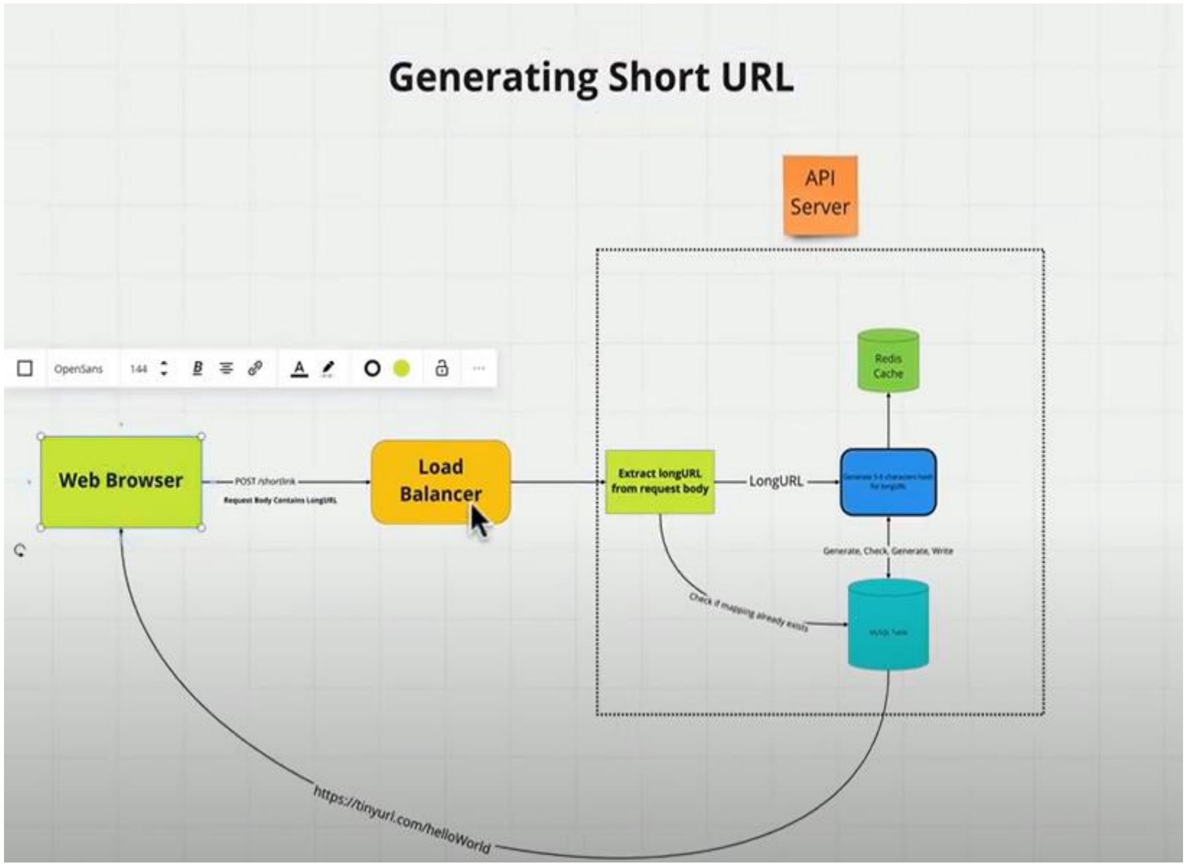
Design a URL shortener

Archana Goli

1.Generating short URL from long URL

Use deterministic hash functions
Take first or last 4-5 characters
For collisions, append pre-determined string to the end


Generating Short URL
API Server
Redis Cache
Web Browser
POST /shortlink
Request Body Contains LongURL
Load Balancer
Extract longURL from request body
LongURL
Generate 5-6 characters hash for longURL
Generate, Check, Generate, Write
Check if mapping already exists
MySQL Table
https://tinyurl.com/helloWorld

1. Web Browser Request (POST Request for Long URL)
The lengthy URL is sent in the request body of a POST request made by the user after accessing the URL shortening service. This long URL may be something like https://example.com/verylongurl.
To generate a new short URL, the request is submitted to the API of the shortening service.
2. Load Balancer

A load balancer processes the request, much like it does in the first diagram.
The load balancer helps disperse traffic for improved efficiency and dependability by making sure the request is effectively handled by one of the available API servers.
3. Extract Long URL from Request Body

The lengthy URL is taken out of the request body by an API server once it has received the request.

At this point, the API server is prepared to create a new short URL or determine whether the provided long URL already has a short URL.
4. Check Redis Cache

The system looks in the Redis cache to determine if this long URL already has a corresponding short URL before creating a new one.
Scenario 1: Cache Hit: The system fetches the corresponding short URL from Redis and provides it back to the user if the long URL is already cached.
Scenario 2: Cache Miss: The system moves on to step 5 if Redis cannot locate the lengthy URL.

