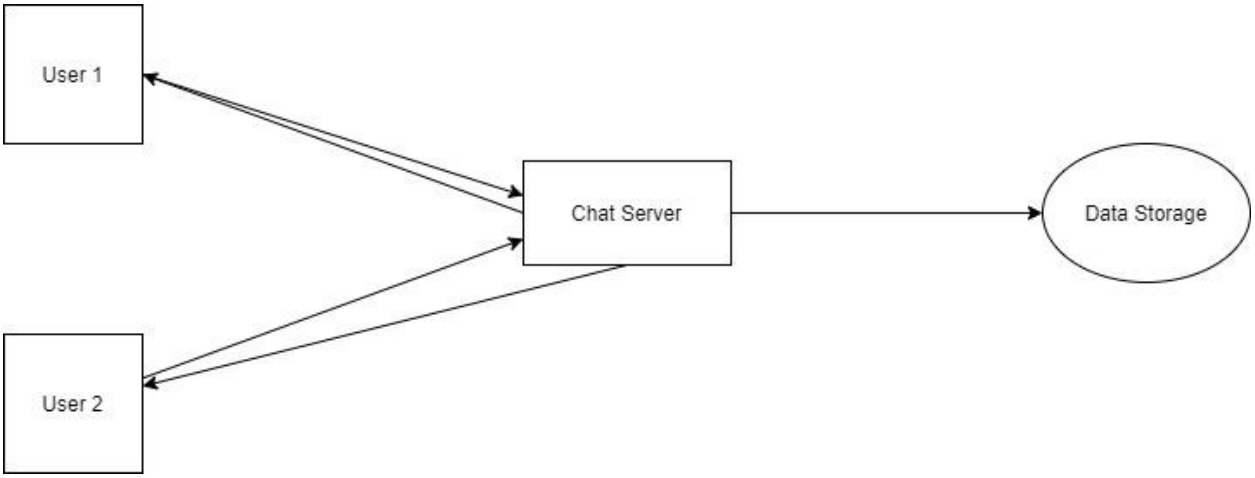
HIGH LEVEL DESIGN

At a high-level, we will need a chat server that will be the central piece, orchestrating all the communications between users. When a user wants to send a message to another user, they will connect to the chat server and send the message to the server; the server then passes that message to the other user and also stores it in the database.



System APIs

sendMessage(senderId, recepientId, messageContent, clientMessageId)

fetchConversation(userId, pageNumber, pageSize, lastUpdatedTimestamp)

fetchMessages(userId, pageNumber, pageSize, lastUpdatedTimestamp)

DB DESIGN

- Require storage for billions of records
- Medium size message (2-10 Mb including phots/videos)
- Service is read heavy and write heavy

User

Columns	Type
userId	integer

Columns	Type
Username	integer
Email	integer
Password	string
create_at	timestamp
Last_login	timestamp

Message

Columns	Type
messageId	integer
from_user_id	integer
to_user_id	integer
content	string
create_at	timestamp

Message Index

Columns	Type
messageId	integer
content	string
create_at	Timestamp

Recent Contacts

Columns	Type
messageId	string
from_user_id	integer
to_user_id	integer

A straightforward approach for storing the above schema would be to use an RDBMS like MySQL since we require joins. But relational databases come with their challenges, especially when we need to scale them.

We propose to utilize Hbase for our system. HBase is a column-oriented key-value NoSQL database that can store multiple values against one key into multiple columns. HBase is modeled after Google's BigTable (https://en.wikipedia.org/wiki/Bigtable) and runs on top of Hadoop Distributed File System (HDFS (https://en.wikipedia.org/wiki/Apache_Hadoop)). HBase groups data together to store new data in a memory buffer and, once the buffer is full, it dumps the data to the disk. This way of storage not only helps to store a lot of small data quickly but also fetching rows by the key or scanning ranges of rows. HBase is also an efficient database to store variable sized data, which is also required by our service

Basic System Design and Algorithm

a. Messages Handling

How would we efficiently send/receive messages? To send messages, a user needs to connect to the server and post messages for the other users. To get a message from the server, the user has two options:

1. Pull model: Users can periodically ask the server if there are any new messages for them.

2. Push model: Users can keep a connection open with the server and can depend upon the server to notify them whenever there are new messages.

If we go with our first approach, then the server needs to keep track of messages that are still waiting to be delivered, and as soon as the receiving user connects to the server to ask for any new message, the server can return all the pending messages. To minimize latency for the user, they must check the server quite frequently, and most of the time they will be getting an empty response if there are no pending message. This will waste a lot of resources and does not look like an efficient solution.

If we go with our second approach, where all the active users keep a connection open with the server, then as soon as the server receives a message it can immediately pass the message to the intended user. This way, the server does not need to keep track of the pending messages, and we will have minimum latency, as the messages are delivered instantly on the opened connection.

Maintain an open connection with the server

We can use HTTP Long Polling or WebSockets. In long polling, clients can request information from the server with the expectation that the server may not respond immediately. If the server has no new data for the client when the poll is received, instead of sending an empty response, the server holds the request open and waits for response information to become available. Once it does have new information, the server immediately sends the response to the client, completing the open request. Upon receipt of the server response, the client can immediately issue another server request for future updates. This gives a lot of improvements in latencies, throughputs, and performance. The long polling request can timeout or can receive a disconnect from the server, in that case, the client must open a new request.

Track of all the opened connection to redirect messages to the users efficiently

The server can maintain a hash table, where “key” would be the UserID and “value” would be the connection object. So, whenever the server receives a message for a user, it looks up that user in the hash table to find the connection object and sends the message on the open request.

