CSE431: Distributed Systems

Major Project Spring 2021

Team Name: **Warriors**

Team Number: **27**

Team Members:

1. **Shubham Agrawal** (2019201085)
2. **Shanu Gandhi** (2019201014)
3. **Shobhit Gautam** (2019201056)
4. **Harshit Pandey** (2019201088)
5. **Yash Upadhyay** (2019201098)

Mentor: **Mayank Musaddi**

Project Topic:

Distributed storage schema for messaging application

Github Link:
https://github.com/ShubhamAgrawal-13/Distributed-storage-schema-for-messaging-application

Video Link:
https://drive.google.com/file/d/1qQe7kUbTAg47ecE6AOms-Nn_rA4FGoMK/view?usp=sharing

Tables of Contents:

1. Introduction

Our project's aim is to make a messaging application which supports group chats, updating and deleting messages with larger scope of scaling it to a high number of users thus the scope of scalability.

2. Objective

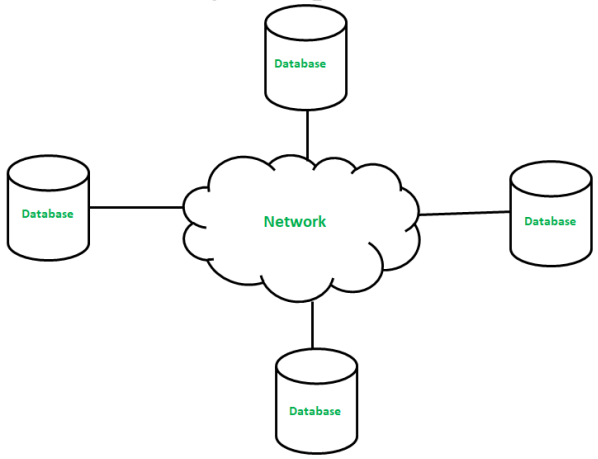
A distributed storage schema for messaging application:

1. Should support group chat
2. Deleting/ Updating messages
3. Scalable

3. Terminologies:

3.1 Distributed database

A Distributed database is a database that consists of two or more files located in different sites either on the same network or on entirely different networks. Portions of the database are stored in multiple physical locations and processing is distributed among multiple database nodes.



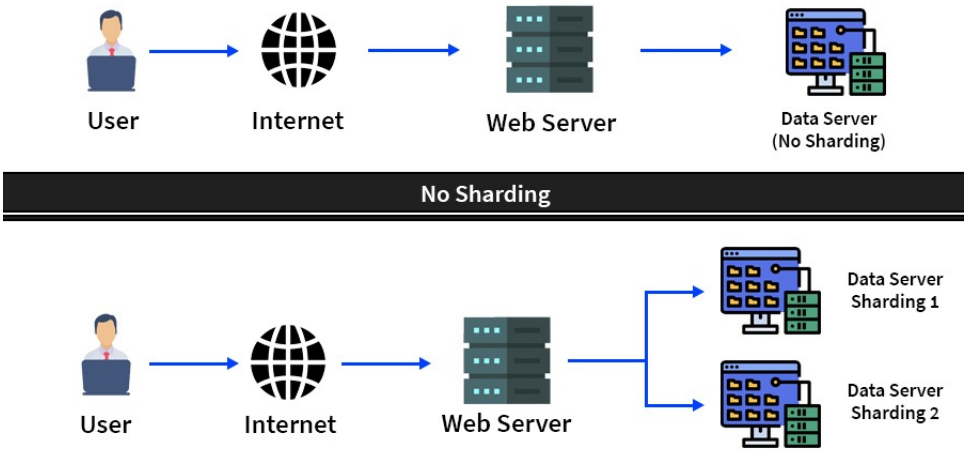
3.2 Replication

Replication as a concept is much like a subset of redundancy. Both involve creating redundant nodes in a system *replication* takes a redundant node one step further; it ensures that the redundant node (a replica) is identical to all its other copies.

3.3 Sharding

A database shard, or simply a shard, is a horizontal partition of data in a database or search engine. Each shard is held on a separate database server for instance, to spread load. Some data within a database remains present in all shards, but some appear only in a single shard.

Below image shows how sharding partitions database.



Below image shows a practical example of sharding based on delivery zone.

4. Technologies:

4.1 Python3

We have used python3.

4.2 Flask Web Framework

Flask is considered more Pythonic than the Django web framework because in common situations the equivalent Flask web application is more explicit. Flask is also easy to get started with as a beginner because there is little boilerplate code for getting a simple app up and running.

4.3 Apache Kafka

Apache Kafka is a framework implementation of a software bus using stream-processing. It is an open-source software platform developed by the Apache Software Foundation written in Scala and Java. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds.

4.4 MongoDB

It is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas. MongoDB is developed by MongoDB Inc.

5. Overview of the Functionalities:

1. Our Project has a Messaging UI through which multiple users can ***register and login*** to start chatting, once registered, a user can send a message to any user in the network and also send messages on his network groups. The sent messages can be changed or ***updated*** if the user wants it to, as well as the user can ***delete*** any sent message.

2. The user info and messages are stored on Distributed Database. The database is replicated to multiple servers and each server does perform ***sharding*** on its own which eventually gives faster access to database entries.

