EXPERIMENT NO: 5

Student Name: Ashmit Agrawal	**UID:** 23BCS11854
Branch: BE-CSE	**Section/Group:** KRG-1A
Semester: 6th	**Date of Performance:** 23/02/2026
Subject Name: System Design	**Subject Code:** 23CSH-314

Aim
to create a scalable real-time messaging program that is akin to Facebook Messenger or WhatsApp and that facilitates group and one-to-one messaging with dependable message delivery, low latency, and high availability.

Objective
• Recognize a real-time messaging system's architecture.
• Determine the functional needs for things like media sharing, chat, and authentication.
• Determine non-functional needs, such as latency, availability, and reliability restrictions.
• Examine the trade-offs for messaging systems in the CAP theorem.
• Create WebSocket and REST APIs to facilitate chat.

Procedure

1. Studied real-world messaging systems like WhatsApp and Facebook Messenger.
2. Identified core entities such as Users, Groups, and Messages.
3. Analyzed real-time communication using WebSockets.
4. Collected functional and non-functional requirements.
5. Designed APIs for user authentication and messaging.
6. Evaluated storage and scalability requirements for large-scale systems.
7. Analyzed availability and eventual consistency trade-offs.

Functional Requirements
1. The client ought to be able to sign up and access the system.
2. One-to-one messaging should be supported by the system.
3. Group communications should be supported by the system.
4. Both text and media message sending should be supported by the system.
5. The entire communication history should be preserved by the system.
6. The client should be able to view and read receipts for deliveries.
7. Real-time message reception is required.

Core Entities of the System

- Users
- Groups

- Messages

API Design

1. **User Registration API**: POST /user/register
2. **User Login API**: POST /user/login

One-to-One Messaging

A. Send Message (WebSocket): WS /messages/send
B. Get Chat List: GET /chat/{user_id}(Pagination Supported)
C. Get Messages of Specific Chat: GET /messages/{user_id}/{receiver_id}

Group Messaging

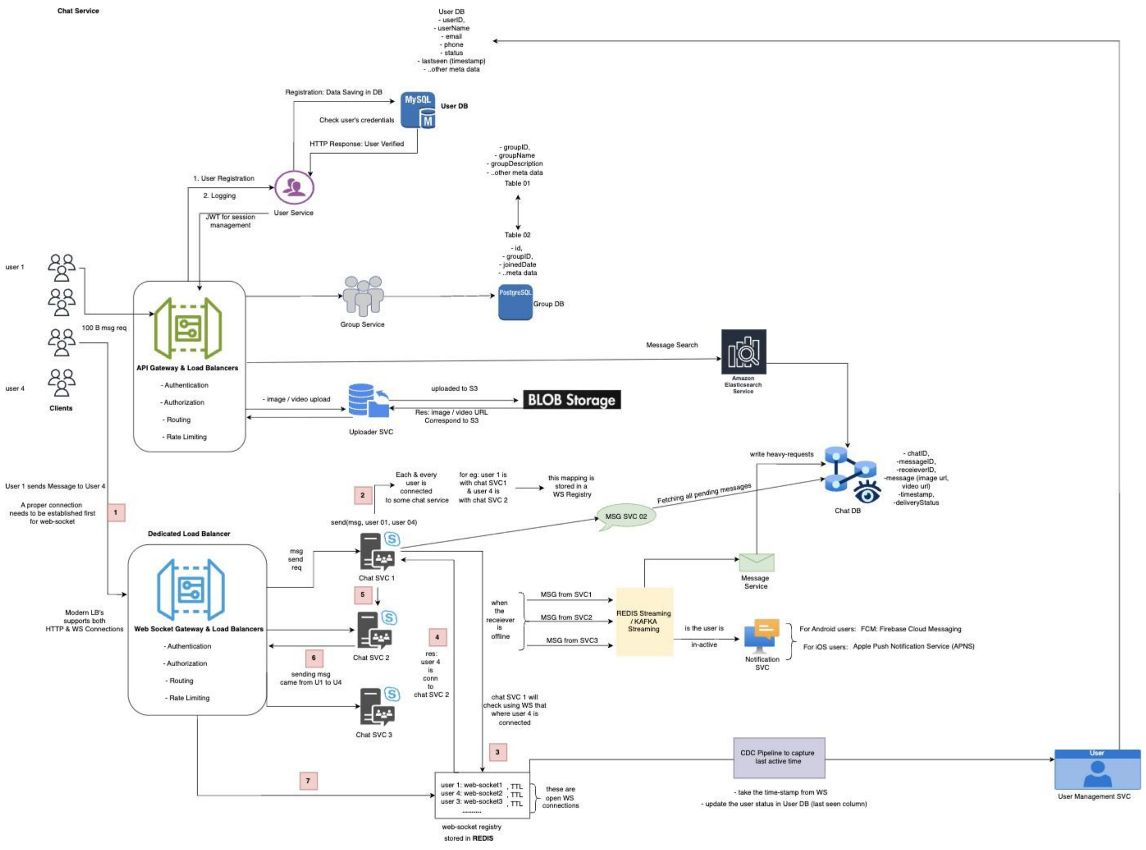
A. Create Group: POST /groups/create
B. Send Group Message (WebSocket): WS /messages/send
C. Get Group Chats: GET /groups/{group_id}(Pagination Supported)
D. Add Member: POST /groups/{group_id}/add
E. Remove Member: DELETE /groups/{group_id}/remove

