Designing a Messaging Platform like WhatsApp

In this post I am going to discuss the design of a messaging platform similar to whatsapp in detail. This is a common question in many system design interviews and I am hopeful that this article will be able to explain it in good detail.

The messaging platform we are going to design will support the following features

1. 1–1 messaging
   a. when receiver is online
   b. when receiver is offline
2. Presence (online/offline)
3. Sent, Delivered and Read acknowledgments
4. Media sharing
5. Group Messaging
6. Storing messages in the cloud

1–1 Messaging

A Simple chat server:

Let us begin with a very simple design where two users A and B need to communicate with each other. We will build on top of this and add more components along the way to scale it to a few million users.

Whenever we talk about a chat application it is always desirable to have a persistent two way communication channel between the server and client because there is a frequent exchange of messages between sender and receiver. Web-socket is the preferred protocol for such persistent two way communication.

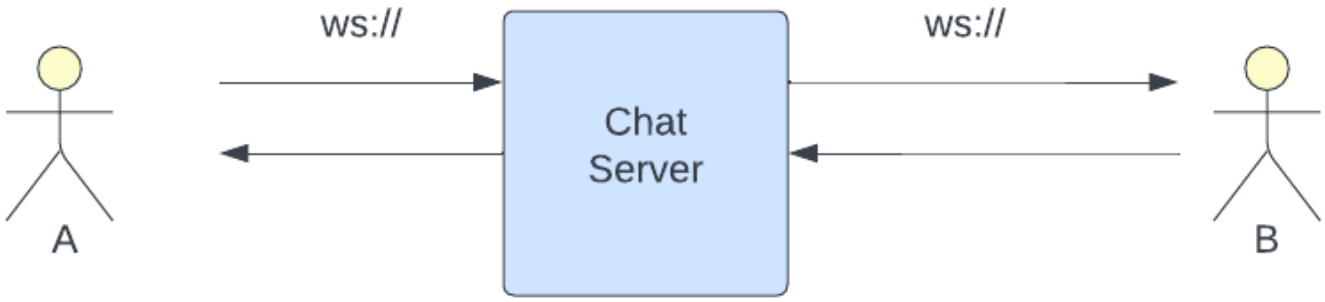


figure-1

As simple as that A and B are both connected to a chat server using the web socket protocol and they send messages to each other.

Authentication:

Even before both A and B connect to the chat server they need to be authenticated by the system. Authentication is a separate design problem in itself. We will talk about it in a different article. Let us assume there are two services called User Service and Authentication Service which takes care of user profile management and authentication respectively.

