System Design: WhatsApp

Interview Daemon Jan 17 · 7 min read

(Peer to Peer Messaging App)

Source: interviewdaemon.com (Link)

System Design Whatsapp. **WhatsApp Messenger**, or simply **WhatsApp**, is an American centralized messaging and Voice over IP (VoIP) service. It allows users to send text messages and voice messages, make voice and video calls, and share images, documents, user locations, and other media.

Introduction

It allows users to send text messages and voice messages, make voice and video calls, and share images, documents, user locations, and other media.

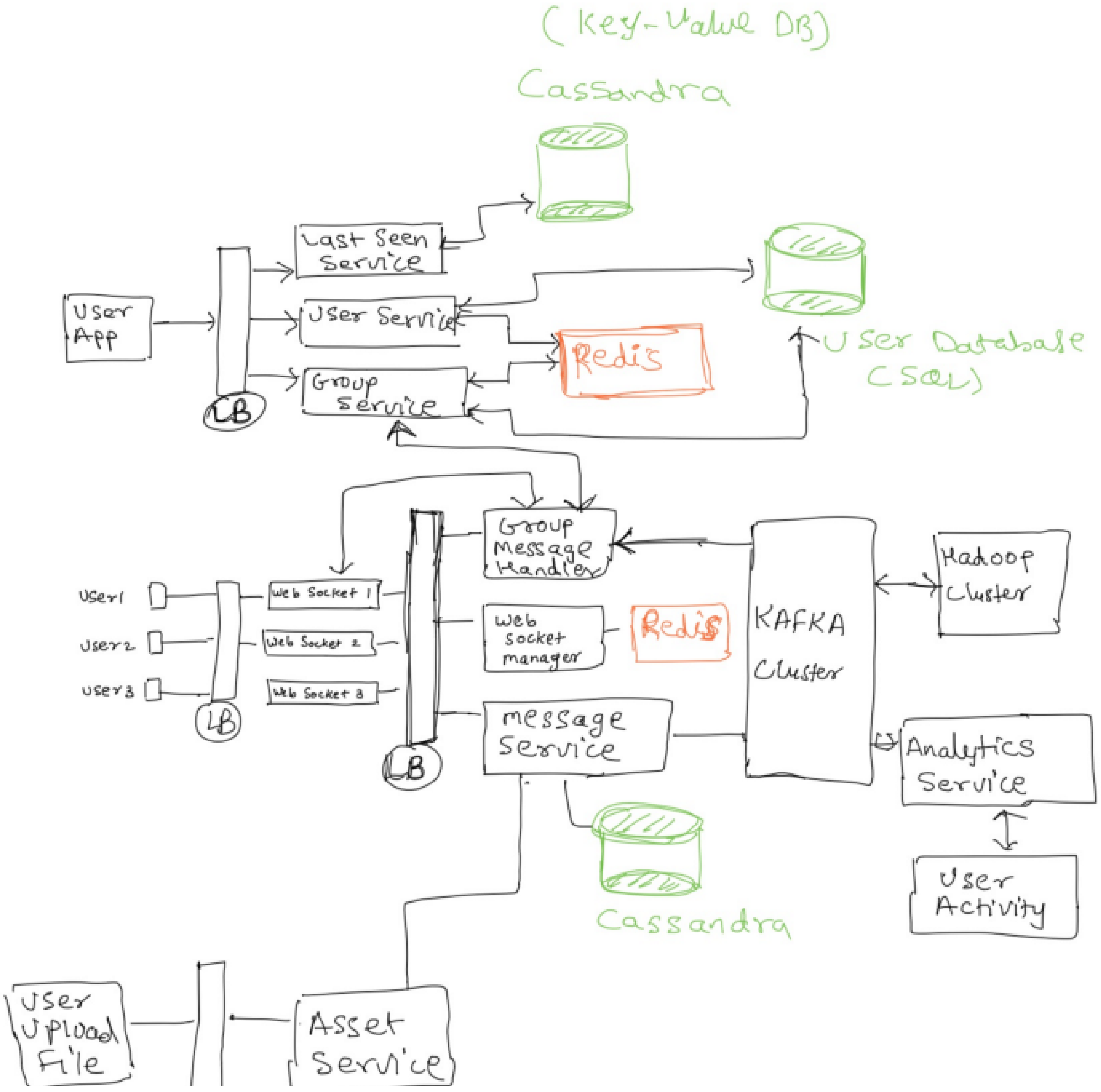
Requirements

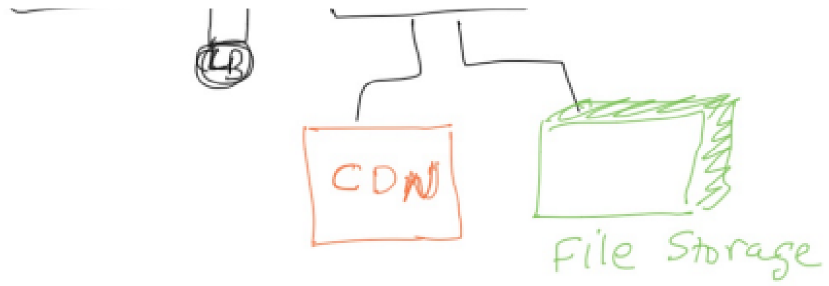
- Support **One to One Chat**
- Support **Group Chat**
- Support **Image, File, Video Sharing**

- Support **Last Seen Time** for User
- Support **Read/Delivered Tick** Receipt for messages
- System should be **Highly Available**
- System should be **Scalable**
- System should be **Low Latency**

High Level System Design

At a high-level, we need to support multiple micro-services in the current system.





Whatsapp Message Flow

- **User-1 sends message to User-2**
- User-1 is connected to the Web socket handler WSH1. So when User-1 decides to send a message to User-2, it communicates this to WSH1. WSH1 will then talk to the web socket manager to find out which web socket handler is handling User-2, and the web socket manager will respond that User-2 is connected to WSH2.
- WSH1 will make a parallel call to **Message service** which will save the message into **Cassandra** and assign it a **message id say M1**. Now that WSH1 has a message id M1 and knows that U2 is connected to WSH2, it will talk to WSH2
- **User-1 sends in Offline (Sending the message without Internet)**
- We can store the message in device cache.
- As soon as User-1/Device is online we will pull all the messages from device local database and send them.

If User-2 is Online

- User-2 could be on the chat screen for User-1, in which case M1 will be immediately delivered and seen or it might not be on U1's chat screen in which case the message will only be delivered.

If User-2 is Offline

- WSH1 talks to the web socket manager to find out which handler is connected to User-2, and finds out User-2 is not connected to anyone, so that part of the flow ends here.

- It is also talking to the Message Service where the message is stored with an id, say, M2.
- Now when User-2 comes online, let's say it connects to WSH3, the first thing WSH3 does is to check in with message service if there are any undelivered messages for User-2 and these messages will be sent to User-2 via WSH3.