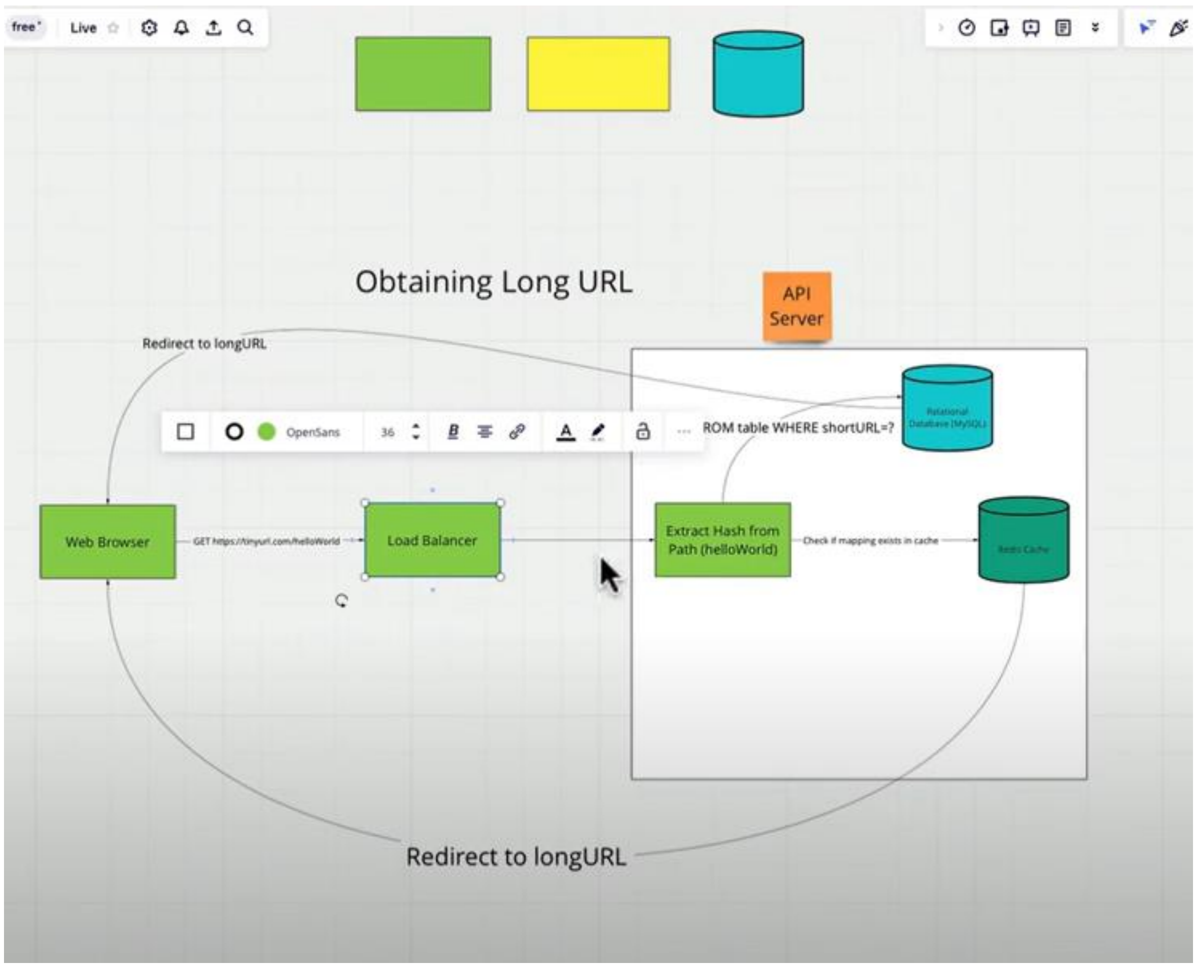
5.Query Relational Database (MySQL)

The system verifies whether the long URL has previously been mapped to a short URL by checking the relational database (MySQL) if the long URL is not cached.
Scenario 1: Found in DB: The system fetches the current short URL if a mapping is already present in the database.
Scenario 2: Not Found in Database: In this case, a new short URL (hash) must be created if the long URL isn't present in the database.
6. Generate Short URL (Hash Creation)
In the event that no mapping is found, the API server creates a fresh, distinct hash (such as helloWorld) to function as the short URL's identification.
Typically, encoding, hashing, or randomization are used to create this hash, guaranteeing that the short URL that is produced is distinct.

7. Store Mapping in MySQL

Next, for permanent storage, the newly created mapping (short URL to long URL) is kept in the MySQL database.
This guarantees that a database query can be used to answer any requests for the long URL or short URL in the future.
8. Cache Update (Redis)
The system can update the Redis cache with the updated mapping after saving it in MySQL.
As a result, requests for this lengthy URL will be processed more quickly in the future because Redis can handle them without requiring the database.
9. Return Short URL
Ultimately, the user receives the freshly created short URL from the system (such as https://tinyurl.com/helloWorld).
Now, the user can access or share the long URL in the future using this shorter URL.

1. Obtaining the long URL from short URL


free
Live
Obtaining Long URL
API Server
Redirect to longURL
OpenSans
36
ROM table WHERE shortURL=?
Relational Database (MySQL)
Web Browser
GET https://tinyurl.com/helloWorld
Load Balancer
Extract Hash from Path (helloWorld)
Check if mapping exists in cache
Redis Cache
Redirect to longURL

Web Browser Request (GET request for Short URL):

A person launches their web browser and types a shortened URL (like https://tinyurl.com/helloWorld).
The lengthy URL is first looked up by the URL shortening service once the browser submits a GET request to them.
2. Load Balancer
A load balancer, which is essential for dividing traffic across several servers, receives the request first.

Reasons for using it: Load balancers help manage the load in high-traffic services by making sure no single server is overloaded with requests. It improves fault tolerance and makes sure that traffic is handled by other servers in the event of a server failure.
The request is sent to one of the API servers by the load balancer.

3.Extract Hash from Path

The URL path (/helloWorld) is where the API server retrieves the unique hash (helloWorld in this example).
The hash's purpose: For the system's lengthy URL, this hash serves as a unique identification. When the short URL (explained in the second diagram) was constructed, it was created.

4. Check Redis Cache
It is the Redis cache that the system searches first for the mapping between the short URL and the long URL.

Rapid in-memory key-value storing is provided by Redis. Its purpose is to lessen the strain on the relational database by caching often retrieved data.
To look up the lengthy URL, use the hash helloWorld as a key.

Why utilize Redis? Comparing Redis to a regular database query, the former is substantially slower. It lessens the database load and lowers user latency.

Scenario 1: Cache Hit

If the mapping for helloWorld is present in Redis, the lengthy URL is obtained from the cache. Without contacting the database, the system can then instantly reroute the user to the lengthy URL.

Scenario 2: Cache Miss
The system advances to step 5, where it queries the relational database, in the event that Redis lacks the mapping.

