

CS 536

Assignment 3: Implementing multi user chatting system

Due Date: March 12, 2006 midnight

Purpose

The objective of the assignment is to extend the two user chatting system to support multiple user conference chats. We will be using UDP sockets to send messages between machines.

Background and Reading

Please read chapter 3 of the textbook and get familiar with the BEEJ guide to internet linked through the Labs page. Understanding the socket programming slides (downloadable from the labs page) will also help. A review of signals in Linux will also help.

Overview

For this assignment, we will be focusing on client server architecture and will be using socket programming to initiate chat requests. A detailed description of a possible architecture is described below.

Architecture

This chatting system will consist of two programs

1. The **chat program** capable of both initiating and receiving chat requests. This program will be running on every user machine.
2. The **chat directory server** is a central server that has a login name and IP address of each and every user. Once the chat program starts, it connects to the directory server to update its online user lists and also lets the directory server know of its own login name and IP address.

The **chat program** will have the following subsystems

1. A **directory client** that will connect to the directory server to update the online list and let the directory server know of its own login and IP address information. The directory client should also update information after an interval of time so that sockets of offline users can be closed.
2. A **message sender** that will send your message using a particular socket to a specified user or a group. For example, if you are talking to USER jain, than your sender should respond appropriately to a message like SEND jain <msg>. It should also send messages to group.
3. A **message receiver** system will read messages from sockets and print on the screen the message and its sender.
4. A **conference chat monitor** system keeps track of all current conference groups.

5. The program should have the capability to both start a conference group by inviting other users and it should also respond to invitations of other people by either accepting or rejecting invitations. Once you decide to form a group, store all the user logins in the conference chat monitor table.

Please note this is just a sample description, you can come up with your designs along the same lines if you want. There are a few general things to keep in mind when designing the system:

- You want to be able to support many users.
- You would like to keep network use to a minimum.
- System response time should be minimized.
- You would like the system to be somewhat scalable (able to grow).
- You would like the system to be portable.
- You would like the system to be robust (able to handle minor network interruptions without going down).
- You would like to provide cool services (perhaps more than simple text messages?).

Procedure and Details

Please note that this lab does not require any specific machines to work on but it will be tested on the XINU front end machines. We will be using UDP sockets to write our chatting system. In a multi-user setting, users can go offline at any moment and hence, using reliable TCP mechanisms will lead to unnecessary traffic.