- And then WSH3 will communicate this change in status to WSH1 in the previously discussed manner.

When User-1 Sends Message and is is Offline

- User-1 sends a message to User-2 and then User-1 goes offline.
- User-2 has received the message M1 but User-1 doesn't know about it.
- So the status will be stored in **Cassandra** against the message as delivered or seen for at least as long as it is not communicated to User-1.
- Once User-1 receives the status of the message it can be deleted if required.

Race Condition

- If User-2 was offline and when it comes online Web Socket Manager talks to WSH3 to get all messages that were not delivered to User-2.
- While talking User-2 again goes back offline.
- 2 things an happen

1. WSH3 gets the message and send to WSH1. WSH1 thinks User-2 is offline and stores the message in message service, and thinks that its job is complete. (In reality User-2 is not online)
2. User-2 was online for sometime and all the messages were delivered.

- To avoid this Web Socket Handlers will do Web Polling to keep track of messages after fixed interval

Component Design

1. Web Socket Handler

- A web socket handler is a **lightweight** server that will keep an open connection with all the active users.
- One Web Socket can have around 65k — 70k connections.
- Scale of Facebook/Whatsapp One web socket handler is not enough to support all users, so there will be multiple web socket handlers in our system.
- Reduce calls to Handler by caching 2 things
- Which users are connected to itself, so if U1 and U2 were both connected to the same handler, that call to the web socket manager could be avoided.
- Information about recent conversations, like which user is connected to which handler.

2. Web Socket Manager

- A web socket handler will be connected to a **Web Socket Manager** which is a repository of information about which web socket handlers are connected to which users.
- It sits on top of a **Redis** which stores two types of information
- Which user is connected to which web socket handler
- What all users are connected to particular web socket handler

3. Message Service

- **Message Service**, which is a repository of all messages in the system.
- A Web socket handler, while talking to a web socket manager will also, in parallel, talk to a **Message Service**, which is a repository of all messages in the system.
- It will expose APIs to get messages by various filters like user id, message-id, delivery status, etc.
- This **Message Service**, sits on top of **Cassandra**.

