Designing a URL Shortener System:

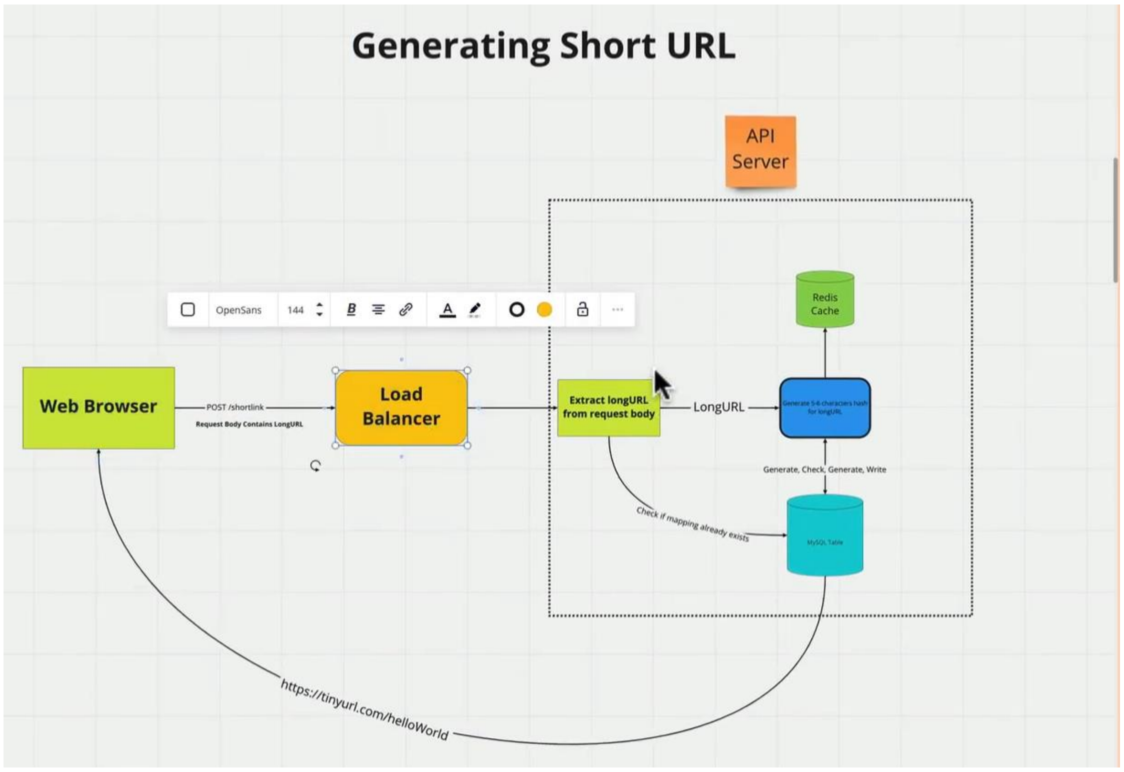
URL Shortener Design Overview

A URL shortener is a web service that converts long URLs into shorter links, facilitating easier sharing and accessibility. Its primary functions include:

- Generate a short URL from a long URL.
- Retrieve the long URL from a short URL.

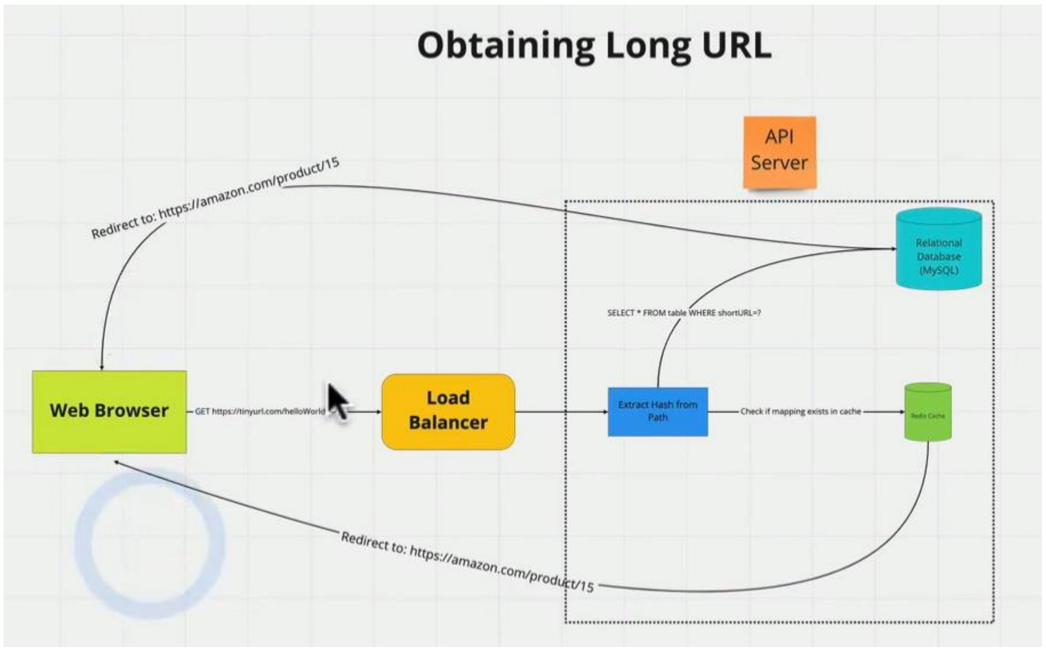
Generating Short URLs:

- User sends a long URL via an HTTP POST request to the server.
- The server checks if the long URL already exists in the database.
- If it exists, return the corresponding short URL.
- If not, generate a hash from the long URL using a deterministic hashing function.
- To ensure uniqueness, check for collisions in the database.
- If a collision occurs, modify the long URL (e.g., append a string) and regenerate the hash.
- Store the mapping of long URL to short URL in a relational database and a cache (e.g., Redis).


