What’s the best data store key value to use for a chat service as mentioned before?

**How many websocket connectison can each service have?**

**65,536 socket connections**

For better understanding of WS load balancing let's dive a bit deeper into TCP sockets background. By default, a single server can handle 65,536 socket connections just because it's the max number of TCP ports

**A diagram of a flowchart

Description automatically generated**

Do not use the diagram here for the genearl set up, check the figures below for the detailed flow

**What are the key components**

1. Send message service/ chat service (most likely websocket)

2. Connection management service

* Handles the WebSocket connections between devices and the server infrastructure.
* Manages the lifecycle of WebSocket connections, including establishing, maintaining, and closing connections.
* **Maintains a mapping of user/device IDs to the respective WebSocket connections.**

3. authenticatoni svc

4. Service discovery service (can also be zookeeper)

* Routes incoming messages from the Send Message Service to the appropriate WebSocket connections based on the user/device mapping.
* Utilizes the mapping stored in a distributed cache to determine the target server and port for message delivery.
* Implements load balancing techniques to evenly distribute message traffic across multiple servers
* The above is absolutely necessary here

5. Distributed cache   
 - stores the user id and the websocket service id here

6. Presence svc

Let user no when sb is online or offline

7. Key value store db

Store chats and user id/webscoket device mapping

**We have 2 services here**

1. Stateful and stateless services

Stateless service is

used to manage the login, signup, user profile, etc. These are common features among many websites and apps.

Stateful service:

The only stateful service is the chat service. The service is stateful because each client maintains a persistent network connection to a chat server. In this service, a client A diagram of a computer server

Description automatically generated

The above is what can happen on many levels:

* Chat servers facilitate message sending/receiving.
* Presence servers manage online/offline status.
* API servers handle everything including user login, signup, change profile, etc.
* Notification servers send push notifications.
* Finally, the key-value store is used to store chat history. When an offline user comes online, she will see all her previous chat history.

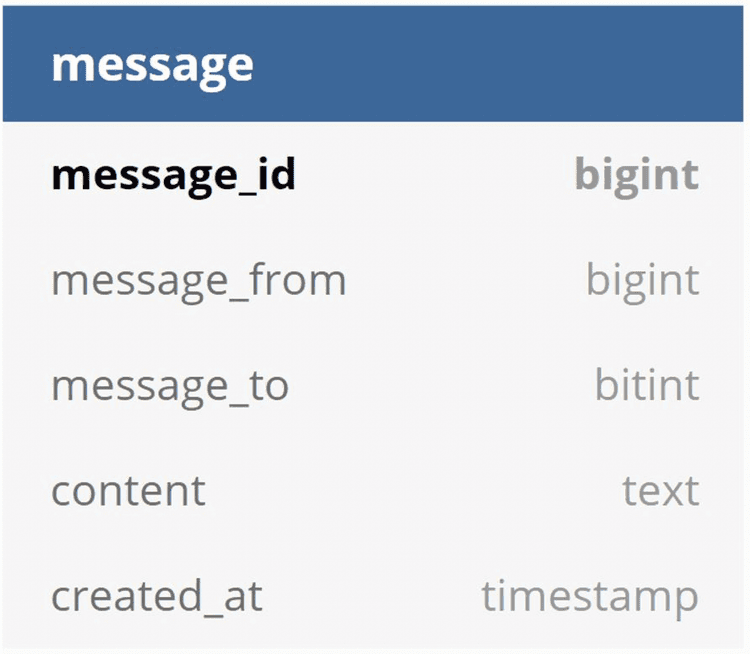
What does the database table look like?

**Data models**

Just now, we talked about using key-value stores as our storage layer. The most important data is message data. Let us take a close look.

**Message table for 1 on 1 chat**

Figure 9 shows the message table for 1 on 1 chat. The primary key is *message\_id*, which helps to decide message sequence. We cannot rely on *created\_at* to decide the message sequence because two messages can be created at the same time.



Msg table for group chat:

Figure 10 shows the message table for group chat. The composite primary key is *(channel\_id, message\_id).* Channel and group represent the same meaning here. *channel\_id* is the partition key because all queries in a group chat operate in a channel.

