Overview

Design the architecture of system like WhatsApp/Facebook messenger.

Functional Requirements

- 1:1 chat
- Group chat.
- Supports multiple logins on different devices.
- Support online/offline indicator.

Non-Functional Requirements

- Realtime/Very low latency.
- Reliable.
- Highly available.

Capacity Estimates

- Daily active users = 500million.
- Ave message sent by user in a day = 40.
- Avg group size = 10.
- How many groups?
- Considering avg message takes 100 bytes, storage required to store them will be 100bytes*40*500million =4KB*500million = 20KB*1000000 = 2 TB per day.

Communication Protocol

Polling vs push based.

- Short Polling -> Server establishes a connection parodically and ask if there are any messages available. Will depend on polling frequency (to make Realtime). Need to make unnecessary request to server which is going to consume resources.
- Long Polling -> Server establishes a connection with keep alive timeout and keep on asking if there are messages available. Although this will be Realtime, we will end up also making unnecessary request to server which is going to consume resources.
- WebSocket -> Initiated by client. Although the connection is established but we don't need to send the message every time (as in case of long polling) because bidirectional.

Choice of database

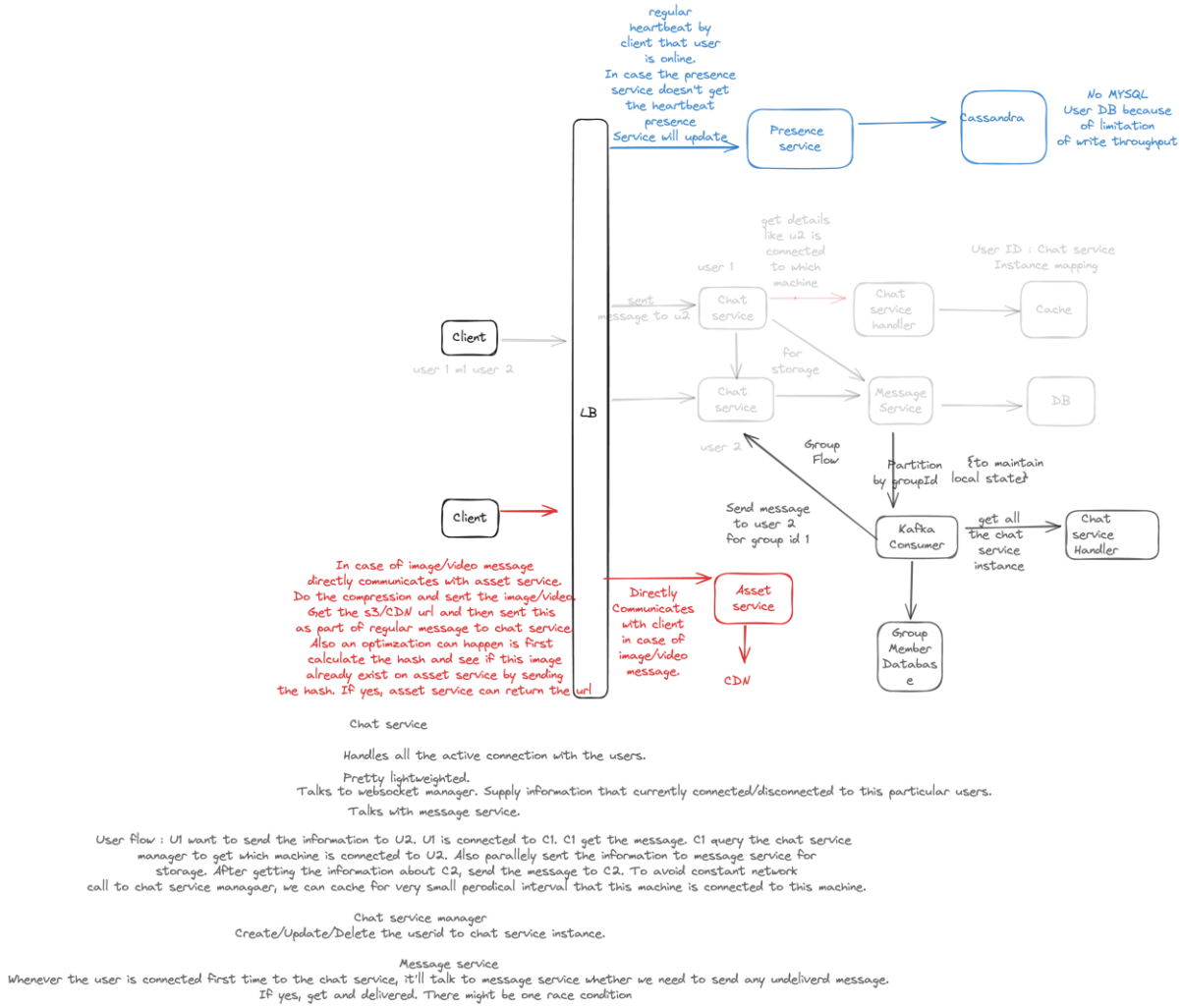
There are 3 types of information that we need to store.