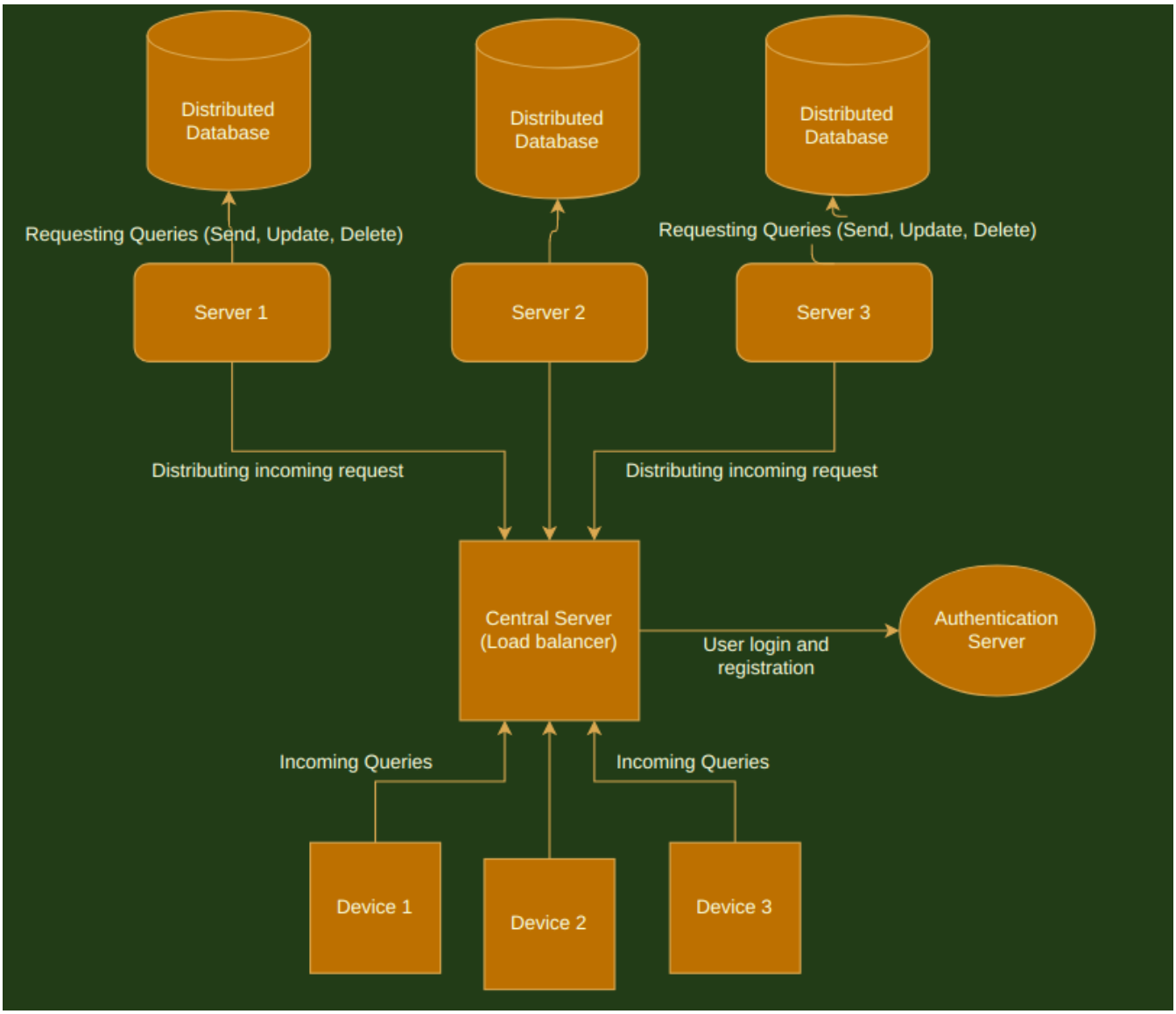
3. The **data integrity** is maintained by active or online servers which have command over each database and communicate through each other thus ensuring deletion and updating of messages.

4. The **load Balancer** is a central server which manages loads of servers and user

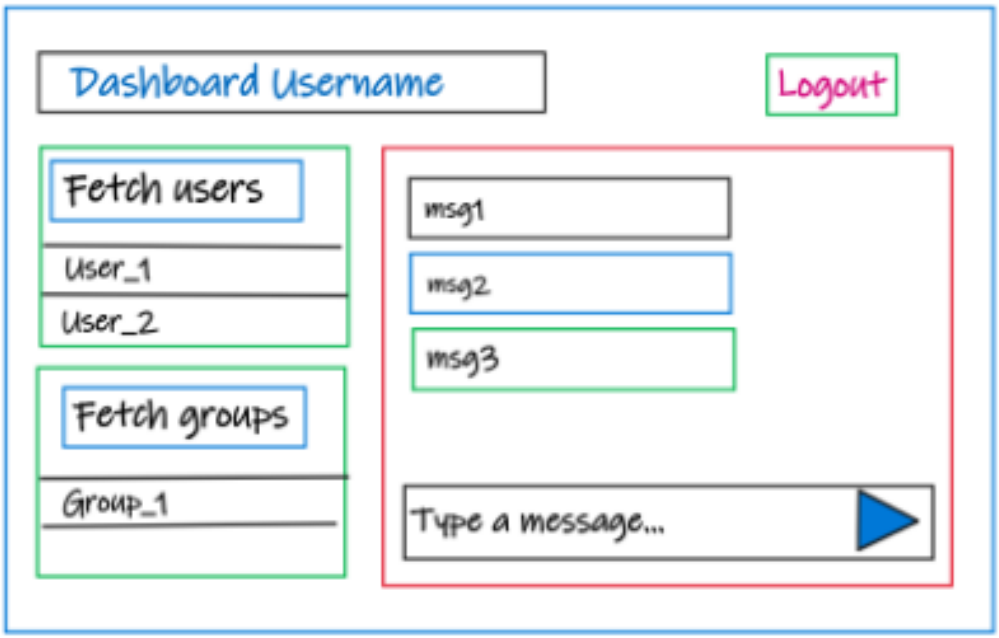
requests and sends it to the server as per its algorithm to minimize load. It also manages user authentication as well as registration of applications.