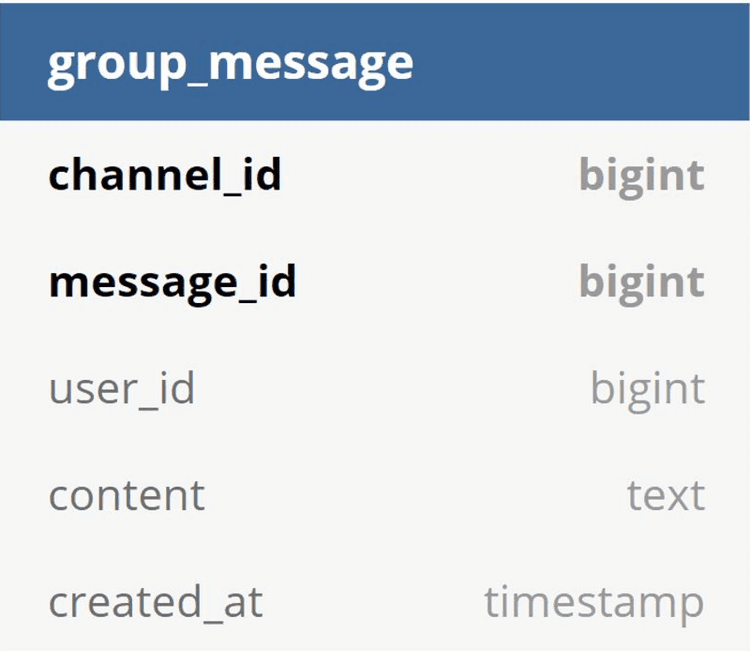


Figure 10

**How to generate a message id?**

2 requirements

1. Ids must be unique here

2. Ids should be sortable by time, meaning new rows have higher IDs than old ones.

The first one using auto-increment

The second one:

use a global 64-bit sequence number generator like Snowflake [6]. This is discussed in the “Design a unique ID generator in a distributed system” chapter.

The deep dive section here

**Step 3 - Design deep dive**

In a system design interview, usually you are expected to dive deep into some of the components in the high-level design. For the chat system, service discovery, messaging flows, and online/offline indicators worth deeper exploration.

**Service discovery**

The primary role of service discovery is to recommend the best chat server for a client based on the criteria like geographical location, server capacity, etc. Apache Zookeeper [7] is a popular open-source solution for service discovery. It registers all the available chat servers and picks the best chat server for a client based on predefined criteria.

Figure 11 shows how service discovery (Zookeeper) works.

