WEEK-4

Designing a URL Shortener System:

- Creating a service that creates a short alias or URL that points people to the original long URL is the first step in designing a URL shortening system, such as TinyURL or Bit.ly. The system needs to be dependable, scalable, and effective.
- URL shortening services such as bit.ly or TinyURL are often used to generate shorter aliases for long URLs. You must create this type of web service in which if a user enters a long URL, the service returns a short URL and if the user enters a short URL, the service returns the original long URL.

Example:
Long URL: https://www.example.com/marketing/new-product-launch-campaign
Short URL: shortyx.com/aBc123

- Whenever a user visits shortyx.com/aBc123, they are redirected to the long URL.

- These short URLs are not only easier to share but also help track user interactions efficiently.

- The core function of any URL shortener is to generate a short, unique URL that maps to a longer one. This is achieved using hashing functions, which convert long URLs into shorter, fixed-length strings.

- Hashing ensures that every long URL has a corresponding short URL, which is easy to generate and retrieve. However, as with any system involving hashes, there is a possibility of collision.

- where two different URLs produce the same short URL. To address this, a collision handling strategy is implemented, such as appending additional strings to the hash before re-hashing, ensuring each short URL is unique.

- A key highlight of the design is the use of caching mechanisms, particularly with Redis, to enhance system performance. Caching plays a crucial role in reducing the load on the database by storing recently accessed URL mappings.

- When a user repeatedly accesses a specific short URL, the cache provides the corresponding long URL directly without querying the database. This reduces the retrieval time and improves the overall system's speed and scalability.

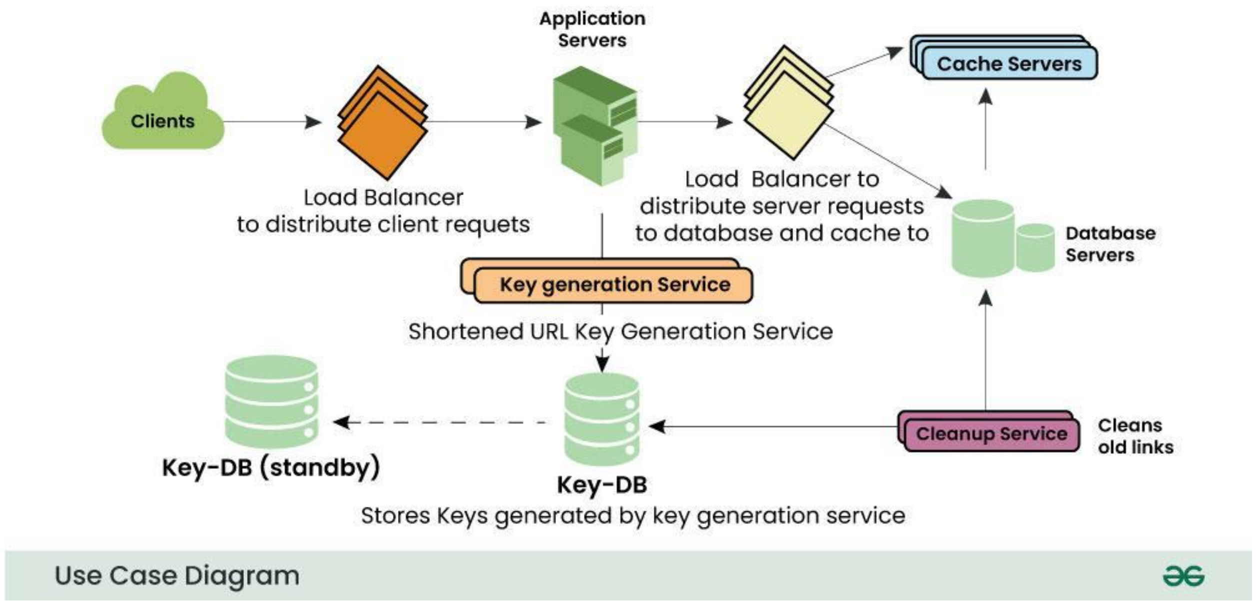
- At the heart of the URL shortener is its database design, which stores mappings of long URLs to short URLs, along with an identification number (ID). This relational database setup ensures efficient lookups and data storage. The system must also account for the lifespan of these links and provide quick access to both frequently and rarely accessed URLs. Deterministic hashing functions ensure that the same long URL always maps to the same short URL, avoiding inconsistencies in URL retrieval.

