Week 4 Learning Submission

System Design Overview

This week, I worked on understanding two essential system design problems: creating a **URL Shortener** and designing a **Chat Application**, along with exploring how **WhatsApp** handles system design for real-time communication. Below are my learnings, broken down into the main aspects of each system.

1. URL Shortener System Design

A URL Shortener is a tool that converts long URLs into short, manageable ones. The shortened version needs to work just like the long one and should be able to redirect users to the original long URL. The service must be simple but should handle a lot of requests efficiently.

Key Features:

- **Shortening URLs**: Takes long URLs and converts them into short ones.
- **Redirecting URLs**: Redirects from the short URL back to the original long URL when accessed.
- **Tracking (Optional)**: Optionally, it can track how many times each short URL is used.

Functional Requirements:

- **Generate Short URL**: The system should take a long URL and return a short version.
- **Redirection**: When someone uses the short URL, the system should redirect them to the long URL.
- **Collision Handling**: Ensure that each short URL is unique so there are no duplicates.

Non-Functional Requirements:

- **High Availability**: The system should always be accessible and handle a lot of users.
- **Low Latency**: The shortening and redirection processes should be quick.
- **Scalability**: The system should scale as more URLs are shortened.

Architecture:

The URL Shortener needs several key parts:

1. **API**: Users will interact with the system via an API. This API should allow users to shorten URLs and redirect them.
2. **Database**: The shortened URLs need to be stored somewhere, usually in a database. We can use NoSQL databases like DynamoDB to store mappings between short URLs and long URLs.
3. **Key Generation**: The system generates unique short URLs by encoding them using characters from base62 (which includes numbers, lowercase, and uppercase letters).

4. **Cache**: Caching systems like Redis can be used to store frequently accessed short URLs, making retrieval faster.

Optimization:

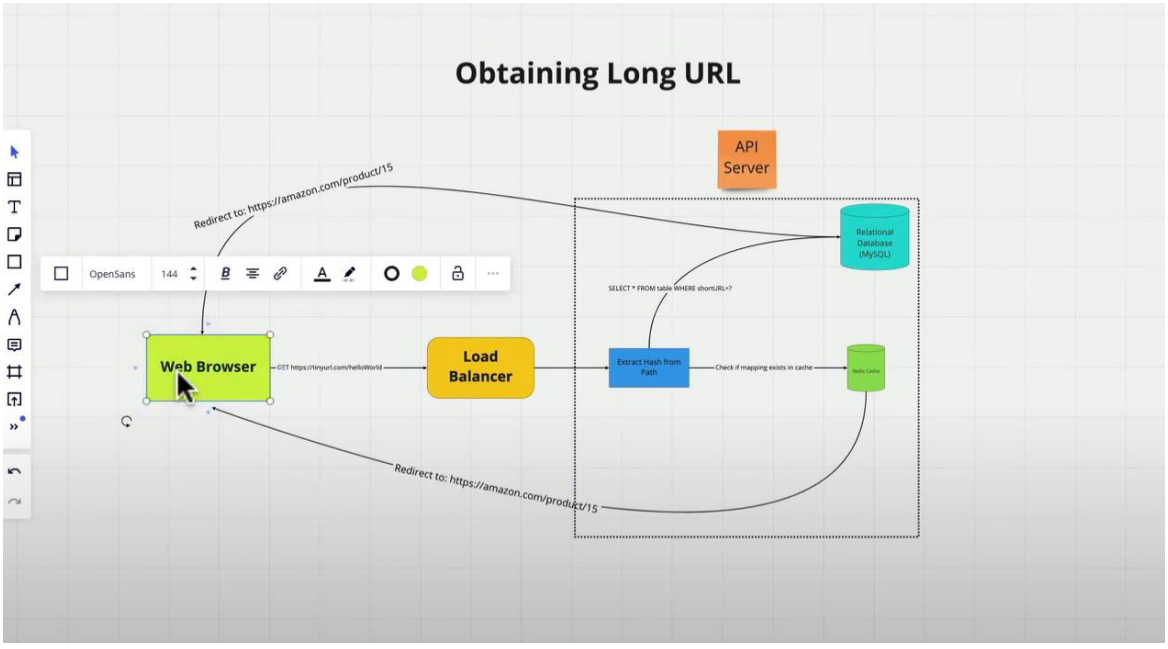
- **Caching**: Popular short URLs can be stored in a cache (like Redis) to avoid repeated database lookups.
- **Load Balancing**: A load balancer will help distribute traffic evenly so that no single server is overloaded.

Retrieve Long URLs from Short URLs

After shortening the URL, when a user tries to access the short URL, the system needs to retrieve the original long URL. The process is straightforward:

1. The user submits the short URL in their web browser.
2. The **load balancer** directs the request to an API server.
3. The API server extracts the **hash** from the short URL.
4. The server checks if the hash exists in the **cache**. If found, the original URL is retrieved from the cache, reducing database load.
5. If not found in the cache, the server queries the **relational database (MySQL)** to fetch the original URL.
6. The original URL is then sent back to the user's browser and redirects them to the original page.

