[redacted] Computer Networks
Programming Assignment 1 - Simple Chat Application

[redacted]

due: 03/22/2022, 23:59PM, EST

1 Introduction

The objective of this programming assignment is to implement a *simple* chat application with at least *3* clients and a server using UDP. You are asked to create one program. The program should have two modes of operation, one is the client, and the other is the server. The client instances communicate directly with each other. The server instance is used to set up clients and for book-keeping purposes. The server is also used to store off-line messages from clients and broadcast channel messages to all clients within a predefined communication channel (group chat). 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 five functions outlined be low. Each function involves either the client part or the server part or a combination of the two. The four functions and their respective parts in both the server and the client are explained in the following sections.

2.1 Registration

For the registration function, the server has to take in a registration or a subscription request from a 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 clients. This functionality involves both client and server modes.

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].

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

  `$ ChatApp -s <port>` : Initiates the server process

  `$ ChatApp -c <name> <server-ip> <server-port> <client-port>` : Initiates client communication to the server. Client name is like a username for this chat client. Server IP address should be given in decimal format and the port number should be 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 will be: `$ ChatApp -c client-name 198.123.75.45 1024 2000`. If arguments are taken in a proper format, a prompt like `'>>>'` should be displayed. The application should also be able to perform basic error checking where the IP addresses are valid numbers, and assigned ports are within the range. Otherwise, an appropriate error message should be displayed.

- Successful registration of the client on the server should also display the status message to the client:
  `$>>> [Welcome, You are registered.]`
- Every client should also maintain a local table with information about all the other clients (name, IP, port number, online-status). Every client should update (overwrite) its local table when the server sends information about all the other clients (further detail on this in upcoming section).
- When the table has been successfully updated, the client should display the message :
  `$ >>> [Client table updated.]`

  There should be two ways to 'disconnect/close' as a client:
- Silent leave: Once a client disconnects/closes, the server will not be notified. You can expect that the client will not register again using the same information after it exits via Silent leave. To exit or close, a client uses `$ >>> ctrl + c` or simply closes SSH window that the client is running on (both actions need to be implemented, and the system should not crash).
- Notified leave: De-registers the client, and the de-registration action will be notified to the server. The client status in the server table should be changed to offline. More detailed information is covered in 2.3.

Server mode:

- The server process should maintain a table to hold the 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 so that they may update their local information. This should happen whenever the server updates its table.

2.2 Chatting

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

Client:

- A client should communicate to another client with the information from its local table (including communicating with itself).

  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 sends the message has to wait for an *ack* and likewise, the client which receives 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 offline, and so the message has to be sent to the server. The server has to save these messages and show them later to the appropriate clients when they come back online and re-register (details in offline-chat section).

  The appropriate status messages also need to be displayed for each scenario:
  `$ >>> [Message received by <receiver nickname>.]`
  `$ >>> [No ACK from <receiver nickname>, message sent to server.]`

2.3 De-registration

This is a book-keeping function to keep track of 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 change its status to offline).
- It then has to *broadcast* the updated table to all the active (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 an *ack*, 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.
  Please note: you should not close the SSH window of this client after de-registration. You will be expected to register the client back later to receive offline messages (if there are any). You do not need to consider a case in which another client uses the same information to register while the client is de-registered.
- Successful de-registration from the server should display the following status message in the client:
  `$ >>> [You are Offline. Bye.]`

2.4 Offline Chat

Another functionality of the chat application is to implement an offline chat. When the client is offline, the server records the chat messages that a client receives from other clients and provides them later when the client comes back online. In a similar fashion, when a client quits the chat session, the server should save the offline chat messages. This has both client and server parts.

