Designing Whatsapp Messenger | System Design

Last Updated : 18 Mar, 2024

Ever thought about how this widely-used messaging app actually works behind the scenes? This article is your guide to the system design of WhatsApp. From handling tons of messages to making sure your chats are secure, we'll explore the technical aspects that keep this app running smoothly and managing things like managing data, keeping your messages private, and the challenges of making sure your texts arrive lightning-fast.

Important Topics for WhatsApp System Design

Requirements

The WhatsApp messenger design should meet below requirements:

Functional Requirement

- **Conversation**: The system should support one-on-one and group conversations between users.
- **Acknowledgment**: The system should support message delivery acknowledgment, such as sent, delivered, and read.

- **Sharing**: The system should support sharing of media files, such as images, videos, and audio.
- **Chat storage**: The system must support the persistent storage of chat messages when a user is offline until the successful delivery of messages.
- **Push notifications**: The system should be able to notify offline users of new messages once their status becomes online.

Non-Functional Requirement

- **Low latency:** Users should be able to receive messages with low latency.
- **Consistency**: Messages should be delivered in the order they were sent.
- **Availability**: The system should be highly available. However, the availability can be compromised in the interest of consistency.
- **Security:** The system must be secure via end-to-end encryption. The end-to-end encryption ensures that only the two communicating parties can see the content of messages. Nobody in between, not even WhatsApp, should have access.
- **Scalability**: The system should be highly scalable to support an ever-increasing number of users and messages per day.

Capacity Estimation

Storage Estimation:

> *100 billion messages are shared through WhatsApp per day and each message takes 100 bytes on average*
> *100 billion/day∗100 Bytes =* ***10 TB/day***
> *For 30 days, the storage capacity would become the following:*
> *30∗10 TB/day =* ***300 TB/month***

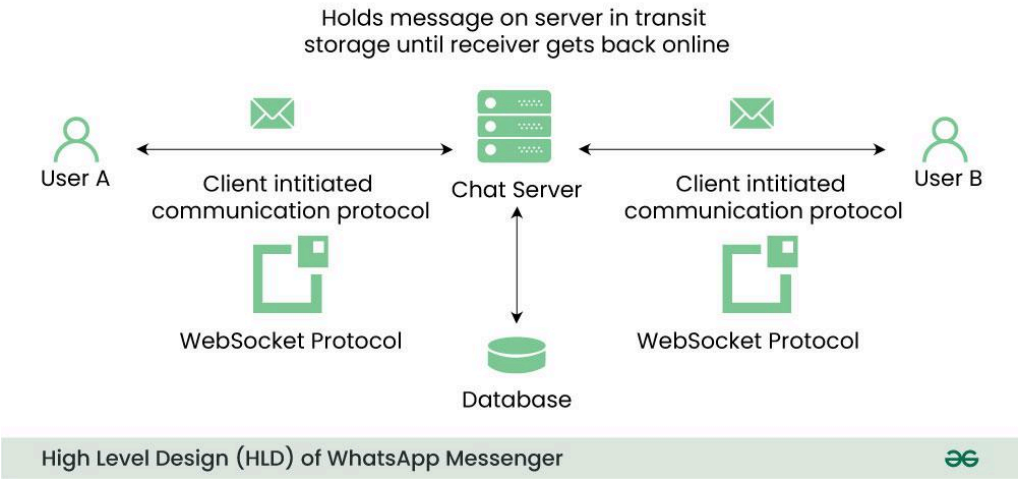
Bandwidth Estimation:

> *According to the storage capacity estimation, our service will get 10TB of data each day, giving us a bandwidth of 926 Mb/s.*
> *10 TB/86400sec* ***≈ 926Mb/s***

Number of servers estimation:

WhatsApp handles around 10 million connections on a single server, which seems quite high for a server.
No. of servers = Total connections per day/No. of connections per server = 2 billion/10 million = ***200 servers***
So, according to the above estimates, we require 200 chat servers.

