System Design: Design Messenger

Again, We follow the same structure and broadly we divide it into 4 sections,

1. Defining the MVP
2. Estimation of Scale: Primarily to determine 2 things,
   2.1 Whether we need Sharding.
   2.2 Whether it's a read heavy system, or write heavy system or both.
3. Design goals.
4. API+Design.

MVP:

- Send a message to the recipient.
- Realtime chat.
- Message history.
- Most recent conversations.

Estimation of Scale:

Let's say starting with 20 billion messages/day.
Every single message is 200 bytes or less.
That means 4TB/day.
If we want to let's say save our messages for 1 year. So for 1 year, it will be greater than PB. In reality, if we are building for the next 5 years, we need multiple PB of storage.

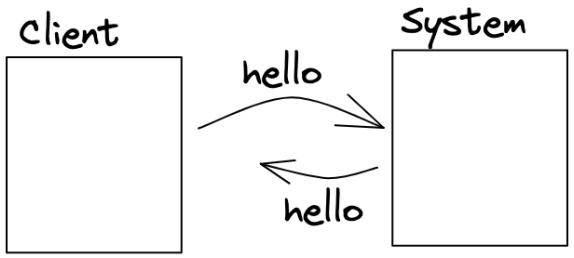
1. We definitely need Sharding!!
2. It's both a read + write heavy system.

Design Goals:

System should be **Highly Consistent** because inconsistency in communications can lead to issues.
Latency should be low.

APIs

In a case of an app like messenger where consistency is super important, one thing to consider should be that your write APIs are idempotent. You need to consider this because your primary caller is a mobile app which could have an intermittent network. As such, if the API is called multiple times, due to application level retries, or if data reaches multiple times due to network level retries, you should not create duplicate messages for it.



Let's say we have a client that is trying to talk to your backend.
Imagine I send a message "Hello" to the backend. The backend gets the message, successfully stores the message, but the connection breaks before it could return me a success message. Now, it's possible I do a retry to ensure this "Hello" message actually gets sent. If this adds 2 messages "Hello", followed by another "Hello", then the system we have is not idempotent. If the system is able to deduplicate the message, and understand it's the same message being retried and hence can be ignored, then the system is idempotent.

How to make the system Idempotent:

We can use a messageId - something that is different across different messages, but same for the same message retried.
Imagine everytime we send a message "hello", the moment "hello" is generated, a new messageId is generated.
Now, when we send this message to the backend, instead of saying user A is sending user B a message "Hello", we say user A is sending userB a message "Hello" with messageId as xyz. Then even if the system gets the same message again then it can identify that it already has a message with messageID xyz and hence, this new incoming message can be ignored.
This however, won't work if messageID is not unique across 2 different messages (If I type "Hello" twice and send twice manually, they should be considered 2 different messages and should not be deduplicated).

How to generate Unique messageId:

We can possibly use the combination of:

- Timestamp(date and time)
- senderID
- deviceID
- recipientID (To be able to differentiate if I broadcast a message).

APIs

Before thinking of the APIs, think of the usecases we would need to support. What kind of views do we have?
First view is the view when I open the app. Which is a list of conversations (not messages) with people I recently interacted with (has name of friend/group, along with a snippet of messages. Let's call that getConversations.

If I click into a conversation, then I get the list of most recent messages. Let's call that getMessages.
And finally, in that conversation, I can send a message.
So, corresponding APIs:

1. SendMessage(sender, recipient, text, messageId)
2. getMessages(userId, conversationId, offset, limit)

Offset: Where to start
Limit: How many messages after that is limit. Offset and limit are usually used to paginate (if page sizes can be different across different clients).

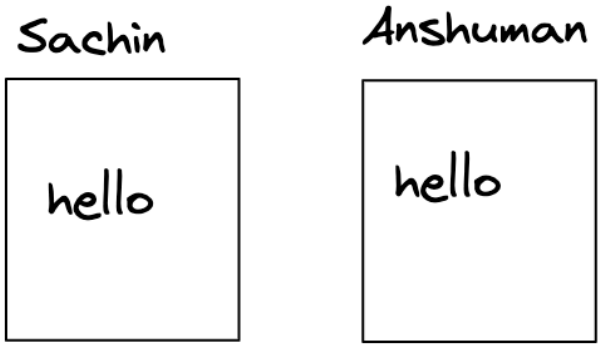
3. getConversations(userId, offset, limit)
4. CreateUser(---).

System Design:

Problem #1: Sharding:

1. userId: All conversations and messages should be on the same machine. Essentially, every user has their own mailbox.
2. ConversationId: Now all messages of a conversation go on the same machine.

userID based sharding:
So every user will be assigned to one of the machines.