5. Query Relational Database (MySQL)

The system queries the relational database (in this case, a MySQL database) if the mapping is not discovered in Redis.

6. API Server & Cache Update

The lengthy URL is sent to the API server after it has been obtained from the MySQL database.
Update on the Redis Cache: In case of a cache miss and the mapping was retrieved from the database, the system can now save the outcome in Redis for further queries. This process guarantees faster queries for the same short URL in the future.
7. Redirect to Long URL
The system sends the user to the lengthy URL after obtaining it.
The lengthy URL that was the user's intended destination is now displayed to them.

Designing a chat Application

Functional prerequisites:

Generate Short URL: The system creates a condensed version of a large URL upon a user's click.
Redirect to Long URL: A shorter URL will link users to the equivalent long URL when they visit it.

Non functional requirements:

Extremely low latency: Users should be redirected quickly and receive brief URLs from the system.
Extremely high availability: Users should always be able to shorten URLs and get redirected since the system should always be up and running.

Explanation: Systems for URL shortening are built with great performance and scalability in mind. A long URL is often maintained in a database, and a unique ID is used to construct a condensed version of the URL. The service looks up the matching lengthy URL and redirects the user when the abbreviated link is clicked.

URL shortening services employ caching techniques (like Redis) to keep mappings between long and short URLs for quick retrieval in order to minimize latency.

Load balancing and replication techniques—such as utilizing cloud services like AWS S3, DynamoDB, and replication mechanisms—are used to prevent single points of failure and provide high availability.

WhatsApp Chat App Design

Functional prerequisites:
One-on-one private chat: A function that enables users to have private conversations with one another.
Users can participate in group chats where they can speak with other users at the same time.
Join/Leave a group: Depending on the access guidelines of the group, users can choose to join or leave a group chat.
Monitor a user's online or offline status: Determine if a user is currently online or not.
Notify the recipient: If they are not online, the system ought to alert them when they join the

network.
Non-essential Conditions:
Minimal latency: Real-time messages have to be sent and received with the least amount of delay possible.
High availability: To guarantee no downtime, the chat service should be accessible around-the-clock.
Scalability: Millions of users should be supported by the system, which may scale horizontally.

High-level Architecture for APIs:

The groups a user has joined are returned by the function getAllGroups(userId).
This API delivers a list of all the group IDs the user is connected to after querying a database to obtain them all.
The function leaveGroup(groupId, userId) eliminates a user from a group.
By changing the membership table of the group, this API enables a user to leave the group.
Using joinGroup(groupId, userId), a user can become a member of a group.
To add the user to the group, this API modifies the membership table of the group.
Sends a message from one user to another or to a group using the sendMessage function (userId, receiver, channelType, message).
The message is added to the appropriate message table (direct message or group message) based on whether it is a private or group chat.
Gets chat messages between users with the parameters getMessage(userId1, userId2, channelType, earliestMessageId).

Database Design Data Access Patterns: Selecting the right database is essential because in messaging systems, particularly group chats, sending messages is usually more common than reading them.

LSM Tree: Log-Structured Merge (LSM) trees are perfect for storing messages since they are designed for workloads that involve a lot of writing.
B Tree: B-trees work well for workloads requiring a lot of reading and fast lookups.
WhatsApp has more writes than reads, which makes LSM-based systems like Cassandra more suitable (particularly in group chats).

Database Schema: Table for Direct Messages:

UserA and UserB (the two chat participants) are the partition key.
Key for Sorting: messageId.
Additional fields: Message, Timestamp.
Explanation: Messages between the two users are stored together because they are used as a partition key. The chronological sorting of messages is guaranteed by the messageId.

Table of Group Chat:

Key to Partition: groupId.
Key for Sorting: messageId.
Additional fields: userId, message, and timestamp.
Explanation: While the sorting key (messageId) arranges messages chronologically, the partition key (groupId) groups all of the messages in a chat.
Table of Group Membership:

Key to Partition: groupId.
Key for Sorting: userId.
Explanation: This database tracks which users belong to which groups.
Table of User Membership:

Storage Key: userId.
Sorting Key: groupId.
Explanation: The groups that a user has joined are tracked in this table.

User Profile Table:

fields: profileImage, online/offline status, hashed password, and userId.
Justification: Information about user profiles is kept in this table.
Upper-Level Architecture:
Clientele:

HTTP is used by the user's device for message retrieval, group join/leave, and login.
The client sends and receives messages via WebSocket in real-time.
Internet Socket:

User A sends messages via WebSocket 101.
The message is sent to WebSocket 102 (User B) after being saved in the database.
In the event that WebSocket 102 is down (meaning User B is not online), the message is queued for further transmission.
Alert System:

Messages are queued by the Notification Service, which notifies the user when they are back online or if the message is delivered while they are offline.
User Mapping Facility:

