INTERNSHIP REPORT

WEEK 4

Designing a URL Shortener System

&

Designing a Chat Application

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

RECRUITMENT LEAD

SUBMITTED BY

SANJEEV REDDY SIRIPINANE

URL Shortener

A URL shortener is a service that converts a large web address (URL) into a shorter, more distinctive identifier. Users can more easily share these shortened URLs on venues that have character constraints, such as social media. The reduced URL redirects to the original URL, maintaining functionality while enhancing shareability and aesthetics. Examples are Bitly and TinyURL.

Create a short URL

To generate a short URL, each lengthy URL is assigned a unique, compact identification. This can be accomplished through:

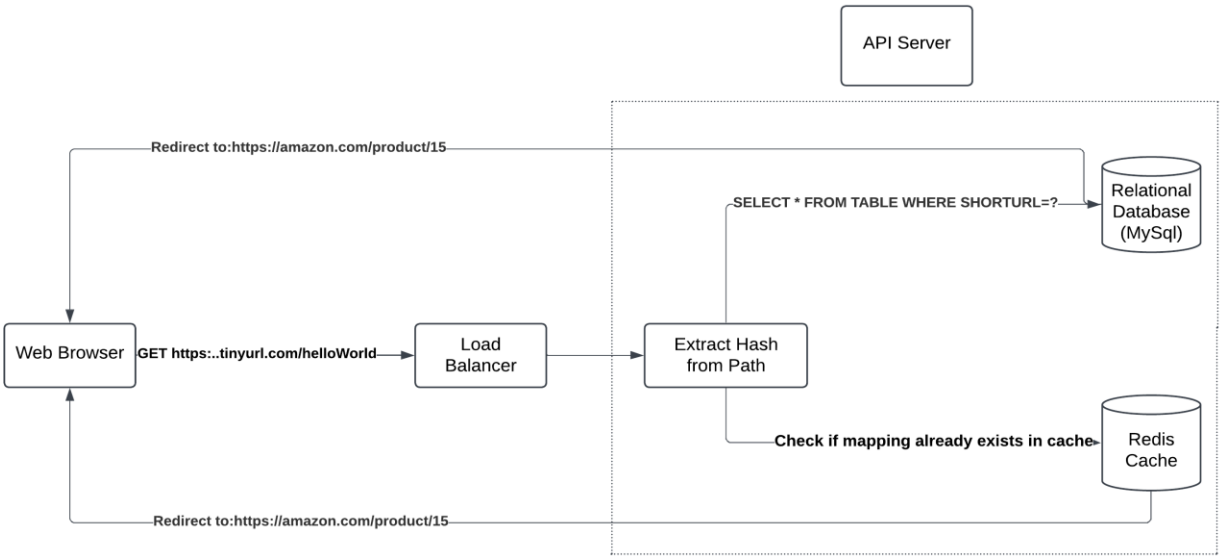
- Hashing: Use a hash function to convert the large URL into a shorter one.
- Incremental IDs: Assign consecutive numbers and encode them in a reduced format (such as Base62).
- Random Generation: Create a random string while assuring uniqueness using database checks. The created short URL is then saved in the database alongside the original URL.

Database

- The URL Mapping Table stores the mapping of short URLs to long URLs.
- Analytics Table: (Optional). Tracks use information such as clicks and geographic location.
- Indexes: Use indexes to quickly look up both short and lengthy URLs.
- Scalability: To manage significant traffic, use distributed databases (such as Cassandra and DynamoDB) or in-memory databases (such as Redis).