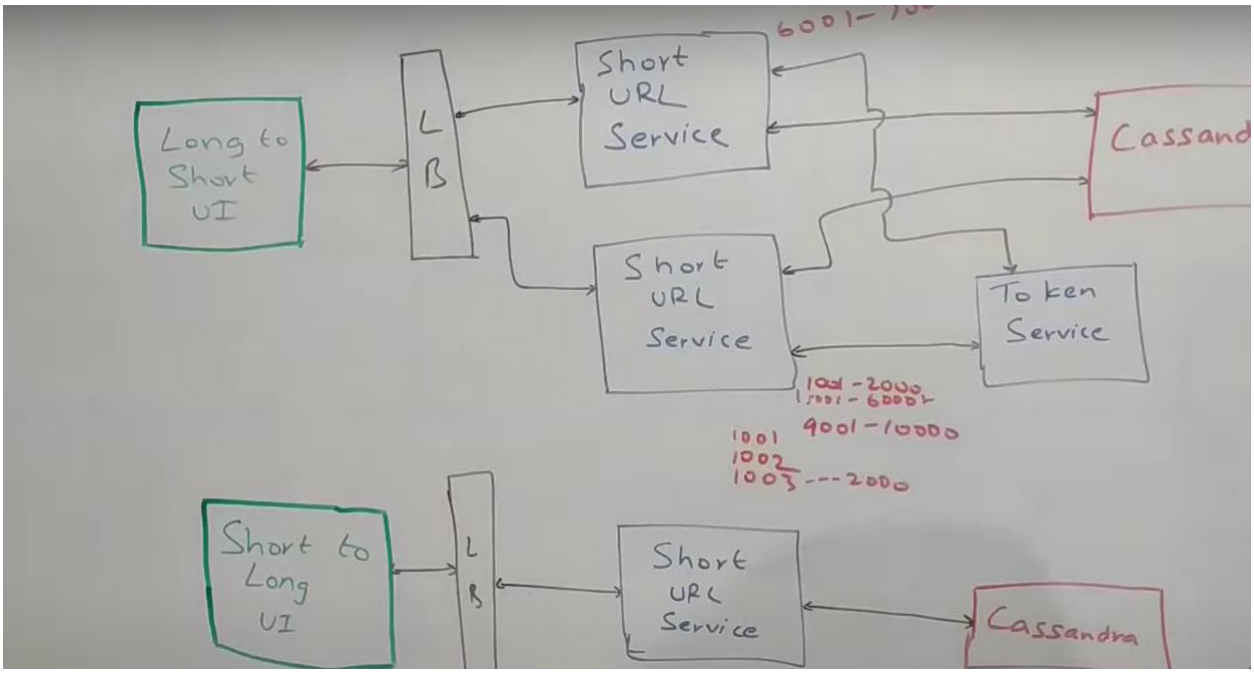
Below is a diagram that visualizes this process:



Generating Short URLs from Long URLs

When a user inputs a long URL, the system shortens it by generating a unique, short URL. This is done using a key generation process, which involves creating a token that maps to the original long URL. The process follows these steps:

1. **User Interface**: The user submits a long URL to the **Long to Short UI**.
2. **Load Balancer**: The request is routed to one of the available **Short URL Services** through the **Load Balancer**, which distributes the incoming requests.
3. **Short URL Service**: The service generates a short URL and stores the mapping in the database (Cassandra).
4. **Token Service**: The **Token Service** ensures that each short URL is unique by maintaining and distributing unique ID ranges.



2. Chat Application System Design

The Chat Application design focuses on enabling real-time communication between users. The system must support both individual conversations and group chats, while ensuring messages are delivered instantly and securely.

Key Features:

- **One-on-One Messaging**: Users can chat with each other in real time.

- **Group Chats**: The system allows multiple users to communicate in a group setting.
- **Media Sharing**: Users should be able to share images, videos, and documents.
- **Notifications**: Users receive notifications for new messages, even if the app is in the background.

Functional Requirements:

- **User Registration/Login**: Users should be able to register and log in to use the app.
- **Message Delivery**: Messages must be delivered in real-time with status updates for sent, delivered, and read messages.
- **Group Messaging**: Group chats should allow users to communicate with multiple participants.
- **Push Notifications**: Users should get notified when they receive new messages.

Non-Functional Requirements:

- **Low Latency**: Messages should be sent and received quickly.
- **High Availability**: The chat application should be available 24/7.
- **Security**: All messages should be encrypted for security.

