CS 39006: Networks Lab

Assignment 5: Implement a Peer-to-Peer Chat Application

Date: 8th February, 2018

Objective:

The objective of this assignment is to understand the functionalities of the select() system call through a network application - a Peer-to-Peer Chat Application.

Submission Instructions:

The code needs to be written in either C or C++ programming language. You need to prepare a report that will contain the followings.

1. **Documentation** of the code along with **compilation and running procedure**, **sample input** and **sample output**

Also include a **makefile** to compile your code. You need to submit the report, code (only C/C++ source) and Makefile in a single compressed (tar.gz) file. Rename the compressed file as `Assignment_1_Roll1_Roll2.tar.gz`, where `Roll1` and `Roll2` are the roll numbers of the two members in the group. Submit the compressed file through Moodle by the submission deadline. The submission deadline is: **February 15, 2018 02:00 PM.** Please note that this is a strict deadline and no extension will be granted.

Please note that your submission will be awarded zero marks without further consideration, if it is find to be copied. In such cases, all the submissions will be treated equally, without any discrimination to figure out who has copied from whom.

The objective of this assignment is to develop a peer-to-peer chat application. A peer-to-peer (P2P) application is an application primitive, where there is 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 as 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.

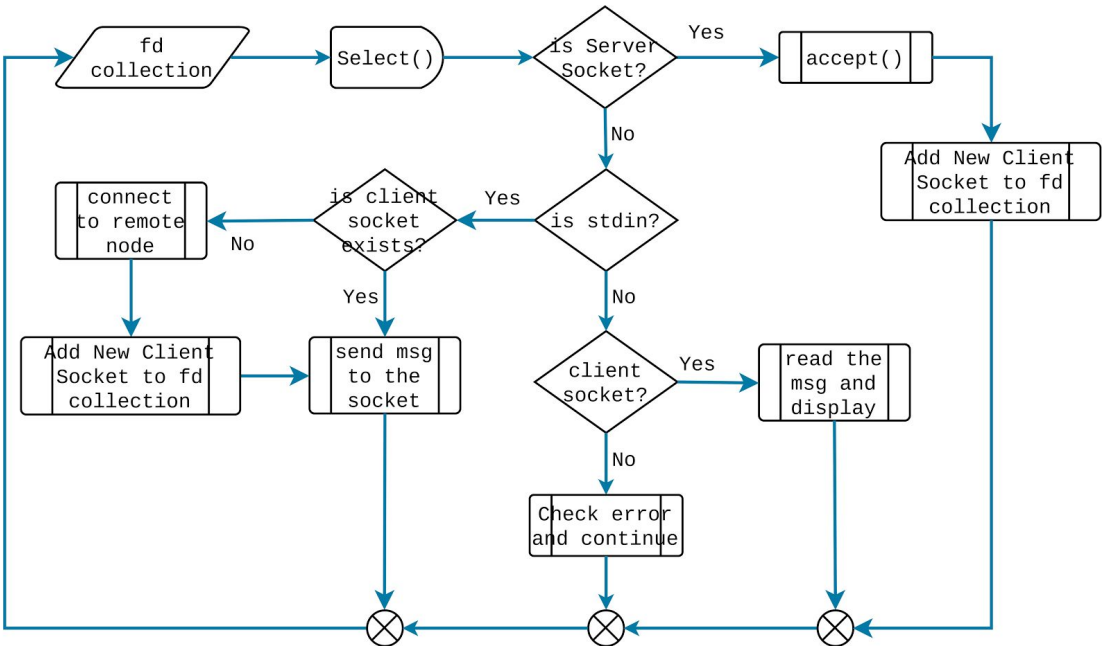
Design Requirements:

1. The entire chat application runs under a single process. So, the peer-chat server is an iterative server, and NOT a concurrent server.
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.*
3. For the chat purpose, a user enters 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 message, and `msg` is the corresponding message.
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 well known port and keep on listening for the incoming connections. There can be a maximum of 5 peers (users), so a maximum of 5 connections need to be supported.
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.
3. Once a user enters a message through keyboard,
    a. The message is read from the `stdin`, the standard input file descriptor.
    b. The application checks whether a client socket to the corresponding server already exists. If a client socket does not exist, then a client socket is created to the corresponding server based on the lookup over `user_info` table
    c. The message is sent over the client socket
4. If there is no activity over the client socket for a timeout duration, then the client socket is closed.

As all the above functionalities are executed over a single process, you need a way to maintain multiple file descriptors, and handle them in an iterative way. This is accomplished by the `select()` system call. A flow diagram of the entire process is given next.


fd collection
Select()
is Server Socket?
Yes
accept()
Add New Client Socket to fd collection
No
is stdin?
Yes
is client socket exists?
No
connect to remote node
Add New Client Socket to fd collection
Yes
send msg to the socket
No
client socket?
Yes
read the msg and display
No
Check error and continue