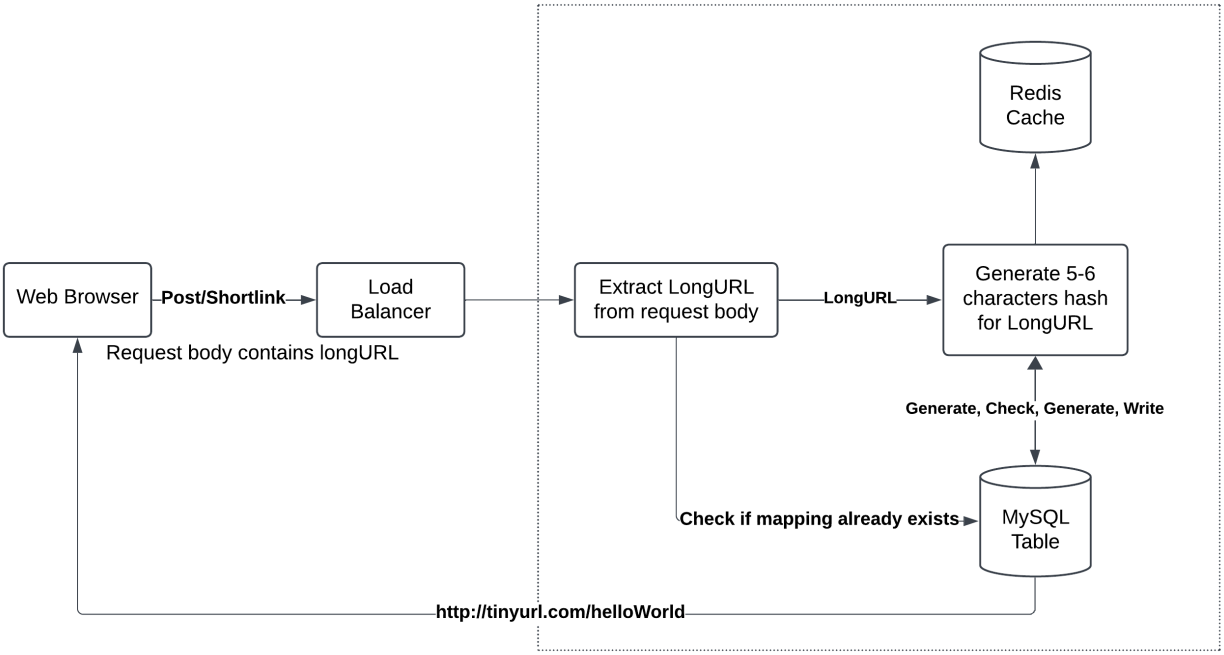
Design: LongURL to ShortURL

1. Accepting a long URL from the user.
2. Validating the URL format.
3. Generating a unique short URL using one of the methods (hashing, incremental IDs, or random generation).
4. Storing the mapping in the database.
5. Returning the short URL to the user.



Design: ShortURL to LongURL

1. The service queries the database to find the corresponding long URL.
2. If found, the system issues an HTTP redirect to the long URL.
3. If not found, return a 404 error.
   Caching frequently accessed short URLs further reduces lookup times.



HTTP Redirects (301 vs. 302)

- **301 Redirect:** Indicates a permanent redirect. It signals search engines to update their records to the new URL.
- **302 Redirect:** Indicates a temporary redirect, keeping the original URL indexed by search engines.
  In most URL shortener use cases, a 302 redirect is preferred to preserve the original URL's SEO and allow future changes.

Design a URL shortening service

A URL shortening service transforms a large, complicated web address into a shorter, more manageable connection. This shortened URL takes readers back to the original site, making sharing links quicker and cleaner—especially on networks with character constraints, such as Twitter. Common examples are Bit.ly and TinyURL, which provide short, easy-to-remember links.

Functional requirements:

- Given a lengthy URL, the service should provide a shorter and distinct alias for it.
- When a user clicks on a short link, the service should redirect to the original URL.
- Links will expire after the regular default time period.

Non-functional needs:

- The system should be very available. This is critical to consider since if the service goes down, all URL redirections will fail.
- URL redirection should occur in real time, with minimal delay.
- Shorter connections should not be predicted.

Capacity calculation for system design of URL shortener:

Let's say our service generates 30 million new URL shortenings every month. Assume we save every URL shortening request (and its accompanying shortened link) for five years. For this period, the service will create around 1.8 billion records.

30 million * 5 years * 12 months = 1.8 billion.

Assume we are using 7 characters to construct a short URL. These characters are a mixture of 62 characters (A-Z, a-z, 0-9). Something like: http://ad.com/abcdef1.

Data Capacity Modeling

We need to know how much data we will have to enter into our system. Consider the various columns or qualities that will be stored in our database and determine the data storage requirements for five years. Let's make the assumptions listed below for various qualities.

- Consider the average long URL size of 2KB ie for 2048 characters.

- Short URL size: 17 Bytes for 17 characters
- created_at- 7 bytes
- expiration_length_in_minutes -7 bytes

The above calculation will give a total of 2.031KB per shortened URL entry in the database.
If we calculate the total storage then for 30 M active users
total size = 30000000 * 2.031 = 60780000 KB = 60.78 GB per month. In a Year of 0.7284 TB and in 5 years 3.642 TB of data.

System Architecture for Short URL Service

1. UI: Accepts long URLs, interacts with the Short URL Service (SUS) to generate/fetch short URLs.
2. Short URL Service (SUS):
    - Create Short URL: Generates a unique short URL, stores it in the database, and returns it.
    - Fetch Long URL: Retrieves the long URL from the database using the short URL and redirects the user.

Short URL Generation Challenges & Solution

Key Issues

1. Collisions: Multiple SUS instances generating the same short URL.
2. Single Point of Failure (SPoF): Dependency on a single Redis instance.
3. Scalability: Handling load when Redis capacity is exceeded.