Client:

A client sends offline messages in two cases:

- When the recipient is offline in its local table of clients. (exit via notified leave - section 2.1)
- When there is a timeout on a message sent to a client. (exit via silence leave - section 2.1 or potentially bad connection between the clients)

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

- Name of the intended recipient
- Message

If successful, 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 offline 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 with the recipient.
- The saved messages 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 offline messages for that client :
  - If *yes*
    - Send all the offline 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 offline 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 (logs back in) this should be printed in client 1

Client 1 :

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

2.5 Channel - Group Chat

Once the clients are registered with the server, they should be added to a channel where they can communicate with all other clients and server. Messages sent to this channel should be broadcast to all the online clients, except the sender client itself. For offline clients, message should be saved and sent to clients once they *reg* back in. To make the implementation easier, clients can use the server to forward messages in the channel.

Client:

- Upon registration, client should be added to a predefined channel. To simplify the implementation, You can assume all clients are members of this channel without additional channel registration process
- Use the following command to send messages to the channel: :

  `$ >>> [send_all <message>]`

  This command should send the message to the server in order to distribute to the clients (again, message length should be variable).
- The client which sent the message has to wait for an *ack* from the server within 500 msecs. Once *ack* is received from the server, appropriate status messages also need to be displayed:

  `$ >>> [Message received by Server.]`
- If the client (sender) does not receive an *ack* response from server within time limit, the client should retry for five times. If it fails all five times the client should display the message:

  `$ >>> [Server not responding.]`
- Clients who received channel message from the server, should send an *ack* back to the server and clients should display the received message:

  `$ >>> [Channel_Message <nick name(sender client)>: message ].`

  Note: 'Channel Message' should be a hard-coded string

Server:

- Upon receiving a channel message from a client, server should send an *ack* back to the sender client.
- The server can use the same table to broadcast messages to all the active (online) clients since all clients automatically become members of this predefined channel without additional channel registration process.
- Server should also expect an *ack* from all the active (online) clients (except the sender client), The server can wait up to 500 msecs to receive an *ack* from all clients.
- If the server does not receive an *ack* response from a client within time limit, the server should check for the status of the intended recipient (mentioned in offline chat).

- 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 (mentioned in offline chat)
- For offline clients, messages should be stored in the same file as the offline chat. The messages should also be saved with their associated time-stamp information (mentioned in offline chat) and *Channel-Message* should be attached at the start of the message as an indication of group chat message.
- When the client returns, the server needs to check for any offline messages and send them to the client based on the time of receipt

For example:
Assume :

- There are three clients
- Client 1 goes offline
- Client 2 sends channel message : Welcome to Channel
- Client 2 sends message to client 1 : Hello
- Client 3 sends message to client 1 : Hello!!

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

```
>>> Channel-Message client 1:  Welcome to Channel
>>> client 2:  Hello
>>> client 3:  Hello!!
```

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

```
>>> You Have Messages
>>> Channel Message client 1:  <timestamp> Welcome to Channel
>>> client 2:  Hello
>>> client 3:  Hello!!
```

Messages that client 2 should display: (Only shown output from SSH, client input not included in the example):
Client 2 :

```
>>> Message received by server.
>>> No ACK from client 1, message sent to server.
>>> Messages received by the server and saved.
```

Messages that client 3 should display: (Only shown output from SSH, client input not included in the example):
Client 3 :

```
>>> Channel Message client 2:  Welcome to Channel
>>> No ACK from client 1, message sent to server.
>>> Messages received by the server and saved.
```

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. Full points will be awarded only if you handle dynamic lengths. You must handle business-logic errors such as a user trying to log in with an already connected nickname.