System Components:

1. **WebSocket Server**: This allows users to send and receive messages instantly by keeping a connection open between the client and the server.
2. **Database**: NoSQL databases like MongoDB store messages and user data.
3. **Message Queue**: A message broker like Kafka ensures messages are delivered reliably, even under high traffic.
4. **Media Storage**: Media files (images, videos) are uploaded to cloud storage like AWS S3.
5. **Push Notifications**: Services like Firebase notify users of new messages.

Process Overview:

- When a user sends a message, it is sent through a WebSocket to the server, which immediately acknowledges receipt and delivers it to the recipient.
- In group chats, the system uses a message queue to ensure that all members of the group receive the message.
- If a user is offline, messages are stored in the database and delivered when they come back online.

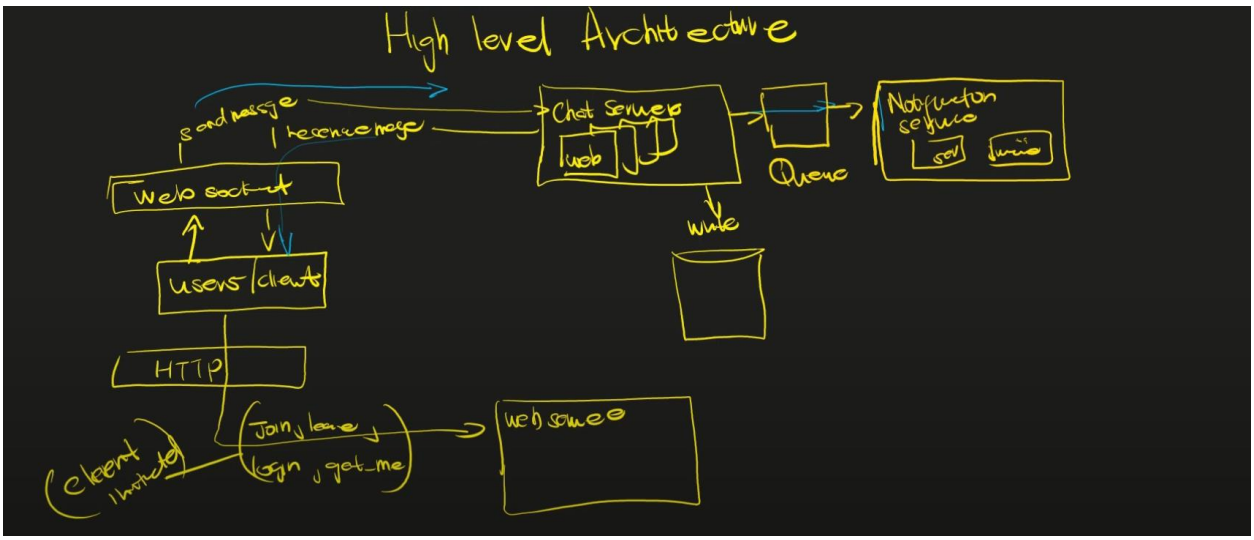
High-Level Architecture

The chat application follows a typical real-time communication architecture, where WebSocket connections maintain live interactions between users and the server. The system is composed of several components working together to ensure messages are delivered quickly and reliably:

1. **WebSocket Servers**: These are responsible for maintaining open, real-time communication between users.
2. **Chat Servers**: Process messages sent by users, store them if needed, and send them to the appropriate recipients.
3. **Notification Service**: Responsible for sending notifications to users, especially when they are not actively using the app.

The message flow looks like this:

- A message is sent from the user through the WebSocket connection.
- The message is processed by the chat server.
- If the recipient is not online, the message is placed in a queue and delivered once the user is available.
- The notification service sends alerts for new messages.



3. WhatsApp System Design

WhatsApp is a real-time messaging app that allows users to send messages, share media, and have group conversations. It also provides end-to-end encryption to ensure privacy.

Key Features:

- **Real-Time Messaging**: WhatsApp uses WebSockets to ensure messages are delivered instantly.

- **Media Sharing**: Users can send images, videos, and documents.
- **Group Messaging**: Multiple users can chat within groups.
- **Offline Messaging**: Messages are stored and delivered when users come back online.
- **Push Notifications**: Users receive notifications for new messages.

Functional Requirements:

- **Messaging**: Send and receive text messages, with status updates for sent, delivered, and read.
- **Media Sharing**: Share images, videos, and other media files.
- **Group Chats**: Messages in group chats should be delivered to all members.
- **Offline Delivery**: Messages should be delivered once users reconnect.

Non-Functional Requirements:

- **Low Latency**: Messages should be sent and received instantly.
- **Scalability**: WhatsApp must handle millions of messages per second.
- **Security**: End-to-end encryption ensures only the sender and recipient can read messages.
- **High Availability**: WhatsApp must be available 24/7, even under heavy load.