Now, Let's do each of the operations:

1. getConversation: It's pretty easy.
   Imagine if Sachin says, get the most recent conversation. We go to a machine corresponding to Sachin and get the most recent conversations and return that.

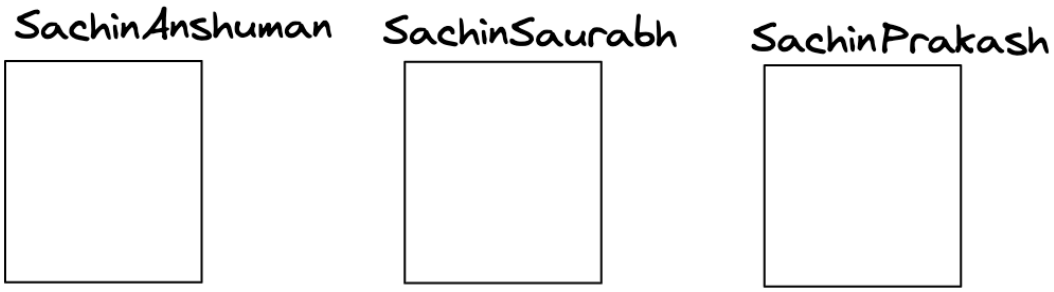
2. getMessages:
   Same, We can go to a machine corresponding to Sachin and for that we can get the messages for a particular conversation.

3. sendMessage:

Imagine if Sachin sends a message “hello” to Anshuman, Now this means we have 2 different writes, in Sachin’s machine and in Anshuman’s machine as well and they both have to succeed at the same time.
So, for sendMessage in this type of sharding, there should be 2 writes that need to happen and somehow they still need to be consistent which is a difficult task.

conversationID based sharding:
Here for every conversation we will have a separate machine.

For example,



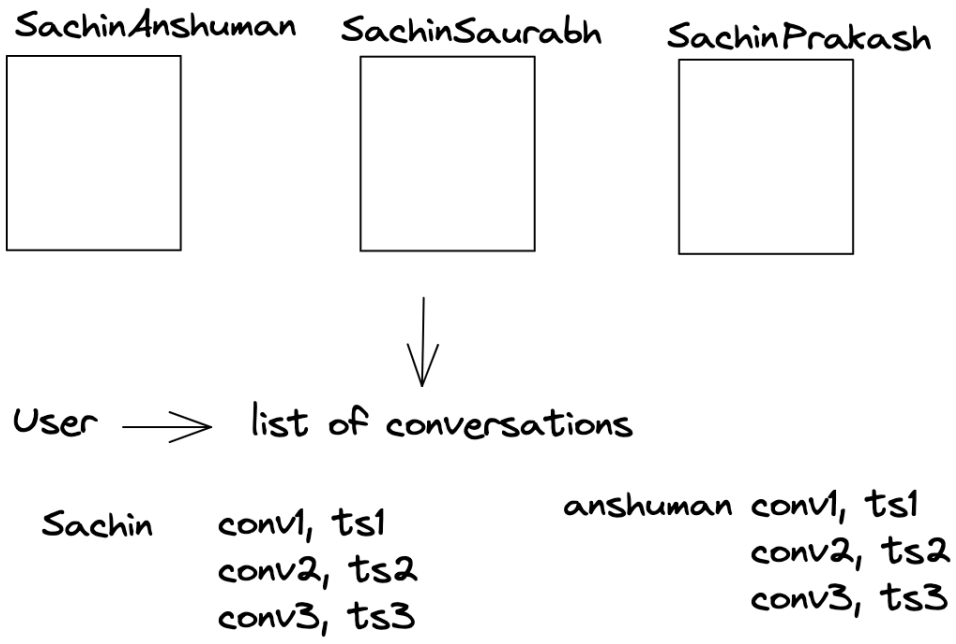
Now, Let’s do each of the operations again:

1. getMessages:
   Say, We want to get the last 100 messages of conversation b/w Sachin and Anshuman.
   So we will go to the corresponding machine which has Sachin/Anshuman messages and fetch the messages inside.

2. sendMessage:
   This is also fairly simple. If Sachin wants to send a message to Anshuman, we go to the machine corresponding to Sachin/Anshuman, and add a message there.

3. getConversations:
   For example, we want to get the latest 10 conversations that Sachin was part of.
   Now in this case, we need to go to each and every machine and fetch if there is a conversation which has the name Sachin in it. ***That is very inefficient.***

One solution might be to have a **Secondary database:**
In this database we can have user to list of conversations (sorted by recency of the last message sent - along with metadata of conversations - snippet, last Message Timestamp, etc.).