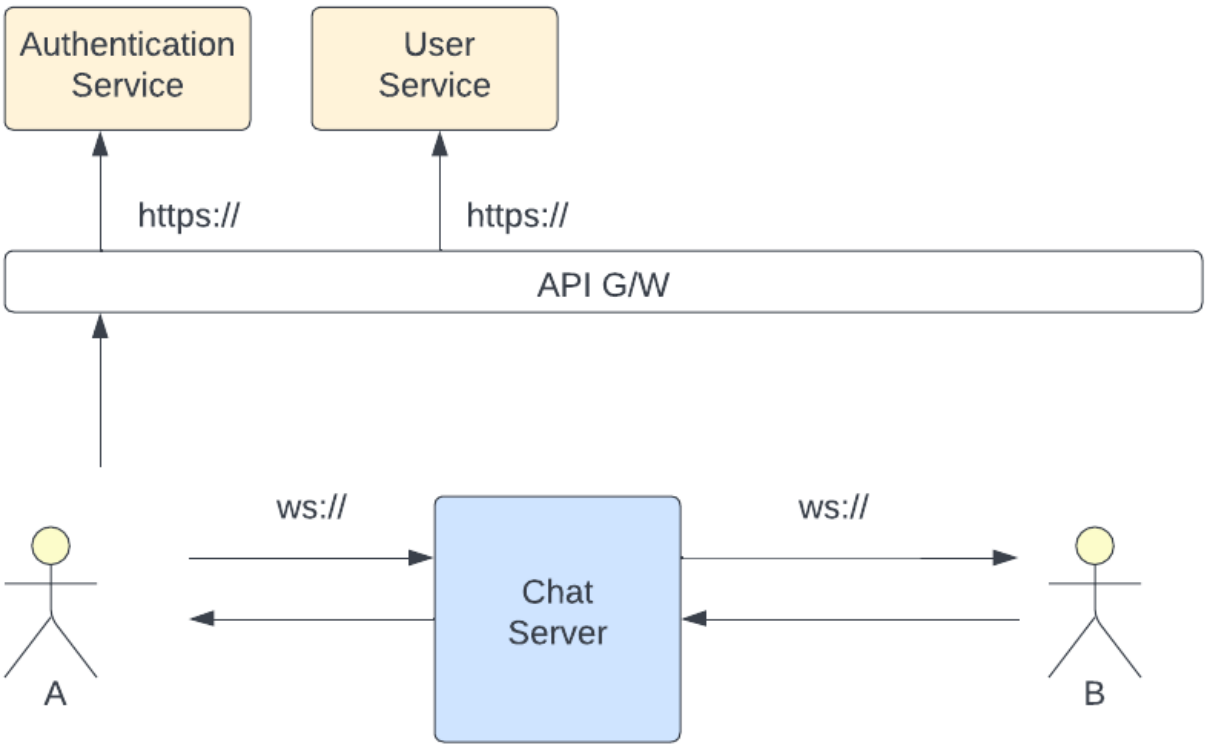


figure-2

Scaling up the design:

We are talking about whatsapp scale here so a single chat server is not enough. We are going to scale the application by increasing the number of chat servers and putting them behind a load balancer.

Having multiple chat servers brings in a new challenge altogether. A and B could be connected to two different chat servers. In order to deliver the messages to the receiver we need to store the information about which user is connected to which server.

We introduce a new micro service called Session Service to store this information. We can use NoSQL key value database like dynamo or Redis to store the user to chat server mapping. (user_id, connected_to, connection_id) is the information stored in the table.

Below is how it works

1. A sends a message to B which lands on CS1 (Chat Server 1).

2. CS1 contacts Session Service to know which chat server B is connected to (CS3 in this case).
3. CS1 send the message to CS3 using a simple rest call.
4. CS3 will deliver it to B using the web socket connection