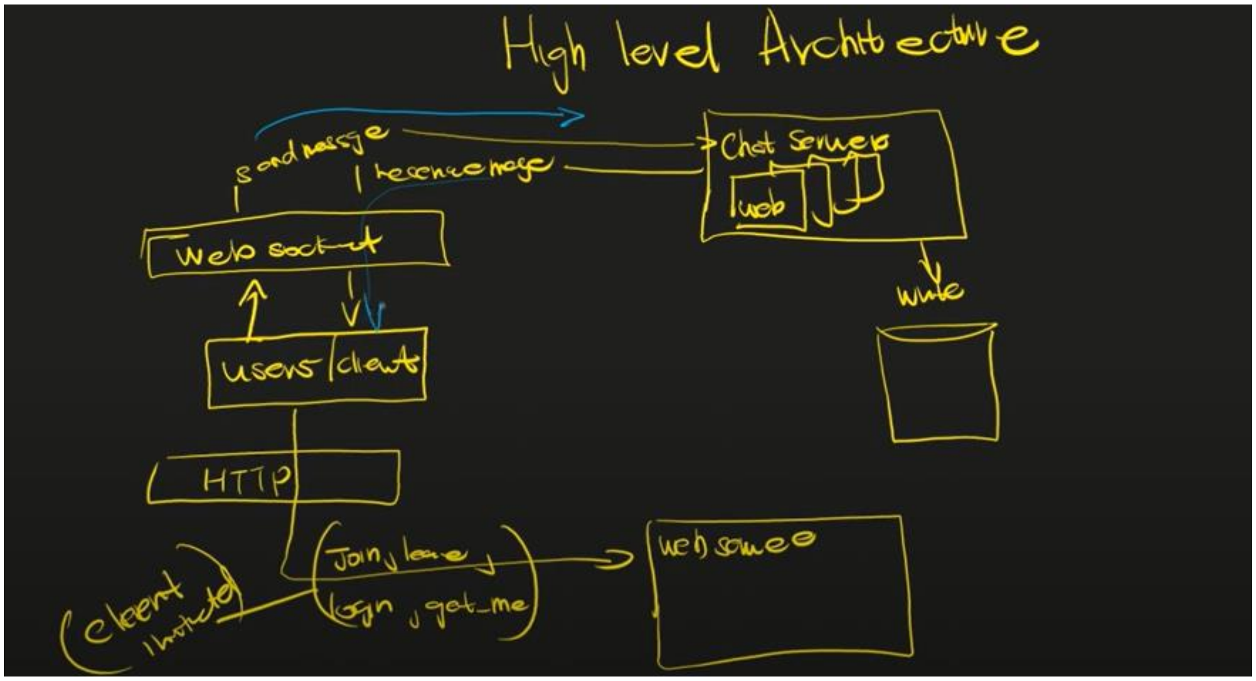
This service indicates which WebSocket is open at any given time.

Database:
NoSQL databases that store all messages and user/group relationships, such as Cassandra, manage reads and writes.
Catch-Up on History:

By using the getHistory(A, B, after timestamp) API, users can retrieve messages that they may have missed.
Database Management:
Compose Actions:
Remove/Include a user in the group: modifies the table of group membership.
Adds the message to the group chat table by clicking "Commit message in the group."
Messages between users can be committed by adding them to the direct message table.
Examine Procedures:
Private conversation: Get every message from the direct messages database given userA and userB.
Group chat: Get every message from the group chat table using the groupId as a guide.
Obtain the IDs of group members: Retrieve every user from the membership table of the group.
Find every group an individual has joined: Obtain every group ID from the user membership table.

Cassandra Concepts:
In the Cassandra cluster, the partitioning key controls how data is split up across the nodes.
Direct message recipients: userA, userB.
groupId is used for group messaging.
Each partition's data is arranged using the sorting key, also known as the clustering key.
MessageId is used for direct messaging.
The messageId for group messages.

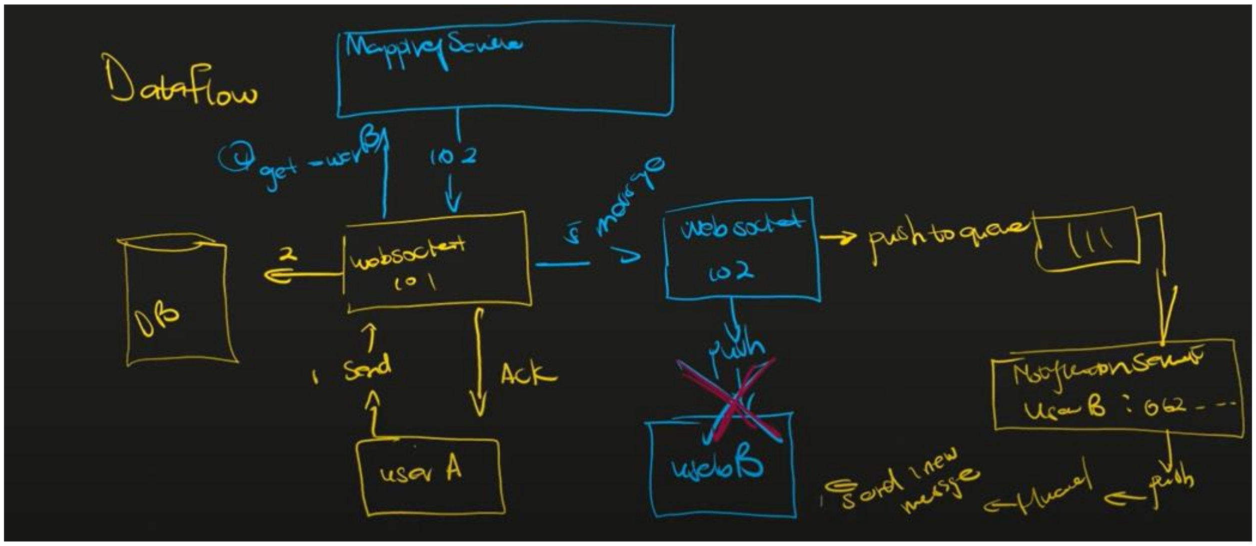
High level Architecture:



1. Client Interaction: The client, which may be a mobile app or a web browser, represents the user.
The HTTP and WebSocket protocols are used by the client to communicate with the system.
2. HTTP (Initial Communication): When a client wants to join or exit a chat room, log in, or perform other crucial chat-related operations, it first makes an HTTP request to the server. APIs are used to handle these tasks, sometimes in conjunction with HTTP-based REST endpoints.
3. WebSocket for Real-Time Communication: Following the initial connection, the chat service and the users/client establish a WebSocket connection. Real-time, two-way communication between the client and server is made possible by WebSockets.
Sending and receiving messages and other chat-specific operations are handled by the WebSocket.