- An interesting aspect of URL shortening is how the system handles user redirects. Two common types of redirects—301 (permanent) and 302 (temporary)—play a significant role in shaping the user experience.

- A 301 redirect tells browsers and search engines that the original URL has permanently moved to the new address, ensuring that search engines update their records. In contrast, a 302 redirect indicates that the change is temporary, so search engines continue to associate the original URL with its content. This distinction is critical for managing SEO and tracking how often a URL is accessed.

- Understanding user behavior is another vital component in designing an effective URL shortener. Many users are likely to revisit newly created short URLs within a short period, making caching strategies especially useful. By anticipating these patterns, designers can implement intelligent caching mechanisms that boost performance during peak usage times. Additionally, future improvements in the system could include analyzing user access patterns to create a more dynamic and adaptive caching solution.

- The design of a URL shortener is an intricate process, involving everything from hash functions to caching and database design. By understanding how each component fits together, from short URL generation to performance optimization and user access patterns, we can create a robust and efficient system.


Application Servers
Cache Servers
Clients
Load Balancer to distribute client requets
Load Balancer to distribute server requests to database and cache to
Database Servers
Key generation Service
Shortened URL Key Generation Service
Cleanup Service
Cleans old links
Key-DB (standby)
Key-DB
Stores Keys generated by key generation service
Use Case Diagram

- Designing a URL shortening service, to platforms like TinyURL, presents an intriguing system design challenge. The goal is to shorten a long URL into a more manageable short URL and, when accessed, redirect the user to the original long URL. To ensure an effective solution, both functional and non-functional requirements must be considered.

Functional Requirements:

- URL Shortening: The system must be capable of generating a short URL from a long URL, ensuring each is unique to avoid collisions.
- Redirection: When a short URL is accessed, the system must redirect the user to the corresponding long URL.
-

Non-Functional Requirements:

- High Availability: As URL shortening services often operate within large-scale platforms like social media, the system needs to be available consistently.
- Low Latency: Quick responses are crucial for a smooth user experience, especially when dealing with large volumes of traffic.
-

Determining the Length of a Short URL:

- The length of a short URL must account for the expected scale. For a small-scale system, 2-3 characters may suffice, but for services on the scale of Google or Facebook, longer short URLs are necessary to avoid running out of possible combinations. To calculate the necessary length, it is essential to consider the number of unique URLs that need to be shortened over a given period. The total number of possible short URLs is determined by the character set (commonly alphanumeric characters, 62 total). The equation

- $62^n > Y$ can be used to calculate the required URL length,
- where ,
- n is the length of the short URL, and
- Y is the expected number of unique URLs.

Architecture:

- The architecture involves a Short URL Service that handles two primary tasks: generating short URLs and fetching the original long URL from a database when accessed. One potential design flaw is the risk of collisions when generating unique short URLs. To avoid this, a unique ID generation mechanism is required.

Token Service:

- One approach is to use a Token Service that assigns unique ID ranges to each service instance. The instances convert numbers in their assigned ranges to base 62 to generate short URLs. A MySQL database could serve as the backend to maintain records of the assigned ranges, ensuring no overlap. When an instance starts or exhausts its token range, it requests a new range from the Token Service, which assigns a unique range.

Database:

- To store the mapping between long and short URLs, a high-performance database like Cassandra is used due to its ability to scale with a large number of records. Cassandra's distributed architecture prevents a single point of failure, making it an ideal choice for global services.

Handling Failures:

- A potential issue arises if a service instance shuts down before using all tokens in its range, resulting in unused tokens being lost. However, given the vast number of possible short URLs, the loss of a small number of tokens is acceptable. To simplify the design, the system can discard unused tokens and request a new range upon restart.

