Project 2

Replicated Group Chat

Design Specification

Dipunj Gupta
DIG44@pitt.edu

James Tomko
JAT190@pitt.ed

March 19th, 2023

Note: Please use the Project 1 design doc submitted by Dipunj Gupta when we refer to Project 1. We have omitted those details to ensure that the following document doesn't mention redundant information that is already available in project 1 doc and it is easier to see what's new in contrast to Project 1.

Problem Statement

To design and implement a **replicated** group chat service that is

- Reliable
- Efficient
- Simple
- As consistent as possible for the given network conditions; and
- Available as long as at least one server is up and reachable.

System Specification

1. Any number of client program instances that can join and leave chat groups and send messages in those groups.
2. 5 instances of a server program that **store messages**

Assumptions

1. Server instances are not reliable and can crash at any time.
2. There is no upper bound on the number of clients.
3. Clients can crash at any time.
4. The network between server and client is unreliable and doesn't guarantee the in-order arrival of messages.
5. Clients have correct system clocks and they use NTP to correctly set their local clocks.

Service Specification

Same as project 1. One new command on client side

- **Command v**
    - print the view of the connected server.
    - Show the ID and IP address of the peer servers

Client-Server Interaction

Same as project 1. The interaction between server and client is unchanged.

Server-Server Interaction

- One of the core system requirements is that the chat service should be available even if just one server is online. Which means that at best, we can only make an effort to replicate the data across replicas, if that fails and the data is saved to the local disk of the server then we have to accept it as a successful write.
- Because clients decide which server they want to connect to, therefore all the servers must be ready with all the data for any group and any user – hence we cannot shard our data across replicas.
- **Replication Protocol**: Hence the replication protocol is to copy-on-write (COW) any message to every visible available replica on a best effort basis.
- We use **server-side vector clocks** to establish ordering between messages. Using vector clocks keeps the system very simple, and satisfies the problem constraints.

Two important scenarios:

1. What happens when a client sends a new message to server A:

   The server A writes that message to its own database and forwards it to all the online replicas that it can see.
2. What happens when a network partition heals:

   The system would be brought to sync. Both the server of the new edge in the network graph would send the latest vector clock that they have and the other server would send all the messages that are newer than that timestamp. The ordering of messages would be done by ordering the vector timestamps. Therefore merging the states of two partitions is straightforward, the order of timestamps dictates the order of messages.

Software Structure

The software structure is similar to project 1 with additional logic for server-server interaction.

The server consists of two servers:

1. Port 12000: the public server, this handles communication between server and client
2. Port 11000: the internal server, this handles communication between server and other replicas.

The directory structure is as follows:

```
├── docs/
│   ├── final_design.pdf
│   └── initial_design.pdf
└── src/
    ├── client/
    │   ├── client.go
    │   ├── network/
    │   ├── state/
    │   └── ui/
    ├── go.mod
    ├── go.sum
    ├── pb/
    │   ├── comm.pb.go
    │   ├── comm.proto
    │   └── comm_grpc.pb.go
    └── server/
        ├── db/
        ├── network/
        └── server.go
```

Backend Data Model

Same as Project 1. We have just added the field called **vector_ts –** an array of size 5 to store the corresponding vector timestamp of each message.

Client Data Structures

Same as project 1

High Level Logic

1. Each replica maintains a local copy of the chat history and a list of active clients.
2. When a client sends a request to a server, the server increments it's vector clock and appends the message to its local copy of the chat history and broadcasts the timestamped request to all other replicas.
3. Each replica receives the broadcasted request and adds it to its local copy of the chat history.
4. The replicas then use the vector timestamp of the messages, to establish the order in which the messages should be applied to the chat history. This ensures that all replicas have an identical copy and ordering of the chat history.
5. If a replica crashes, the other replicas can continue to process client requests, while also attempting to connect to the crashed replica every 1 second. When the replica crashes any immediate users of the crashed replica are marked as offline on all servers. All the users who were connected to the replica are immediately logged out and notified about the loss of connection.
6. When the crashed replica recovers, the connection is reestablished and its SyncReplica() function is called, which synchronises the crashed replica and the server. Both these parties send each other their vector timestamp of the latest message that they know of, and the other replicas all messages that are newer than that timestamp are exchanged.
7. Each replica server uses a postgres database that provides durability for fail stop failures using a WAL, which can be used to recover the replica to its previous most consistent in case of a crash.

Failure Semantics

The system could fail due to the following reasons:

1. <u>Network failure:</u>
    a. The link between server and client could fail – same reasons as project 1.
    b. The link between server and peer/replica could fail.
        i. If the network failure is ephemeral, TCP would attempt a retransmit and alleviate any problems due to dropped messages.

    ii. If the network failure is long-lived, then the peer replica is considered to be in another network partition. When the partition heals, the “visible” server count for both the servers would increase and they will exchange and merge their histories to bring each other up to date.
2. Client Failure: Same as project 1.
3. Server Failure: When the server crashes, any client connected to it is notified of the connection loss and is disconnected. All the peer servers update their list of active users (we keep a hash map between which user is active on which server ID on all servers)

Limitations

1. 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.
2. 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.
3. If a message X was received by a server A (from a client), and A went offline before it could replicate X to other servers, then as long as A is offline, users wouldn’t be able to see X. However, once A recovers, message X will be replicated to other servers. (eventually consistent if the crashed replica eventually comes up online)