4. Users/Clients: This is a representation of every user or client that is currently logged into the chat system. To send and receive messages in real time, each client connects with the other across the established WebSocket connection.
5. Chat Service: Web: This backend service is in charge of overseeing the chat feature. All incoming communications are processed by it, and it forwards them to the relevant parties (other connected users).
The chat service will also likely contain features like user authentication, message history, and room administration.
6. Database (Persistent Storage): User data and chat messages are kept in a database, which is visualized as a storage block. This is probably where the chat history is stored, so users can access previous exchanges.

7. Send/Receive Message Flow: A message goes through the following steps when it is sent by a user:
It is transmitted from the Users/Clients over the WebSocket connection.
The Chat Service receives the message and processes it there.
The WebSocket connection on their client is then used to send the message to the receiver or recipients.
The chat provider may choose to save the message in a database so that it can be accessed at a later time.



WebSocket 1.0:

For User A, this is the connection.
When User A clicks "send," WebSocket 101 manages the message transfer.
An acknowledgment (Ack) is sent back to verify delivery of the message after it has been sent.
WebSocket No. 102:

represents User B's connection.
If the WebSocket connection is open, messages sent to User B are relayed straight to them.
The message is sent to the notification service instead of User B if WebSocket 102 is not active (User B is offline or disconnected).

DB (Database):

saves the message sent by User A before processing it.
The message is first sent to the database by WebSocket 101, and then it is pushed to WebSocket 102 or the queue.
In line:

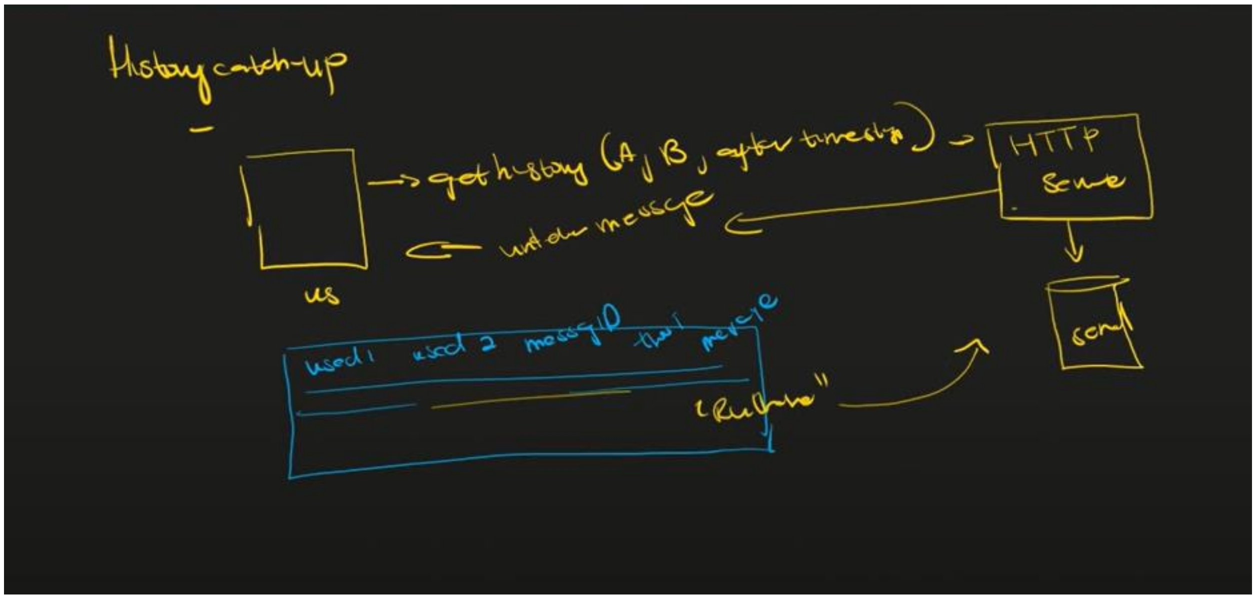
In the event that WebSocket 102 is unable to send the message (due to User B being offline), the message is queued for subsequent delivery.
Alert System:

When User B returns online, it is this service's responsibility to send the message.
Following its receipt from the queue, it uses a push notification to deliver the message to User B's device.
Map Services:

assists in determining which WebSocket connection (101) belongs to a certain user.
This service determines User B's WebSocket so that the message is routed correctly.

Data Flow Explanation: WebSocket 101 is used by User A to send a message.
The message is acknowledged (Ack) to User A and is kept in the database.
The Mapper Service identifies if WebSocket 102 (User B) is active.
In the event that WebSocket 102 is accessible, User B receives the message directly.
The message is pushed to the queue for delayed delivery if WebSocket 102 is unavailable.
When User B returns online, the Notification Service pulls the message from the queue and pushes User B with the notification.

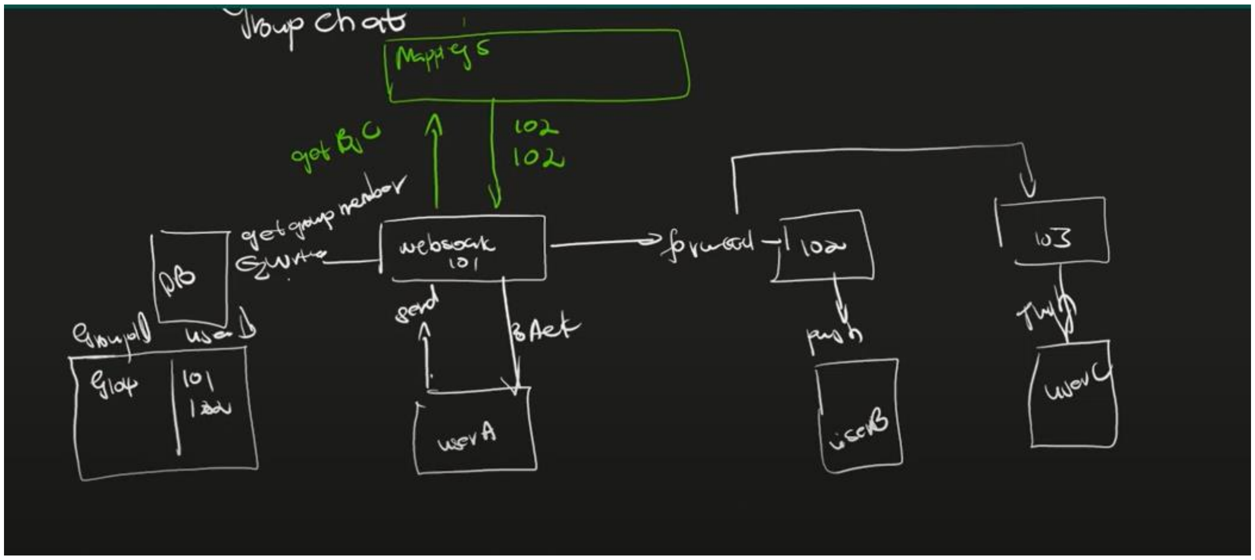
History catch-up:



1. User (us) Requests Conversation History:
Individual/Customer: This is a representation of the user wishing to "catch up" on messages

they might have missed because of a timeout or disconnections.
After experiencing a timeout or requiring older messages, the client (us) submits a get history request.
2. Request Sent to HTTP Service: The HTTP Service receives the request for the chat history. This system component interacts with the database and responds to requests that are not processed instantly.
This service responds to the client's request and retrieves the required conversation history from the database.

1. Database (Historical Storage): Database (scan) storage This is an image of the database containing the chat history. It keeps track of previous chat messages so they can be accessed at a later time.
The user's desired chat history is retrieved by the HTTP service by sending a query to this database.
4. Chat History Fetched: The HTTP service queries the database and returns to the client the requested chat history (messages A, B, etc.).
After receiving these outdated messages, the client might show them to the user.
5. Incoming Messages: The client is concurrently receiving live messages over the established Web-socket connection and retrieving the history. By doing this, the user can catch up on previous conversations and remain current on those that are still going on.



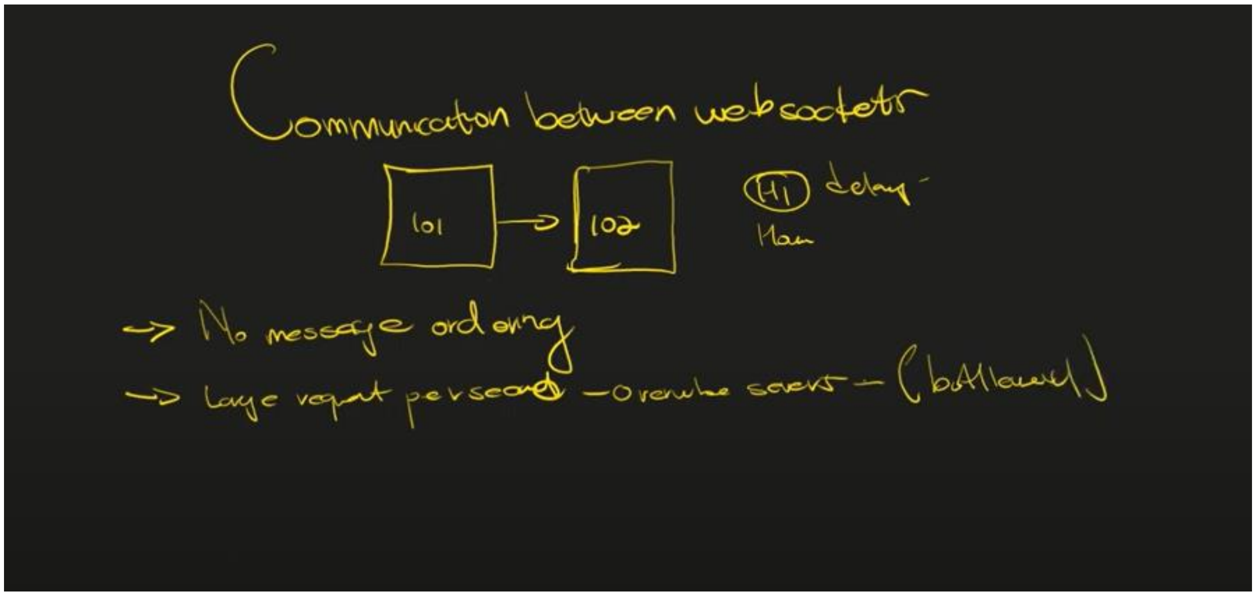
1. Group Information Retrieval: "got group" and "get group member" are mentioned at the top. This points to the possibility of a feature to get details about the organization and its constituents.
The users participating in the group chat have their member IDs (like 101, 102, etc.) and group ID (like G) retrieved by the system from the database.

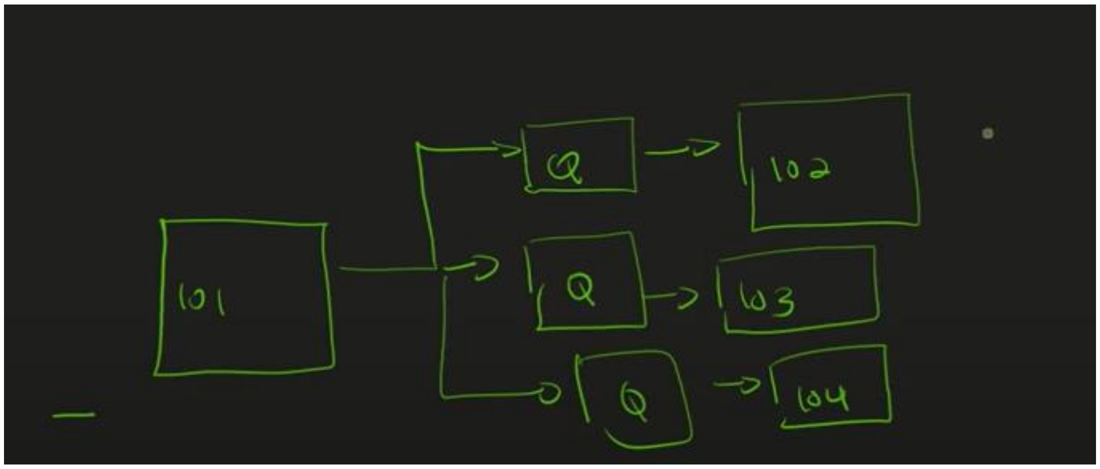
2. WebSocket Connection: A WebSocket connection (designated as "websocket 101") is shown in the diagram.
For users to send and receive messages in real-time throughout the chat, this connection is essential since it eliminates the need for frequent HTTP requests or refreshes.