Non-Functional Requirements

1. System should handle extremely large scale traffic (millions to billions of messages per day) requiring approximately 100 TB storage.
2. High Availability should be prioritized under CAP theorem.
3. System should follow Eventual Consistency model.
4. End-to-end latency should be approximately 200–300 ms.
5. System should be highly reliable with no message loss or packet drop.
6. Horizontally scalable distributed architecture.

High Level Design (HLD)

The system consists of Client Applications (Mobile/Web), API Gateway, Authentication Service, Messaging Service (WebSocket Servers), Group Service, Media Service, Message Queue, Distributed Databases, Cache Layer (Redis), and Object Storage for media files. Messages are delivered through WebSocket servers for real-time communication and stored in distributed databases for persistence.


Chat Service
User DB
- userID,
- userName
- email
- phone
- status
- lastseen (timestamp)
- ..other meta data
Registration: Data Saving in DB
MySQL
User DB
Check user's credentials
HTTP Response: User Verified
1. User Registration
2. Logging
JWT for session management
User Service
- groupID,
- groupName
- groupDescription
- ..other meta data
Table 01
Table 02
- id,
- groupID,
- joinedDate
- ..meta data
PostgreSQL
Group DB
Group Service
user 1
user 4
Clients
100 B msg req
API Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Message Search
Amazon Elasticsearch Service
- image / video upload
uploaded to S3
BLOB Storage
Res: image / video URL Correspond to S3
Uploader SVC
write heavy-requests
- chatID,
-messageID,
-receieverID,
-message (image url, video url)
-timestamp,
-deliveryStatus
Chat DB
User 1 sends Message to User 4
A proper connection needs to be established first for web-socket
1
2
Each & every user is connected to some chat service
for eg: user 1 is with chat SVC1 & user 4 is with chat SVC 2
this mapping is stored in a WS Registry
Fetching all pending messages
MSG SVC 02
send(msg, user 01, user 04)
Dedicated Load Balancer
msg send req
Chat SVC 1
5
Modern LB's supports both HTTP & WS Connections
Web Socket Gateway & Load Balancers
- Authentication
- Authorization
- Routing
- Rate Limiting
Chat SVC 2
6
sending msg came from U1 to U4
4
res: user 4 is conn to chat SVC 2
Chat SVC 3
when the receiever is offline
MSG from SVC1
MSG from SVC2
MSG from SVC3
REDIS Streaming / KAFKA Streaming
Message Service
is the user is in-active
Notification SVC
For Android users: FCM: Firebase Cloud Messaging
For iOS users: Apple Push Notification Service (APNS)
chat SVC 1 will check using WS that where user 4 is connected
3
7
user 1: web-socket1, TTL
user 4: web-socket2, TTL
user 3: web-socket3, TTL
these are open WS connections
web-socket registry stored in REDIS
CDC Pipeline to capture last active time
- take the time-stamp from WS
- update the user status in User DB (last seen column)
User
User Management SVC

Learning Outcomes

- Designed a scalable real-time messaging system.
- Identified functional and non-functional requirements.
- Designed REST and WebSocket APIs.
- Understood high availability and eventual consistency trade-offs in messaging systems.