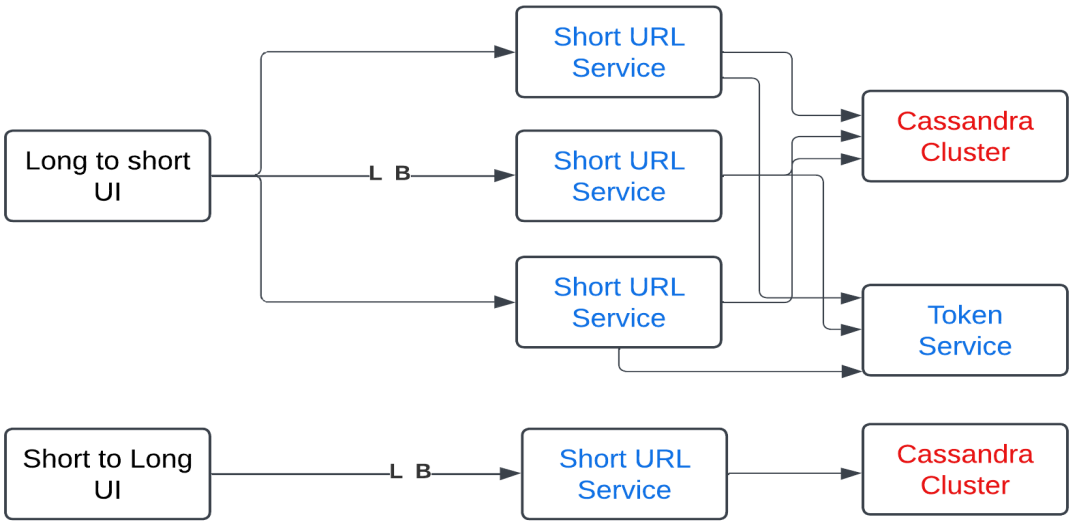
Optimized Solution

- Distributed Redis Instances:
    - Assign a unique range of IDs to each Redis instance.
    - Generate unique numbers incrementally within each range.
    - Convert the number to Base62 for the short URL.
- Managing Redis Ranges:
    - Use a central management system to allocate ranges dynamically.
    - When adding a new Redis instance, assign a new range without affecting existing instances.

Benefits

- Collision Avoidance: Each Redis operates within its allocated range.

- Scalability: Adding Redis instances increases capacity seamlessly.
- Fault Tolerance: Multiple Redis instances prevent a single point of failure.



Process:

- On startup, each SUS requests a unique range from the Token Service.
- When an SUS depletes its range, it fetches a new range.
- For example:
    - SUS1 gets range 1–1000.
    - SUS2 gets 1001–2000.
    - SUS3 gets 2001–3000.

Scaling the Short URL Service:

- Distribute MySQL Instances:
  Deploy multiple MySQL instances geographically to reduce latency and avoid a bottleneck. Each instance manages a subset of ranges for its region.
- Increase Range Size:
  By allocating larger ranges to each service, the frequency of requests to the Token Service decreases. This reduces load and improves overall system performance.

Handling Missing Ranges

If a service shuts down before using its entire range, we won't try to track or recover the unused numbers. Here's why:

- Negligible Loss: With over 3.5 trillion possible unique numbers available, losing a few thousand isn't significant.
- Simplicity Over Complexity: Attempting to track and reassign unused ranges would complicate the system and potentially slow it down, which isn't worth the tradeoff.

Redirecting Short URLs to Long URLs

When a request is made for a short URL:

1. The service fetches the corresponding long URL from the database.
2. It performs a redirect, taking the user to the correct page.

We use Cassandra as the database because it handles 3.5 trillion records efficiently. However, other databases like MySQL with sharding can also work. Cassandra is a simpler choice for this scale, but you can choose a database that suits your needs.

Analytics:

To include analytics, each request to generate or redirect a short URL collects metadata such as platform (e.g., Facebook, Twitter), user agent (iOS, Android, browser), and IP address, which is then submitted to Kafka for processing. To decrease latency, this write operation takes place asynchronously on a different thread. Instead of sending to Kafka for each request, data can be queued locally and written in bulk when the queue hits a threshold or at predetermined intervals (for example, every 30 seconds). While this minimizes CPU and bandwidth utilization, a machine failure may result in greater data loss than a single write—a trade-off to consider.

Designing a Chat Application System Design:

The goal is to design a chat application like WhatsApp or Facebook Messenger, focusing on key features and non-functional requirements. The primary features of the system include:

1. Private Chat: One-on-one conversations between users.
2. Group Chat: Users can create and participate in group conversations, with the ability to join and leave groups at will.
3. Online User Tracking: The system needs to monitor user status (online or offline).
4. Offline Notifications: When a user is offline, the system should send notifications to their device (e.g., Samsung or Huawei phones).

Non-Functional requirements:

Minimum Latency: Prioritize low latency to ensure real-time chat experiences with instant message delivery.

High Availability: Ensure the system remains available, with occasional delays (e.g., 2 seconds) acceptable.