Scaling and High Availability:

- To ensure the system remains available and scalable, it can be distributed across multiple geographies and data centers, reducing the impact of failures or network latency. Additionally, by increasing the length of token ranges assigned to each instance, the frequency of requests to the Token Service is reduced.

Analytics:

- While the system can function without analytics, tracking usage data, such as top URLs, geographic locations of users, and device types, provides valuable insights. This can be achieved by capturing attributes from incoming requests and sending the data to a message queue system like Kafka for processing. Batch writes to Kafka improve performance by reducing the I/O overhead and network traffic. The data can then be processed using tools like Hadoop or Spark Streaming to generate insights.

<u>Designing a Chat Application: System Design:</u>

- A chat application serves as a software messenger, facilitating text-based instant messaging between users, to popular platforms like WhatsApp and Facebook Messenger.

Key Features and Requirements

- Essential features of a chat application encompass private chats, group chat functionalities, and mechanisms for users to join or leave groups. A critical aspect is the ability to monitor user status, indicating whether they are online or offline, and providing notifications when users are unavailable. Non-functional requirements further shape the application's architecture, emphasizing minimal latency, high availability, and scalability. Prioritizing minimum latency enhances the real-time chat experience, even at the expense of consistency.

- To accurately design the system, capacity estimation is necessary. Assuming a user base of 500 million daily active users, each sending approximately 80 messages daily, the application will generate around 150 gigabytes of data daily.

- Consequently, the database must be horizontally scalable, supporting the anticipated message volume while ensuring high write efficiency. As the usage pattern mirrors that of WhatsApp—where message writes outnumber reads—the design must focus on optimizing write operations in a distributed database environment.

Database Design and Choice

- For the chat application, a NoSQL database such as Cassandra emerges as the optimal choice. This database is particularly suited for high-volume write operations, allowing for rapid data insertion, which is essential for managing small, frequent messages. When modeling data with Cassandra, understanding access patterns is crucial since the database does not utilize traditional relationships found in relational databases. Instead, data is organized in tables based on how it will be accessed, which includes user interactions like sending and retrieving messages, as well as managing group memberships.

- Cassandra's architecture employs partitioning and clustering to optimize data retrieval. The system can define direct message tables with user identifiers, timestamps, and messages, where sender and receiver identifiers serve as partition keys. This design minimizes latency by ensuring related data is stored together, facilitating efficient retrieval. Group chat messages can be similarly structured, although careful consideration of partitioning is necessary to prevent performance degradation due to excessive partitioning.

API and System Architecture

- The architecture of the chat application integrates multiple components, including user interfaces, chat services, and databases. Real-time communication is facilitated through a WebSocket protocol, enabling instantaneous message exchange.

