until the chat-server returns a success message.

After having established a client-id, the user launches the read- and the write-processes (see below.) Subsequent to that, this process will be used to create new chat-rooms, to change the client's default chat-room, etc.

After having established a client-id, the user launches the client's read-process at the shell-prompt. This process should require the IP and port-number of the server as well as the client-id that the client just established. This process sends a *read* command to the server and identifies itself using the client-id. After having received a success status, it just waits for chat-messages from the server.

Finally, the client launches the writer-process from the command prompt with arguments similar to those used by the read-process at the shell prompt. Once the server comes back with a success status, the client uses this shell session to write chat-message to its default chat-room. To write to a chat-room that this client has subscribed to or has created (through the use of the command-process), the user has to use the command-process to change its default chat-room first and then use this process to post a message to that chat-room.