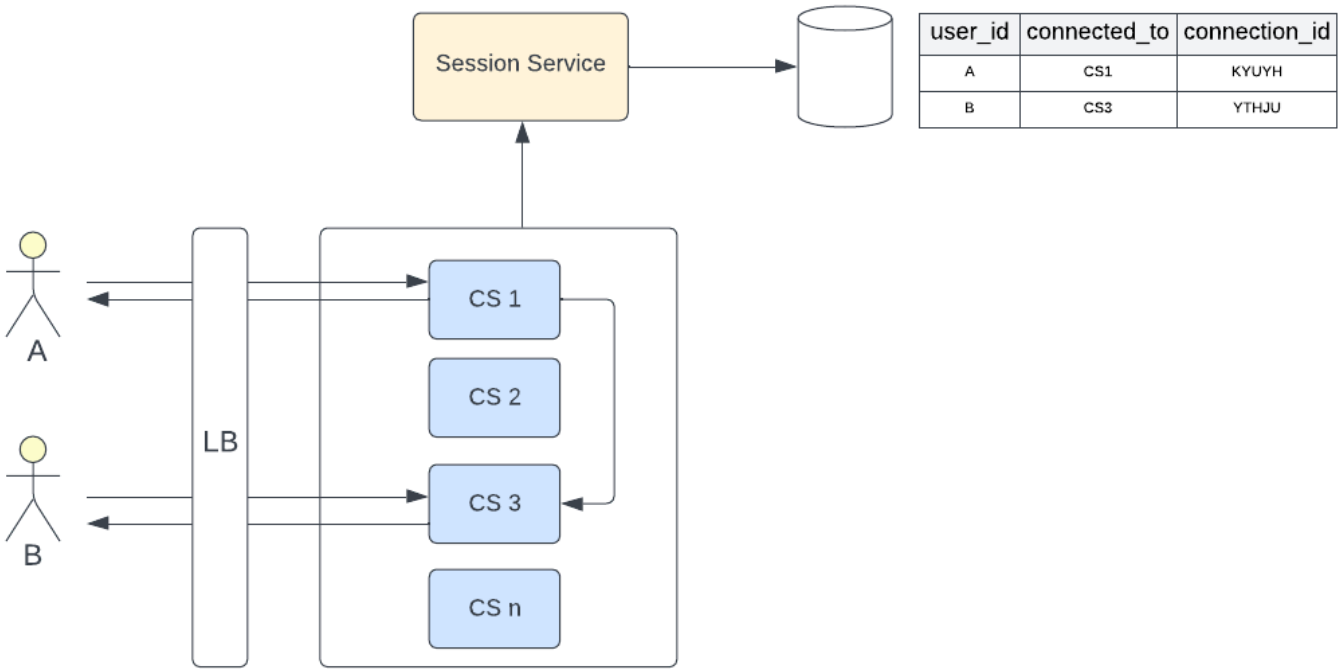


figure-3

To reduce the clutter between the chat servers we will introduce a topic to which the chat servers publishes the messages they receive. This way we can do additional operations on the messages (in later sections) without overloading the chat servers.

1. A sends a message to B which lands on CS1 (Chat Server 1).
2. CS1 publishes the message to a Kafka topic.
3. Messaging Service consumes the messages contacts Session Service to know which chat server B is connected to (CS3 in this case).
4. Messaging Service sends the message to CS3.
5. CS3 will deliver it to B using the web socket connection.

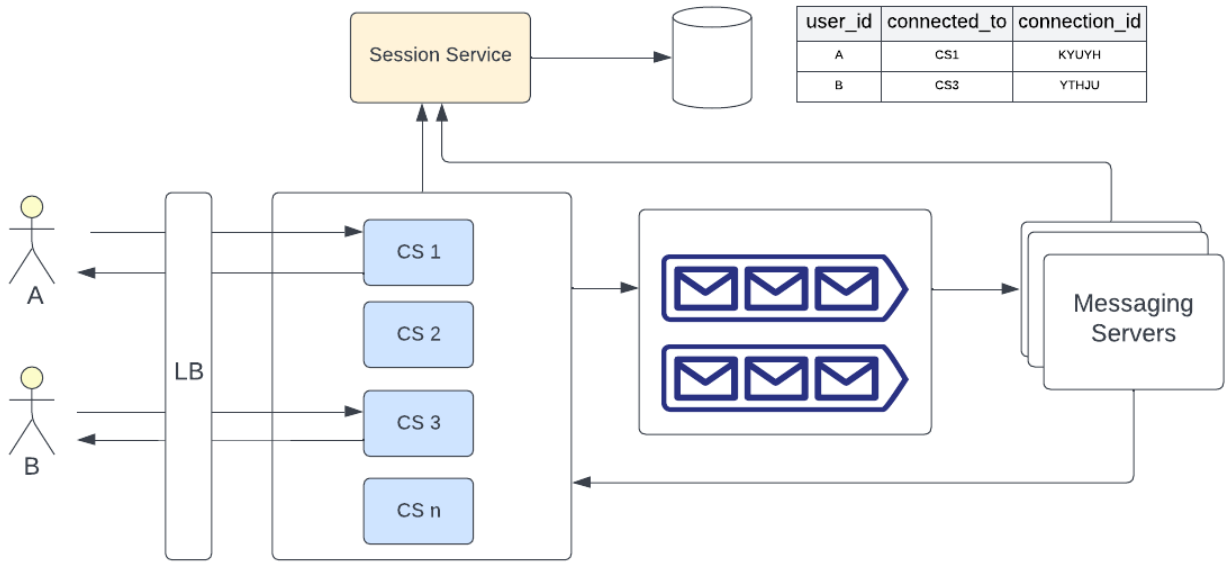


figure-4

We can use Kafka as the messaging system given the scale. A single topic can handle upto 4000 partitions. In Kafka we can have consumers equal to the number of partitions. If time taken to process one message is ~200 ms then one consumer can process 5 messages in one second and with 4000 partitions we can get a throughput of 20K TPS in a single region.

