Project 2: A Simple Chat Application

Due: 4/8/2021

Project Objectives

1. Experiencing signal handling and thread programming
2. Mastering socket based networking application development
3. Getting familiar with application-layer network protocol design and development

Project Description

A chat application has two components: a chat server and chat clients. A chat server is an application program that maintains the user account information, and facilitate the chat functionality . A chat client is an application program that can send either broadcast or targeted messages to other clients.

In this project, you need to write two programs chat_server and chat_client, which correspond to the chat server and chat client, respectively. The BSD TCP sockets are used for communications between the client and server. The functionality of each of these programs is described below.

chat_server

usage: chat_server configration_file

The parameter configuration_file is a configuration file for the server program. Each configuration entry occupies one line with the following format: **keyword: value.** Currently only one keyword **port** is defined. The value of port is a nonnegative integer, which specifies the port number on which the server program should run. When the specified port number is 0, the server let the system assign any unused port number for the server.

After the server program starts, it will first read in the configuration information. Then it will create a socket using the socket() system call. It binds its address to the socket using the bind() system call. The server machine's fully-qualified domain name and the (assigned) port number are displayed on the screen. Then listen() call is used to indicate that the server is now ready to receive connect requests from clients.

The server's job is to wait for either connection or disconnection requests from clients. If it receives a connect request, it accepts the connection. After the connection has been established, the server will wait for messages/requests from the client (see below for the commands a user may issue in chat_client, which may trigger communication with chat_server).

If the server receives a disconnect request from a client, it should close the corresponding TCP socket connection.

chat_client

usage: chat_client configuration_file

The configuration_file has the same format as the configuration file of the server program. The client configuration file defines two keywords: **servhost** and **servport,** they specify the chat server's hostname (or dotted-decimal IP address) and port number, respectively. The client first creates a

socket using the socket() system call. It then connects to the server using the connect() system call. The whereabouts of the server are specified in the configuration file as discussed above.

A user can perform the following four operations in chat_client:

1. quit the client program using command "exit". If a TCP connection is open, the client should close the connection before exiting the progrma. The "exit" command is only issued when the user has logged out from the server (see below for login and logout commands).
2. log into the server using command "login" in the format of "login username". In this project, we do not consider user password, and we also assume that no two users will give the same username when they log in. After receiving the login command, the server will simply maintain some proper information for the username, in particular, the socket connection information for the username, so that later it can properly forward messages to this username.
3. log out from the server, in the format "logout". The server will remove the socket connection information for the corresponding client after it receives the logout command. After the logout command is sent to the server, the user will have the same interface as when the client program first starts. That is, the user can issue two commands "login" to log into the server, and "exit" to quit the client program.
4. send a chat message in the format of "chat [@username] some_message", where @username is optional parameter of the chat command. If @username is not present in the command, the corresponding some_message is a broadcast message, the server should send to all the users who are currently logged in. If @username is present in the chat command, the some_message should only be forwarded to the corresponding "username". Note that "chat" command can only be issued after the user has logged into the server.

When the server forwards a message from user1 to another user2, it should include the user1 username in the message. When user2 receives the message originated from user1, it should display the message in the following format: user1 >> some_message

In summary, a user can issue commands "exit", "login" after the client first starts. After it logs into the server, it can issue commands "chat" and "logout". After a user logs out from a server, it can issue the same two commands "login" and "exit" as when the client first starts.

Application layer protocol

In this project, you have the freedom to design your own application layer protocol, in particular, the message format exchanged between client and server. However, you should follow the above mentioned requirements and the well-established principles as illustrated in the design of HTTP and SMTP.

Mandatory implementation requirements

- Pthread and select() must be used in your project to support I/O multiplexing on the server and/or client programs. For example, if the I/O multiplexing in the client program is implemented using POSIX threads, the I/O multiplexing in the server program can be implemented using select() system call. Note that other techniques can also be used in your project including concurrent server model etc, or the combination of these techniques. However, you must have pthread and select() in your project and they are used in a meaningful way (for I/O multiplexing). This requirement is mainly for you to get a chance to practice pthread and select() based programming. You need to document in the README file how pthread and select() are used in your project. If you prefer, you can also discuss other server models that you have used in your project.
- signal process. Your programs must handle the signal SIGINT, generated when a user types Ctrl-C. In particular, your programs must close the opened socket connections before they exit.

- properly closing opened socket connections: for example, when a user issues the command "exit", the opened socket connection should be properly closed before exiting the program

Grading Policy

A program with compiling errors will get 0 point (we must be able to compile and run your program on linprog). Total points: 100.

1. proper README file and makefile (5)
2. program initialization with configuration file (5)
3. user commands login, logout, and exit (15)
4. client chat command "chat" (30)
5. handling SIGINT signal (15)
6. usage of pthread (15)
7. usage of select() (15)

Extra Points (choose one of the following two options, maximum extra points are 25 points. You will not get more than 25 extra points even if you implement both)

First option (25 points):

Implement a GUI on linprog. If you implement a GUI for this project, you do not need to follow the commands we specified above, but you must have the same functionalities (you can certainly add more functionalities). The popular choices of GUI packages on Linux include Qt, wxWidgets, and GTKmm. Note that "text-based user interface" or "GUI-like" solutions such as "ncurses" are not considered in this project for the 25 extra points. If the GUI package you would like to use is not available on linprog, it is OK for you to develop the extra-point part on your own laptop (but with a Linux system), and you will give a demo to the TA using your own laptop for this part.

Second option (25 points):

In the required components of the project, a targeted message in the format of "chat @user1 some_message" is delivered from the client to the server, and then from the server to the corresponding receiver. In this extra point part of the project, you should directly deliver the message from the sender to the receiver, without relying on the server to do the forwarding. For example, assuming user1 issues a command "chat @user2 some_message". The message should be directly delivered from the chat_client of user1 to the chat_client of user2 (without forwarding to the chat_server first). One way to achieve this is to create an additional listen socket in chat_client (in addition to the TCP client socket that is used to communicate with the chat_server). This listen socket information (IP address and port number) is forwarded to the chat_server when a user logs into the chat_server, which is in turn forwarded to other users (chat_client). In this way, when user1 sends a direct way to user2 using command "chat @user2 some_message", user1 can look up the listen socket of user2 and connect to it to directly send the message, instead of forwarding to the chat_server first. In this way, a chat_client program is actually both a server program and a client program, which is commonly referred to as "servent" in TCP/IP socket programming.

If you have implemented the extra points component, please indicate which option you choose and specify how you implement it in a README file. Please note that no partial points will be given to the extra point components of the project. If you decide to do it, you must fully implement its functionality. Please also note that, even if you have implemented the extra points part, you will still need to submit the required components (and the extra-points part).

Deliverables Requirement

Tar all the source code files including the header files, the makefile, and README file in a single tar file (name it as proj2.tar) and submit via the submission page on Canvas. As mentioned above, if you have implemented the extra-points , you will need to submit both the required components and

the extra-point parts. Here is a simple example on how to use the Unix command tar (assuming all your files related to this projects are under one directory, say project2). To tar all the files:

tar -cvf proj2.tar *

To check the contents in the tar file:

tar -tvf proj2.tar

To untar a tar to get all files in the tar file (to avoid potential overwriting problems, it is better to do this under a different directory instead of the original project2 directory):

tar -xvf proj2.tar