Consistency: Sacrifice consistency to some degree, as it's less critical in a chat application.

Scalability: Design the system to handle growth in users and message traffic without compromising performance.

Capacity Estimation and API Design for Chat Application:

Capacity Estimation:

- Active Users: Assumed 5 million daily active users.
- Message Volume: On average, each user sends 80 messages per day, with each message containing around 100 characters.
- Data Volume: This results in approximately 150 GB of data per day, so the system should be horizontally scalable to handle this load.
- Global Distribution: The system should be globally distributed to ensure low latency and handle traffic spikes efficiently.
- Write vs. Read: Writes (message sending) are more frequent than reads (message retrieval), a characteristic similar to WhatsApp, influencing database selection and design.

API Design:

1. Get All Groups: An API to fetch all groups a user has joined, requiring the userID.
2. Leave Group: An API allowing a user to leave a group, requiring the userID and groupID.
3. Join Group: An API to allow a user to join a group, using the userID and groupID.
4. Send Message: An API to send messages, which includes:
    - userID (sender),
    - receiverID (recipient),
    - channelType (private or group).
5. Receive Messages: An API to fetch messages, taking in:
    - userID1, userID2 (participants),
    - channelType (private or group),

- ensuring messages are retrieved in order.

Database Design

Key Design Decisions:

Write-Heavy Load:

- Because the application is more write-heavy than read-heavy, we want a database designed to handle a large volume of writes, particularly tiny messages that must be added often.
- This brings us to Column-Oriented Databases (CODs) such as Cassandra, which are better suited for quick writes and good availability, particularly when working with enormous datasets.

Partitions and clusters:

- Partition Key: The partition key is required to distribute data among nodes and reduce reading latency. For example, for direct conversations between two users, the partition key can be a combination of user_and user_b, guaranteeing that these two users' messages are saved on the same node for faster retrieval.
- Clustering Key: The clustering key determines the order of rows within the division. For direct messages, sorting by message_id guarantees that they are arranged by creation time.

Avoiding Complicated Relationships:

- In this architecture, the application does not require complicated interactions between entities, as seen in e-commerce systems. As a result, a NoSQL database is more suited because we are only interested in basic CRUD operations without complicated joins.
  Optimizing for retrieval:

Optimizing Retrieval:

- Access patterns, such as obtaining messages for two users or a certain group, have a significant impact on schema design. By understanding how data will be retrieved, we eliminate superfluous joins or difficult searches and instead utilize denormalization to store data in ways that improve retrieval for our specific use cases.

Scalable and fault-tolerant:

- Cassandra is a distributed database that scales horizontally. It also provides fault tolerance through replication. This means that even with the high volume of writing, the system can scale efficiently and still provide availability and durability.

Schema Design

1. **Direct Messages (Private Chat):**
   - Partition Key: user_a, user_b (to ensure both users' data is in the same partition for faster access).

- Clustering Key: message_id (to sort the messages by timestamp, assuming message_id is sequential).
- Columns: timestamp, message_content, message_id, etc.

```
CREATE TABLE direct_messages (
    user_a UUID,
    user_b UUID,
    message_id UUID,
    timestamp TIMESTAMP,
    message_content TEXT,
    PRIMARY KEY ((user_a, user_b), message_id)
);
```

2. **Group Chat**:
   - **Partition Key**: group_id, message_id (to ensure the messages are partitioned by group and sorted by message_id).
   - **Clustering Key**: message_id (to order messages within a group).
   - **Columns**: timestamp, message_content, user_id, etc.

   Example:

   - CREATE TABLE group_messages (
     group_id UUID,
     message_id UUID,
     timestamp TIMESTAMP,
     message_content TEXT,
     user_id UUID,
     PRIMARY KEY ((group_id, message_id), message_id)
     );
3. **Group Membership**:
   - Partition Key: group_id (to store the users in each group together).
   - Clustering Key: user_id (to order users in the group).
   - Columns: user_id, status, etc.

- CREATE TABLE group_membership (

  group_id UUID,

  user_id UUID,

  status TEXT,

  PRIMARY KEY (group_id, user_id)

  );

4. **User Profile**:
    - Partition Key: user_id (unique identifier for each user).
    - Columns: password_hash, status, profile_image_url, etc.
    - CREATE TABLE user_profile (

      user_id UUID,

      password_hash TEXT,

      status TEXT,

      profile_image_url TEXT,

      PRIMARY KEY (user_id)

      );

- Snowflake ID Generation: For message_id and other IDs, a service like Twitter's Snowflake ID may be used to generate globally unique IDs across distributed systems.
- Avoiding Hotspots: Cassandra's partitioning method must avoid producing "hot spots" (overloaded nodes). Partition keys should be carefully selected to spread the load equitably across the cluster.
- Scalability: The schema takes advantage of Cassandra's distributed architecture to grow horizontally, with partitions spread across numerous nodes and clusters.
- Broadcasting Messages: The group_membership table aids in broadcasting messages to all group members, allowing the system to effectively retrieve all users inside a group.