3.Users and Messaging: User IDs, such as userA and userB, are used to identify the users in the group chat.
For the purpose of monitoring who is sending and receiving messages, the system assigns each user a unique identification.

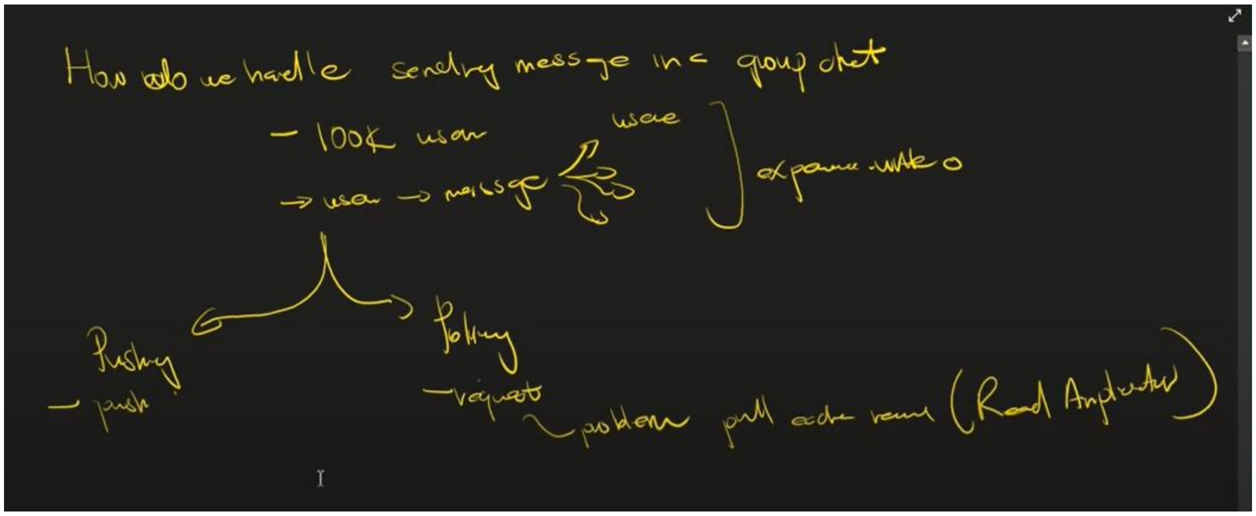
4.Sending and Receiving Messages: A message sent by user A is routed over the WebSocket connection:
Sending a message from userA is started by the send action.
The WebSocket server receives the message.
The message is subsequently broadcast by the WebSocket to every participant in the group chat, including userB and others.
The "back" action suggests that user A may additionally receive a confirmation or acknowledgement from the system confirming the successful transmission of their message.
5. User Notifications: The push action implies that the WebSocket server publishes the message to every group member (such as userB) as soon as it receives it.
The flow of notifications or communications to other group members is shown by the arrows number 102.

6. Managing Multiple Users: The table called "group" in the diagram's lower section lists every user in the conversation, including userA, userB, and maybe more.
This structure represents each user, and their individual IDs facilitate communication and message delivery.


Communication between websockets
101
102
Hi delay
Man
→ No message ordering
→ Large request per second — overwhelm servers — (bottleneck)



Synchronous HTTP solution


How do we handle sending message in a group chat
- 100k users
→ user → message
user
expensive write
Pushing
- push
Polling
- request
polled pull each resume (Read Amplification)