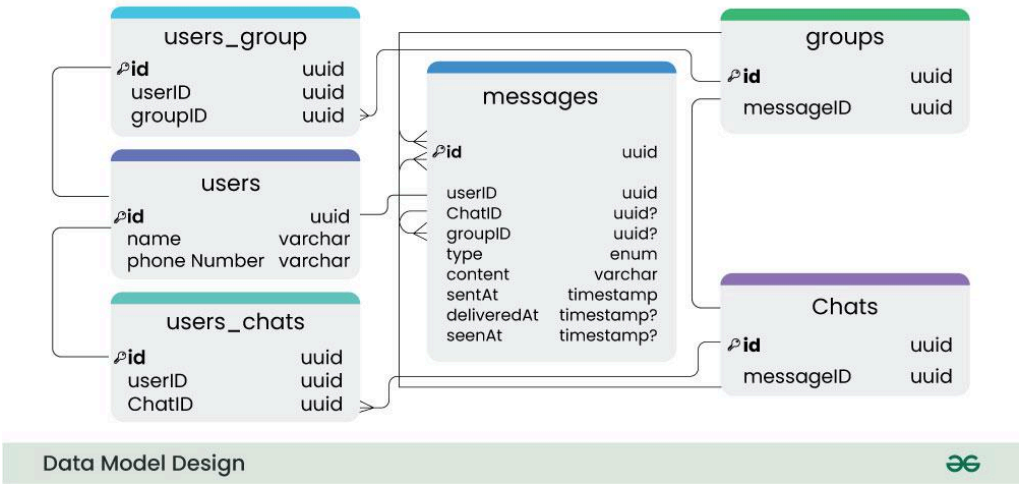
High Level Design (HLD) of WhatsApp Messenger



The following steps describe the communication between both clients:

- User A and user B create a communication channel with the chat server.
- User A sends a message to the chat server.
- Upon receiving the message, the chat server acknowledges back to user A.
- The chat server sends the message to user B and stores the message in the database if the receiver's status is offline.
- User B sends an acknowledgment to the chat server.
- The chat server notifies user A that the message has been successfully delivered.
- When user B reads the message, the application notifies the chat server.
- The chat server notifies user A that user B has read the message.

Data Model Design:



- **users**: This table will contain a user's information such as name, phoneNumber, and other details.
- **messages**: This table will store messages with properties such as type (text, image, video, etc.), content, and timestamps for message delivery. The message will also have a corresponding chatID or groupID.
- **chats**: This table basically represents a private chat between two users and can contain multiple messages.
- **users_chats**: This table maps users and chats as multiple users can have multiple chats (N:M relationship) and vice versa.
- **groups**: This table represents a group between multiple users.
- **users_groups**: This table maps users and groups as multiple users can be a part of multiple groups (N:M relationship) and vice versa.

API Design

Send message

sendMessage(message_ID, sender_ID, reciever_ID, type, text=none, media_object=none, document=none)

This API is used to send a text message from a sender to a receiver by making a POST API call to the /messages API endpoint. Generally, the sender's and receiver's IDs are their phone numbers.

Get Message

getMessage(user_Id)

Using this API call, users can fetch all unread messages when they come online after being offline for some time.

Upload File

uploadFile(file_type, file)

We can upload media files via the uploadFile API by making a POST request to the /v1/media API endpoint. A successful response returns an ID that's forwarded to the receiver.

