CS 39006: Networks Lab

Assignment 4: Implement a Peer-to-Peer Chat Application

Due Date: February 09, 2024

The objective of this assignment is to develop a simple peer-to-peer chat application. A peer-to-peer (P2P) application is an application primitive, where there are no dedicated central server nodes. Every node in the network can connect with each other and transfer data among themselves.

In this P2P chat application, we consider a close group of friends who want to chat with each other. The chat application uses TCP as the underlying transport layer protocol. For P2P implementation, every instance of the chat application runs a TCP server (we call this a peer-chat server) where the chat application listens for incoming connections.

The protocol details are as follows. The same process of the application also runs one or multiple TCP clients (based on the number of peers) to connect to the other users and transfer messages. For this assignment, assume that you can have a maximum of only 3 peers, so each will run one server and two clients max.

Design Requirements:

1. The entire chat application runs under a single process.
2. Every participating user maintains a `user_info` table that contains - (a) name of the friend, (b) IP address of the machine where the chat application is running, (c) port number at which the peer-chat server is running. *This table is static and shared a priori with all the participating users.* For your assignment, name your users user_1, user_2, and user_3, all your IP addresses will be 127.0.0.1 and use port no.s 50000, 50001, and 50002 for running the servers of the three peers respectively.
3. For the chat purpose, a user enters a message using the keyboard. The message format is as follows: `friendname/<msg>` where `friendname` is the name of the friend to whom the user wants to send a message, and `msg` is the corresponding message (a string of maximum length 250 characters). The msg will be sent to that friend only.
4. The communication mode is asynchronous, indicating that a user can enter the message anytime. The console should always be ready to accept a new message from the keyboard, unless the user is already typing another message.

Protocol Primitives:

1. As we mentioned earlier, every instance of the chat application runs a TCP server which binds itself to a port (see no. given above) and keeps on listening for the incoming connections.
2. Once the server receives a connection, it accepts the connection, reads the data from that connection, extracts the message, and displays it over the console. The message should be displayed with a starting text “Message from <friend name>: “
3. Once a user enters a message through the keyboard,

    a. The message is read from the `stdin`, the standard input file descriptor. The message entered is one line of alphanumeric characters, space, tab, punctuation marks etc. terminated by the \n character.
    b. The application checks whether a client connection to the corresponding server already exists. If a client connection does not exist, then a client connection is created to the corresponding server based on the lookup over `user_info` table. (You can use other data structures to keep track if a client connection exists or not).
    c. The message is sent over the client connection
4. If there is no activity over the client connection for a timeout duration (use 5 minutes), then the client socket is closed. The connection can be open again later if the user wishes to send another message to that friend later.
5. The chat application runs perpetually.

As all the above functionalities are executed over a single process, you need a way to maintain multiple file descriptors (one server socket, two client sockets to the other two peers, and the standard input (file descriptor 0)), and handle them in an iterative way. This is accomplished by the `select()` system call. The call will need to wait on the server socket (to receive connections/data from other users) and the standard input (to read data from the keyboard and send). The actions taken on each are already given above.

Submission Instructions:

You should write two C programs - peerserver.c (contains the server program) and peerclient.c (contains the client program). Keep these two files in a single compressed folder (zip or tar.gz) having the name <roll number>_Assignment4.zip or <roll number>_Assignment4.tar.gz. Upload this compressed folder at the Moodle course page by the deadline.