Generating Short URL
API Server
Redis Cache
OpenSans
144
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

Retrieving Long URLs:

- User sends an HTTP GET request to the short URL.
- The server extracts the hash from the short URL.
- Check the Redis cache for the long URL.
- If found, redirect the user to the long URL.
- If not found, query the MySQL database to retrieve the long URL.


Obtaining Long URL
API Server
Redirect to: https://amazon.com/product/15
Relational Database (MySQL)
SELECT * FROM table WHERE shortURL=?
Web Browser
GET https://tinyurl.com/helloWorld
Load Balancer
Extract Hash from Path
Check if mapping exists in cache
Redis Cache
Redirect to: https://amazon.com/product/15

Redirection Types

- **301 Redirect (Permanent)**:
  - Caches the response in the browser, reducing server hits for subsequent requests.
- **302 Redirect (Temporary)**:
  - Does not cache the response, allowing tracking of every access to the short URL.

Summary:

Users generate short URLs by sending long URLs to the server, which checks for existing mappings and generates unique hashes if necessary. Long URLs can be retrieved by sending requests to the short URLs, with caching mechanisms in place to improve performance.

URL Shortening Service Design:

Functional Requirements (FRs):

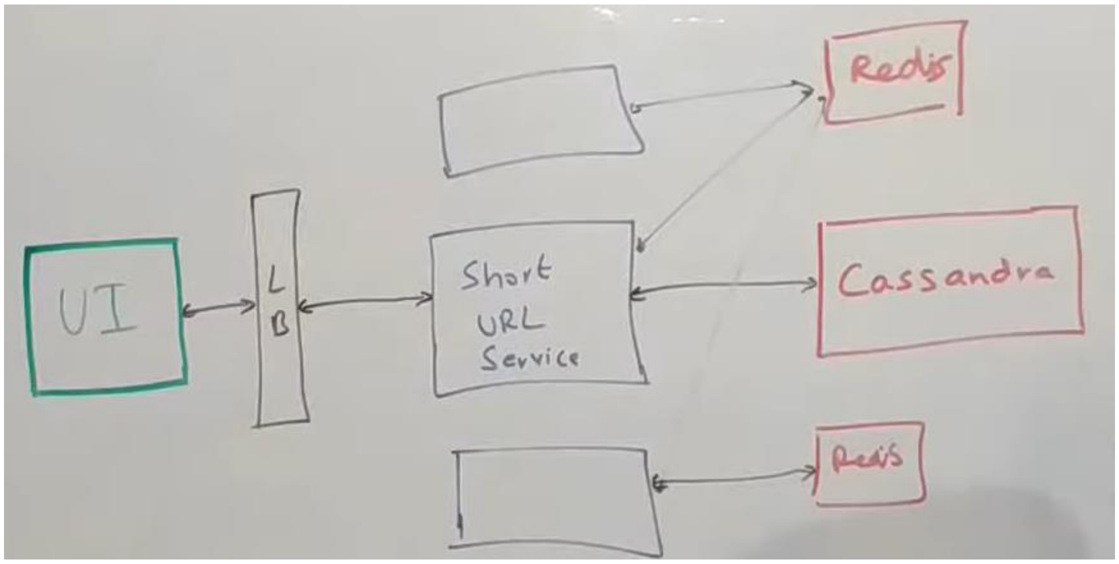
- **Shortening URLs**: Given a long URL, the service returns a shorter URL.
- **Redirecting**: When a short URL is accessed, it redirects to the original long URL.

Non-Functional Requirements (NFRs):

- **High Availability**: The service must always be available, especially for high-traffic platforms like Facebook or Twitter.
- **Low Latency**: The service should respond quickly to user requests.

Short URL Length Considerations:

- The length of the short URL depends on the expected scale of usage. For small scales, 2-3 characters may suffice, but larger scales require more characters.
- Calculate the number of unique URLs needed based on traffic estimates (requests per second).



Character Set for Short URLs:

- Typically includes lowercase letters, uppercase letters, and numbers (62 characters total).

Calculating Required Length:

- To determine the length *n* of the short URL, ensure 62^n is greater than the total number of unique URLs needed (denoted as Y). For example, if Y leads to n=5*n*=5, use a length of 5.

System Architecture Overview:

- **UI**: Accepts long URLs and returns short URLs.
- **Short URL Service**: Generates short URLs and stores them in a database.