6. Big Picture:

Here in the Big picture, we have shown how our end-to-end system will work. Here the user will first **register/login** itself to the system which will be done via central server to authentication server. Now users will be allowed to query databases in whatever way he wants.



This is the design of initial dashboard that we thought of:



7. Modules:

1. **User:** he/she who is using our platform will be given a url (***https:ip:port/home***),with this user can use the services of our messaging platform like chatting to other users or in groups.

2. **Load Balancer:** It is responsible for handling user requests and giving it to the server, it decides which request to give to which server and if load is more on a single server, then transfer requests to other servers.

3. **Authentication Database Server:** It is maintained by a loaded Balancer server that registers new users and checks validity of existing users.

4. **Active Servers:** It handles requests sent by load balancers and acts accordingly. Ex: adding message to chat, deleting, updating message. It also messages other servers about its operation and updates other databases to maintain integrity.

5. **Distributed Database:** It is a database that is replicated and stored on different servers, when a new request comes in, data is fetched mainly chats and then stored back on the database by servers.

6. **Dashboard:** It is a UI which is viewed by users for chat related operations. The initial wireframe of Dashboard which we finalised is as follows.

7. **Sharding:** We divided the database on the basis of user name. Here for instance tables with names A to B (2 characters) are stored in one fragment, C to F (4 Characters) in another fragment. Interesting fact here is that we have given ratio to each character in the alphabet that is since it is obvious that A will appear in more names so we have given it a lesser ratio whereas for some other character which will appear less in name (like X) we give it a higher ratio.

8. Services:

1. Registration:

It registers users to Application for services.

Semantics of Registration Service:

User_id	Email	Password
123	abc@gmail.com	*****

Config JSON of Registration:

1. Request Part:

```
{
"uid":"user1",
"email":"abc@gmail.com",
"password":"12345"
}
```

2. Response Part:

```
{
"uid":"user1",
"ack":"ok"
}
```

From User End, register with ID, password.

2. Login:

User can login to Application, after logging in, dashboard will open. Users can perform operations now.

User_id	Email	Password
123	abc@gmail.com	*****

Config JSON of Login:

1. Request Part:
   {
   "uid":"user1",
   "password":"12345"
   }

2. Response Part:
   {
   "uid":"user1",
   "ack":"ok"
   }

3. Send Message:

1. **User Side**: send (userid1_userid2_msg) here user2 is receiver/group name from user Side.
2. **At database level:**

Database_id	Conversation_id	Group_Owner_Id	Cretaion_Time_Stamp
		-1 (simple chat bw 2)	
		1234 (else group)	

Table 1: CONV ID1 (user1_user2) For Two user Chat

Messsage_id	sender_user_id	Message_text	Time_Stamp

Table 2: CONV ID2 (group1) For Group Chat

Messsage_id	sender_user_id	Message_text	Time_stamp

3. **Load Balancer Level:** When msg comes apply round robin to select server and send request with corresponding operation.

4. **At Server Level:** Selected server will identify the sender and receiver. Then update the receiver with msg. Apply sharding and parallelly update all databases from the same server.

5. **Config of Send Message:**

Request Part:

```
{
    "uid1":"user1",
    "uid2":"user2",
    "msg":"text"
}
```

Response Part:

```
{
    "ack":"ok",
    "msg_id":"123",
    "timestamp":"123456787",
    "uid1":"user1",
    "uid2":"user2",
    "msg":"text"
}
```

4. Update Message:

For updating messages sent by the authenticated user.

Syntax: Fetch Messages to get msgid, Update(msgid, updated msg, load balancer).

Json:

Request Part:

```
{
    "op_type":"update",
    "msg_id":"1223",
    "isGroup":"true",
    "uid1":"user1",
    "uid2":"group1"
}
```

Response Part:

```
{
    "uid":"user1",
    "ack":"ok"
}
```

5. Delete Message:

For deleting messages sent by the authenticated user.

Syntax: Fetch Messages to get msg id, Delete(msgid, load balancer) .

