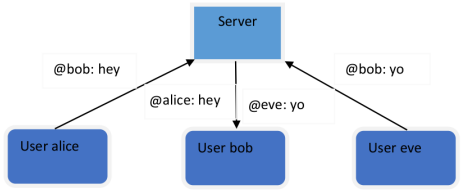
COL334/672 Assignment 2

September 2021

In this assignment, we will build a chat application that allows users to send plain text messages with one another. The figure below shows the framework we will build. Users can direct messages to other users using an @ prefix, and the server needs to forward these messages to the intended recipients. The message could be intended to be sent for a single client (Unicast) or all clients (Broadcast). The messages would be communicated as plain text, so the server will be able to read the messages. This is a simpler version of any commercial chat-based application which make use of centralized servers for relaying the messages, except the fact that messages would be encrypted in the case of commercial applications.



1 Message Format

Let us start with building a protocol to be used for communication. You will have to write a client application and a server application. TCP Sockets will be used for communication between a client and server . The protocol format can be as follows, much like HTTP, with some modifications as shown in the tables below. In contrast to HTTP's stateless behavior, we will preserve the state for each TCP connection at server.

1.1 User registration

Each client application will open two TCP sockets to the server, one to send messages to other users, and one to receive messages from other users. We can also do it over a single TCP connection but it will require some concurrency management, hence we will keep it simple for now. The client application

1	Client to server registration for sending messages	REGISTER TOSEND [username] \n \n
2	Server to client, if username is well formed	REGISTERED TOSEND [username]\n \n
3	Client to server registration, for receiving messages	REGISTER TORECV [username]\n \n
4	Server to client, if username is well formed	REGISTERED TORECV [username]\n \n
5	Server to client, if username is not well formed	ERROR 100 Malformed username\n \n
6	Server to client message in response to any communication until registration is complete	ERROR 101 No user registered \n \n

Table 1: Message Format for User Registration

will need to start with registering the user on both the TCP sockets using the following method.

Upon receiving the registration messages, the server will check whether the username is well formed, i.e. only alphabets and numerals, with no spaces or special characters. It will acknowledge the registration requests if the username seems good, else it will return an error message. As will also become obvious soon, the server will need to maintain a list of users who have registered and the corresponding TCP Socket objects for the sending and receiving connections over which they registered.

The user registration should be the very first activity that is done when a client opens the sockets to the server. The server should return an error message to all other commands it receives on the socket until the user registration has been done on both the sockets. The format for user registration is specified in Table 1.

1.2 Message Sending from Client to Server

On the TCP socket dedicated to sending messages to other users, the following method will be used.

The format of the messages being exchanged is specified in Table 2. The format of the SEND method is similar to that of various HTTP methods, with a header section followed by the data section. The header section has a field for content length which tells the server how many bytes to read after the blank line. If the Content-length field is missing, then the server should send an error message back to the client and close the socket. This is because in this case the server will not know how to parse any further data it receives on the TCP connection. The client can then reattempt to open a new connection and start do the registration again. As mentioned earlier, the messages sent may either me Unicast or Broadcast. The keyword ALL in the recipient field indicates it is supposed to be broadcasted, otherwise all messages are unicast by default.

1	Client to server message	SEND [recipient username]\n Content-length: [length]\n \n [message of length given in the header above]
2	Server to client message, if message is delivered	SEND [recipient username]\n \n
3	Server to client message, if message is undelivered	ERROR 102 Unable to send\n \n
4	Server to client message, if header is incomplete	ERROR 103 Header incomplete\n \n

Table 2: Message Format for Sending Message

If the message is received correctly by the server ,then the server should check if the recipient has been registered. If a registered user is found, the server shall attempt to forward the message to the recipient over the socket on which the recipient had registered to receive messages from other users. If a registered user is not found, ERROR 102 message will be sent back to the sender. This is why we mentioned earlier that the server will need to maintain a list of users who have registered, and the corresponding TCP Socket objects for the sending and receiving connections over which they registered.

At the receiver, if the message is received successfully, RECEIVED message will be sent to the server. The server then sends a SENT message to the sender, thus completing a successful transaction. If the receiver senses some error in the received message, it responds to the server with a valid error message. The server then forwards the same to the server. This will of course be done on the TCP socket of the sender which is dedicated to sending messages to other users.

1.3 Message Forwarding at server

The format of the forwarded message received from sender to the intended receiver is similar to the originally sent message, and the parsing procedure to be followed by the server will also be similar. The detailed format is specified in Table 3.

1	Server to client message	FORWARD [sender username]\n Content-length: [length]\n \n [message of length given in the header above]
2	Client to server message, if message is received	RECEIVED [sender username]\n \n
3	Client to server message, if header is incomplete	ERROR 103 Header Incomplete\n \n

Table 3: Message Format for Forwarded Message

2 Application Design

In this section, we define the behavior of Client-side application and Server-side application.

2.1 Client-side application

The client application should take a command-line input for the username and the server's IP address (localhost or 127.0.0.1 if you are running the server locally), and then start with opening two TCP sockets to the server, one for sending messages to other users and one for receiving messages. When the sockets are opened, the client should first send REGISTER messages and read the acknowledgements.

The client will then need to start two threads. On one thread, it should read one line at a time from stdin, i.e. what the user types on the keyboard. Each time this thread reads a line, it should parse it to get the intended recipient and the message. Each line typed by the user should be of the form: @[recipient username] [message].

If the line is not in this format then the thread should reject the line and ask the user to type again. The thread should then send a SEND message to the server and wait for a response. Upon getting a successful response, it should inform the user that the message was delivered successfully, else convey an error to the user. The application can then loop back to waiting to read the next line typed by the user.

On the other thread, it should wait to read FORWARD messages from the server. Upon receiving a message successfully, it should send the appropriate response to the server, display the message to the user, and go back to waiting to receive more messages from the server.

2.2 Server-side application

The server application would begin with listening for connections on some port number. You can choose any port number. Upon receiving new socket acceptances from a client application, the server should spawn threads for each socket

to communicate with client. The threads will begin with expecting to receive REGISTER messages.

For the thread which receives a TORECV message, the thread should add the user and corresponding Socket object to a global Hashtable (or any other suitable data structure), and close the thread, but of course not the Socket.

For the thread which receives a TOSEND message, the thread should begin to wait to receive SEND messages from the client application. It will parse these messages, and upon receiving well-formed messages the thread should send a FORWARD message to the recipient on the recipient's Socket with which it registered through the TORECV message. This Socket will need to be looked up from the Hashtable being maintained by the server. The thread will then wait for a response, which it will convey back to the sender on the TOSEND Socket of the sender, and then begin to wait for more messages from the sender.

This should give a nice and simple chat application through which users can talk one-on-one with each other.

3 What to submit

You should program this assignment completely in either of C, C++, Java or Python. Submission folder should be a single .zip or .tar.gz file contain an src directory with the source code of your assignment and a README on how to execute the server and client applications, and a doc directory containing a report. The report should specifically contain answers to all the questions asked throughout the assignment document above. Submit a zip file containing the above mentioned files as <Entry-Number>.zip