High-level Architecture: This high-level architecture describes the major components and interactions for developing a real-time messaging system with peer-to-peer communication. The architecture's basic elements and flow are as follows:

Users (clients) Real-time Communication: Users exchange messages using the WebSocket protocol (which runs over TCP). WebSocket enables low-latency, persistent, bidirectional communication, which is critical for real-time messaging systems.

Message Management:

- This service handles both incoming and outgoing messages. It guarantees that messages are sent over the WebSocket server and saved in a database.
- WebSocket Server: Each WebSocket server manages individual user sessions and routes messages to the appropriate user.

Groups:

- Users can join and leave groups. The HTTP protocol is used to handle requests like joining, leaving, and managing groups since these actions are client-initiated and do not require real-time communication.

Notifications Service:

- The Notifications Service queues messages for offline users and pushes them to their device when they return online.
- Mobile Alerts: If the user is offline, the notification service sends a push notice (for example, to a mobile device) informing them of the availability of a new message.

Presence Service

- The Presence Service records a user's online/offline status. This guarantees that messages are sent to the appropriate WebSocket server or saved for later delivery if the user is not currently available.

User Mapping Service

- Session Mapping: Connects user sessions to particular WebSocket servers. When a user delivers a message, the system must determine which WebSocket server is accountable for the receiver and route the message appropriately.

Data Flow Example (Message Sending Process)

- User Sends a Message: A user sends a message that is first written to the database for persistence.
- Server Mapping: The system uses the User Mapping Service to identify which WebSocket server should handle the message for the recipient.
- Message Delivery: If the recipient is online, the message is delivered directly through the WebSocket server. If the recipient is offline, the message is queued for future delivery via the Notification Service.

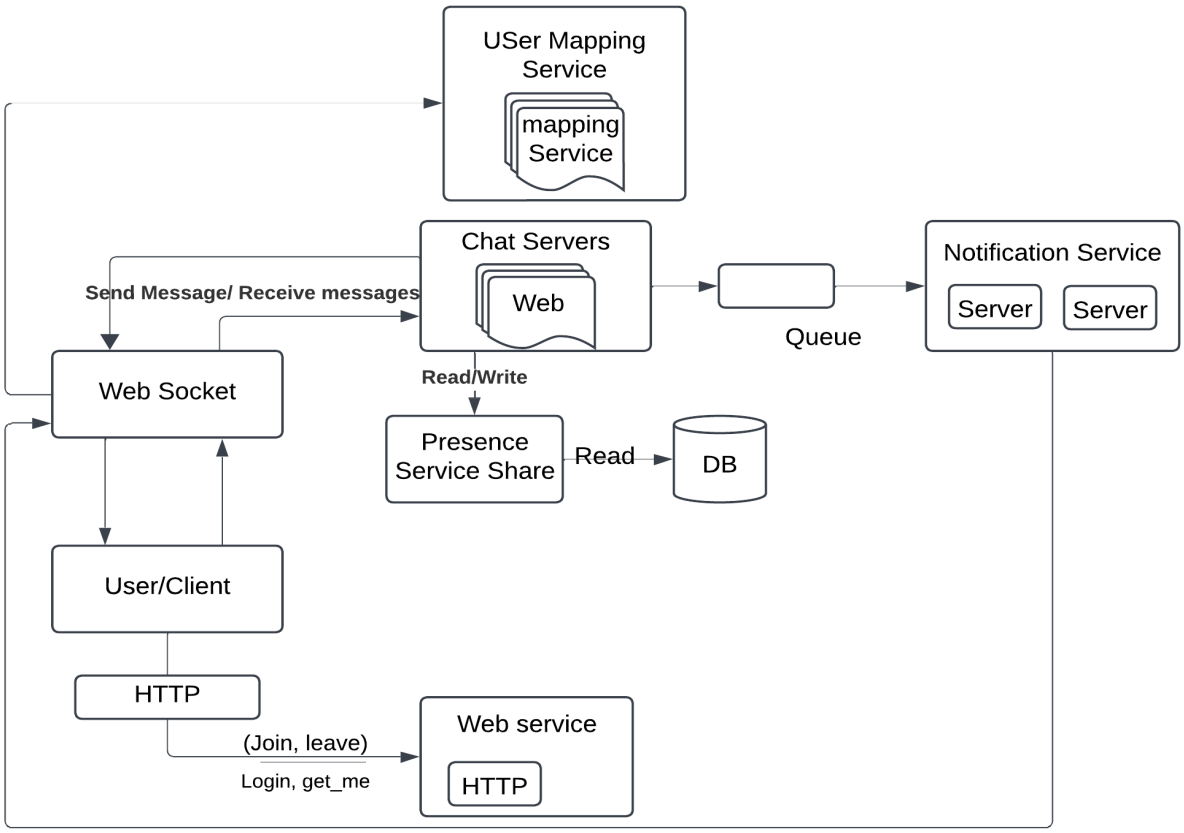
Message Queue and Notification

- If the recipient is offline, the system stores the undelivered messages in a queue.
- Push Notifications: The Notification Service sends a push notification to the offline user, notifying them about the new message.
- Device-Specific Notifications: The message is delivered as a notification to the recipient’s device (e.g., mobile phone).