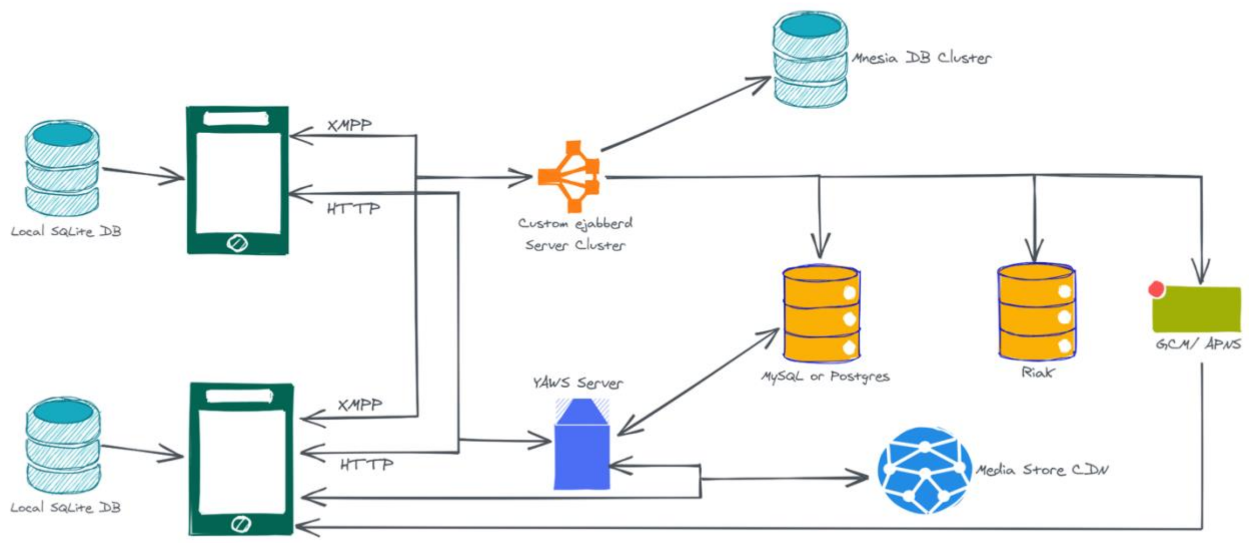
- The API design must accommodate various functionalities, such as joining or leaving groups and sending messages, with a robust authentication mechanism to ensure secure interactions. The inclusion of notification services is also critical for alerting users to new messages when they are offline, enhancing overall usability.

- To manage user presence, the system should implement efficient algorithms to detect user status without overwhelming the infrastructure. Rather than a constant heartbeat check, which can strain resources, a smart approach involves monitoring only frequently interacted users, ensuring timely status updates while optimizing resource usage.

- Designing a chat application necessitates a comprehensive understanding of user needs, system requirements, and architectural components. By prioritizing essential features, optimizing database interactions, and implementing a scalable architecture, developers can create a robust platform that meets the demands of modern communication. The focus on user experience, combined with efficient data handling and real-time capabilities, positions the chat application as an essential tool for users seeking instant connectivity and engagement.

WhatsApp System Design: Chat Messaging Overview

- Designing WhatsApp or any large-scale messaging system involves creating a real-time, highly scalable, and fault-tolerant system that can handle millions of active users, messages, and media exchanges simultaneously. Below is a detailed breakdown of the core components, architecture, and challenges involved in designing such a system.



Key Features of WhatsApp Messaging:

1. User Authentication:
    - WhatsApp uses phone numbers as identifiers, with SMS-based verification for authentication.
    - A session management system ensures users stay logged in across devices.
2. Real-Time Messaging:
    - Real-time communication is facilitated using WebSockets or XMPP (Extensible Messaging and Presence Protocol), enabling a persistent connection between the app and server for fast message delivery.
3. Message Delivery:
    - Point-to-point messaging is used to send messages from the sender to the server, which relays them to the recipient.
    - Offline messaging is supported, where messages are queued on the server and delivered when the recipient reconnects.
4. End-to-End Encryption (E2EE):
    - WhatsApp ensures user privacy with end-to-end encryption, meaning only the sender and recipient can decrypt the messages.
    - Encryption keys are managed on the user devices, and not stored on the server.
5. Group Messaging:
    - Group messaging allows multiple users to communicate, with messages broadcasted to all members.
    - The server manages group membership and ensures the correct delivery of messages to all participants.
6. Media Sharing:
    - Users can share images, videos, documents, etc. Media files are uploaded to the server, stored temporarily, and delivered to the recipient.

High-Level Architecture:

1. Client Side (Mobile App):

    - The WhatsApp mobile app (iOS/Android) provides the user interface, allowing users to send messages, share media, create groups, and view message statuses (delivered, read).
    - The client connects to the server using WebSockets to maintain an always-on connection for real-time communication.
2. Backend Servers:
    - WebSocket/XMPP Server: Handles real-time message delivery, keeping connections open for instant messaging between users.
    - REST API: Provides non-real-time functionalities such as user registration, fetching chat history, or managing user profile settings.
    - Media Server: Handles media uploads and storage (e.g., images, videos), typically using cloud storage solutions.
    - Authentication Service: Verifies users based on phone numbers via SMS and manages user sessions.
3. Database (Message Storage):
    - NoSQL databases like Cassandra or HBase are used to store messages, user data, and metadata due to their scalability and ability to handle high write/read traffic.
    - Message Queuing: Messages are queued if the recipient is offline and delivered when they reconnect.
4. Encryption and Security:
    - End-to-End Encryption (E2EE) ensures messages are encrypted on the client-side and only decrypted by the recipient.
    - Encryption keys are never shared with the server, ensuring privacy.
    - Push Notifications: For offline users, push notifications are used to alert them of new messages.
5. Media Storage and Delivery:
    - Media files (images, videos, audio) are uploaded to cloud storage (e.g., AWS S3) and temporarily stored on the server.
    - Once uploaded, a download link is sent to the recipient, who retrieves the media from the storage server.
6. Group Management:
    - Groups are maintained on the server, including metadata about group members and roles (admin, participants).
    - Group messages are broadcast to all participants in the group via WebSocket connections.
7. Load Balancers:
    - Load balancers distribute incoming traffic across multiple WebSocket servers to avoid bottlenecks and ensure scalability.
    - Used to handle millions of simultaneous users and message exchanges.
8. Distributed Caching:
    - Redis or Memcached can be used for caching frequently accessed data like user sessions or recently sent messages to reduce load on the database.

Detailed Message Flow:

1. User Authentication:
    - The user logs in by verifying their phone number via an SMS-based token.

- Upon successful verification, the backend generates a session token for the user.
    - The token is stored locally and used for subsequent API and WebSocket requests.
2. Establishing WebSocket Connection:
    - Once the user is authenticated, the client establishes a persistent WebSocket connection with the server.
    - This WebSocket channel is used for sending and receiving messages in real-time.
3. Sending a Message:
    - The sender's client app sends a message through the WebSocket connection.
    - The server processes the message, adds metadata (timestamp, sender ID, etc.), and stores it in the database.
    - The server forwards the message to the recipient(s) via their WebSocket connections if they are online.
    - If the recipient is offline, the message is queued in the message queue, and a push notification is sent to notify the recipient of the new message.
4. Message Persistence and Syncing:
    - The server stores the message in a distributed database (e.g., Cassandra) for future retrieval.
    - If the user logs in from another device, the chat history can be synced from the server.
5. Read Receipts & Status Indicators:
    - The server tracks the delivery status of each message (sent, delivered, read).
    - The sender's client is updated when the message is delivered or read.
    - Typing indicators are handled similarly, with real-time updates broadcast to the appropriate users.

Scaling and Performance Considerations:

1. Sharding & Partitioning:
    - To handle billions of messages daily, WhatsApp uses sharding, where messages and user data are divided across multiple servers based on some partitioning strategy (e.g., user ID).
2. Load Balancing:
    - Load balancers distribute traffic across multiple WebSocket and API servers to prevent any server from becoming a bottleneck.
3. Message Queues for Offline Users:
    - For users who are offline, messages are queued on the server (e.g., using Kafka or RabbitMQ) and delivered once the user reconnects.
4. Eventual Consistency:
    - In large-scale systems like WhatsApp, eventual consistency is employed where different replicas of data might not be immediately consistent, but they will eventually converge to the same state.

Challenges and Solutions:

1. Handling Millions of Connections:
    - Managing millions of WebSocket connections requires efficient server and resource management.

    - Horizontal scaling of WebSocket servers and using load balancers helps distribute the load.
2. Message Delivery Reliability:
    - Using idempotency ensures that messages are only delivered once, even in the case of network issues or retries.
    - Implementing acknowledgment mechanisms ensures that both the sender and the recipient can track message status (sent, delivered, read).
3. Privacy and Security:
    - End-to-end encryption ensures messages remain private between the sender and the recipient.
    - Data encryption at rest and in transit further protects sensitive information.
4. Handling Media Files:
    - Large media files are handled by offloading them to cloud storage services like AWS S3 or Google Cloud Storage and delivering download links rather than streaming them directly through the server.