University of Arkansas

Department of Computer Science and Computer Engineering

CSCE 3613 – Operating Systems — Spring 2019
Due: April 17 (Wednesday) 3:05PM — Homework #6
Required for Honors students, Optional for other students

DISTRIBUTED CHAT SERVICE

The objective of this assignment is to provide some hands-on experience on **client-server programming** using **sockets** and **TCP/IP protocol suite** in a **distributed environment**. We consider a simple distributed chat application consisting of a Chat Server and multiple clients. Chat Server (CS) will manage the chat rooms and the clients will be able to 'join' or 'exit' specific chat rooms by sending messages to CS. When a client wants to send a message to a specific chat room, it will contact the Chat Server, which in turn will broadcast the message to all the clients in that room. You will write the client and chat server programs. In this assignment, you will implement the system using **TCP** as the transport layer protocol.

Chat Server (**CS**): There will be only one chat server process in the system. It will be identified by an IP address (CHATSERV-IP-ADDRESS) and the port number (CHATSERV-PORT-NUMBER), which can be assumed to be globally known to all the clients. CS will manage the chat rooms and keep track of the clients in each room. For simplicity, assume that there is a fixed number (i.e., three) of chat rooms. Upon the receipt of specific commands from clients, CS must take the appropriate actions:

- `Join <Chat Room Number> <handle>`: CS will update its records to indicate that a new client has joined the specific chat room with the given handle. Depending on your design, it may need to record the connection data for the involved client, such as the handle, IP address and Socket/Port information.
- `Exit <Chat Room Number>`: CS will update its records to reflect that the client is no longer in the specified chat room.
- `Msg <Chat Room Number> <Message text>`: CS should transmit the message text to all other clients that previously joined (and not yet exited) that chat room. **You must not use Socket Multicast/Broadcast**. Each client in the chat room must be contacted separately while forwarding the message. The handle of the message-sending client and the chat room number must be displayed in the consoles of other clients in that room, preceding the message text.

CS should react reasonably to unexpected message sequences. For example, it should refuse to distribute messages from the clients that have not yet joined the target chat room. Unrecognized commands should be rejected, and a client should not be able to join a given chat room under multiple handles. After processing each client request, the server must return an informative reply, which must be also displayed on the client's console (e.g., "Your message to Room 2 has been successfully distributed", "Unrecognized Command", etc.).

Client Programs: Each client program will wait for input from the user, forward the request to CS, and display incoming messages from CS. The user should be able to send any of the following commands, in any order:

- `Join <Chat Room Number> <handle>`: to join the specified chat room under the specified handle.
- `Exit <Chat Room Number>`: to leave the specified chat room.
- `Msg <Chat Room Number> <Message text>`: to send a message to the specified chat room.

To get full credit, the incoming messages should not be delayed while waiting for input from the user at the client side; they must be displayed as soon as they are received.

Provided that you comply with the rules above, the details of the interface between the client programs and the users are left unspecified. You can use a simple text-based interface, but it should be efficient and easy to use. Similarly, the format of socket messages between clients and the server is left to you; however, you should be able to justify your design decisions. When designing your system, keep in mind the advantages of creating new processes/threads for sub-tasks and/or concurrency, if applicable. You should identify any potential race conditions on shared variables (if any), and take the preventive measures.

Programming Languages, Socket Interfaces: For this assignment, please use C as the programming language. You can use any socket interface as long as you abide by the rules above. However, you are encouraged to use Berkeley Sockets since you are likely to find a large number of references. Berkeley Sockets are also available at Turing.

Choosing Computers and Port Numbers: Ideally, you should develop and test your program by running each client and the chat server on different machines. If you are unable to find multiple computers, you can create a unique process to represent the server and the clients on the same computer. You should be careful when choosing port numbers for your server process(es). All port numbers below 1024 are reserved, and the ports used by other services/students may not be available either. In addition, make sure that you 'close' every socket you use in your programs. If you abort a program, the socket may still hang around and the next time you bind a new socket to the same port you previously used (but never closed), you may get an error. When considering socket management, you can assume that servers/clients never crash during the execution.

Running Your Programs with a Specific Message Sequence: In this assignment, you have to run your programs with the following sequence of messages. In order to be able to follow the interactions, please allow 2-3 seconds after each initialization for socket link establishment phase.

1) Initialize Chat Server
2) Initialize Client 1
3) Initialize Client 2
4) Client 1 sends the command -join 1 mike
5) Client 2 sends the command -join 1 bob
6) Client 1 sends the command -msg 1 hi.. anyone here?
7) Client 2 sends the command -msg 1 hi mike!
8) Client 1 sends the command -join 2 mike
9) Client 1 sends the command -join 3 mike
10) Client 2 sends the command -join 2 bob
11) Client 2 sends the command -join 2 larry
12) Initialize Client 3
13) Client 1 sends the command -msg 2 no one here?
14) Client 3 sends the command -join 3 susan
15) Client 3 sends the command -msg 2 are you there, bob?
16) Client 3 sends the command -msg 3 are you there, bob?
17) Client 1 sends the command -msg 3 I think Bob is in Room 1.
18) Client 3 sends the command -exit 2
19) Client 3 sends the command -exit 3
20) Client 3 sends the command -join 1 susan
21) Client 3 sends the command -msg 1 what's up, bob?
22) Client 2 sends the command -msg 1 Working on CSCE 3613 project. What about you?
23) Client 3 sends the command -msg 1 same here...
24) Initialize Client 4
25) Client 4 sends the command -join 1 mary

26) Client 3 sends the command -join 3 mary
27) Client 1 sends the command -exit 1
28) Client 4 sends the command -msg 1 is it possible to see the list of users?
29) Client 4 sends the command -msg 3 who are the people in this room?

Submission: Please create a tar file using the name hw6_<your last name>.tar including the following items.

1) The source code of your programs (the server and client programs), Makefiles, and/or any other information necessary to compile and run your program if you are using additional facilities/libraries. *Your code should include meaningful comments and variable names*.
   Please note that the I will use Turing to compile and verify your code.
2) A brief report consisting of the following contents:
   - A write-up describing/explaining the major design decisions you made (for example, using multiple threads or processes on the server and/or client side, the format of socket messages, etc.).
   - The output of four clients for the message sequence above.
3) A README file specifies the names of the above files.

Resources: You are strongly encouraged to refer to socket programming links available at http://hthreads.csce.uark.edu/wiki/Howtos.

Suggestions: First of all, allow sufficient time for your assignment unless you have prior experience in socket programming. As the first step, you will have to study the basics of socket communication by reading Web tutorials and examining/writing simple client-server codes. Then you can incrementally develop your system, by adding multiple clients, enhancing the server, designing the user interface module for the clients, and so on.

Grading: The relative weights of grading components will be approximately as follows:

- Write-up and pseudo-code: 20 points
  The output of four clients for the message sequence are necessary for grading
- Correct Design and Implementation: 80 points
- **Total Points: 100**

Late Penalty: No late submission will be accepted unless you consult with the instructor in advance.