Project 1: Group Chat

Design Specification

v2.0.0

Dipunj Gupta
(dig44@pitt.edu)

Jan 30th, 2023

Problem Statement:

To design and implement a group chat service.

System Specification:

1. A client program that can join and leave chat groups and send messages in those groups.
2. A server program that is responsible for **storing messages**

Assumptions:

1. Server is 100% reliable or in other words – once the server is started, it won't stop abruptly due to external factors such as system failure / power outage etc.
2. There is no upper bound on the number of clients.
3. Clients can crash.
4. The network between server and client is unreliable and doesn't guarantee the inorder arrival of messages.

Service Specification:

The system provides the following services to a user:

1. **c <hostname | IP Address>** :connect to chat server.
2. **u <username>** : client logs into the server with identity **username.** What should be the behavior if another client instance is already using the "*username*"?
3. **j <chat_group>:**
    a. The "j" command can only be issued after the client has selected some username, what should be the behavior if "j" is invoked before the client program has selected a username?
    b. Upon successfully joining the *chat_group*, the following information is presented to the new user:
        i. All the users of the group who are online at the moment.
        ii. The 10 most recent messages in chronological order along with the number of unique likes for each of them.
    c. If there is any change in the above information(online users, unique likes, 10 recent messages), the screen should refresh to show the updated information
4. Once the user is in some ***chat_group***, the following commands can be issued by the user:
    a. **a <message>:** sends the message with content **message** to current **chat_group**, the message length is limited to 80 chars, and doesn't contain newline characters. What happens if the user tries to send more than 80 chars/newline chars?
    b. **j <new_chat_group>**: the user leaves the current *old_chat_group* and joins the group with the identifier **new_chat_group**. What if chat_group == new_chat_group??
    c. **u <new_username>:** client is exited from the current group, and switched to the user with identity **new_username**. What if username = new_username?
    d. **l <n>**: likes the message with id n, only if the message wasn't sent by the current user themselves. What if the user tries to like their own message ??
    e. **r <n>**: if the current user had liked the message with id n, then this command removes the like, What if the user tries to unlike a message that they haven't liked??
    f. **p** : print the full chat history for **chat_group**.

    I have additionally added the following command
    g. **c**: clear screen (works on some terminals)

NOTE: I missed the first "c" command in the project description that was supposed to be used to connect to the server. However, I instead provided the same functionality as a command line option, similar to the sample project files Prof. Babay provided us with. A client can connect to a specific server by invoking the client binary as follows:

```
./client -host <server_addr/ip> -port <server_port_number>
```

Design Decisions

For all the above open ended issues highlighted in red, I propose to resolve them as follows:

What should be the behavior if another client instance is already using the "*username*"?

- The system allows the same "username" to be logged in from multiple client instances.
- A username can be online in the same group from different clients.
- A username can be online in different groups from different clients.
- A username would be counted only once in the online user list for a particular group.
- The username would be removed from the online user list for a group only when that username is offline in **ALL** client instances.

What should be the behavior if "j" is invoked before the client program has selected a username?

The client program would show the user an error that they must select a username before they can join a group.

What happens if the user tries to send more than 80 chars/newline chars ?

The client application's UI **ensures** that the user cannot enter more than 80 characters, and any attempt to press enter/return key to input a newline character causes the message to be sent across in the current group.

What if chat_group == new_chat_group??

If the user tries to invoke the command **"j group_id"** , whilst being active in the group with "**group_id**", then the user is notified that they are already in the group with **"group_id".**

What if username == new_username??

If the user with identity **username** tries to invoke the command **"u username"**, then the user is notified that they are already the user with identity **username.**

What if the user tries to like their own message ??

The system notifies the user that they can't like their own message

What if the user tries to unlike a message they haven't liked ??

The system notifies the user that they can't un-like a message unless they have previously liked it.

Client-Server Interaction

We decided to use gRPC for the client and server communication. Even though in the initial design doc we thought of using raw TCP connection, using the argument that “Using RPC over a public internet connection wouldn't be optimal, as network latency could negatively impact its intended goal and prevent it from delivering the desired location transparency.” However, we concluded in favor of gRPC because we felt that local transparency is meaningful only to the developer and not the user of the application, because under the hood gRPC also uses TCP connections to interact with the remote server, but it provides a ready to use IDL that is much faster to implement.

Here is the brief service description:

```
service GroupChat {

  rpc CreateNewMessage(TextMessage) returns (Status) {}
  rpc UpdateReaction(Reaction) returns (Status) {}

  rpc SwitchUser(UserState) returns (Status) {}
  rpc SwitchGroup(UserState) returns (GroupDetails) {}

  rpc PrintGroupHistory(GroupName) returns (GroupHistory) {}
  rpc Subscribe(google.protobuf.Empty) returns (stream GroupDetails){}
}
```

We use the subscribe() rpc to maintain a long lived connection from the server to the client, this stream is used to broadcast group changes to the users who are part of the relevant group.

