ECE361 COMPUTER NETWORKS

Text Conferencing

H. TIMORABADI

Objective

In this lab, you will use UNIX TCP sockets to implement a simple conferencing application.

Reference

- Section 2.4 and related sections on Berkeley API from Chapter 2 of the Communication Networks by Alberto Leon-Garcia and Indra Widjaja, McGraw Hill, 2nd Edition, 2004.

- The network socket programming "Beej's Guide to Network Programming" available on the University Portal under the course section.

Assignment Description

In this assignment, you need to implement a text conferencing application. It consists of a server with registered users, and several client nodes which are the end points for a conference session. The clients may be in different networks. TCP will be used to communicate between the clients and the server.

Each client has a unique name referred to as the identifier (ID). The ID of a node is unique among participating nodes. The server is assumed to have a list of client IDs that it will accept, and each ID is associated with a password (perhaps taken through a user registration procedure). A client must first log into the server with its ID and the server's IP address.

When a client wishes to communicate with other clients, it must create a conference session on the server. A conference session is simply a list of clients that are sending text messages to one another. Any message sent by a single user is seen by all clients that are participating in the conference session. It is the server's responsibility to multicast the text data to all participating clients.

You will implement both the server and the client processes.

Protocol

We use two categories of messages: data and control. Control messages are used for initiating the client-server interaction, session creation, invitation, termination, etc. Since you are using TCP, you do not need to use acknowledgements for reliability. You only need acknowledgements for success/failure conditions at the server, e.g. whether login is successful.

You **must** use the following structure for data and control messages:

```
struct message {
        unsigned int type;
        unsigned int size;
        unsigned char source[MAX_NAME];
        unsigned char data[MAX_DATA];
};
```

The type field indicates the type of the message, as described in the table below. The size field should be set to the length of the data. The source field contains ID of the client sending the message.

The following table shows all control packets that **must** be implemented:

Table 1. Packet types

Type	**Packet Data**	**Function**
LOGIN	<password>	Login with the server
LO_ACK		Acknowledge successful login
LO_NAK	<reason for failure>	Negative acknowledgement of login
EXIT		Exit from the server
JOIN	<session ID>	Join a conference session
JN_ACK	<session ID>	Acknowledge successful conference session join
JN_NAK	<session ID, reason for failure>	Negative acknowledgement of joining the session
LEAVE_SESS		Leave a conference session
NEW_SESS	<session ID>	Create new conference session
NS_ACK		Acknowledge new conference session
MESSAGE	<message data>	Send a message to the session or display the message if it is received
QUERY		Get a list of online users and available sessions
QU_ACK	<users and sessions>	Reply followed by a list of users online

Although you are required to use this protocol and struct, the serialization and deserialization approach is up to you. For example, you can use ":" to separate fields of the struct when you serialize it (as in the File Transfer lab), but if you want to experiment with a different approach that is fine too.

Section 1

Client Program (client.c):

You should implement a client program, called "client.c", in C on a UNIX system. Its command line input should be as follows:

```
client
```

Upon execution, the client program will wait for user input on the command line.

The login process should be clear from the available messages and command parameters. You may assume that the passwords are sent and received in plain text. TCP is a bidirectional protocol, so once a connection has been established between the client and the server, communication may proceed.

Once the client has logged in and joined a session, it will need to wait for messages from the server and at the same time continue to wait for user input on the command line. You will need a strategy for dealing with these two I/O sources simultaneously (e.g., starting a receive thread, or using non-blocking sockets).

The client must implement the following commands on the stdin file stream:

Table 2. Client commands

`/login <client ID> <password> <server-IP> <server-port>`	Log into the server at the given address and port. The IP address is specified in the dotted decimal format
`/logout`	Log out of the server, but do not exit the client. The client should return to the same state as when you started running it
`/joinsession <session ID>`	Join the conference session with the given session ID
`/leavesession`	Leave the currently established session
`/createsession <session ID>`	Create a new conference session and join it
`/list`	Get the list of the connected clients and available sessions
`/quit`	Terminate the program
`<text>`	Send a message to the current conference session. The message is sent after the newline

Here is an example of how clients in a session would work:

Client 1:

1. Run the program: ./client
2. /login jill eW94dsol 128.100.13.140 5000
3. /createsession lab_help

Client 2:

1. Run the program: ./client
2. /login jack 432wlFd 128.100.13.140 5000
3. /list /* Returns a list of users and the sessions they joined */
4. /joinsession lab_help
5. Hi Jill! This is Jack. How are TCP sockets different from UDP sockets?

