Fall 2011 - CSEE W4119 Computer Networks
Programming Assignment 1 - Simple Chat Application

Prof. Gil Zussman

due: 10/26/2011, 11 AM, EST

1 Introduction

This programming assignment is to implement a *simple* chat application with *3* clients and a server using UDP. You are required to create one program. The program should have two modes of operation, one is the client and the other is the server. The clients communicate directly with each other. The server is used for setting up the clients and for book-keeping purposes. The server also needs to be used for storing off-line messages from clients. The functionalities and specification of each program are described in detail below. ***Please start early and read the entire homework before you start!!***

2 Functionalities

The complete chat application can be broadly classified into four functions. Each function involves either the client part or the server part or a combination of both. The different functionalities and their respective parts in both the server and the client are explained in the upcoming sections.

2.1 Registration

In this function the server has to take in a registration or a subscription request from the client. The server needs to be started before the client. The server maintains a table with the IP addresses, port numbers, and nick-names of all the clients. This functionality has both client and server parts,

Client mode:

- The client has to communicate with the server using the IP address and the port number of the server. [assume all clients by default know the server information]

  `$ UdpChat <mode> <command-line arguments>` : Start the program for server or client, (for example : UdpChat -c for client and UdpChat -s for server). The server takes its listening port as an argument and the client mode should take four arguments, the client's nick-name, server-ip address, server port number, and client's port number for listening.

  `$ UdpChat -s <port>` : Initiate the server
  `$ UdpChat -c <nick-name> <server-ip> <server-port> <client-port>` : Initiate the communication to the server. The nick-name is like the username for a chat client. The server IP address is given in dotted-decimal format. The port number is an integer value in the range `1024-65535`. For example if the server IP is `198.123.75.45`, the server port is `1024`, the client's port number for listening is `2000` , then the command string is going to be:
  `$ UdpChat -c client1 198.123.75.45 1024 2000`. If arguments are taken in a proper format, a prompt like `'>>>'` should be displayed. Otherwise appropriate error messages should be displayed.

- Successful registration of the client should also display the status message in the client:
  `$>>> [Welcome, You are registered.]`

- Every client should also maintain a local table with all the client information (nick-name, ip, port number, online-status). Every client should update (overwrite) its local table when the server sends the information about all the other clients.
- When the table has been successfully updated, the client should display the message :
  `$ >>> [Client table updated.]`
- To exit or close a client use: `$ >>> ctrl + c`
  Assume a client once exited does not return.

Server mode:

- The server should maintain a table to hold the nick-names, IP addresses, and port numbers of all the clients.
- When a client sends a registration request, it should add the client information (name, IP address, port number, online-status) to the table.
- The server should *broadcast* the complete table of active clients to all the online clients to update their local information.

2.2 Chatting

Once the clients are set up and registered with the server, the next step is to implement the actual chat feature. The clients should communicate to each other directly and must not use the server for forwarding chat messages. Since it does not involve the server there is just the client part for this functionality.

Client:

- A client should communicate to another client with the information from its local-table.

  The client should support the following command for sending messages
  `$ >>> send <name> <message>`. : This command should make the client look up the IP address and port number of the recipient client from its local table and send the message to the appropriate client. (message length should be variable)
- The client which sent the message has to wait for an *ack* and likewise, the client which received the message has to send an *ack* once it receives the message.
- If *ack* times out (*500 msecs*) for a message sent to a another client, it means the client at the receiving end is off-line, and so the message has to be sent to the server. The server has to save these messages and show it later to the appropriate clients when they register back. (details in offline-chat section).

  The appropriate status messages also needs to be displayed

  `$ >>> [Message received by <receiver nickname>.]`

  `$ >>> [No ACK from <receiver nickname>, message sent to server.]`

2.3 De-registering

This is a book-keeping function to keep track of the active clients. This functionality involves both client and server parts.

Server:

- When the server receives a de-registration request from a client, it has to change the respective client's status to offline in the table (do not close or exit the client to make it offline).
- It then has to *broadcast* the table to all the active or online clients.
- The server then has to send an *ack* to the client which requested de-registration.

Client:

- When a client is about to go offline, it has to send a de-registration request to the server to announce that it is going offline.
- The client has to wait for an *ack* from the server within *500 msecs*. If it does not receive, the client should retry for *5* times. If it fails all five times the client should display the message:
  `$>>> [Server not responding]`
  `$>>> [Exiting]`
  and exit.
- All the other active clients, when they receive the table from the server, should update their respective local tables (just overwrite the existing table).

  `$ >>> dereg <nick-name>` : This is a de-registration request to the server from the client to go offline. We don't expect clients to log back in with the same IP and port number after they have exited.
- Successful de-registration from the server should display the following status message in the client:

  `$ >>> [You are Offline. Bye.]`

2.4 Off-line Chat

The last functionality of the chat-application is to implement an off-line chat. This is similar to the feature in google-chat. When the user is off-line, the server records the chat messages the user gets and provides it later when the user is online. In a similar fashion, when a client quits the chat session, the server should save the off-line chat messages. This also has both client and server parts.