We can use userId as the key so that it will be hashed to get the partition. We can stick to DefaultPartitioner of Kafka — even distribution across all partitions. We can add more topics and have producers and consumers connect to them controlled through configuration to achieve higher throughput.

Presence (Online/Offline):

We can have separate microservice for monitoring the status of a user — whether the user is online/offline. But it might be an overkill. We will introduce a new field called `last_seen_timestamp` in the sessions table . It makes sense as the sessions service is anyways dealing with the client connections.

Clients send a heart beat message say every 5s to indicate that they are connected. Either the chat servers or the messaging servers can update the last_seen_timestamp.

user_id	connected_to	connection_id	last_seen_timestamp
A	CS1	KYUYH	1672883793
B	CS3	YTHJU	1672883794

figure-5

User Status can be derived from the last_seen_timestamp by calculating the difference between current instance and the last_seen_timestamp and comparing it with heart beat time interval.

Sending messages to offline users:

The design we put together till now works well when connected to chat server and is online. We need to enhance it to handle the cases where receiver is offline. There are two ways to handle message when receiver is offline

1. Send the message to a Push Notification Service (PNS)like One Signal or Firebase.
2. Store the messages in a database and deliver it to the receiver when the receiver comes online.

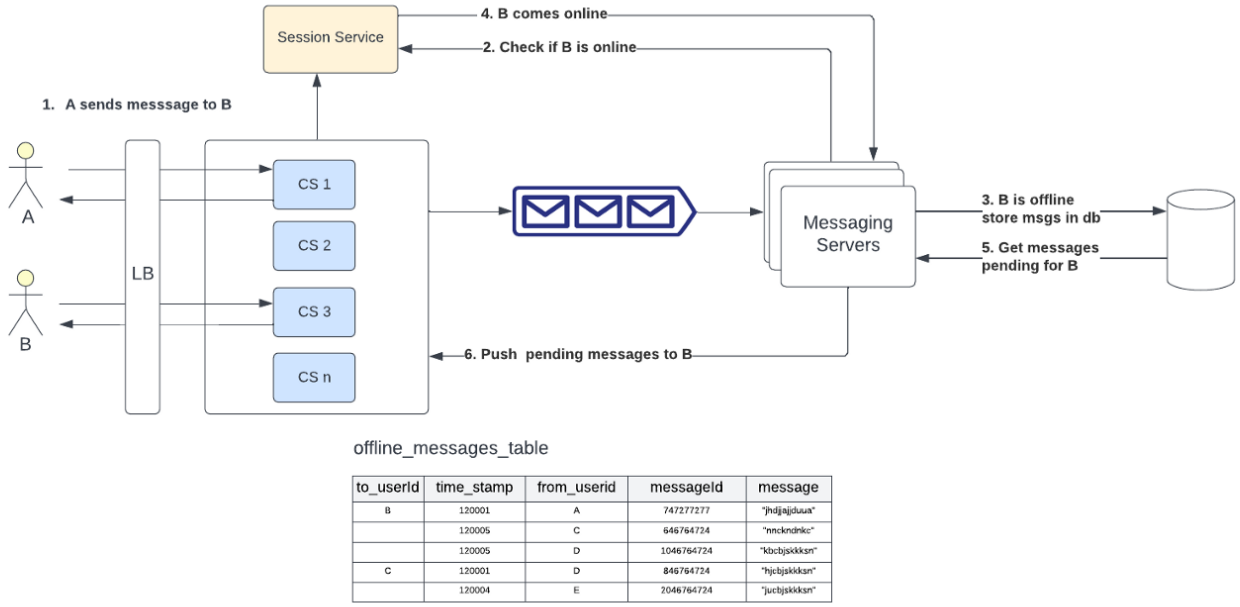
With option #1 the PNS will take care of delivering the message when the user connects to internet.