Json:

Request Part:

```
{
    "op_type":"delete",
    "isGroup":"false",
    "uid1":"user1",
    "uid2":"user2",
    "msg_id":"1223",
}
```

Response Part:

```
{
    "uid":"user1",
    "ack":"ok"
}
```

6. Fetch Message:

Fetch Message Json:

Request Part

```
{
"op_type":"delete",
"uid1":"user1",
"uid2":"group1",
"isGroup":"true",
}
```

Response Part:

```
{
"uid":"user1",
"msgs":{
        "uid2":"user1",
        "msg":"text",
        "timestamp":"123456787654",
        "msg_id":"123"
}
}
```

7. Fetch User:

Fetch User Json:

Request Part

```
{
"op_type":"fetch_user",
"isGroup":"false",
"uid1":"user1",
"uid2":"user2",
"msg_id":"1223",
}
```

Response Part:

```
{
"uid":"user1",
"ack":"ok"
}
```

```
{
"uid":"user1",
"users":["user1", "user2", "user3"]
}
```

8. Fetch Group:

Fetch Group Json:

Request Part

```
{
"isGroup":"true",
"uid":"user1"
}
```

Response Part

```
{
"uid":"user1",
"groups":["group1", "group2", "group3"]
}
```

9. Logout:

For logging out users from the application. It can be done by clicking the button in the UI.

Syntax: Logout (0, flask server, userid)
Login (1, flask server, userid)

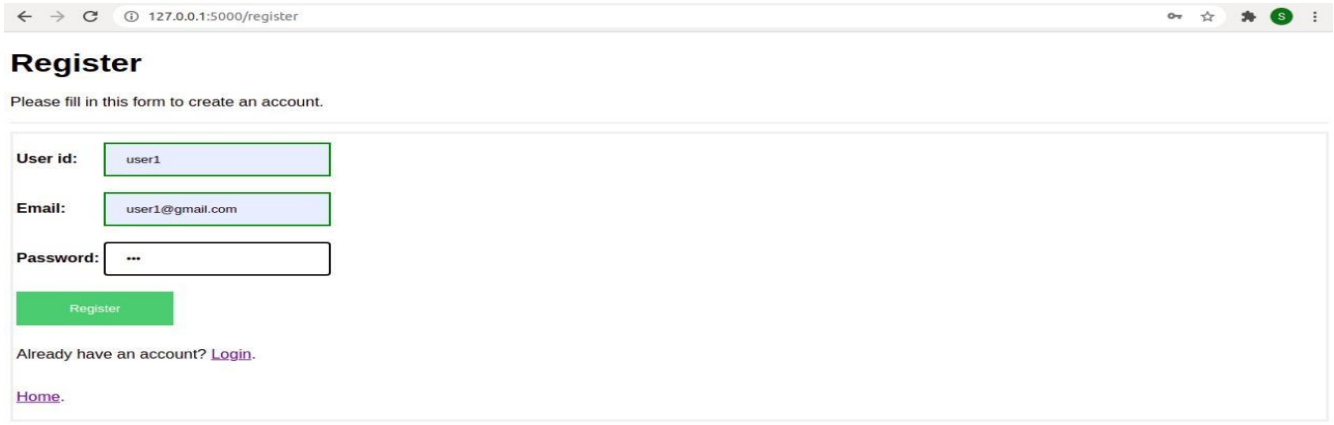
9. Scalability Aspect:

In this model we developed and discussed above, we bring increasing scalability by two ways, first, we are having multiple servers online and active simultaneously by which we can do load balancing of multiple queries coming on the network. Secondly, we have done sharding on the database we stored. Here sharding will partition the database into smaller fragments which flash server will further help us to reduce query time and make access to data faster which eventually help us to handle many requests.

10. Use Cases:

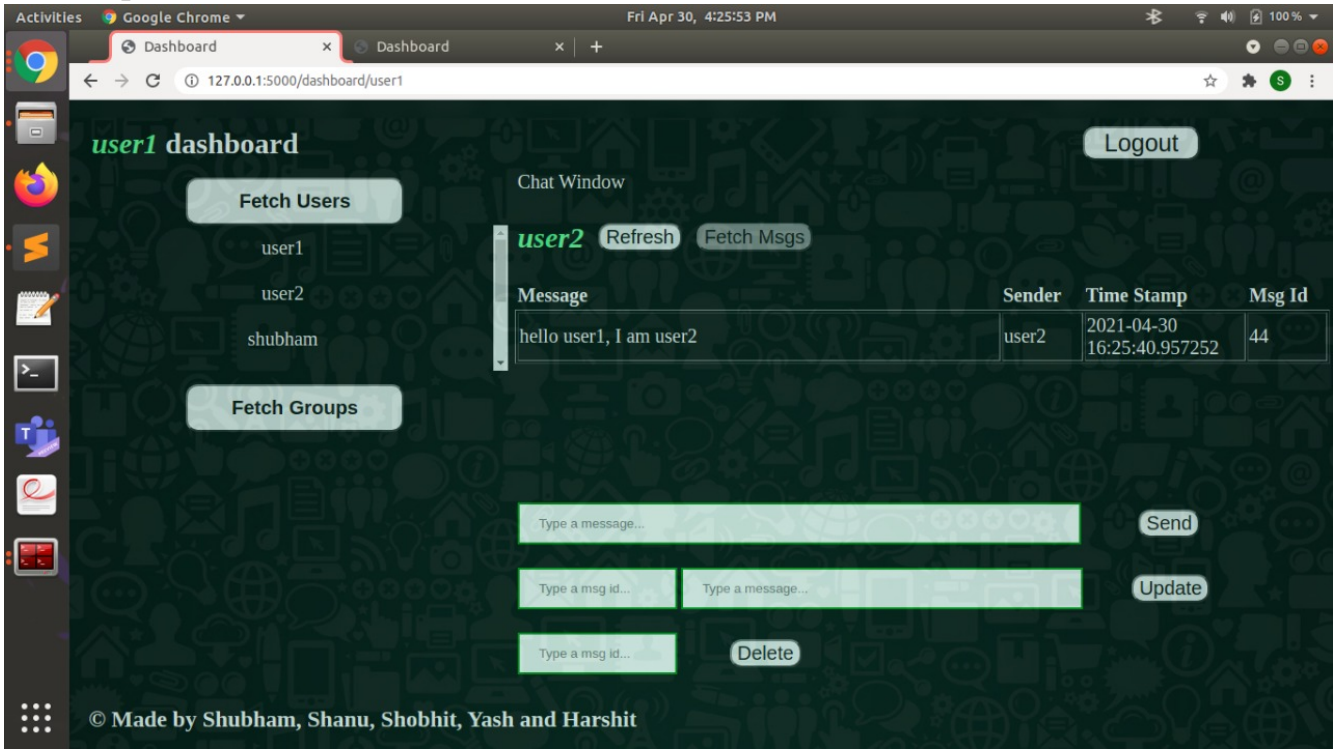
Below we have pictorially explained all the possible scenarios, of all kinds of queries that can happen in this messaging system.