Client:

A client sends off-line messages in two cases:

- When the recipient or end-client is offline in its local-table of clients.
- When there is a time-out on a message sent to a client.

In both the above given cases the client has to send an automatic *save-message* request to the server. This request should also include the

- Nick-name of the intended recipient
- Message

On success, the following status message should be displayed in the client:
`$ >>> [Messages received by the server and saved]`

A logged-out client should be able to log back in using :
`$ >>> reg <nick-name>` : Instruct the server to sign-in or register the client (i.e., change the associated client's status to online in the table).

Server:

- When the server receives an offline-message, it has to save it separately for different clients. (For example, you can use files for each client and save all off-line messages for a client in its appropriate file).
- When a server receives a *save-message* request from a client it has to check for the status of the intended recipient.

- If the recipient client is still active, then send the client which sent the *save-message* request an *err* message :
  `$ >>> [Client <nick-name> exists!!]`
  and also send the table to the client for it to get updated.
- If the recipient client is not active, then the server should change the status of the appropriate client to offline, broadcast the updated table to all active clients and save the messages in the files associated.
- The messages while saved should also have their associated *time-stamp* information. (You can get this using `gettimeofday()`).
- An *ack* also needs to be sent to the client which made a save-message request.

When a logged out client returns :

- The server needs to check for any off-line messages for that client :
  - If *yes*
    * Send all the off-line messages to the client
    * Clear them in the server
    * Change the status of the client to online
    * *broadcast* the table to all the online clients.
  - If *no*
    * Change the client's status to online
    * *broadcast* the table to all the clients.
- Clearing the messages in the server makes sure that the server does not send the same off-line messages repeatedly. This status message should also be displayed in the client before the offline messages are displayed:
  `$ >>> [You have messages]`

For example:
Assume :

- There are three clients
- Client 1 goes offline
- The other two clients send messages to client 1

The off-line messages in the server for client 1 should be saved as

```
>>> client 2:  Hi!
>>> client 3:  Hello!!
```

When client 1 returns (log back in) this should be printed in client 1
Client 1 :

```
>>> You Have Messages
>>> client 2:  <timestamp> Hi!
>>> client 3:  <timestamp> Hello!!
```

3 Testing

Before submitting your work, please do **test your programs thoroughly**. Your chat application should *at least* work with

- *One* instance of the program in server mode.
- *Three* instances of the program in client mode.

To start-off with you can assume fixed sizes for the client table and extend your implementation to handle dynamic length if you have time, however full points will be awarded only if you handle dynamic lengths. You must handle business-logic errors such as a user trying to login with an already connected nick-name.

Two simple example test cases have been provided for you. You should also test your program with your own test cases.

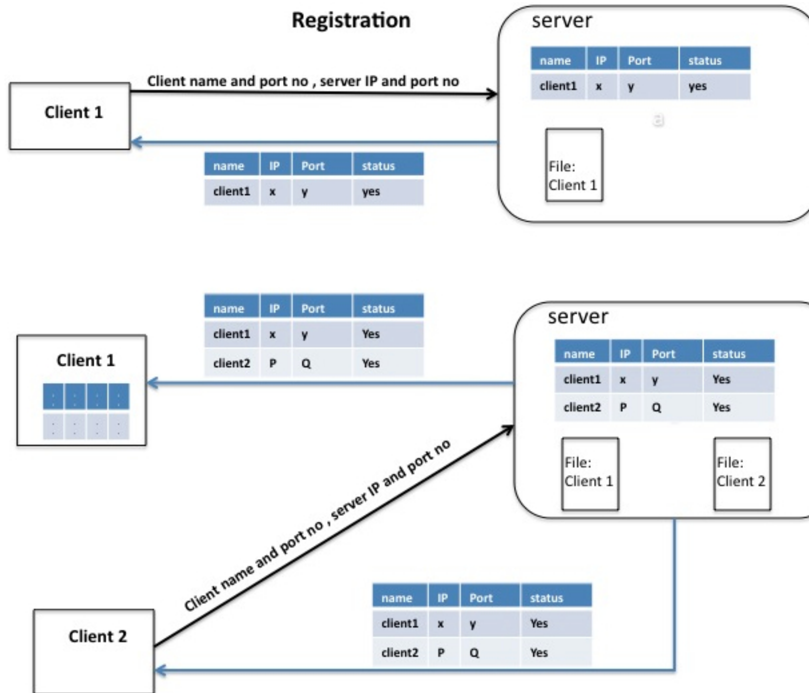
Test-case 1:

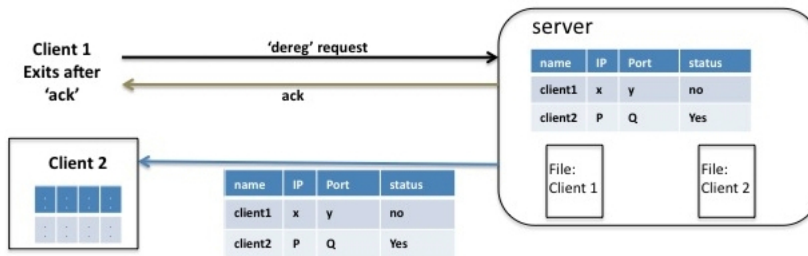
1. start server
2. start client `x`(the table should be sent from server to x)
3. start client `y`(the table should be sent from server to x and y)
4. start client `z`(the table should be sent from server to x and y and z)
5. chat `x -> y, y->z, ... , x ->z` (All combinations)
6. dereg `x` (the table should be sent to y, z. `x` should receive 'ack')
7. chat `y->x` (this should fail and message should be sent to server, and message has to be saved for `x` in the server)
8. chat `z->x` (same as above)
9. reg `x` (messages should be sent from server to x, x's status has to be broadcasted to all the other clients)
10. `x, y, z` : exit

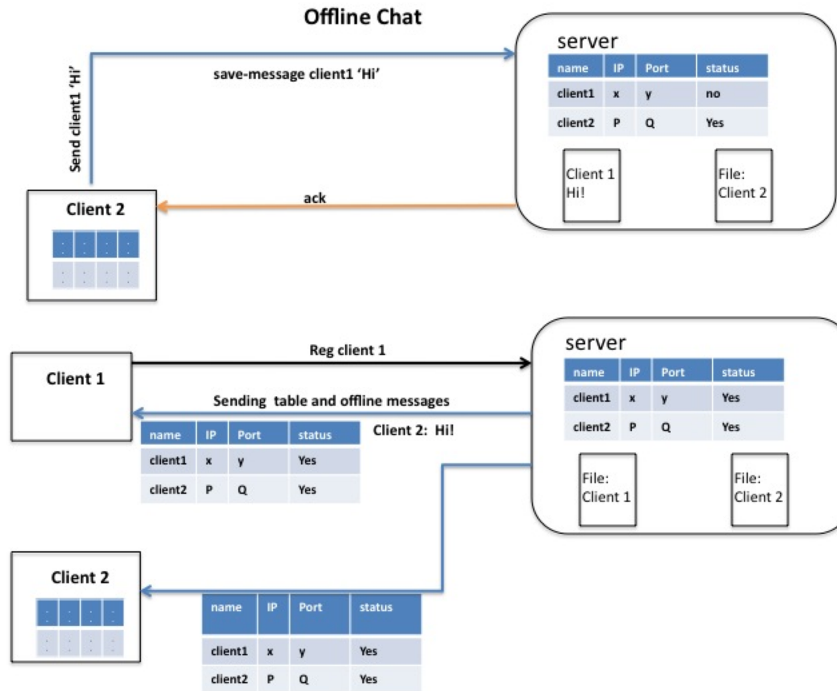
Test-case 2:

1. start server
2. start client `x` (the table should be sent from server to x)
3. start client `y`(the table should be sent from server to `x` and `y`)
4. dereg `y`
5. server exit
6. send message `x-> y` (will fail with both `y` and server, so should make *5* attempts and exit)

The figures below shows the registration process, de-registration process and offline messaging involving two clients. To provide some more *clarity!!!*.


Registration
server
Client 1
Client name and port no , server IP and port no
name IP Port status
client1 x y yes
File:
Client 1
Client 1
Client 2
Client name and port no , server IP and port no
client2 P Q Yes
File:
Client 2


De – Registration
Client 1
Exits after
'ack'
'dereg' request
ack
server
name IP Port status
client1 x y no
client2 P Q Yes
Client 2
File:
Client 1
File:
Client 2



4 Submission Instructions

Please use either C or Java for developing the chat application. Both are available at CS department CLIC or EE ILAB. Your submission package should include the following deliverables.

- `README`: The file contains the project documentation, program features, usage scenarios, brief explanation of algorithms or data structures used, and the description of any additional features/functions you implemented.
- `Makefile`: This file is used to build your application. If you have written the programs in C (Unix), the output file name should be `UdpChat`. If you used Java, the file name should be `UdpChat.class`.
- Your source code.
- `test.txt`: This file should contain some output samples on several test cases. This would help others to understand how your programs work in each test scenario. It is optional to include this, as part of your `README` document.

The submission should be made via Courseworks. Zip all the deliverables mentioned above, and name the zip file as `<your UNI><CLIC or ILAB>PA1.zip` (e.g. `gz2136ILABPA1.zip` for Professor Zussman). Upload the file in the Courseworks, under `Class File -> Shared Files -> PA1`.

No windows programming environments like (.net, Visual studio, VC++ etc) will be allowed. Programs have to be compiled using gcc. You can use the cygwin environment that comes with gcc utility however we do not recommend using a windows environment. If you decide to use cygwin, we cannot guarantee that the TAs will be able to help debug any associated errors. If you have any issues with your environment, please see the TAs early on.

In the grading of your work, we will take the following points into account.

- The documentation clearly describes your work and the test result.

- You handle all errors (Exceptions, memory management and business-logic) and exiting the program gracefully.
- The source code is complied properly by using the `Makefile` and generate appropriate output files.
- The programs run properly, including 1) take appropriate commands and arguments, 2) handle different situations and support required functions, and 3) display necessary status messages.

Happy Coding and Good luck!!