Handling Offline Users (History Catch-Up)

- Retrieving Undelivered Messages: When an offline user reconnects, the system ensures that they retrieve any undelivered messages. This is achieved through an HTTP request to fetch the message history, which includes the undelivered messages.
- Undelivered Message Table: Messages are stored with metadata like user ID, message ID, and timestamp to ensure proper retrieval when the user reconnects.

High Level Architecture



WhatsApp System Design

The goal of developing WhatsApp (or any other chat-based program) is to grasp the underlying architecture, prioritize critical features, and build a scalable, efficient, and dependable messaging platform. The design should include user-centric features while assuring great performance and simplicity**.**

Key Features to Design

- One-to-One Chat: Foundational to group messaging.
- Group Messaging: Built on the one-to-one messaging design.
- Read Receipts: Notifications for sent, delivered, and read statuses.

The design process:
client-server communication

- Users connect to WhatsApp using a gateway.
- Gateways support external protocols (such as HTTP) but employ lightweight internal communication.

Decoupling Connections:

- A Sessions Microservice controls which users are connected to which gateway boxes.
- Gateways forward communications and delegate session management to a specialized service.

Message Flow

One-to-One Messaging:

- Sender (User A) connects to the gateway and sends a message.
- Gateway forwards the message to the Sessions Microservice.
- Sessions Microservice determines the recipient's (User B's) gateway box and routes the message.

Protocols:

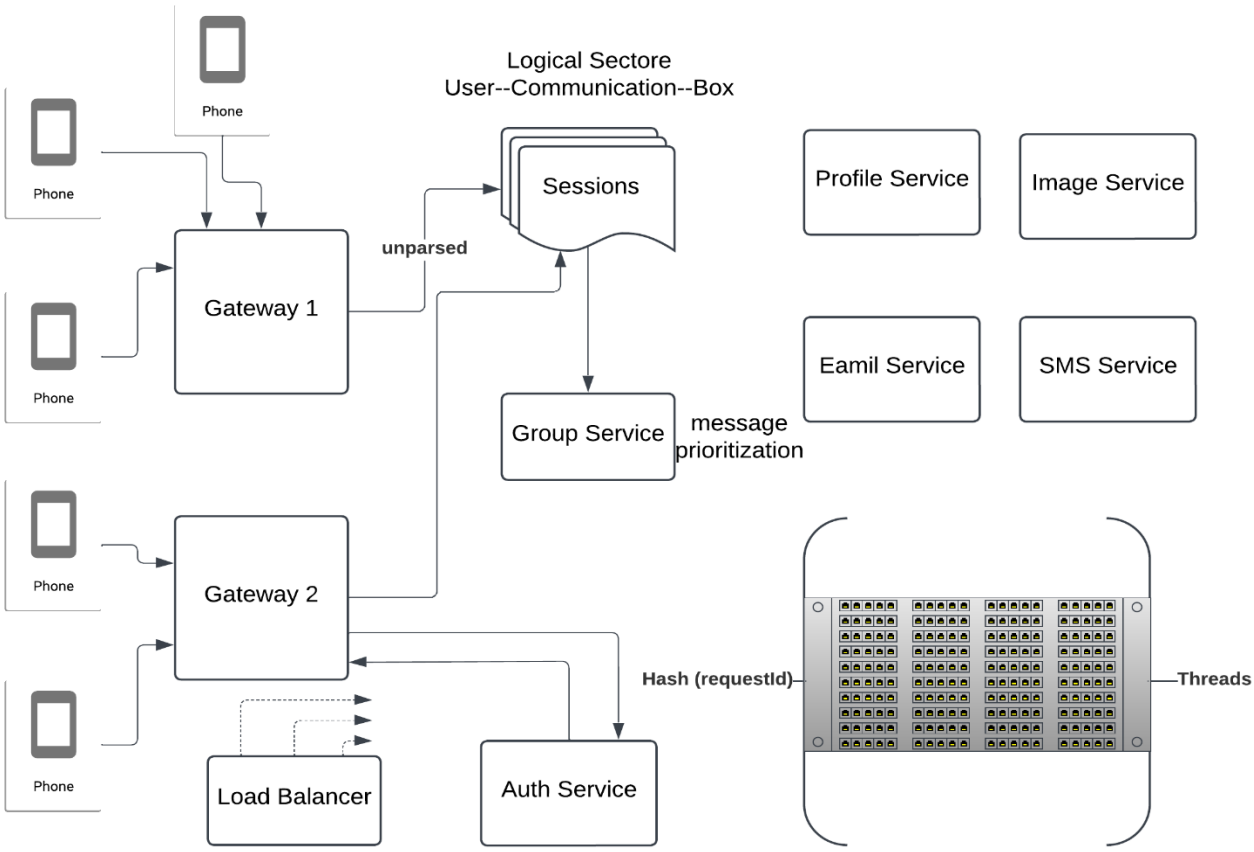
- WebSockets are used for real-time, bidirectional communication. They replace HTTP, which lacks server-to-client messaging capabilities.

