COL334 Assignment-2

Shrey J. Patel, 2019CS10400

September 2021

1 Approach and Implementation

The entire approach and implementation has been highlighted in the code itself as detailed comments. Each and every function and non-trivial data structures have been accompanied by appropriate explanations.

2 Design Decisions

There are certain design considerations which I have taken to ensure continuity in terms of protocol implementation, exception handling, sockets and threading. Most of them are as below:

- I have used Java to implement the client-server model of the chat application. The main reason is that the constructs used for sockets and threading are very reliable and robust in terms of correctness checking and implementation. This is one of the primary reasons that I have been able to implement extensive error handling. Additionally, Java sockets and threading are known to take care of low level operations automatically without introducing too much programming overhead. And lastly, I preferred Java over C++ because of machine independence of Java bytecode.
- For client side implementation, I have created two different threads, one for the sending socket and other for the receiving socket, so that I can keep both the message flows independent and avoid concurrency management.
- However, the registration of both the sockets with the server is done before each of the threads are initiated. In other words, the registration messages are not sent in parallel, but in sequence to ensure that both the send and receive sockets are registered before any one of them starts dealing with the actual messages. Thus, in the beginning, this decision keeps both the sockets synchronised. So, a user can only type messages once both the threads have been registered.
- In server side implementation, if a particular client is successfully registered, then two separate sockets(send and receive) are created and stored in two separate global hash maps, both of which are indexed by the username of the client. In this way, the send and receive sockets of any client on the server are also kept separate, so that they can work independently, i.e. If a person A wants to send a message to some other person B, and at the same time a third person C wants to send a message to A, then both of these operations can be done simultaneously without stalling the other since the send and receive sockets have been intentionally kept separate.

- Just like the client side implementation, the send and receive sockets are also allotted to different threads so that all the clients can function in parallel. However, the major difference between client and server side implementations are in the threading of the receive socket. In client, each receive socket was allotted to a specific thread which kept running till the disconnection of the client, but in the server, the receive socket is allotted a thread only when a message is to be sent to that particular recipient, unlike the receive socket which keeps running until the client disconnects.

- So, the thread is spawned only when the receive socket is to be used to forward a message through the server. This is not significant in case of a Unicast message since only one thread would be active in that case. However, this decision is much more relevant in case of a Broadcast message, in which case multiple threads are spawned, one for each of the recipients registered to the server and all the messages are forwarded in parallel by the server through these multiple threads, thus saving a lot of memory and time.

- And when a message is delivered to and acknowledged by all the recipients, then all the threads are joined and closed simultaneously and once again all that remains are only the receive sockets without threads. This is beneficial for receiving sockets since they are invoked only when a particular message is to be forwarded, as opposed to send sockets, which have to be active all the time to look for any incoming message. Thus, we can save a significant amount of resources by assigning threads dynamically to the receive sockets.

- Thus, I have supported parallel broadcasting instead of Stop and Wait. So, instead of waiting for the response of one recipient and then sending the message to the other recipient, my implementation sends messages to all the recipients at once and receives responses from all of them. The broadcast is considered successful only if all of the recipients receive and acknowledge the message.

- All the message parsing in both the client and the server has been handled using a single global encoding function called parseResponse, which maps every type of message, whether it is allowed by the protocol or not, to an integer response code. Thus, all the operations on the messages (including error messages) can be done using these response codes.

- **ERROR Handling:** I have accounted for all the errors defined under the protocol definition. However, there are many other possible errors concerning the structure of the messages exchanged between the server and the client, and I have collectively classified them under ERROR 103 for uniformity. Some of the responses of the client and the server towards different error messages are:

    1. **ERROR 100: Server to Client** The client exits and prompts the user to use a valid username in the next attempt.
    2. **ERROR 101: Server to Client** The client exits since registration is mandatory for sending messages.
    3. **ERROR 102: Server to Client** This is the only error message which doesn't force the client to disconnect. This error is thrown in two scenarios. First, when the recipient is not registered to the server, and second when the message forwarded to the recipient is not acknowledged. In both cases, the user is prompted to try sending the message again.
    4. **ERROR 103: Server to Client** This error indicates that the client violates the protocol in sending the message and thus, the client is immediately disconnected and deregistered from the server.
    5. **ERROR 103: Client to Server** The server is never disconnected due to any error message, unless it is explicitly interrupted from the console. Any error message to the server arrives only during forwarding, and ERROR 103 arises because the message sent to the recipient doesn't follow the protocol. So, instead of disconnecting, the server reports failure to the sender in the form of ERROR 102.
    6. All other errors like the abrupt disconnection of server or client or other errors related to sockets or threads in general are handled using try catch blocks.