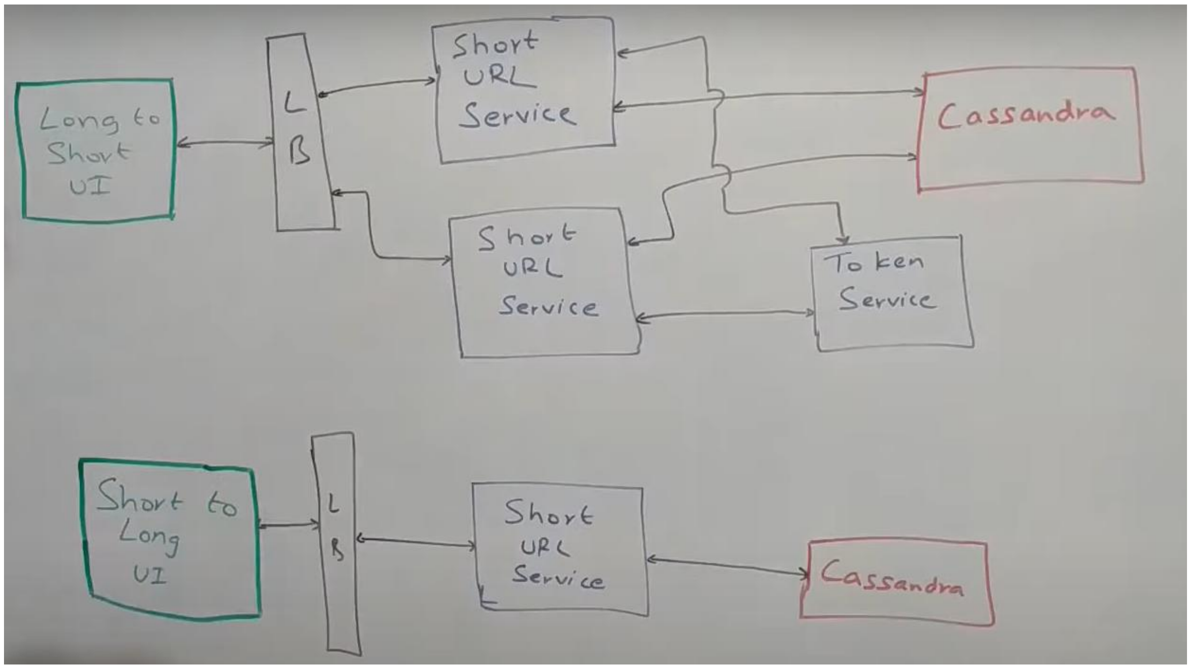
- **Database**: Stores mappings of short URLs to long URLs, with options like Cassandra for scalability.

Collision Handling:

- To avoid collisions (same short URL for different long URLs), use a unique number generator, such as Redis, to ensure unique short URLs.

Token Service:

- Implement a Token Service that assigns unique ranges of numbers to different service instances to prevent duplication.



Analytics Integration & Storage:

- Collect data on URL usage, including traffic sources and user agents, by integrating with Kafka for real-time analytics.
- Use batch processing to reduce latency, aggregating requests before sending to Kafka.
- Store analytics data in a system like Hadoop or use Spark Streaming for real-time analysis.

Conclusion:

The design of a URL shortening service involves careful consideration of functional and non-functional requirements, unique URL generation, and analytics capabilities to ensure a robust and efficient system.

Designing a Chat Application: System Design:

Chat Application Design Notes:

Overview of Chat Applications:

- Chat applications, like WhatsApp and Facebook Messenger, facilitate text-based instant messaging between users.

Key Features:

  - **Private Chat**: Supports one-on-one conversations.
  - **Group Chat**: Allows multiple users to converse in a shared space, with features for joining and leaving groups.

User Status Tracking:

    - Implement a mechanism to track online/offline status of users.

Latency and Availability:

- Prioritize minimum latency for real-time messaging, accepting some delay in message delivery.
- Focus on high availability over strict consistency due to the nature of chat applications.

Capacity Estimation:

- Estimate load based on user activity:
  - 500 daily active users sending an average of 80 messages per day, each around 100 characters.
  - Total data storage requirement estimated at 150 GB per day.

Database Design:

- Use a horizontally scalable NoSQL database (e.g., Cassandra) to handle high write operations.
- Focus on write-optimized structures due to the nature of messaging.

API Design:

- **Get All Groups**: Retrieve all groups a user has joined.
- **Join/Leave Group**: Manage group memberships.
- **Send Messages**: API to send messages between users.

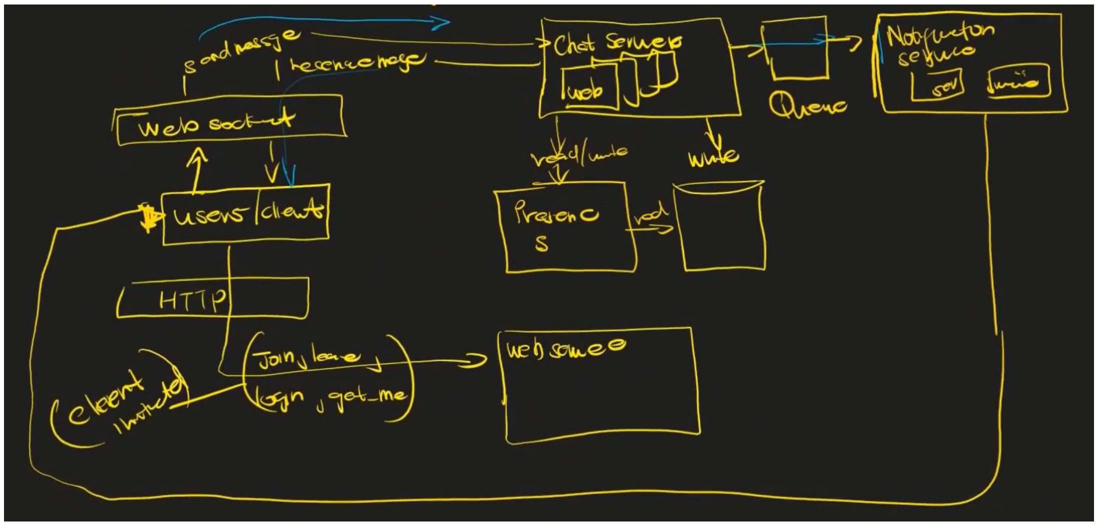
Data Modeling:

- **Direct Messages Table**: Partitioned by user IDs to optimize retrieval.