Action for user who has gone offline

If the receiver has disconnected, the server can notify the sender about the delivery failure. If it is a temporary disconnect, e.g., the receiver’s long poll request just timed out, then we should expect a reconnect from the user. In that case, we can ask the sender to retry sending the message. This retry could be embedded in the client’s logic so that users do not have to retype the message. The server can also store the message for a while and retry sending it once the receiver reconnects.

Total Chat Servers Required

Let us plan for 500 million connections at any time. Assuming a modern server can handle 50K concurrent connections at any time, we would need 10K such servers.

How do we know which server holds the connection to which user? We can introduce a software load balancer in front of our chat servers; that can map each UserID to a server to redirect the request.

How should the server process a ‘deliver message’ request? The server needs to do the following things upon receiving a new message: 1) Store the message in the database 2) Send the message to the receiver and 3) Send an acknowledgment to the sender.

The chat server will first find the server that holds the connection for the receiver and pass the message to that server to send it to the receiver. The chat server can then send the acknowledgment to the sender; we do not need to wait for storing the message in the database (this can happen in the background). Storing the message is discussed in the next section.

How does the messenger maintain the sequencing of the messages? We can store a timestamp with each message, which is the time the message is received by the server. This will still not ensure correct ordering of messages for clients. The scenario where the server timestamp cannot determine the exact order of messages would look like this:

1. User-1 sends a message M1 to the server for User-2.

2. The server receives M1 at T1.

3. Meanwhile, User-2 sends a message M2 to the server for User-1

4. The server receives the message M2 at T2, such that T2 > T1.

5. 5. The server sends message M1 to User-2 and M2 to User-1.

So, User-1 will see M1 first and then M2, whereas User-2 will see M2 first and then M1.

To resolve this, we need to keep a sequence number with every message for each client. This sequence number will determine the exact ordering of messages for EACH user. With this solution, both clients will see a different view of the message sequence, but this view will be consistent for them on all devices.

b. Storing and retrieving the messages from the database

Whenever the chat server receives a new message, it needs to store it in the database. To do so, we have two options:

1. Start a separate thread, which will work with the database to store the message.

2. Send an asynchronous request to the database to store the message.

We must keep certain things in mind while designing our database:

1. How to efficiently work with the database connection pool.

2. How to retry failed requests.

3. Where to log those requests that failed even after some retries.

4. How to retry these logged requests (that failed after the retry) when all the issues have resolved.

How should clients efficiently fetch data from the server? Clients should paginate while fetching data from the server. Page size could be different for different clients, e.g., cell phones have smaller screens, so we need a fewer number of message/conversations in the viewport

c. Managing user's status

We need to keep track of user's online/offline status and notify all the relevant users whenever a status change happens Since we are

We can easily figure out the user's status from this. With 500M active users at any time, if we must broadcast each status change to all the relevant active users, it will consume a lot of resources. We can do the following optimization around this:

1. Whenever a client starts the app, it can pull the status of all users in their friends' list.

2. Whenever a user sends a message to another user that has gone offline, we can send a failure to the sender and update the status on the client.

3. Whenever a user comes online, the server can always broadcast that status with a delay of a few seconds to see if the user does not go offline immediately.

4. Clients can pull the status from the server about those users that are being shown on the user's viewport. This should not be a frequent operation, as the server is broadcasting the online status of users and we can live with the stale offline status of users for a while.

5. Whenever the client starts a new chat with another user, we can pull the status at that time.

Data Partitioning and Replication

Data partitioning

Partitioning based on UserID: Let us assume we partition based on the hash of the UserID so that we can keep all messages of a user on the same database. If one DB shard is 4TB, we will have "3.6PB/4TB ~= 900" shards for five years. For simplicity, let us assume we keep 1K shards. So, we will find the shard number by "hash(UserID) % 1000" and then store/retrieve the data from there. This partitioning scheme will also be very quick to fetch chat history for any user.

In the beginning, we can start with fewer database servers with multiple shards residing on one physical server. Since we can have multiple database instances on a server, we can easily store multiple partitions on a single server. Our hash function needs to understand this logical partitioning scheme so that it can map multiple logical partitions on one physical server.

Since we will store an unlimited history of messages, we can start with a big number of logical partitions, which will be mapped to fewer physical servers, and as our storage demand increases, we can add more physical servers to distribute our logical partitions.

Partitioning based on MessageID: If we store different messages of a user on separate database shards, fetching a range of messages of a chat would be very slow, so we should not adopt this scheme.

Replication

We cannot have only one copy of the user's data, because if the server holding the data crashes or is down permanently, we do not have any mechanism to recover that data. For this, either we must store multiple copies of the data on different servers or use techniques like Reed-Solomon encoding to distribute and replicate it.

Caching

One of our design goals was to ensure tight consistency. Most distributed caching system are good with availability and they become eventually consistent. But they are not tightly consistent.

One way to resolve this is to make sure that caching for a user completely resides on one server. The same server can also have other users as well, but users are assigned to exactly one server for caching. To further ensure consistency, all the writes for that user should be directed through this server and this server updates its cache when the write is successful on DB before confirming success.

Multiple concurrent writes : The caching server will also multiple indices corresponding to the mailbox (the ones for recent conversations / recent messages). A single write would affect multiple columns. While a NoSQL DB might guarantee atomicity on a row level, in the caching layer, we will have to guarantee it artificially. One simple way of solving it would be to have a user level lock in the caching server for the user which allows only one write operation to go through at a time.

Load Balancing

We will require a load balancer in front of our chat servers; that can map each UserID to a server that holds the connection for the user and then direct the request to that server. Similarly, we would need a load balancer for our cache servers