- User profile information. We need acid guarantees. Also not write heavy so MYSQL can be used.
- Group Members Database (Group Id to user ID mapping (A given user belongs to which group). Will not be write heavy (user leaving/joining the group is way smaller). Can use MySQL.
  Considering the daily active user against the group, we might need to shard the db.
  Partitioning key can be groupId as we want all the user of the same group on the same shard.
  groupid , userId
- Message Table
  Need 2 separate table. One for 1:1 and another one for group.
  This will be write heavy. To get historical messages/get message on another device are not very frequent. Can use Cassandra as it offers good read/write performance.
  1:1 Chat
  messageId, message_from,message_to,content,created_at
  Can use partition key as message_from as we want all the messages of a given user on the same shard. Index can be on message_to. Sorting key can be
  Group chat
  chatId, messageId,message_from,content,created_at
  Can use partition key as chatId as we want all the messages of a given chat group on the same shard. Index can be on chatId. Sorting key can be created at.

  How to make sure the ordering of message ?
  created_at ? Server generated timestamp. Distributed timestamps are unreliable. Depend on if we want strict ordering guarantees' not, good to go.
  message_id : We can use unique id generator algorithm which make sure the ids are increasing over time (can be local as we must maintain the order within a single group and not with the other group).

Architecture

Link



Components

Chat service

- Pretty lightweighted service responsible for maintaining the WebSocket connection with client.
- Update the information to chat service manager that connecting to this particular user.
- Communicates with message service to store/get historical data.

Chat Manager Service

- Updates the information in cache that this chat service is managing this user.

Message Service

- Stores the message in message DB. Also sent this information (CDC) to Kafka in case of group messaging.

How does 1:1 Flow will look like? (CSM = Chat service manager, CS = Chat service)

- User 1 establishes a connection with the chat service 1. C1 updates this information in cache via chat manager service. U2 connects to C2 and stores this information.
- U1 first query to get if there are any undelivered messages. C1 ask message DB. Message DB to get this information (get the message where status is either not seen or undelivered).
- U1 initiates the sent request to U1. C1 get the sent request. C1 queries the CSM to check what instance of chat service is handling U2. Get the details and sent the information to C2.
- C2 sent the message to U2.
- Parallelly C1 sent the information message service for persistence.
- Also, C1/C2 will be responsible for receiving if user has got the message/seen the message and propagate to each other and to message service for status update.
- Also to avoid query CSM, cache locally for very short interval of time that this CS instance is handling this user.

How does group Flow will look like?

- U1 sent a message to group chat id 1.
- C1 gets a group sent request. Forward this information to message service. Message service persist this information to DB while also sending the information to Kafka for fanning out to another user.
- Kafka consumer needs to know 2 information -> Which user belongs to this group. Information available in MYSQL Group member table. Either this Kafka consumer interaction can be handled by group service or directly. To avoid query the MySQL, partition key can be groupid and consumer can maintain some sort or local state (g1 -> u1,u2, u3 etc). Other

better way can be group service can also CDC the changes of MYSQL table to this same consumer which then it can utilize for maintaining states.

- After getting list of users, query the CSM to get list of CS instances and forward them the message.

Inspired from LINK1 AND LINK2