Scenario 0 for Authentication:

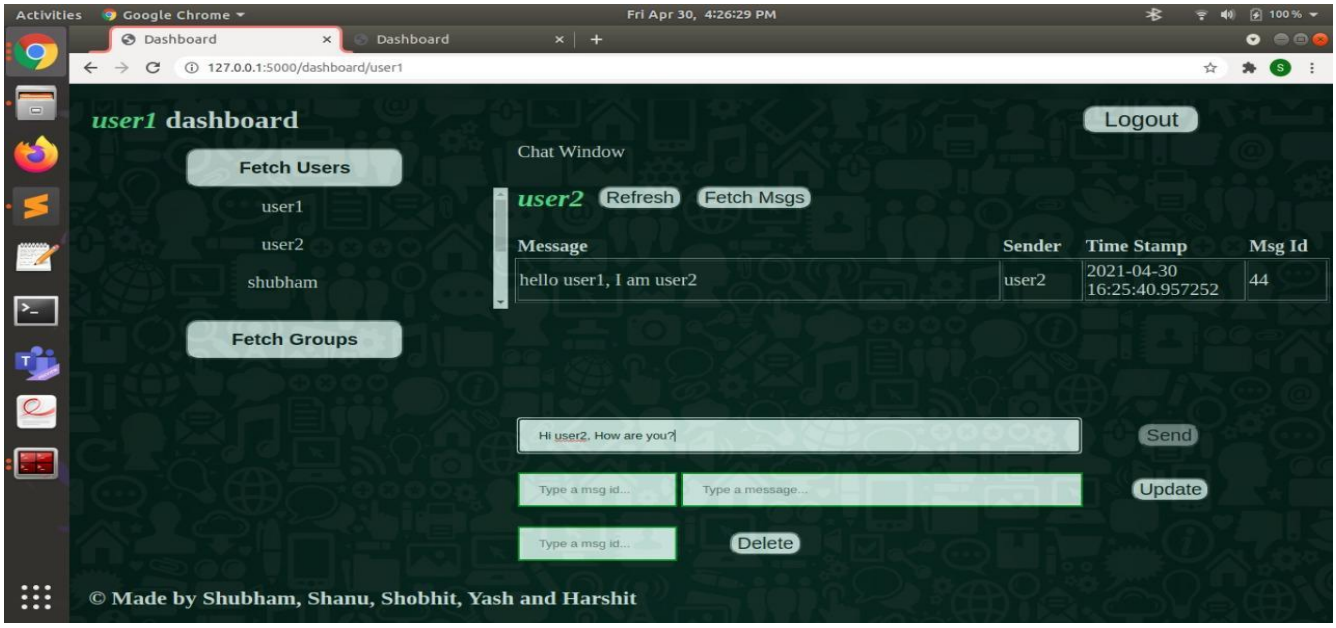


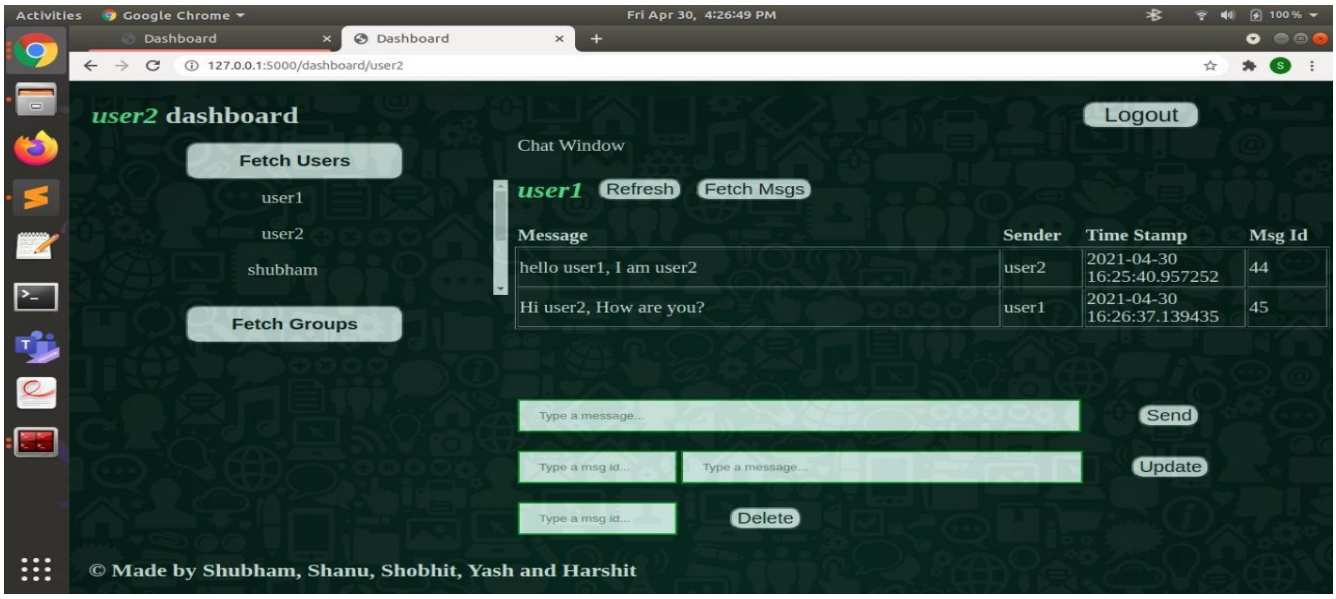
Scenario 1 for Fetching message from user1 to user2:

To load previous chats between users.

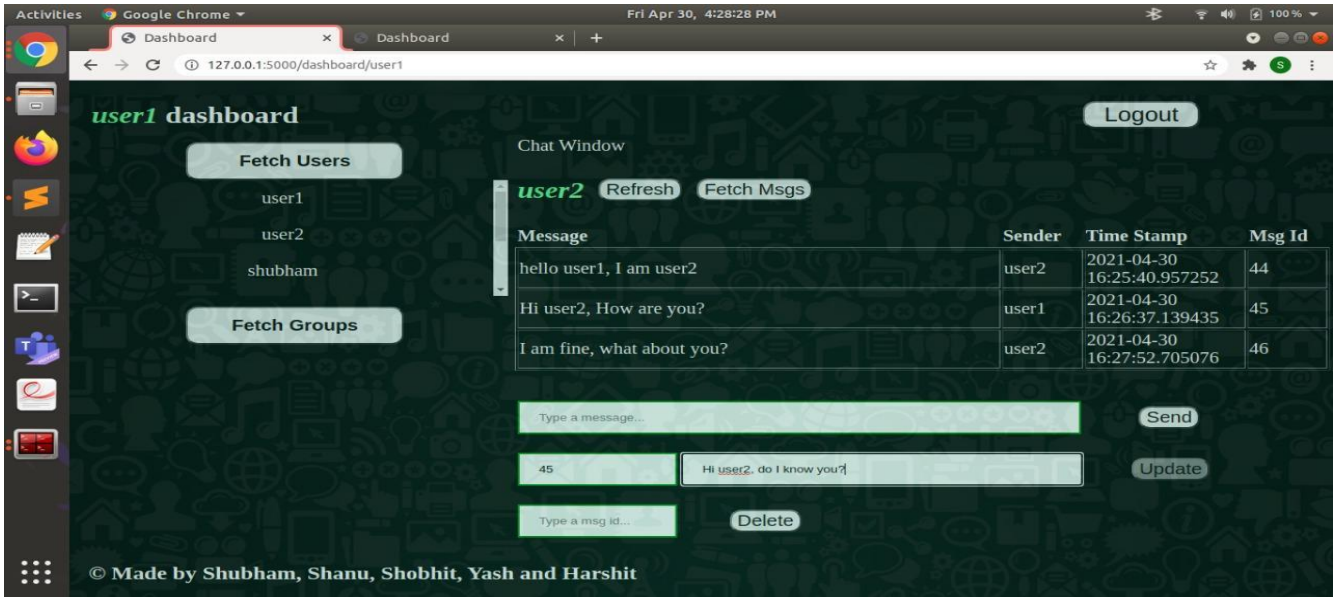


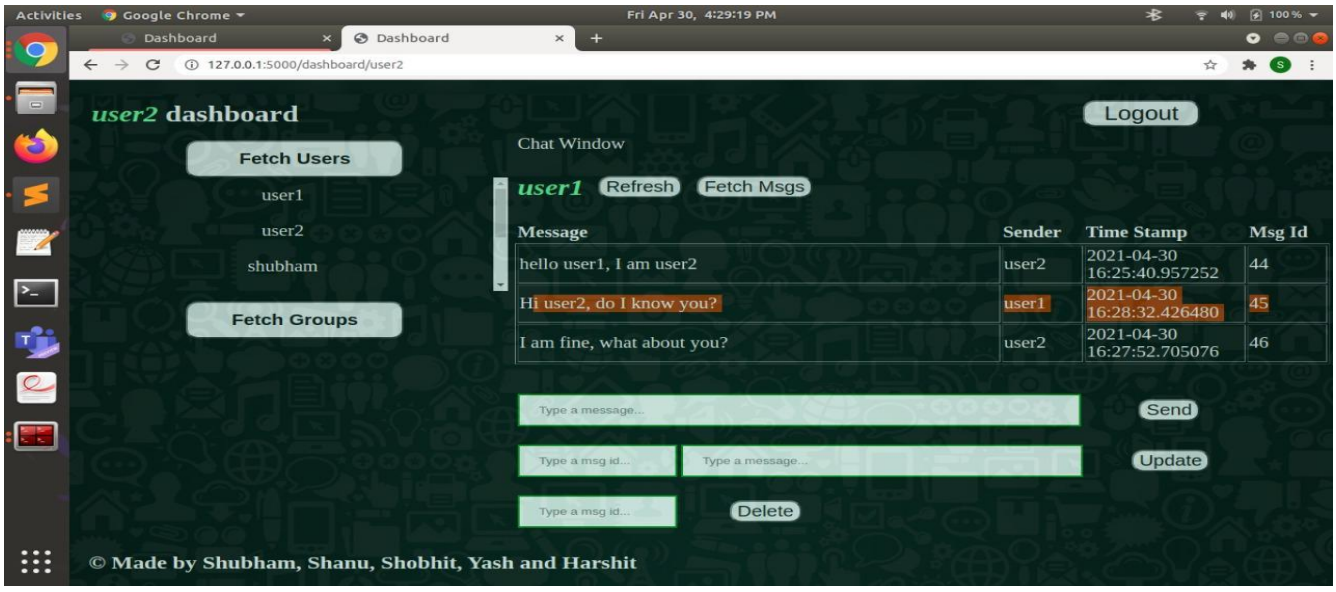
Scenario 2 for Sending message from user1 to user2:





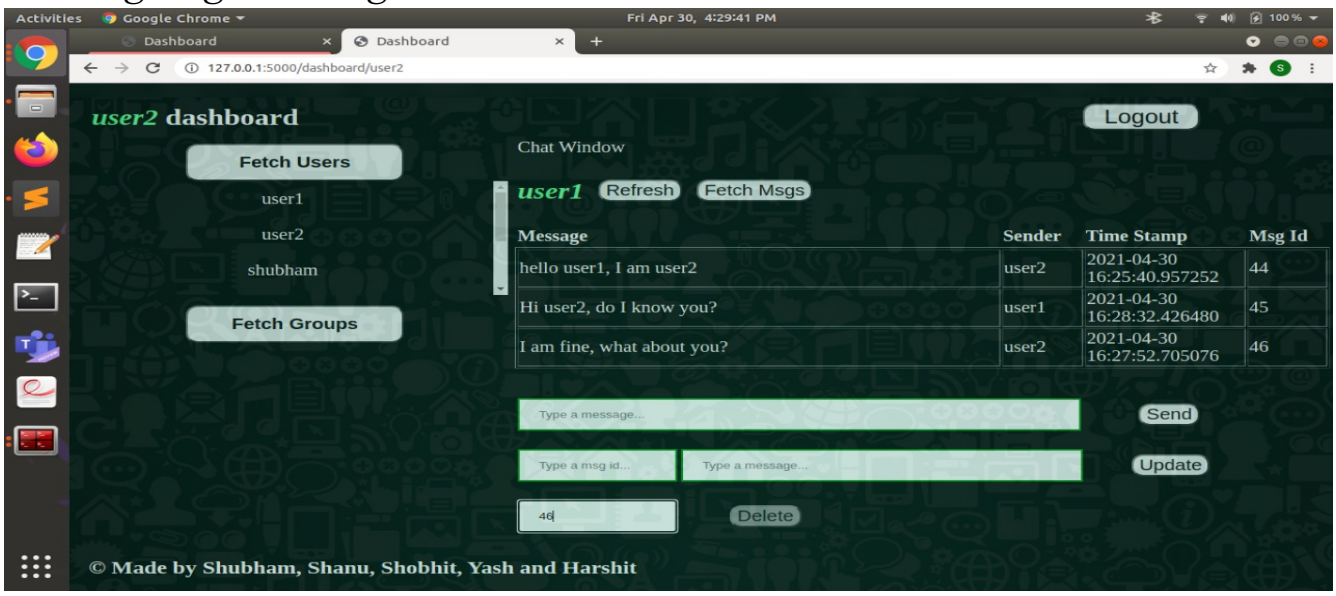
Scenario 3 for Updating message sent earlier from user1 to user2:

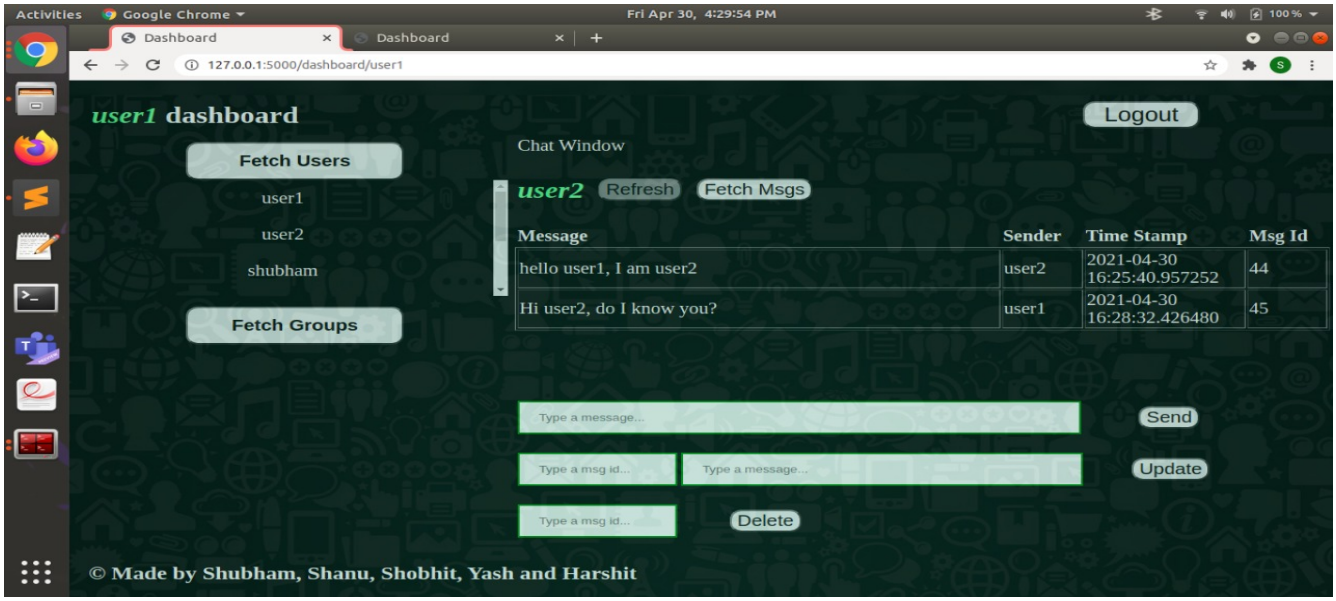




Scenario 4 for Deleting message sent earlier from user2 to user1:

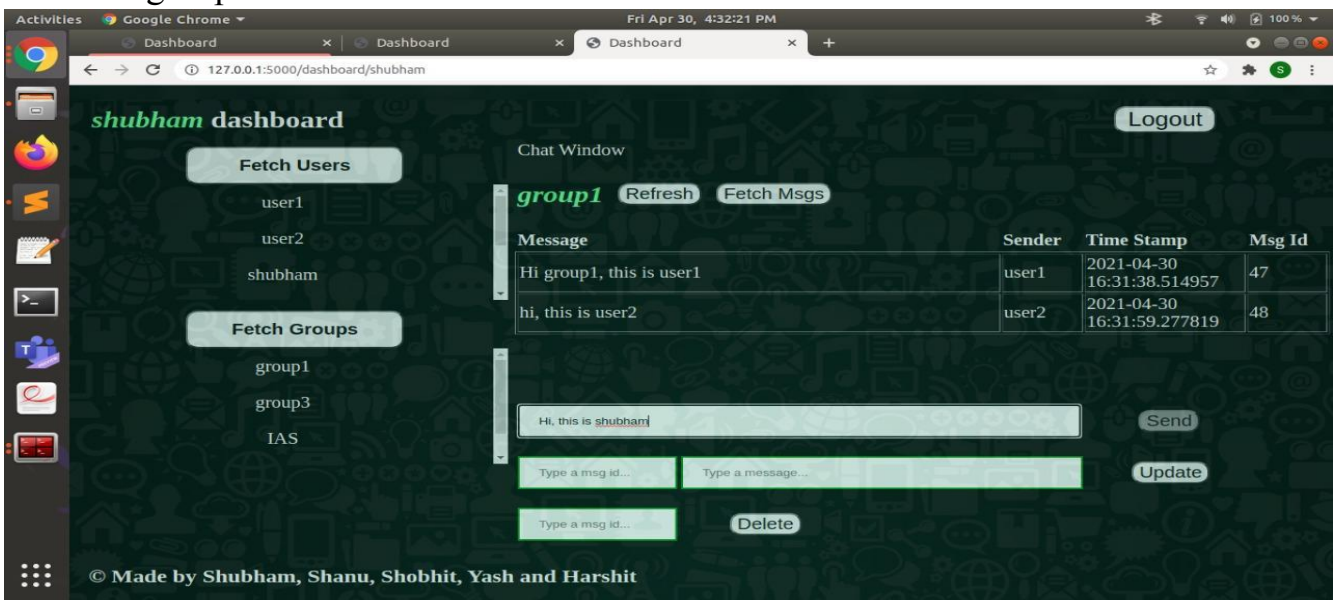
Deleting msg with msg id 46





Scenario 5 for Broadcasting message from user1 to group1:

where group1 is list of people whom user wants to broadcast, in other words, user1 will send it to group



11. References:

- **Apache Kafka:** https://kafka.apache.org/intro
- **MongoDB** : https://www.tutorialspoint.com/mongodb/index.htm
- **Distributed Database**: https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_databases.htm
- **Sharding:** https://www.digitalocean.com/community/tutorials/understanding-database-sharding
- **Messaging App Schema** : https://www.simform.com/how-to-build-messaging-app-whatsapp-telegram-slack/
- **Distributed DBMS Schema**: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/6677/667713/Distributed-database-schema/10.1117/12.734802.short?SSO=1

- **Load Balancer :**
  https://www.educative.io/courses/grokking-the-system-design-interview/3jEwl04BL7Q

- **Scalability** :
  https://medium.com/swlh/building-scalable-distributed-systems-part-1-introduction-to-scalable-systems-9ca471fd77d7#:~:text=Scalability%20Basic%20Design%20Principles,known%20as%20the%20system's%20throughput

12. Video and Code Links:

1. Github Link:

https://github.com/ShubhamAgrawal-13/Distributed-storage-schema-for-messaging-application

2. Video Link:

https://drive.google.com/file/d/1qQe7kUbTAg47ecE6AOms-Nn_rA4FGoMK/view?usp=sharing