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

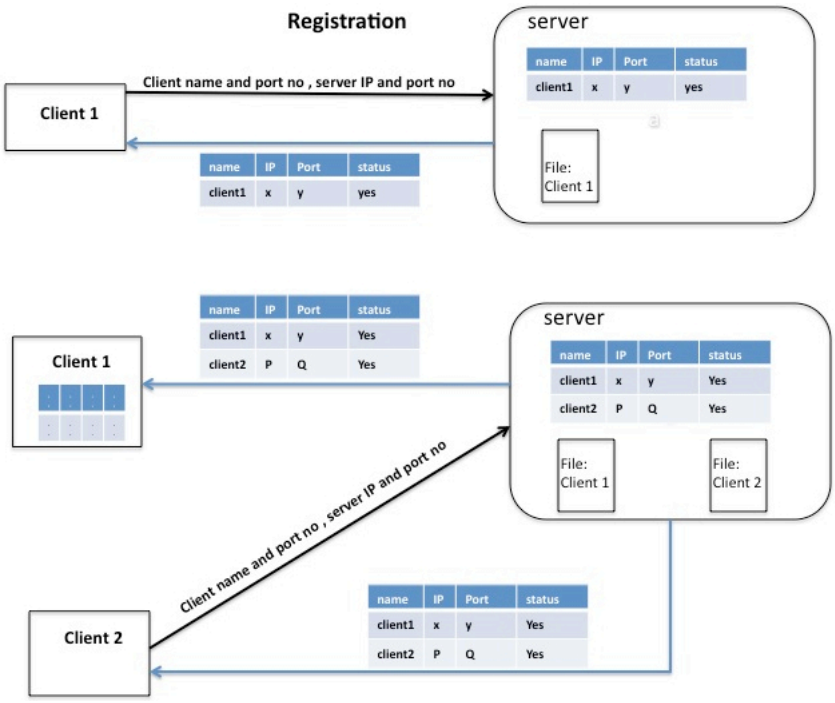
Test-case 1:

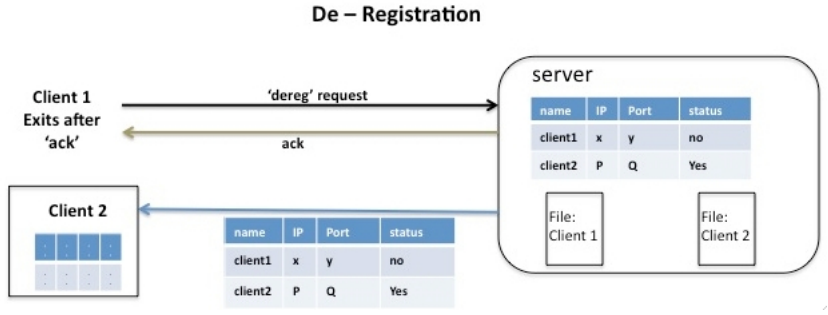
1. start server
2. start client `x`(the table should be sent from server to x)
3. start client `y`(the table should be sent from server to x and y)
4. start client `z`(the table should be sent from server to x and y and z)
5. chat `x -> y, y->z, ... , x ->z` (All combinations)
6. dereg x (the table should 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

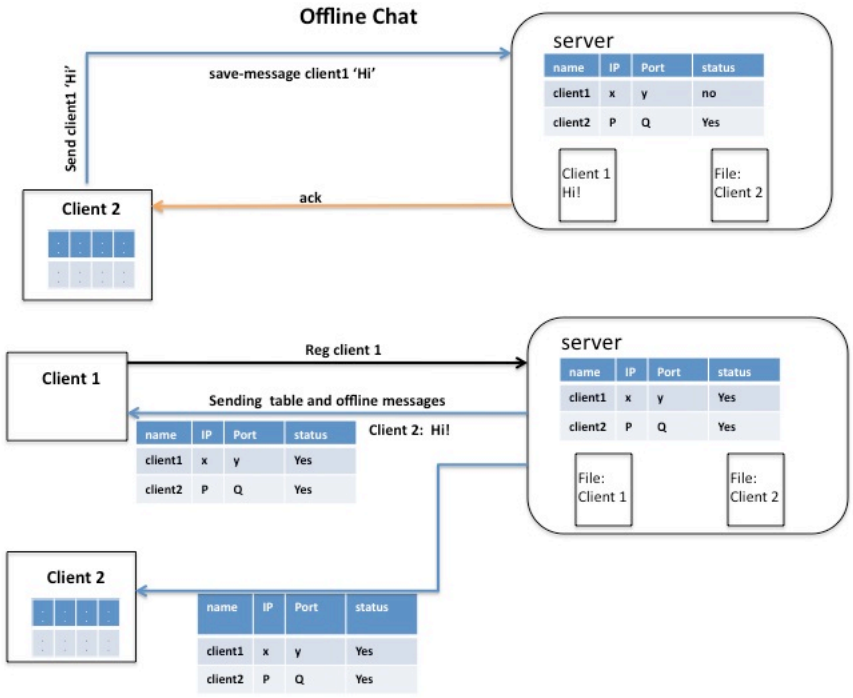
Test-case 2:

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

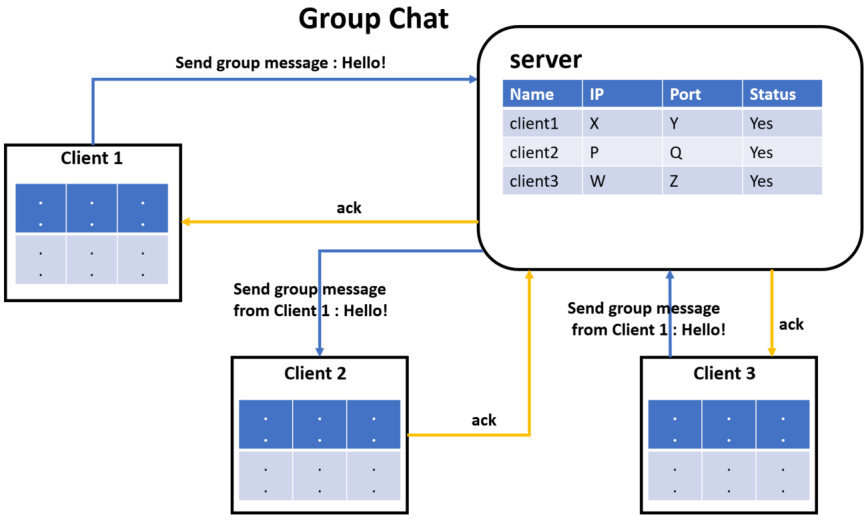

De – Registration
Client 1
Exits after
'ack'
'dereg' request
ack
server
name IP Port status
client1 x y no
client2 P Q Yes
File: Client 1
File: Client 2
Client 2



Test-case 3:

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` , `y` and `z`)
5. send group message `x-> y,z`

The figures below shows the group chat involving 3 clients and 1 server. To provide some more *clarity*.



4 Submission Instructions

You may use either C, Java, or Python for developing the chat application. Your submission package should include the following deliverables.

- `README`: Please put your name and UNI at the top of your README. The next thing in your README should be explicit command line instructions for compiling and running your program. The file should also contain basic project documentation, program features, brief explanation of algorithms or data structures used, a list of known bugs, and the description of any additional features/functions you may have implemented (fully optional).
- `Makefile`: This file is used to compile your program. If you have written the program in C, the output file name should be `ChatApp`. If you used Java, the file name should be `ChatApp.class`. If Python, have your program be called `ChatApp.py`. You do not need to supply a Makefile to compile your code if implementing in Python.
- Your source code. Please comment your code well, and use clear and sensible variable names.
- `test.txt`: This file should contain some output samples from the command line on several test cases. This will help others to understand how your programs work in each test scenario. It is optional to include this as a section of your `README` document.

Your submission should be made via Courseworks.

Please do not utilize Windows programming environments including .NET, Visual Studio, VC++, etc. Programs written in C have to be compiled using gcc, not clang or another compiler. All submissions will be compiled, run, and evaluated on Ubuntu 18.04 LTS. If you have any issues with your environment, please let the TA know early on.

Please comment your code. This not only makes it more likely that you will be awarded partial credit for anything which does not work, but you will thank yourself in six months when you are reviewing your code for a job interview, expanding on it as a personal project, explaining it to your pet fish, etc...

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

- The documentation clearly describes your work and the test result.
- The program takes command line arguments in the ***exact same format as specified by the assignment***.
- You handle all errors (exceptions, memory management and business-logic) and exit the program gracefully.
- The source code can be compiled and run properly, without errors or warnings.
- The programs run properly, including 1) take appropriate commands and arguments, 2) handle different situations and support required functions, and 3) display correct status messages in given scenarios.

Happy Coding and Good luck!!

□