In option #2 we can store the messages in a database so we can deliver them when the user comes back online. Below is the sequence of events that happen when the receiver is offline

1. A sends a message to CS1 to be delivered to B.
2. CS1 pushes the message to the messages topic.
3. One of the messaging servers consumes the message and checks with session service to see that B is offline and stores the messages in the offline messages table in the db.
4. When B comes online Sessions Service informs the messaging servers indicating the same (this can either be via a Rest API or messaging).

5. Messaging server will then look up the messages in the for B and pushes the messages to Chat Server 3.

6. Chat Server 3 pushes the messages to B.



offline_messages_table

to_userId	time_stamp	from_userid	messageId	message
B	120001	A	747277277	"jhdjajduua"
	120005	C	646764724	"nncknclnkc"
	120005	D	1046764724	"kbcbjskkksn"
C	120001	D	846764724	"hjcbjskkksn"
	120004	E	2046764724	"jucbjskkksn"

figure-6

We can use a key value NoSQL db like dynamoDb as all we need to store is messages against the user that they are intended to and query them using the to_userId. *to_userId* will be the partition key for this table.

A relational db will also do just fine here. A single instance of db like Aurora can easily store 64TB of data without impacting performance. Since we need to store only offline messages and they can be deleted once delivered we wont exceed this limit. In case of relational db we can have message_id as primary key and an index on to_user_id.

Note: When a user comes online Session Server can communicate this to messaging server either via simple Rest call or by publishing a message into another queue which the messaging servers can listen to.

Sent, Delivered and Read Acknowledgments:

Sent : Immediately after consuming the message from the message topic the messaging server can send a message of type SENT_ACK to the sender via the chat server the sender is

connected to.

Delivered **:** Messaging server will send an acknowledge message of type DELIVERED_ACK to the sender once the message is delivered to the receiver (if receiver is offline this happens after the receiver comes online and pending messages are pushed to receiver).

Read: Once the receiver gets the message it will send a message of type READ to Chat Server. After the Message server sees this message it will send a READ_ACK to the sender o the sender via the chat server the sender is connected to.

Media sharing

The Client application can upload media directly to a cloud server like Amazon S3 after encrypting it (building cloud storage in itself again is a design problem in itself). After uploading the document/image the link that is received is included as part of the message and sent to the chat servers. When the receiver receives the message if the message is of type Media the receiving client will download media from the cloud storage and decrypt the media. Security is a very important for both text messages and media. All of them should be encrypted. We can use a security protocol like Signal Protocol . This article captures a nice summary of how Signal Protocol is used in Whatsapp.

Group Messaging

We can easily extend our above design to deal with groups. Group information can be stored in a simple table as below.

group_id	admin_user_ids	group_member_ids

figure-7

Primary_key will be group_id. group_member_ids is a list of user_ids in the group and admin_user_ids is a list of user_ids of the admins in the group.

Below is the flow for group messages

1. A sends a message to group G.

2. The message service sees that the message is a group message (using the message_type field).
3. Looks up the group_member_ids from the db.
4. It follows the same process outlined in figure-4 and figure-6 outlined to deliver the message to the group_member.

There might be cases where we need to look up all groups that a user is part of eg: while displaying all the groups in the client. So instead of having group_member_ids as a list we can have an entry for each group_member_id. If we go with a NoSQL db (group_id, group_member_id) will be the composite primary key and group_id will be the partition key. We can add a column called is_admin to represent which members are admins in the group.

group_id	group_member_id	is_admin

figure-8

Storing messages in the cloud

Some messaging systems store messages on the cloud while some don't. For example whatsapp doesn't store messages. They are only stored on the client device. Facebook Messenger stores messages in their cloud.

The amount of messages we need to store in our application is enormous at whatsapp scale. Even if we consider 100 million active users in a day and each user sends 10 messages on average and each message is 10 KB in size we will need have to store 10 TB per day.

We will use a NoSQL db with the below table structure because scaling NoSQL databases is easier compared to scaling a RDBMS (our application needs to implement sharding and consistent hashing). We can have a table with the below structure.

message_id	from_user_id	to_user_id	content	time_stamp	message_type

figure-9

partition key can be from_user_id and to_user and time_stamp can be sort keys in the above table.