- Now we can expect that new users will keep getting added to the system every day and all users, old and new, will keep having new conversations every day i.e. message service needs to build on top of a data store that can handle **ever-increasing data**.
- Facebook keeps messages permanently while whatsapp/snapchat deletes the messages once its read and keeps in permanent storage.

4. User Service

- Stores user-related information like name, id, profile picture, preferences, etc, and usually,
- User can store in **MySQL database.**
- Information is also cached in **Redis**.

5. Group Service

- Maintains all group related information like which user belongs to which group, user ids, group ids, a time when the group was created, a time when every user was added, status, group icon, etc.
- Group Service can also store all the data in **MySQL** with Master and Slaves.
- Information is also cached in **Redis**.
- Web socket handlers won't keep track of groups, it just tracks active users.
- So when User-1 wants to send a message to Group-1, WSH1 gets in touch with Message service, that User-1 wants to send a message to Group-1, and it gets stored in Cassandra as M3.
- Message Service will now communicate with **Kafka**.
- M3 gets saved in **Kafka** with an instruction that it has to be sent to Group-1.
- Now Kafka will interact with something known as **Group Message Handler**.
- The group message handler listens to Kafka and sends out all group messages.

- Group message handler talks to Group service to find out all the users in G1 and follows the same process as a web socket handler and delivers the message to all users individually.

6. Asset Service

- When content like this is sent out it will be compressed and encrypted at the device end and the encrypted content will be sent to the receiver.
- Even while receiving the content will be received in an encrypted format and decrypted on the device end.
- User-1 is sending an image to User-2
- User-1 will upload the image to a server and get the image ID-1.
- Then it will send the image ID-1 to User-2 and User-2 can search and download the image from the server.
- Web Socket Handler are light weight and so no logic is in Web Socket Handler.
- The image will be compressed on the device side and sent to an **Asset Service** through the **Load Balancer** and asset service will store the content on **File Storage (S3)**.
- **Optimization / How to handle Message Image Forwards** (multiple copies of the same image)
- Before uploading an image to asset service, User-1 will send a hash of the image and if the hash is already there in the asset service, the content will not be uploaded again, but the ID against the same image/content will be sent to User-2.
- We can do multiple hashing to avoid collision.

7. Analytics Service

- Events are sent either to an **Analytics service** that sends them to Kafka or in case of messages, directly to Kafka.
- Kafka could further be connected to **Spark streaming service** that either analyzes the content as it flows in or dumps it into a **Hadoop** cluster which can run a wide variety of queries.

8. Last Seen Service

- **Last Seen Service** which keeps track of the last seen time of the user.
- We can use in a **Redis(Cache)** or **Cassandra** in the form of user id and last seen time.
- Events could be two types, the ones fired by the app and the events fired by the users.
- App events — things like when a web socket connection is created, etc. These events will not be tracked by the last seen service.
- User events — things like when the user opens or closes or does something else in the app. These events will be used to track the last seen time.

API Design

We can use REST or SOAP to serve our APIs. Basically, there will be below important API's of photo and video sharing service system.

1. postMessage

```
postMessage(api_dev_key, message_text, user_id, media_id)
```

Database

- **Web Socket Manager** stores the data in **Redis(Cache)** to store following:
- Which user is connected to which web socket handler
- What all users are connected to particular web socket handler
- **Message Service**, uses **Cassandra** as it is easy to query.
- **User Service** uses **Redis(Cache)** and **MySQL** to store user related details
- **Group Service** uses **Redis(Cache)** and **MySQL** to store group related details.
- **Last Seen User** user details like userId and timestamp can be store in **Cassandra**

- We use **Kafka** for **Group Messaging** and **Analytics** or any other data stream.
- **HDFS** for old storage.
- **Asset Service** uses
- **CDN** for Caching Images
- **File Storage (S3)** for storing File Objects

Cache and Load Balancing

We can introduce a cache**(Redis)** for metadata servers to cache hot database rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.
(Whatsapp we can use **Redis**)

How can we build a more intelligent cache? If we go with the eighty-twenty rule, i.e., 20% of daily read volume for photos is generating 80% of the traffic, which means that certain photos are so popular that most people read them. This dictates that we can try caching 20% of the daily read volume of photos and metadata.

References:

- Code Karle

Interview Daemon   System Design Interview   System Design   WhatsApp   Facebook Messenger