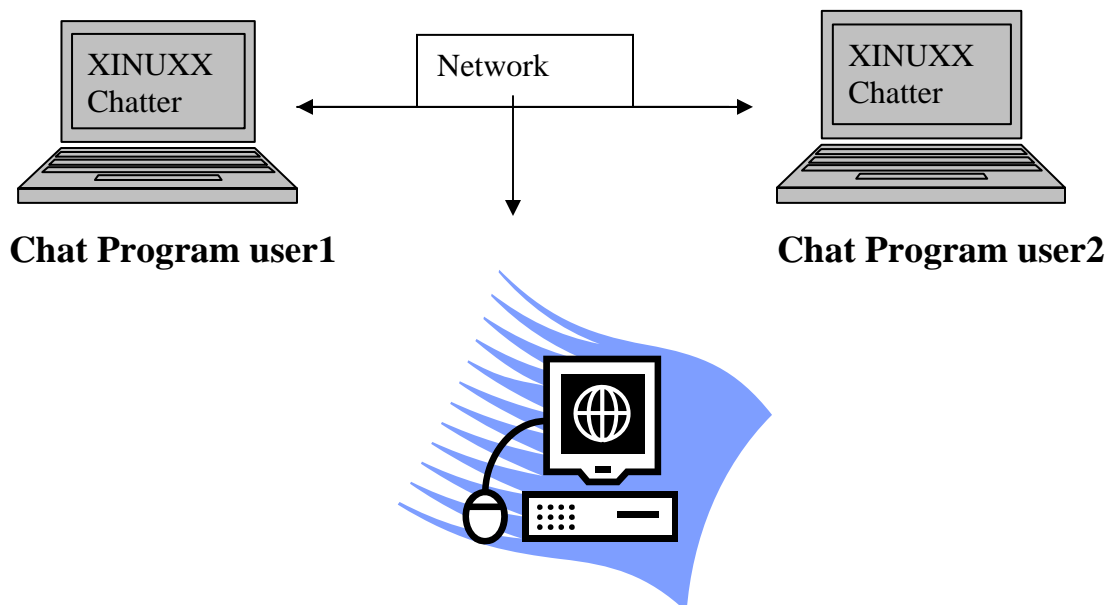


Figure 1: Directory Server XINUXY machine

Step 1: Implement the directory server

The server will keep in memory the list of pairs <name, IP-address> that will be added by the **Directory Clients**. Use **TCP** sockets for this server. The server will be waiting for incoming connections and when a request of the following form arrives:

```
ADD<crLf>
<Name><crLf>
<crLf>
```

it will register this entry in the Directory Server. The server will return:

```
OK<crLf>
<crLf>
or
```

```
ERROR<crLf>
<Error message><cr><lf>
<crLf>
```

The IP address will be taken from the socket information returned by accept.

When a request of the form:

```
DELETE<crLf>
<crLf>
```

is received, the entry that corresponds to that IP address will be removed.

The server will return:

```
OK<crLf>
<crLf>
or
```

```
ERROR<crLf>
<Error message><cr><lf>
<crLf>
```

To get all the user entries the client will send:

```
GET<crLf>
<crLf>
```

and the server will answer with the list of user/ip pairs

```
User1<crLf>
ip address 1<crLf>
```

```
User2<cr>
ip address 2<cr>
...
<cr>
```

The end of the list will be delimited by an empty line. Alternatively, the server will reply:

```
ERROR<cr>
<Error message><cr><lf>
<cr>
```

The telephone program will send an ADD message when the chat program starts. It will also send a DELETE message when the chat program exits.

The directory server should periodically check whether all users are online or not.

Step 2: Implementing directory client

The chat program should send an add message to the directory server when the client starts and the directory client is also responsible to issue get commands periodically to update the local online user record. The directory client should send a delete command when exiting the program.

Step 3: Message sender

The message sender module or thread is responsible for parsing commands of the form SEND <user/group> <message>

The sender then should identify the socket/s based on username/group name and then issue the appropriate send call. It is left to you to decide the optimal data structures (for fast searching) to store the collection of sockets. Please note that all the slavesockets will be running on random ports. And all your machines should have the same master socket this way you know which port the machine is listening on.

A group message should be delivered to all the recipients. A best way to do is to have a separate socket for each member of the group. You can also implement a routing table based algorithm for sending message across group peers.

Step 4: Message receiver

The message receiver module is responsible for printing all incoming messages. You will have to look in to the select; so that we don't have blocking receives. Your module should print the output like

```
<user/group>: <Message>
```

Step 5: Conference Monitor

A user can invite other users to start conference chats and a user can also accept invitations. The user also has the ability to quit the conference at any time. The conference monitor is responsible for monitoring these changes. We will be using TCP for these control messages.

Once a user wants to invite other users, it should send a request like
<Group Name><crLf><source user><crLf><crLf>

Once a program receives this request, it can either accept or deny the request

To accept send,
OK <groupname> <crLf><crLf>
and add the group to the conference monitor.

On receiving an OK, the conference monitor at the source should be updated and a final message should be sent to all peers (only source is aware of it till now).

Now all machines should have the group name entry filled in their conference monitors.

If the user wants to leave conference, it should let all other peers know about its desire to quit by sending

QUIT <group name>

to all peers.

Feel free to make improvements to this suggested design.

Grading

Grading for the project will happen in a one on one presentation to the TA. Session details will be posted later.

Turn In

You will turn in this lab using electronic turnin through the XINU machines. The deadline is March 12, 2006 midnight.

Turn In Command

Go to the parent directory of your working directory and on a xinu frontend machine type

turnin -ccs536 -pproject3 <directory>

Requirements

- 1. Turn in all your source code electronically**
- 2. Write a readme file**
- 3. Write a design document defending your designs.**