- **Group Messages Table**: Include group ID and message ID as partition keys to manage message ordering.

High-Level Architecture

- Use WebSocket for real-time communication.
- Implement a chat service to manage message sending and persistence.
- Include a notification service for offline users.



User Profile Management: Maintain a user profile table with hashed passwords and online status.

System Architecture Overview:

User Interaction and Messaging Flow:

- **User Online Status**: A service checks if users are online or offline, which is crucial for message delivery.
- **Message Delivery**: When a user sends a message, it is first written to the database, then sent to the appropriate websocket server for the recipient.

Server Mapping and Message Routing:

- **Server Mapping**: Each websocket needs to send messages to a specific server. A user mapping service helps track the server ID associated with each user session.
- **Message Routing**: Messages are routed based on the user's websocket server, ensuring they reach the correct recipient.

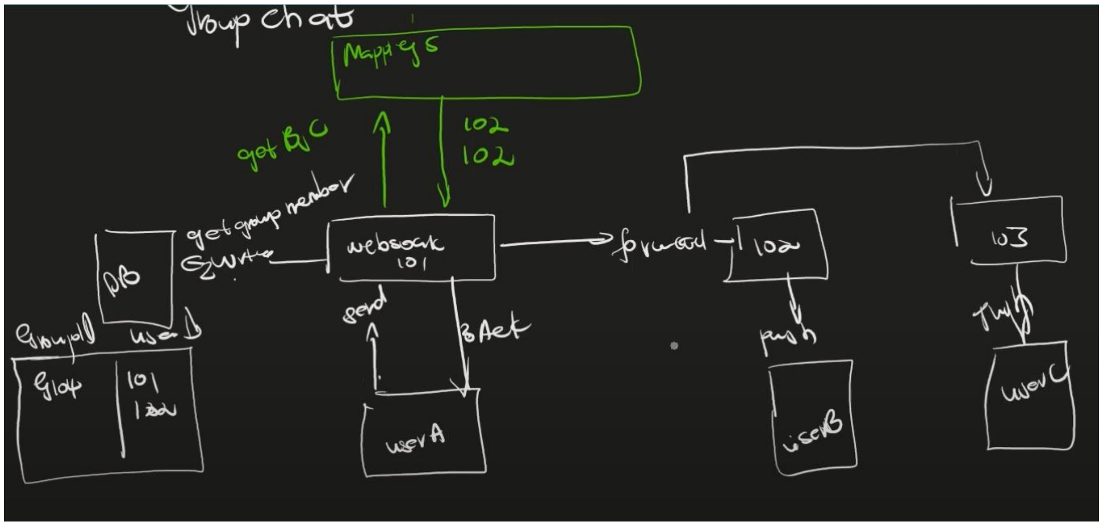
Notification Service:

- **Offline Notifications**: If a user is offline, messages are queued and a notification is sent to their mobile device to alert them of new messages.

- **History Retrieval**: Upon reconnection, users can retrieve undelivered messages through a history API.

Group Messaging:

- **Broadcasting Messages**: When a message is sent to a group, it is written to the database, and the system retrieves all group members to forward the message to their respective websockets.
- **Handling Large Groups**: For large groups, a decision mechanism determines whether to push or pull messages based on group activity.



Presence Detection:

- **Heartbeat Mechanism**: Users send heartbeat signals to check online status, but this can be resource-intensive, necessitating a more efficient algorithm that focuses on frequently contacted users.

Message Ordering and Delivery Guarantees:

- **Message Sequencing**: Each message is assigned a sequence number to maintain order and ensure that messages are processed correctly even if they arrive out of order.
- **Buffering Solutions**: Implementing buffering algorithms on websockets can help manage message exchange and reduce system overload.

System Challenges and Improvements:

- **Handling Server Downtime**: Strategies must be in place to manage user reconnections and message retrieval during server outages.
- **Caching and Replication**: Future improvements could include caching mechanisms and replication strategies to enhance system reliability and performance.

WhatsApp System Design:

Overview:

This focuses on the design of a chat application similar to WhatsApp, covering key components, requirements, and potential optimizations. It is structured to help you understand the system design process and prepare for interviews.

Understanding System Design Interviews:

- System design interviews assess how candidates break down large systems into manageable components and handle ambiguity. Interviewers look for familiarity with current technologies (e.g., AWS, databases) and the ability to communicate ideas clearly.

