URL Shortener system design

I have learned how to create a service that converts long URLs into shorter, more manageable links while ensuring scalability and reliability.

And how the system uses techniques like hashing to generate unique short URLs and employs a database to store mappings between the original and shortened links.

I also learned about the key components of the system, like load balancers, which help distribute traffic evenly across servers to handle a large number of requests, and the service also uses a database to store the original and shortened URLs, and it’s important to ensure there are no collisions (when two URLs generate the same short link). HTTP redirection is used to take users from the short link back to the original website, and the system needs to be designed in a way that is scalable and efficient for handling large volumes of data and traffic.

TinyURL System Design

I’ve learned how the system is structured to meet both functional and non-functional requirements. The main goal is to convert long URLs into shorter ones and ensure seamless redirection when users access the short URLs. To handle high traffic and guarantee reliability, the system must have low latency and high availability.

The process starts by considering factors such as the length of the short URLs, which depends on traffic and longevity. The basic system architecture includes a UI for inputting long URLs, a service to generate the short URLs, and a database to store these mappings. To avoid collisions (when two URLs generate the same short link), the system uses a counter (like Redis or token service) and converts this unique number to a base 62 representation. A key challenge is managing traffic efficiently without overloading Redis or causing a single point of failure, so scaling strategies involve distributing the load across multiple services or databases like MySQL or Cassandra.

The service also incorporates analytics, capturing user data such as platform and location to optimize performance and latency. However, writing this data to Kafka for analytics needs to be done carefully to avoid impacting system speed. To address potential data loss, bulk writes or scheduled writes are used to balance performance with reliability.

This overall design prioritizes scalability, efficiency, and fault tolerance to ensure the service remains highly available and responsive under high demand.

From chat application system design video, I have learned how to structure a real-time messaging platform. The design process focuses on key components such as a client-server architecture, message queue for handling real-time communication, and a database for storing messages and user data. The system emphasizes scalability, reliability, and low latency, often using WebSockets for instant communication and load balancing to manage traffic. Features like user authentication, message history retrieval, and offline support are also essential considerations to create an efficient, user-friendly chat application.

In the video of WhatsApp system design, I learned about how a large-scale chat messaging system is structured to handle millions of users simultaneously. The key elements of the design include real-time messaging, ensuring reliability, and delivering messages even under heavy loads. WhatsApp uses a client-server architecture where each message passes through a server before being delivered to the recipient. This enables synchronization across multiple devices, encryption for security, and the ability to store message history.

**One-to-One Messaging:**

In one-to-one messaging, the system follows a **client-server-client** model. When a user sends a message:

The message is sent from the client to the server.

The server stores the message temporarily and forwards it to the recipient.

If the recipient is online, they receive the message immediately; if not, the server stores the message and delivers it when they come online.

The message queue ensures that no messages are lost, and WhatsApp uses **push notifications** to notify the recipient even if the app is not open.

Key concepts:

**End-to-end encryption** ensures that only the sender and receiver can read the message.

**Ack system**: After the message is sent, the sender receives delivery confirmations in the form of single (sent), double (delivered), and double-blue (read) ticks.

**2. Group Messaging:**

For group chats, the system scales by sending messages to multiple participants:

When a user sends a message to a group, the server handles this as a **fan-out** operation.

The server stores the message and forwards it individually to all group members, ensuring each member gets a copy of the message.

The same encryption applies, where each recipient receives their own encrypted version of the message.

To optimize, group messaging can use **broadcasting** techniques to distribute the message efficiently without sending multiple requests.

**Delivery status** for each group member works similarly to one-to-one messaging, where the sender can see delivery and read receipts for each member.

**3. Sending Images:**

When sending images (or any media files), the process is slightly more complex:

**Client Upload**: The image is first uploaded to a dedicated **media storage server**. The media is stored as a binary object and given a unique identifier (e.g., a URL).

**Message with Link**: Once the image is uploaded, the sender’s client generates a message that includes the link to the image on the media server. This message is then encrypted and sent to the recipient(s).

**Recipient Download**: The recipient(s) receive the message, which contains the link. The recipient's client fetches the image by downloading it from the media server via the link, while keeping the entire operation encrypted.

**Caching**: To optimize performance and reduce load on the server, images are often cached locally on the recipient’s device.

**Media storage** is separate from text message storage to handle large files efficiently.

**Compression**: Images and videos are often compressed to reduce load time and server bandwidth, though WhatsApp also allows users to send original-quality media when required.

Overall, WhatsApp uses robust systems like **load balancers**, **sharding**, and **message queues** to handle the scale, ensuring real-time delivery with low latency, whether for one-to-one, group chats, or media sharing.