One problem with the above data model is the partition for a given from_user_id can grow very big with in a few day causing large partition problems. One way to avoid this is instead of using to_user_id as the partition key we cans use for example a combination of user_id and month *to_user_id:mm-dd* as partition key so the partition key wont grow beyond a month. This will also make it easy to query past messages when user tries to see the history.

Below is the end to end system architecture with all the partswe discussed above.

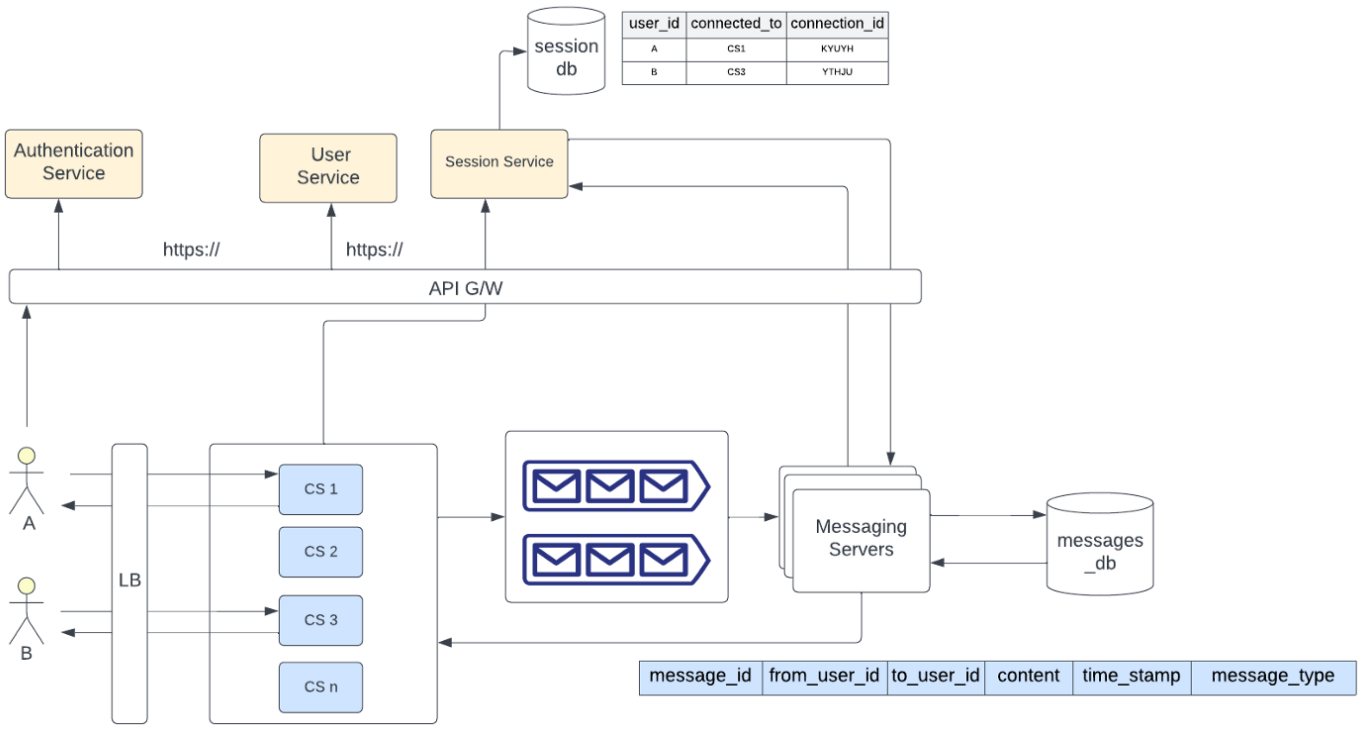


figure-10

That concludes the post. It is a long one but hopefully helps in understanding the messaging system design in good detail.

Cheers!
Divakar Ungatla

References

1. https://medium.com/@panghalamit/whatsapp-s-end-to-end-encryption-how-does-it-work-80020977caa0

2. https://www.amazon.in/dp/B08B3FWYBX/ref=dp-kindle-redirect?_encoding=UTF8&btkr=1

3. https://www.amazon.com/dp/B092CRHS99