Gathering Requirements:

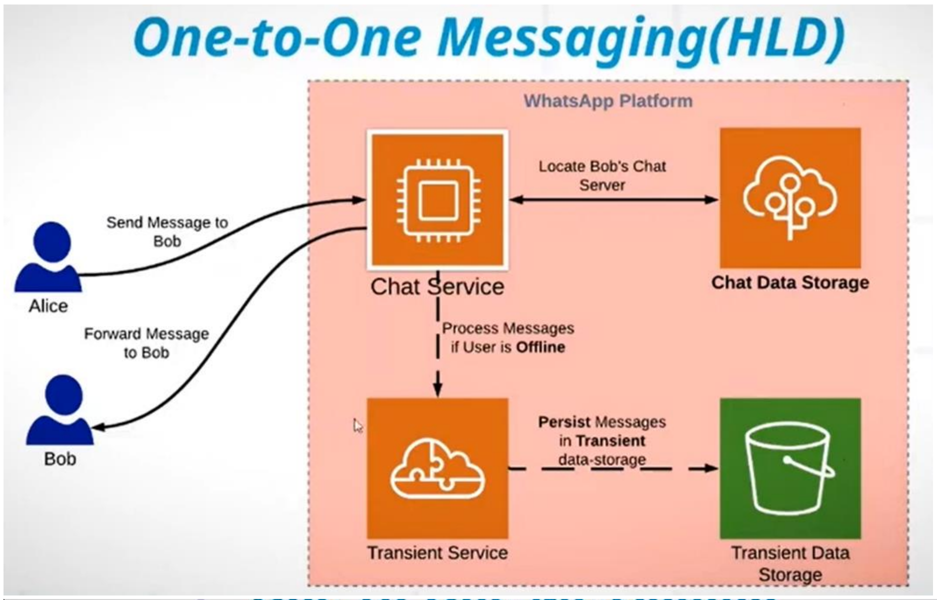
- **One-to-One Messaging**: Users can send messages directly to each other.
- **Delivery Receipts**: Indications for sent, delivered, and read messages (e.g., single tick, double tick).
- **Media Sharing**: Support for sending images and videos.
- **Push Notifications**: Alerts for new messages.

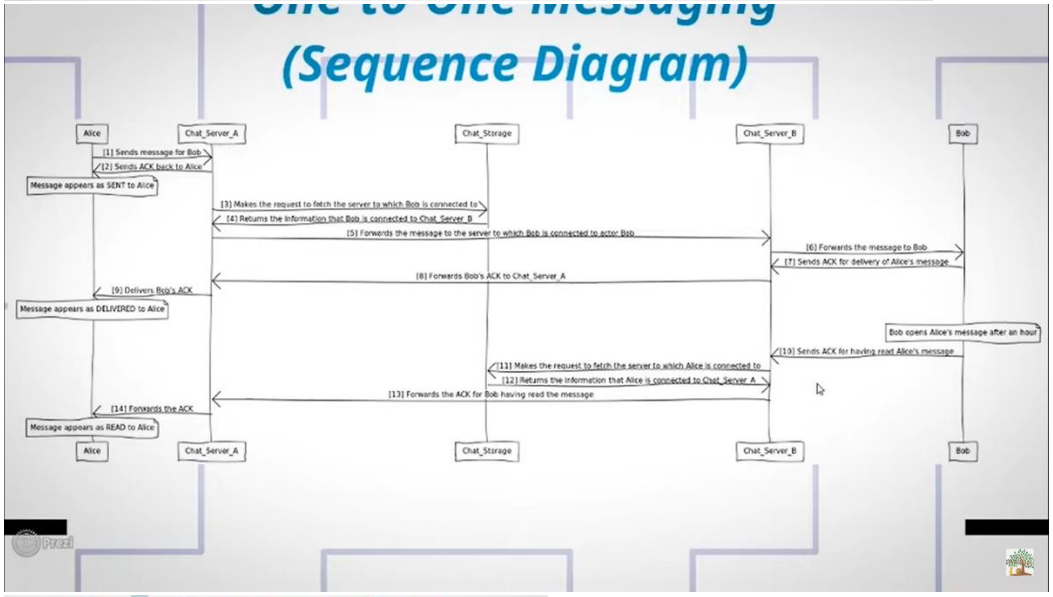
High-Level Architecture:

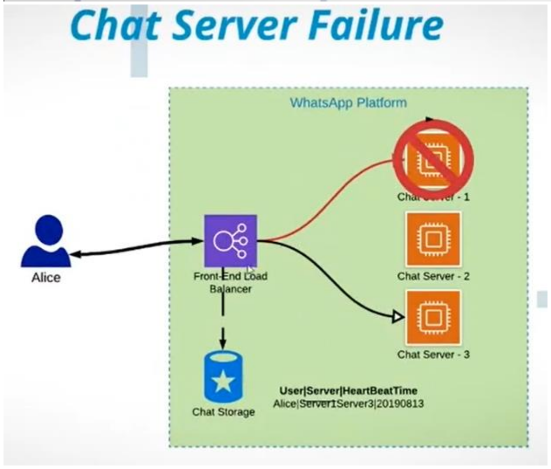
- The architecture can be visualized as a **client-server model**:
  - **Clients**: Users (e.g., Alice and Bob) interact with the app.
  - **Servers**: Handle message routing and storage.
  - Messages are sent to a server, which checks the recipient's status and delivers the message accordingly.

Message Flow:

- **Sending a Message**:
  - User sends a message to the server.
  - Server checks if the recipient is online.
  - If online, the message is delivered; if offline, it is stored temporarily.
- **Receiving a Message**:
  - The recipient's server forwards the message to their device, sending an acknowledgment back to the sender's server.

