Project 1: Group Chat

Design Specification

v1.0.0

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**.

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 a notification, 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 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

The client and server interact by passing JSON payloads to each other over TCP sockets. We choose TCP since the network between client and server is unreliable.

The client can send the following payloads to the server:

1. New Message (Text) due to **a** command
2. Message Update (Like/Unlike)
3. Switch User: { new_user_id }
4. Join Chat group: { group_name, user_id }

The server can send the following payloads to the client:

1. New Message (Text): a JSON object.
2. Message Update (like/unlike / delete):
3. Group Join Info: { online users: the active users in the current group, last_10_messages }

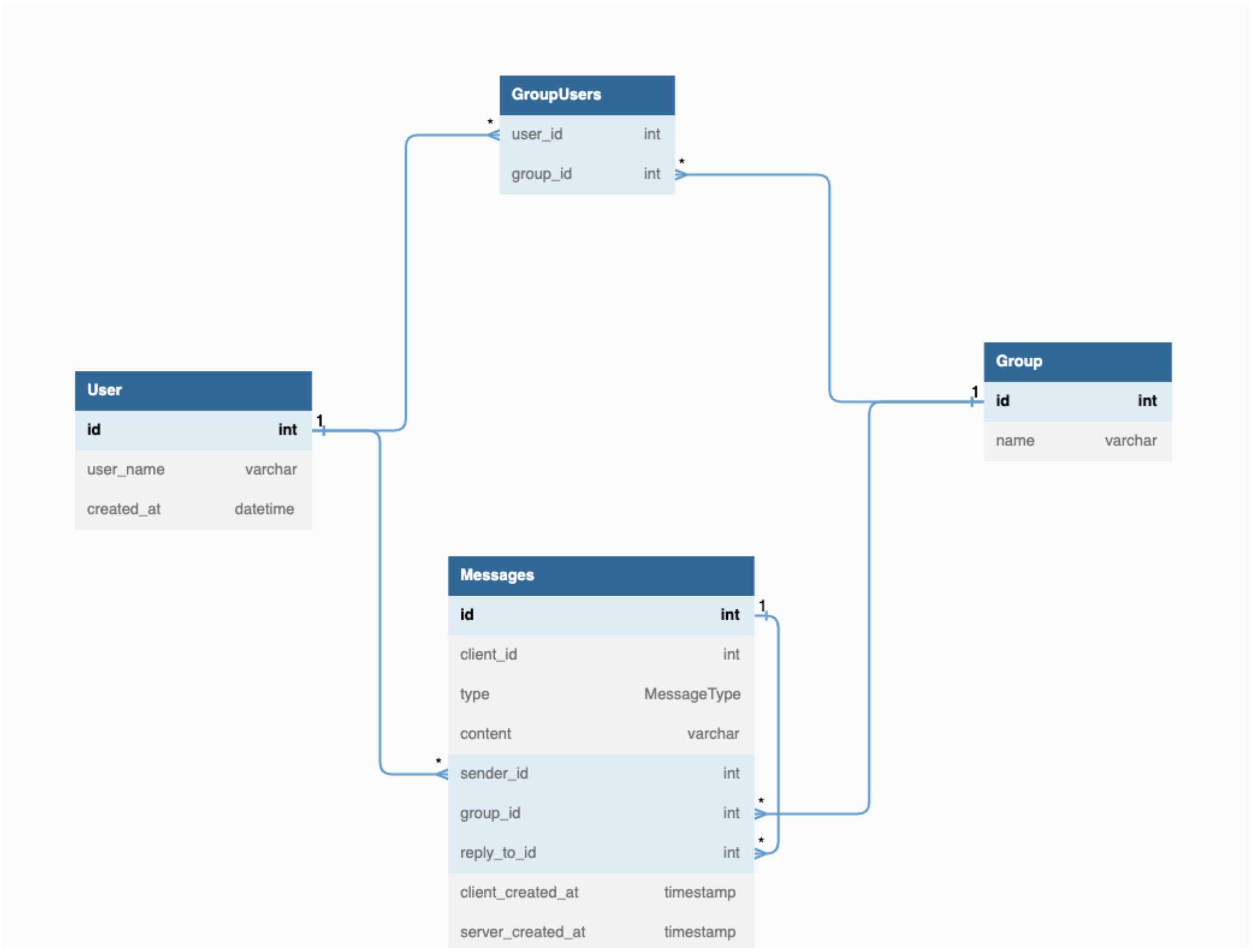
Using RPC over a public internet connection wouldn't be optimal, as network latency could negatively impact its intended goal and prevent it from delivering the desired location transparency..

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



- User table keeps track of all users
- Group table keeps track of all the groups that are created
- Messages table stores all the messages that are sent between clients
- The GroupUsers table keeps track of which user is present in which group at the moment. This table would be heavily updated if a lot of users are using the system, and they change groups frequently.

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 server/client can be started in any order, as soon as the server is online, the clients are able to successfully connect to the server.
2. 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.
3. In case the client is unreachable (due to network/client failure), the associated socket connection would timeout. 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. Choosing a socket timeout: If a client's presence fluctuates too much the consider the following situations:
   a. Timeout is too small: The server would remove and add the client repeatedly from it's server state and send the group info again and again to this client, this would also cause the online member count to incessantly change for all other users in the group – which might not be an ideal user experience.
   b. Timeout is too big: The client could be offline but the server would still wait long before updating its server state and hence for this duration everyone else in the group would have the wrong information about the offline client.
   c. Hence it's important to choose a right amount of timeout.
3. 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.