Delivery Acknowledgment:

- Once User B receives the message, a delivery acknowledgment is sent back to the sender (User A) via the Sessions Microservice.

Data Persistence:

- Messages are stored in a database for reliability and retries if the recipient is offline.



Optimizations and Considerations

Lightweight Gateway:

- Keeps connections "dumb" to avoid storing transient session data.
- Delegates connection mappings to the Sessions Microservice.

Avoid Single Points of Failure:

- Use multiple instances of the Microservice Sessions and database for redundancy.

Read Receipts:

- When User B reads the message, an acknowledgment is sent, updating the state to "read" for User A.

Message Delivery and Read Receipts

- Delivery Receipt: When User A sends a message to User B:
  - Message Routing: The system determines where User B is connected (e.g., Gateway 2).
  - Notification: User B receives the message.
  - Delivery Receipt: The server sends confirmation back to User A that the message was delivered successfully.
- Read Receipt: When User B opens the chat tab, the system:
  - Send a "read" notification to the server.
  - The server informs User A that the message has been read.

Last Seen/Online Status

- A dedicated Last Seen Microservice tracks user activity:
  - User Activity: Any user interaction (e.g., sending or reading messages) updates the "last seen" timestamp in a database.
  - System Activity: Background activities (e.g., message polling) are flagged and excluded from "last seen" updates.
- Real-Time Status:
  - If the last activity is within a few seconds, the user is shown as "Online."
  - Beyond a set threshold (e.g., 20 seconds), the user's last activity timestamp is displayed.

Group Messaging

- Group Service:
  - Maintains mappings of group IDs to user IDs.
  - Ensure scalability by delegating group membership management to a separate service.
- Message Routing:
  - The Session Service queries the Group Service for members of a group.
  - It determines the respective gateways/users and routes the message efficiently.
- Group Size Limitation:
  - Limits (e.g., 200 members) prevent overloading the system and ensure real-time delivery.

Optimization Techniques

- Gateway Responsibilities:
  - Gateways handle WebSocket connections but minimize processing to conserve resources.
  - Messages are forwarded as raw/unparsed data to reduce computational load.
- Parser Microservice:

    - Converts raw messages into structured objects for further processing.
    - Supports lightweight gateways by offloading processing tasks.
- Consistent Hashing:
    - Balances group membership data across servers.
    - Reduces memory duplication by distributing mappings efficiently.

Future Enhancements

- Service Discovery and Heartbeat Maintenance:
    - Tracks active servers and their health status.
    - Ensures reliable routing and load balancing.
- Authentication Service:
    - Manages user verification securely.
    - Simplifies implementation for scalable user access control.

Key Points:

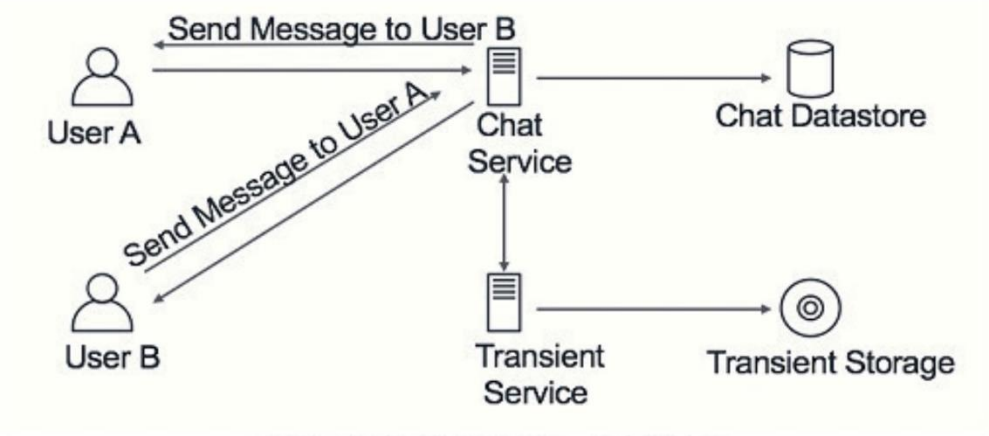
- The architecture effectively separates concerns:
    - Gateways focus on connection management.
    - Services like Group and Last Seen specialize in specific tasks.
- Memory and computational load are minimized through optimizations like raw message handling, consistent hashing, and service delegation.
- Scalability and real-time responsiveness are ensured with efficient routing and data management strategies.