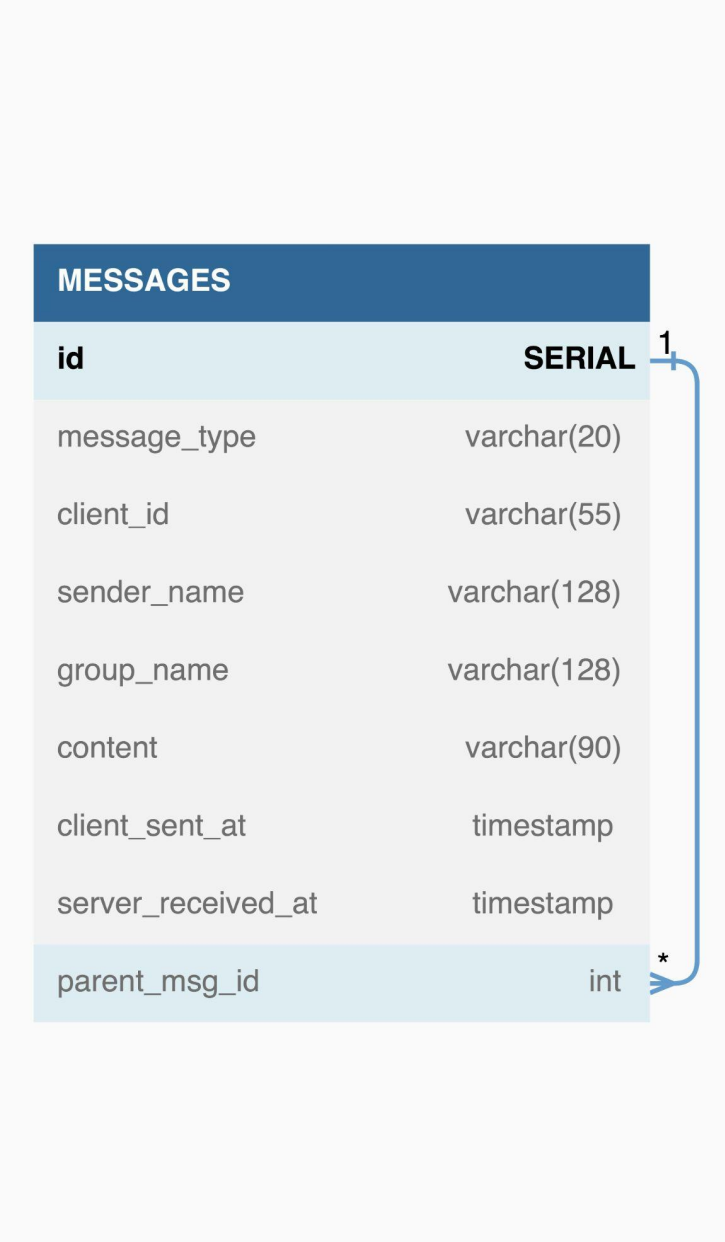
Whenever a group has new info (like someone liked a message, unlike a message, sent a new message, joined/left) we send the 10 recent messages along with the list of online members to each active member of the group over the rpc stream of Subscribe() function.

Failure Semantics

The system could fail due to the following reasons:

1. Network failure: The link between server and client could fail. The network could fail midway when:
    a. Transmission from server to the client: In this case the client won't receive the packet. The TCP protocol ensures that the server gets to know this and hence the server would try again until a predefined timeout. After which the server would declare the client as offline.
    b. Transmission from client to the server: The client first writes a message to its persistent local storage. Only on confirmation of the local write, a network call is made to the server over the TCP connection to send the message. We use the "at least once" semantic, because the client sends the messages with a unique id and hence the server can understand if the message is a duplicate request.
2. Client Failure: The client could crash, in this case the client won't be able to generate any new messages but the server might attempt to send it new ones. However, the corresponding TCP socket at server's end would timeout after a certain time, and as a consequence the server would declare the client as offline. In case the client revives before the timeout, the enqueued messages would be successfully received by the client albeit with a delay, because the TCP protocol would reattempt until the timeout is over.

Backend Data Model



- We ditched the user table because our server is reliable and hence we use an inmemory structure to keep track of all online users and their client ids and the group they are active in.
- Messages table stores all the messages that are sent between clients
- The table has a btree index on group_name so it is effectively faster than splitting the message of each group into a separate file, because the file system is not as optimized for searching data as a DBMS system.

Client Data Structures

The client needs the following data structures:

1. A queue of last 10 messages (with their likes and the associated metadata)
2. A key-value cache for all the messages Since a message cannot be deleted after it has been sent, the messages don't update, therefore we can cache all the messages.

High Level Logic

1. The database must be started and be healthy before the server is started.
2. The server must be started before a client.
3. A client instance creates a message payload and sends it to the server. The server then stores the message and relays it to the intended recipients.
4. In case the client is unreachable (due to network/client failure), grpc stream back to the client times out and In this case, the server treats the client as offline and removes it from the list of online users. Upon restoration of the client, the server sees the client joining the group as no different from a client explicitly joining a group and hence sends the group info with the 10 latest messages and list of online members.

Limitations / Drawbacks / Second Thoughts

1. In the proposed system architecture diagram, the server(and it's database) could be a single point of failure, but as per the system specification, the server is extremely reliable and hence unlikely to face a system failure.
2. The p command could cause the client application to crash/hang, because if a chat group has a big history (in terms of memory) the client might run out of space to hold all that data.
3. The UI isn't the best because if a user had typed some text, and in the meanwhile a rerender happens, then the user's typed data is present in the stdin however, due to the rerender the user might not feel that way.