ECE361 Computer Networks
Text Conferencing

Objective

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

Reference

Before starting this assignment, you should read **Section 2.4 "The Berkeley API" of your textbook**. For further information on network socket programming, you may read ***Beej's Guide to Network Programming*** posted on the course website.

Project Description

In this assignment, you need to implement a conferencing application. It consists of a server with registered users, and several client nodes which are the end points for a particular conference session. The clients may be located 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 through a user registration procedure that took place before the time of this project). A client must first log into the server with its ID and the server's IP address. Upon logging in by a client, the server will create a binding between the client and itself until the client exits.

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

The following protocol is only a **suggestion**. **Feel free to amend, or even use a completely different protocol,** but you must implement the basic commands and functionalities specified in the next two sections. You are required to explain your protocol and its implementation in the demo lab session.

We use two categories of messages: data and control. Control messages are used for client-server binding, 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 lab3message {
        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:

type	packet data	Function
LOGIN	<client ID, 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		Create new conference session
NS_ACK	<session ID>	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

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 further commands on the command line, and start a receive thread. 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.

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.

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

`/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 string dot format
`/logout`	Exit the server
`/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 new line.

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

Client 1:

1. Run the program: ./client
2. /login jill eW94dsol 192.168.1.5 5000
3. /createsession lab_help

Client 2:

1. Run the program: ./client
2. /login jack 432wlFd 192.168.1.55 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. If so, the server should check whether a client has been in one 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>
```

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 joinsession). 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: Project

In this section, more possible functionalities are suggested. You could design your own commands and implement them.

Suggestion Functionalities:

- One client is able to join multiple conference sessions.
    - *Client Program*: if so, the client should clearly indicate on the client's terminal the session ID of every message.
    - *Server Program*: the up-to-date list should be carefully designed.
- A client is able to invite other clients into a conference session.
    - *Client Program*: if so, you must provide a protocol for a client to either accept or refuse an invitation.
    - *Server Program*: the server should be able to forward the invitation and corresponding messages to the specific clients.
- Implement a timer with each client on the server side, to disconnect clients that have been inactive for a long time.

Note that you do not need to implement all functionalities. You can either choose one or design your own functionality. You should provide a text file to explain design and implementation for your functionality**.**

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.
- 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 (a2.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 of your source code into two executables: one named `server` and one named `client`.
- The following command can be used to tar your files:
  `tar –czvf a2.tar.gz <files to tar>`
- Use the following command on the eecg UNIX system to submit your code:
  `submitece361s 2 a2.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:
  `submitece361s -l 2`
- 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.