WhatsApp System Design | Large Scale Messenger or Chat like application

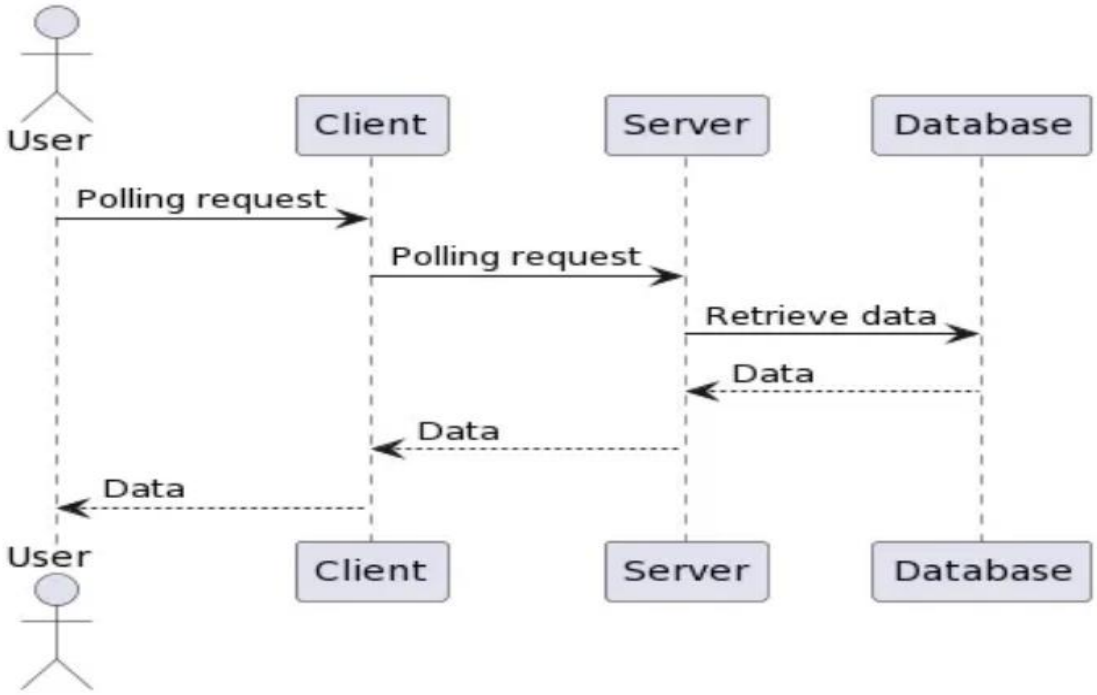
Requirements:

- Should support one on one conversations between users.
- Should show sent/ Delivered/ Read confirmation
- Should support sharing media i.e images/videos/documents.
- Should support push notifications.
- Additional features i.e last seen, encryption.

High Level Diagram



Sequence Diagram:



Transient Data Storage:

- We may use a queue-based technique to store and retrieve transitory messages according to a FIFO principle. We may leverage current cloud-based technologies to accomplish this, such as Amazon SQS or Windows Azure Queue Service. We can utilize these queues to hold temporary messages sent to offline users. All references to these temporary messages are erased from the system once they have been sent to the offline user.

Push Notifications

- There are two techniques to delivering messages to consumers via push technology: client pull and server push. If we use client pull, we may choose between long and short polling. On the other side, the server push technique can be implemented in two ways: WebSocket and Server-Sent Events (SSE). Websockets have become the de facto communication technology for chat apps. We have included more information about it in the area below.
- Using the polling approach, the client requests new data from the server at regular intervals. The trade-off choice to use the polling approach may be made utilizing the data points listed below.

Short Polling (e.g., AJAX-based timer)

  - Pros: more straightforward and not very server-consuming if the time between requests is long
  - Cons: not ideal for scenarios when we need to be notified of server events with minimal delay

Long Polling (e.g., Comet based on XHR)

  - Pros: notifications of server events happen with no delay

- Cons: more complex and consume more server resources

There are two basic approaches for pushing server messages to clients. The first one is WebSocket, a communication protocol. It establishes duplex communication channels over a single TCP connection. Its two-way communication makes it appropriate for scenarios like chat apps. The alternative method is known as Server-sent events (SSE), which allows a server to deliver "new data" to the client asynchronously after the first client-server connection has been established. SSEs are more suited to the publisher-subscriber paradigm, such as real-time streaming stock quotes, Twitter feed updates, and browser alerts**.**