In this section, a client is only allowed to join one session. As such, the server should check whether a client is currently in a session or not.

Server Program (server.c):

You should implement a server program, named "server.c", in C on a UNIX system. Its command input should be as follow:

```
server <TCP port number to listen on>
```

You should have a strategy to deal with unavailable port numbers. For example, you can use the UNIX netstat command to find out which ports are available on the system.

Upon execution, the server should wait for connections from the clients in the system. The server acts as both **a conference session router and a database**.

For the database part:

- The server **has access to a client list** with passwords (which can be hard coded into your program or kept as a database file).
- The server **should keep an up-to-date list** of all connected clients in the system, as well as their session id, and IP and port addresses. The server should **delete** a conference session when the last user of the session leaves.

The server should acknowledge some client requests (i.e. login and join session). You should deal with the possibility of a client attempting to log in with an ID that is already connected.

The server implements the conference functionality of the system **by forwarding messages intended for a conference session to all the clients registered for that session.** This process should be transparent to the receiving clients.

Section 2: Additional Features

In this section, you must implement **TWO** additional features. Some important constraints:

1. **The commands from section 1 (see Table 2) must remain unchanged.** Implement new commands if needed, rather than modifying the existing ones.
2. Your additional features must not be subsets of each other. This is not an issue for the features listed below, but be careful of this if you propose your own features.

A list of approved features is included below (remember: **pick exactly TWO features**). You can also propose your own features, but if you choose to go this route you must get approval from a TA either on Piazza or via email. You are allowed to implement any features that are approved on Piazza, regardless of who proposed them.

1. Multiple sessions.
    a. Client: Allow a clients to join multiple sessions. You should clearly indicate on the client's terminal the session ID of every message.
    b. Server: Keep all sessions that the client joined. You should carefully design the up-to-date list (i.e., the output of `/list`) to reflect this.
2. Session invites.
    a. Client: Implement a procedure for a client to invite other clients into a session. You must provide a protocol for a client to either accept or refuse an invitation.
    b. Server: The server should be able to forward invitations and corresponding messages to the specific clients.
3. Inactivity timer.
    a. Server: Disconnect clients that have been inactive for a long time, regardless of whether they're in a session. Note: the timer and disconnection decision **must** be implemented at the server.
    b. Client: Gracefully handle being disconnected from the server.
4. Private messaging. Clients can send private messages to each other, regardless of which session the sender and receiver are in (if any). Private messages are only viewable by the target client, and it must be clear who the sender is.
5. User registration. If a client does not already have a username and password to log in with, it can register with the server to create an account. This login information must be persistent, i.e., if the server is restarted, any users who registered dynamically must still be able to log in.
6. Session admins. When a client creates a session, they become the admin of that session. A session admin can kick another client out of the session. They can also give the admin position to someone else. There must be a way for other clients to see who the admin of a session is.

You should provide a text file to explain your design and implementation.

Deliverables

The following should be submitted and be available for your demo lab session:

- The client and server programs. All code must be readable, with meaningful comments. If you use encryption for the passwords, make sure you use simple passwords and make them available separately. Your code must be able to run on the UG machines.
- It wouldn't be a bad idea to have a set of meaningful test cases or scenarios, which will test most of the functionality.
- Set of Makefiles and/or UNIX scripts that simplify the compilation.
- The text file to explain Section 2.

Notes

- This lab is long so you should allocate a proper amount of time for finishing it.
- For easy inspection, you may want to separate your code into more than two files; however, you will neither be punished nor rewarded for this.
- For electronic submission, only one person in the group should submit the file. You should create a tar ball (e.g., a4.tar.gz) with all the files needed to compile and run your programs. You should also include a Makefile in the tar ball that compiles all your source code into two executables: one named `server` and one named `client`.
- The following command can be used to tar your files (where **x** is the lab number, ranging from 4 to 5 for the Text Conferencing lab):
  `tar –czvf a`**`x`**`.tar.gz <files to tar>`
- Use the following command on the eecg UNIX system to submit your code:
  `submitece361f` **`x`** `a`**`x`**`.tar.gz`
- You can perform the electronic submission any number of times before the actual deadline. A resubmission of a file with the same name simply overwrites the old version. The following command lists the files you have submitted:
  `submitece361f -l` **`x`**
- You are expected to explain the design choices made in your code to the TAs during your demo lab session. The lab specifications are similar to but different from those in previous years in subtle ways. This will make it easy to spot plagiarized code.