System Components:

1. **WebSocket Servers**: WhatsApp uses WebSocket connections to maintain real-time communication.
2. **Message Storage**: Uses a distributed database (Mnesia) to store messages temporarily until they are delivered.
3. **Media Handling**: Media files are compressed, encrypted, and stored in a content delivery network (CDN) for fast access.
4. **Group Messaging**: A message broker (Kafka) is used to handle the distribution of messages in group chats.
5. **Push Notifications**: Notifications are sent using services like Firebase Cloud Messaging or Apple Push Notification Service.

Security:

WhatsApp ensures **end-to-end encryption**, which means messages are encrypted on the sender's device and only decrypted on the recipient's device. This ensures that no third party, including WhatsApp itself, can access the contents of the messages.

Scalability:

WhatsApp handles billions of messages daily and achieves this by:

- Using **WebSocket servers** to maintain open connections for instant communication.
- Storing messages temporarily in the **Mnesia database** until the recipient is online.
- Storing media in a **CDN** to ensure fast media file retrieval.
- Distributing load across multiple servers globally using **load balancers**.

Fault Tolerance:

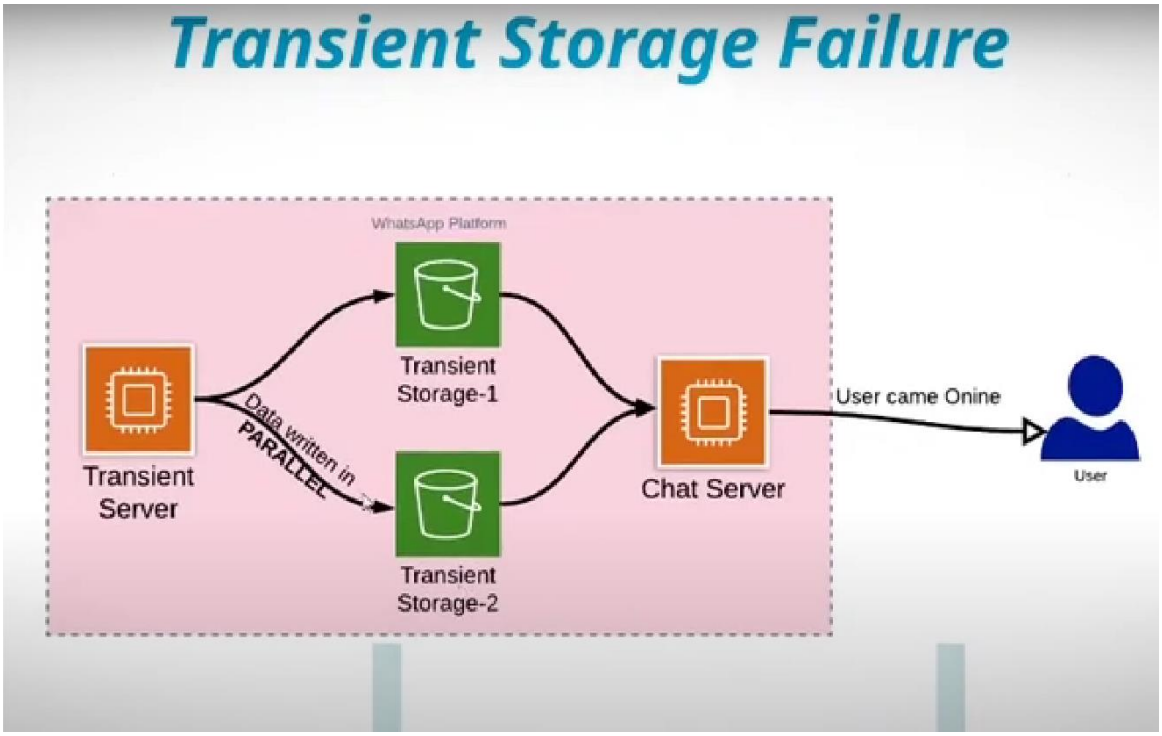
WhatsApp replicates data across multiple servers to ensure that if one server fails, another can take over, minimizing downtime. Regular **backups** are also in place to prevent data loss.

Message Storage and Fault Tolerance

WhatsApp handles billions of messages daily and needs to ensure that no messages are lost, even if there is a storage or server failure. When users are offline, messages are stored in **transient storage** and delivered once the user comes online. The transient storage system ensures that data is stored redundantly to avoid loss in case of failures.

The process involves:

1. Messages are written to **transient storage** in parallel across two systems (Transient Storage-1 and Transient Storage-2).
2. When the user comes online, the **chat server** retrieves the message from one of the storage systems and delivers it to the user.
3. This redundancy ensures fault tolerance, so if one of the storage systems fails, the message can still be retrieved from the other.


Transient Storage Failure
WhatsApp Platform
Transient Server
Data written in PARALLEL
Transient Storage-1
Transient Storage-2
Chat Server
User came Onine
User