Now again if we do these operations:

1. getMessage: it will work fine.
   If we say get the last 10 messages with the conversation of Anshuman and Sachin. Since they are sharded by conversationId, it will have one machine which has all the messages.

2. getConversations: Now again we can't go to any one of the databases, we have to go to the secondary database and have to read from here.

3. sendMessage:
   If let's say Sachin sends the message in the conversation b/w Sachin and Anshuman, In this case, we have to add the message to SachinAnshuman Database and then in the secondary database we have to change the ordering of conversations in both Sachin and Anshuman's list of conversations.
   Therefore a single send message has 3 writes.

For systems like, Slack, MS Team, Telegram that can have large groups, userID based sharding will be ridiculously expensive as every single sendMessage will leads to 1000 writes in a 1000 member group. Hence, they relax on the ordering of threads in getConversations (best effort) and instead use conversationId based sharding.

For 1:1 messages-> UserId seems to be a better choice (2 writes vs 3). That being said, you can't go terribly wrong with conversationID either. For the purpose of this doc, we will use userID.

In sharding based on userId, 2 operations are working perfectly fine: getMessage, gerConversation.
But the problem is with sendMessage, when we send a message hello, it was written to 2 different machines and if one of those writes fails then probably both the machines become inconsistent.

Problem #2: sendMessage consistency

Consistency means: If user1 sends a message “Hi”, and does not get an error, then it should imply that the message has been delivered to user2. User2 should get the message.

If user1 sends a message to user2, how should we sequence the write between user1 DB and user2 DB to ensure the above consistency?

Case1: write to sender/user1 first.

1. If it fails then we return an error.
2. If it succeeds, then:
   We write to recipient / user2 shard: again it can have 2 possibilities:
   2.1: Success: Then the system is in consistent state and they return success.
   2.2: Failure: Rollback and return error.

Case2: Write to recipient/user2 shard:

1. Failure: Simply return an error.
2. Succeed: It has reached to recipient so system is in consistent state.
   Then we write to sender shard:
   1. Success: System is in consistent state, return success.
   2. Failure: Add it to the queue and keep retrying so that eventually it gets added to the sender’s shard.

Out of these 2 cases, case2 is much better because the sender sends the message and the recipient gets the message. The only problem is when the sender refreshes he cannot see the message. Not the good behavior but better of the two behaviors.
Because case1 is very dangerous, Sender sends the message and when he refreshes the message is still there but the recipient never got it.

Problem #3: Choosing the right DB / cache:

Choosing the right DB here is very tricky as this system is both read heavy and write heavy. As we have discussed in the past, both compete with each other, and it’s best to reduce this system to either read heavy or write heavy on the storage side to be able to choose one DB.
Also, this requires a massive amount of sharding. So, we are probably looking for a NoSQL storage system that can support high consistency.

Reduction to read vs write heavy
If we were building a loosely consistent system where we cared about trends, we could have looked to sample writes / batch writes. But here, we need immediate consistency. So, absorbing writes isn't feasible. You'd need all writes to be immediately persisted to remain highly consistent.

That means, the only read option is to somehow absorb the number of reads through heavy caching. But remember that you'd need to cache a lot for it to absorb almost all of reads (so much that it won't fit on a single machine) and this cache has to be very consistent with the DB. Not just that, you'd need to somehow handle concurrent writes for the same user to not create any race condition.

Consistency of cache: We can use write-through cache.
Lots of data to be cached: We would need to shard cache too.
Handle write concurrency in cache: How about we use appservers / business logic servers as cache. We can take a write lock on user then.

A simple way to do this might be to use appservers as cache, and have them be tied to a set of users (consistent hashing of users -> appservers). This would also let you take a write lock per userID when writes happen, so that writes happen sequentially and do not create race condition in the cache.
Since you cannot cache all the users, you can look at doing some form of LRU for users.

Pros:
- Can scale horizontally. Throw more appservers and you can cache more information.
- Race conditions and consistency requirements handled gracefully.

Cons:
- If the server goes down, things are unavailable till reassignment of the user happens to another app server. Since, this might take a few seconds, this causes unavailability of a few seconds every time the appserver is down.
- When the app server restarts, there is a cold cache start problem. The initial few requests are slower as the information for the user needs to be read from the DB.

Right DB for Write heavy, consistent system:
If we successfully absorb most of the reads, so that they rarely go to the DB, then we are looking for a DB that can support write-heavy applications. HBase is good with that. It allows for column family storage structure which is suited to messages/mailbox, and is optimized for high volumes of writes.