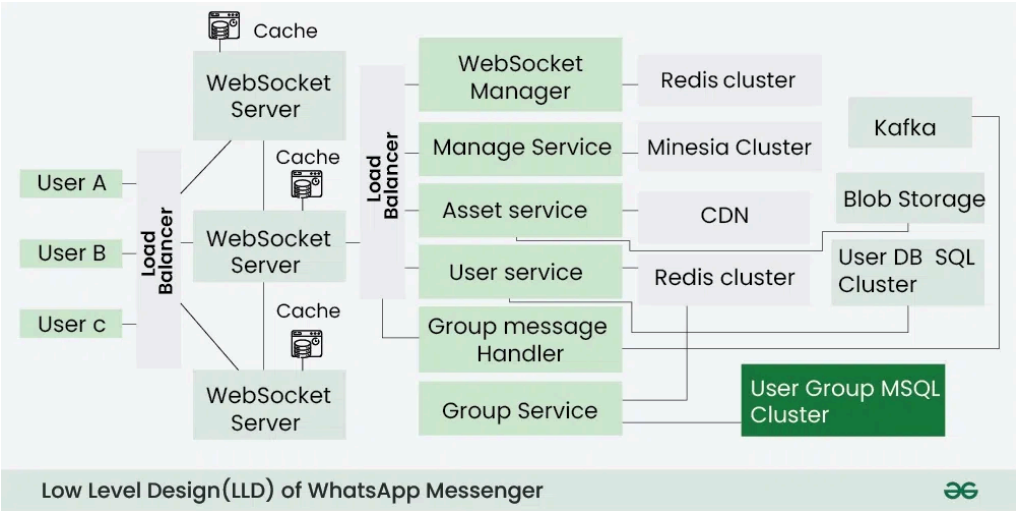
Download Media File

downloadFile(user_id, file_id)

Architecture

We will be using microservices architecture since it will make it easier to horizontally scale and decouple our services. Each service will have ownership of its own data model.

Low Level Design (LLD) of System Design



1. Connection with Websocket Server

In WhatsApp, each active device is connected with a WebSocket server via WebSocket protocol. A WebSocket server keeps the connection open with all the active (online) users. Since one server isn't enough to handle billions of devices, there should be enough servers to handle billions of users.

The responsibility of each of these servers is to provide a port to every online user. The mapping between servers, ports, and users is stored in the WebSocket manager that resides on top of a cluster of the data store. In this case, that's Redis.

2. Send or receive messages

The WebSocket manager is responsible for maintaining a mapping between an active user and a port assigned to the user. Whenever a user is connected to another WebSocket server, this information will be updated in the data store. A WebSocket server also communicates with another service called message service.

Message service is a repository of messages on top of the Mnesia database cluster. It acts as an interface to the Mnesia database for other services interacting with the databases. It is responsible for storing and retrieving messages from the Mnesia database. It also deletes messages from the Mnesia database after a configurable amount of time. And, it exposes APIs to receive messages by various filters, such as user ID, message ID, and so on.

3. Send or receive media files

We have another service called the asset service, which is responsible for sending and receiving media files.

- The media file is compressed and encrypted on the device side.
- The compressed and encrypted file is sent to the asset service to store the file on blob storage.
- The asset service assigns an ID that's communicated with the sender. The asset service also maintains a hash for each file to avoid duplication of content on the blob storage.
    - For example, if a user wants to upload an image that's already there in the blob storage, the image won't be uploaded. Instead, the same ID is forwarded to the receiver.
- The asset service sends the ID of media files to the receiver via the message service. The receiver downloads the media file from the blob storage using the ID.
- The content is loaded onto a CDN if the asset service receives a large number of requests for some particular content.

4. Support for Group messages

Since user A is connected to a WebSocket server, it sends a message to the message service intended for Group A.The message service sends the message to Kafka with other specific information about the group. The message is saved there for further

processing. In Kafka terminology, a group can be a topic, and the senders and receivers can be producers and consumers, respectively.

Now, here comes the responsibility of the group service. The group service keeps all information about users in each group in the system. It has all the information about each group, including user IDs, group ID, status, group icon, number of users, and so on. This service resides on top of the MySQL database cluster, with multiple secondary replicas distributed geographically. A Redis cache server also exists to cache data from the MySQL servers. Both geographically distributed replicas and Redis cache aid in reducing latency.

The group message handler communicates with the group service to retrieve data of Group/A users. In the last step, the group message handler follows the same process as a WebSocket server and delivers the message to each user.

Approach to achieve the below system attributes

Non-functional Requirements	Approaches
Minimizing latency	• Geographically distributed cache management systems and servers • CDNs
Consistency	• Provide unique IDs to messages using Sequencer or other mechanisms • Use FIFO messaging queue with strict ordering
Availability	• Provide multiple WebSocket servers and managers to establish connections between users • Replication of messages and data associated with users and groups on different servers • Follow disaster recovery protocols
Security	• Via end-to-end encryption
Scalability	• Performance tuning of servers • Horizontal scalability of services