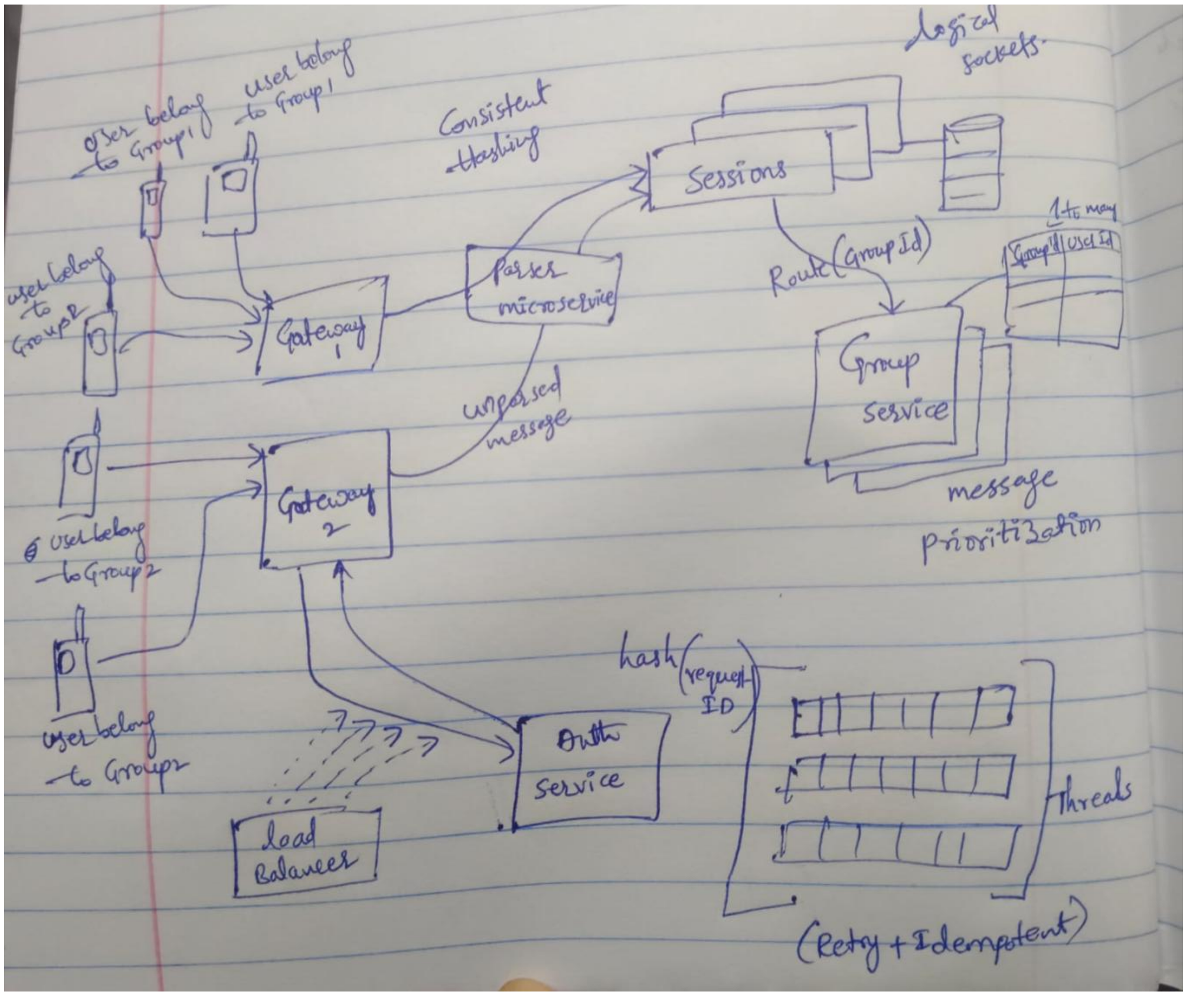
Whatsapp System Design

Features of whatsapp:

- Group messages
- Sent + Delivered + Read + receipts
- Online / Last seen
- Image sharing
- Chats are temporary /permanent
- One to one chat

- TCP Websocket allow peer to peer communication

Architecture for Real time messaging system:



1. Client Devices

The diagram depicts various modern-day client devices, such as smartphones, that act as emitters and receptors of texts/strings.

Users get the advantages of belonging to a few groups and they are then allowed by this feature to send messages to an individual user as well as to the whole group.

2. Gateways

There are two gateways in the architecture, called Gateway and Gateway 2 respectively.

Gateways work as a connection point between client devices and the backend services; they are in charge of:

Taking in the messages from users.

Guiding the traffic to the relevant services.

Session management and routing handling.

Both gateways communicate with the group service for managing group messaging and an unplaced message queue for messages that cannot be delivered immediately.

3. Load Balancer

Load balancer redistributes all incoming traffic to the gateways for the purpose of creating an even load for the system so that no single gateway gets congested.

It improves fault tolerance and makes it possible for the system to support more users at once.
4. Session Management
Users and the system are kept connected over time by the sessions component. For real-time communication, this is essential.
The system is able to keep track of active users and their group affiliations because each user session is given a unique identity.

5. Group Service
The group service is in charge of overseeing communication inside the group.
Using group IDs, it distributes messages to the right group members.
This service guarantees that messages sent to groups are received by all members and permits numerous users to engage in a single conversation.
In order to facilitate effective message routing, the system is able to keep a mapping of users to groups (e.g., Group ID -> User List).

6. Message Prioritization
The system can categorize and respond to messages according to their significance thanks to message priority.
Less important communications can be transmitted later, and more vital messages can be prioritized for timely delivery.
This guarantees that urgent messages are responded to right away.

7. Unplaced Messages
Unplaced messages are held in a queue and cannot be delivered right away.
Message management in the event that the recipient is unavailable or offline is made easier using this. When the user returns online, the system will try to deliver these messages.

8. Auth Service
User authentication and authorization are managed by the Auth Service, which is perhaps short for Authentication Service.
It makes sure that the messaging system is secure by limiting who is allowed to send and receive messages.

9. Hashing and Request ID

Hash(request ID) is mentioned in the diagram, implying that every request is hashed to guarantee that it can be uniquely identified.
This is probably done to keep things consistent and avoid processing messages twice.

10. Retry and Idempotency
Even in the event of temporary problems, messages are finally delivered since the system has a mechanism in place for retrying message delivery in the event of a failure.
Idempotency guarantees that a message will only be processed once even if it is transmitted more than once as a result of retries. By doing this, duplicate messages are kept out of the recipient's inbox.

11. Threads
The system manages several user requests at once by using multi-threading. By processing many messages or user requests concurrently, each thread enhances responsiveness and scalability.
In a real-time messaging service, where numerous users may be sending messages simultaneously, this is essential for managing heavy traffic.

How the system works together:

User Interaction: Messages are sent by users from their connected devices to the gateways.
Gateway Processing: These messages are received by the gateways, which then forward them to the group service if they are group messages, or route them directly to the appropriate recipients.
Load balancing: A load balancer makes sure that traffic is split up equally among several gateways.
Session Management: Real-time updates and message delivery are made possible by the maintenance of active sessions for every user.
Group Management: Using the group ID, the group service distributes messages to the right group members.

Message handling: A message is pushed to the unplaced message queue for delivery at a later time if it is unable to be delivered.
Retry Mechanism: Using request IDs to maintain state and guarantee idempotency, the system tries again to deliver messages that are unsuccessful.
Concurrency: By handling multiple user requests at once, the multi-threaded architecture enhances system performance.