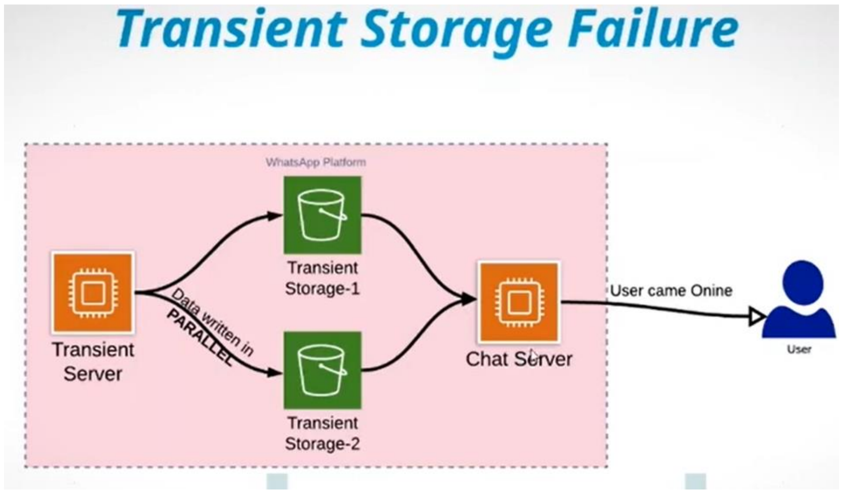

One-to-One Messaging(HLD)
WhatsApp Platform
Alice
Bob
Send Message to Bob
Forward Message to Bob
Chat Service
Locate Bob's Chat Server
Chat Data Storage
Process Messages if User is **Offline**
Transient Service
Persist Messages in **Transient** data-storage
Transient Data Storage


One-to-One Messaging
(Sequence Diagram)
Alice
Chat_Server_A
Chat_Storage
Chat_Server_B
Bob
[1] Sends message for Bob
[2] Sends ACK back to Alice
Message appears as SENT to Alice
[3] Makes the request to fetch the server to which Bob is connected to
[4] Returns the information that Bob is connected to Chat_Server_B
[5] Forwards the message to the server to which Bob is connected to actor Bob
[6] Forwards the message to Bob
[7] Sends ACK for delivery of Alice's message
[8] Forwards Bob's ACK to Chat_Server_A
[9] Delivers Bob's ACK
Message appears as DELIVERED to Alice
Bob opens Alice's message after an hour
[10] Sends ACK for having read Alice's message
[11] Makes the request to fetch the server to which Alice is connected to
[12] Returns the information that Alice is connected to Chat_Server_A
[13] Forwards the ACK for Bob having read the message
[14] Forwards the ACK
Message appears as READ to Alice
Prezi


Chat Server Failure
WhatsApp Platform
Alice
Front-End Load Balancer
Chat Server - 1
Chat Server - 2
Chat Server - 3
Chat Storage
User|Server|HeartBeatTime
Alice|~~Server1~~Server3|20190813

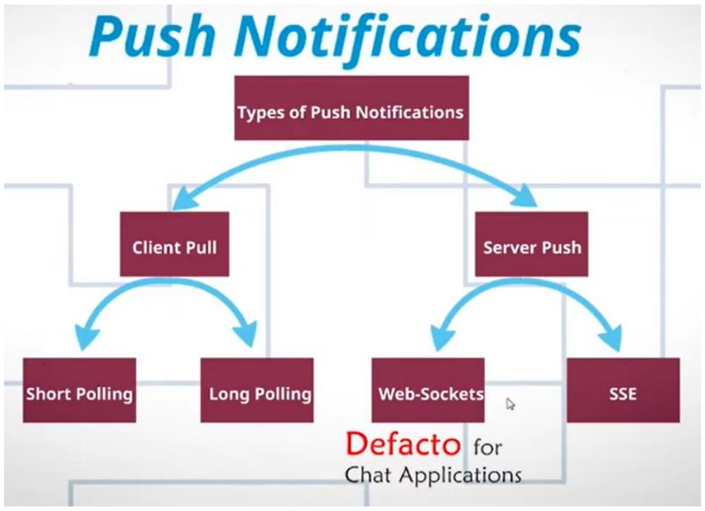
Transient Storage:

- To handle offline messages, implement a **FIFO-based queuing system** for undelivered messages. This allows messages to be stored and retrieved when the user comes back online.



Push Notifications:

- Implement push notifications using techniques like **WebSockets** for real-time communication. This allows the server to send updates to clients without them needing to request them.



Potential Improvements:

- **Scalability**: Discuss how to scale the system to handle millions of users and messages. For instance, if expecting 1 billion users, calculate the required number of servers based on concurrent connections.
- **Redundancy**: Create replicas of transient storage to prevent data loss during server failures.

Overview of Designing WhatsApp

Key Features of WhatsApp:

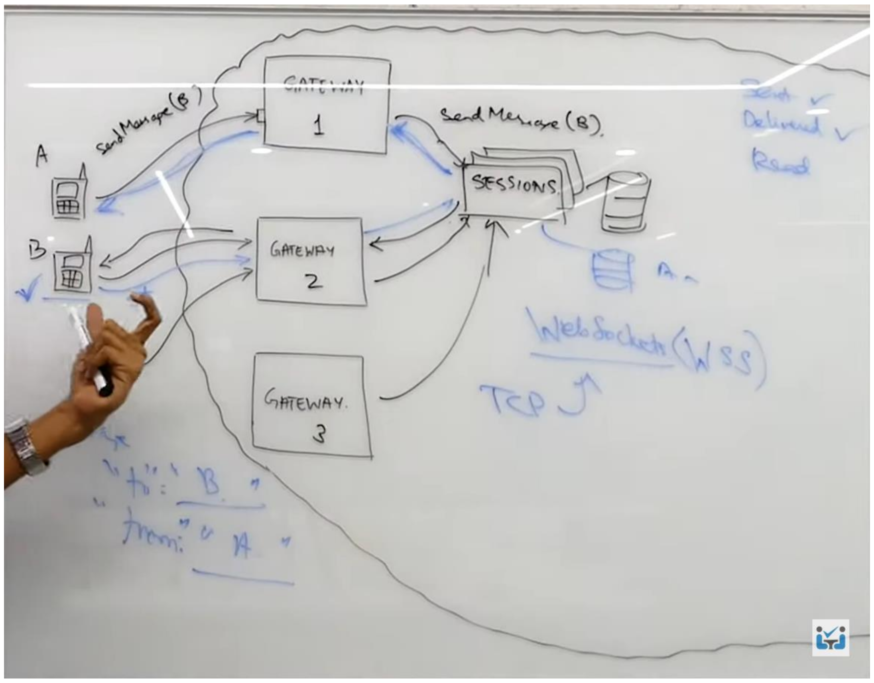
- **Group Messaging**: Supports groups with a maximum of 200 members. Essential for system design interviews.
- **Read Receipts**: Users receive notifications on message status (sent, delivered, read).
- **Image and Video Sharing**: Allows users to share multimedia content.
- **Last Seen Status**: Indicates when a user was last active.

System Architecture:

- **Gateway Connection**: Users connect to WhatsApp through a gateway, which communicates with internal services.
- **Session Management**: A dedicated microservice tracks user connections and routes messages.
- **WebSockets for Real-Time Communication**: Enables peer-to-peer communication, allowing the server to send messages directly to clients.

Message Flow:

- **Sending a Message**: User A sends a message to User B via the gateway, which forwards it to the session service.
- **Delivery Confirmation**: Once User B receives the message, an acknowledgment is sent back to User A.
- **Last Seen Tracking**: The last seen timestamp is updated based on user activity, ensuring accurate online status.



Group Messaging Mechanism:

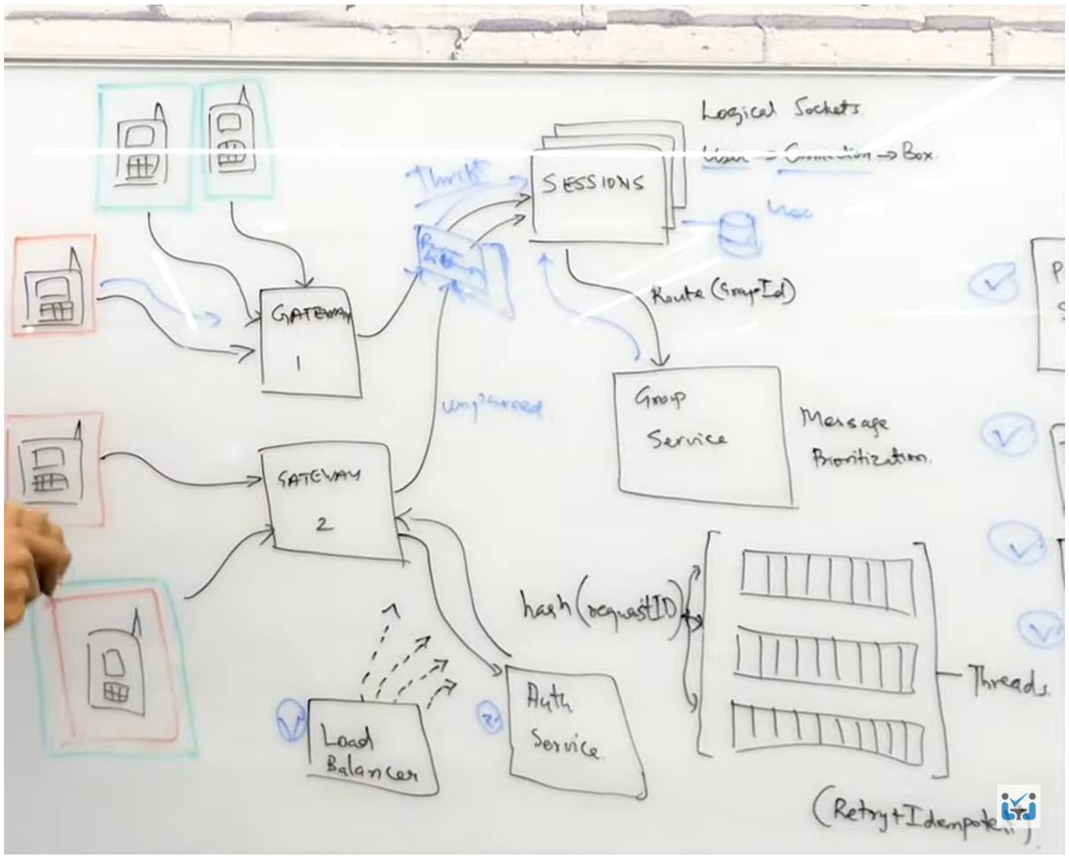
- **Decoupling Group Information**: Group membership is managed by a separate group service to reduce complexity.
- **Message Routing**: The session service queries the group service for member connections and routes messages accordingly.
- **Limitations on Group Size**: WhatsApp limits groups to 200 members to manage load effectively.

Error Handling and Resilience:

- **Message Queues**: Implemented to ensure messages are sent reliably, even in case of failures.
- **Idempotency**: Ensures that repeated message deliveries do not cause duplication.

Performance Considerations:

- **Load Management**: Prioritizes important messages during high traffic events to maintain system performance.
- **Efficient Resource Use**: Minimizes memory usage on gateways by offloading processing to dedicated services.



Notification System Overview:

Introduction to Notifications:

- Notifications can take various forms, including:
    - **Push Notifications**: Alerts sent directly to a user's device.
    - **Email Notifications**: Messages sent to a user's email.
    - **SMS Notifications**: Text messages sent to a user's phone.
- Examples of notifications:
    - A user receives a message alert (push notification).
    - An e-commerce app informs users of a new discount via email.
    - A maps application alerts users about a new restaurant via SMS.

Building the Notification System:

- The notification system consists of a microservice responsible for sending notifications.
- **Triggering Notifications**:
    - Other microservices can call the notification microservice to send notifications.
    - Scripts and cron jobs can also trigger notifications.

Sending Notifications:

- The notification microservice uses third-party services to send notifications:
    - **Email Services**: e.g., MailChimp, AWS Email Service.
    - **SMS Services**: e.g., Twilio.
    - **Push Notification Services**: Utilizes SDKs provided by iOS and Android.

User Information Storage: A MySQL database stores user information necessary for sending notifications, such as email and device ID.

Challenges in the Notification System:

- **Single Point of Failure**: The notification microservice can become a bottleneck.
- **Overloading**: High traffic can overwhelm the microservice.
- **Difficulty in Scaling**: Scaling the entire microservice is resource intensive.

Enhancements to the System:

- **Message Queue**: Introduced to retain notifications before sending, allowing for easier scaling.

– **Workers**: Lightweight functions that consume messages from the queue and send notifications.

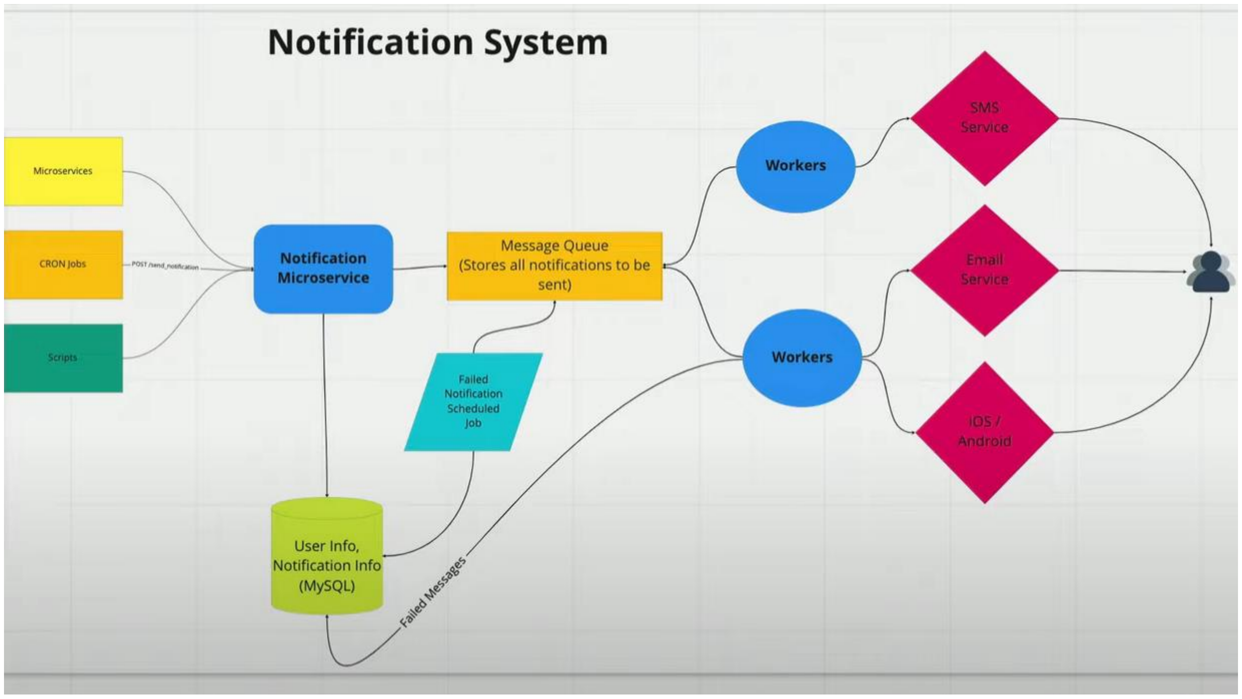
Handling Notification Failures:

– A MySQL table stores the status of each notification (e.g., delivered, pending, failed).

– Scheduled jobs can retry sending failed notifications by re-queuing them.

Summary:

The system consists of a notification microservice, a message queue, and worker functions that handle sending notifications. Each component is designed to improve scalability and reliability, ensuring that notifications are sent efficiently and can be retried if failures occur.


Notification System
Microservices
CRON Jobs
POST /send_notification
Scripts
Notification Microservice
Message Queue (Stores all notifications to be sent)
Failed Notification Scheduled Job
User Info, Notification Info (MySQL)
Failed Messages
Workers
Workers
SMS Service
Email Service
iOS / Android