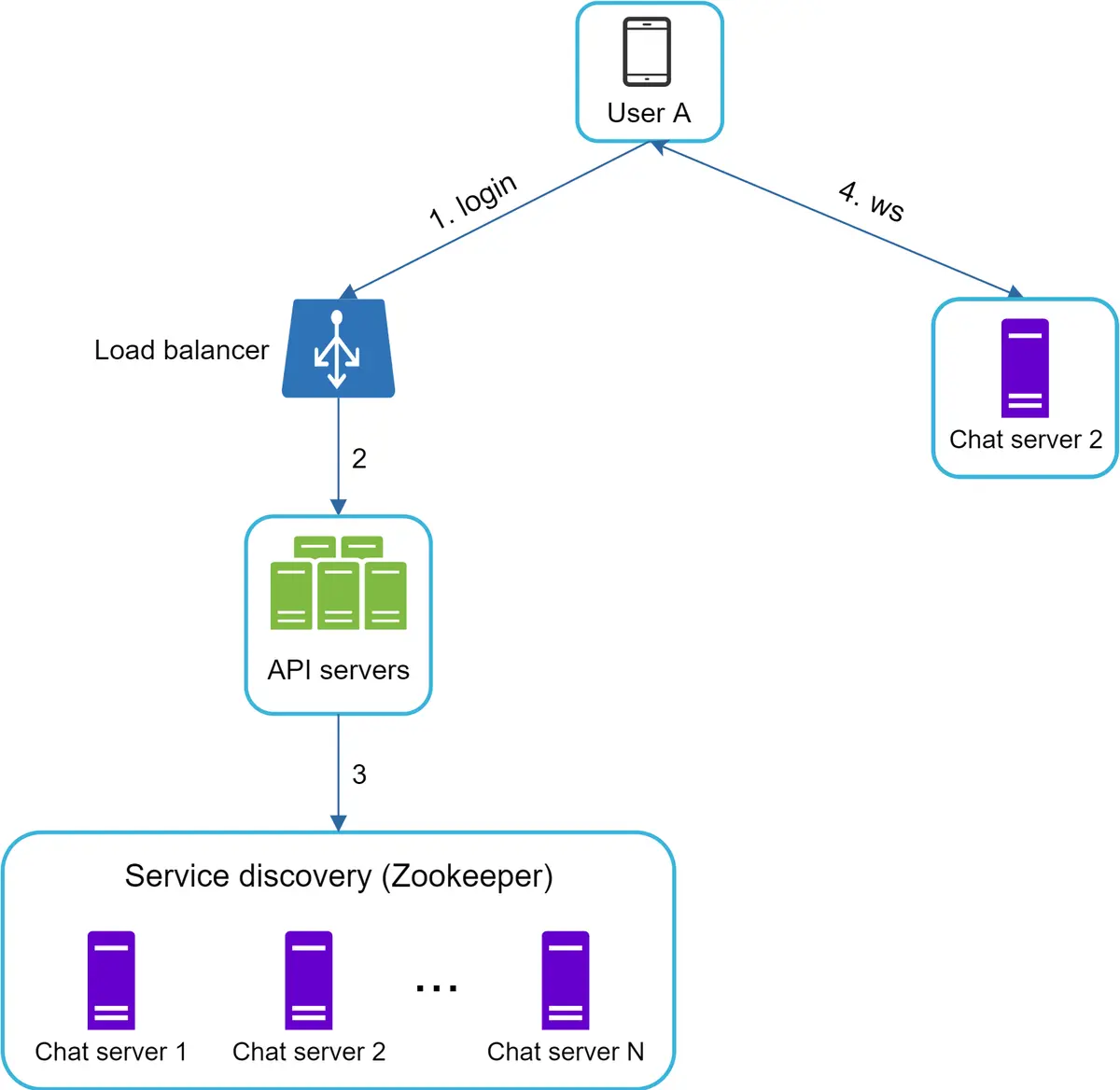


Figure 11

1. User A tries to log in to the app.

2. The load balancer sends the login request to API servers.

3. After the backend authenticates the user, service discovery finds the best chat server for User A. In this example, server 2 is chosen and the server info is returned back to User A.

4. User A connects to chat server 2 through WebSocket.

**Message flows**

It is interesting to understand the end-to-end flow of a chat system. In this section, we will explore 1 on 1 chat flow, message synchronization across multiple devices and group chat flow.

**1 on 1 chat flow**

Figure 12 explains what happens when User A sends a message to User B.

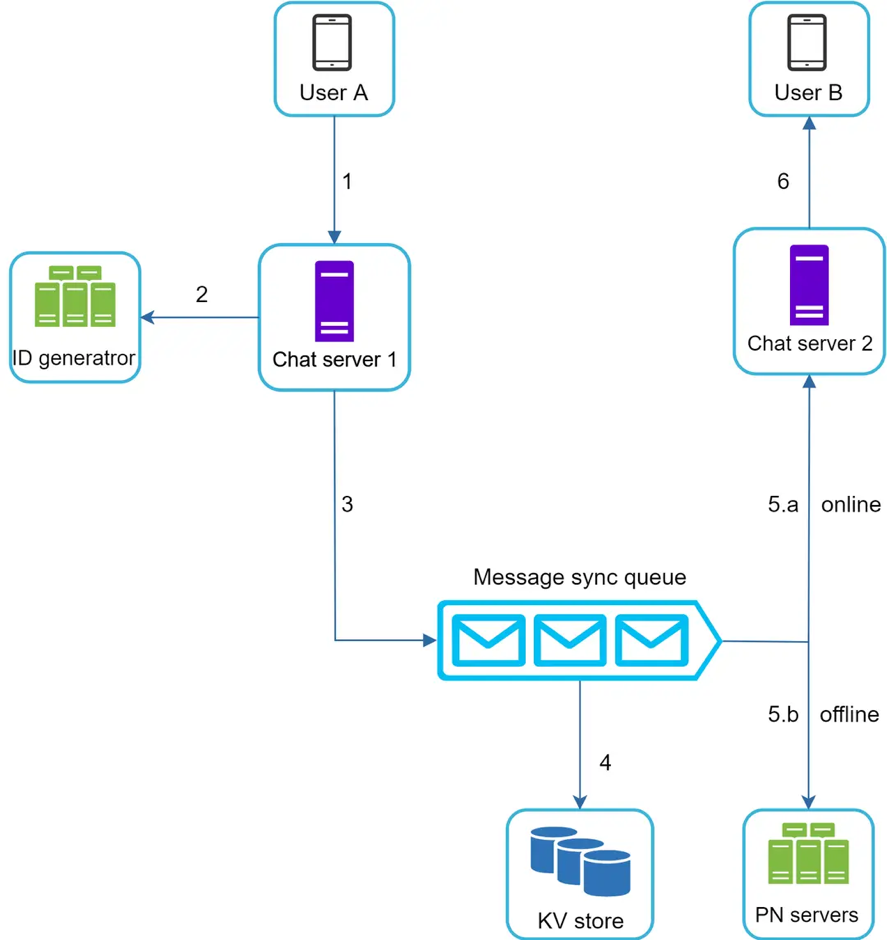


Figure 12

1. User A sends a chat message to Chat server 1.

2. Chat server 1 obtains a message ID from the ID generator.

3. Chat server 1 sends the message to the message sync queue.

4. The message is stored in a key-value store.

5.a. If User B is online, the message is forwarded to Chat server 2 where User B is connected.

5.b. If User B is offline, a push notification is sent from push notification (PN) servers.

6. Chat server 2 forwards the message to User B. There is a persistent WebSocket connection between User B and Chat server 2.

**How do we ensure messages are synced across multiple devices?**

Many users have multiple devices. We will explain how to sync messages across multiple devices. Figure 13 shows an example of message synchronization.

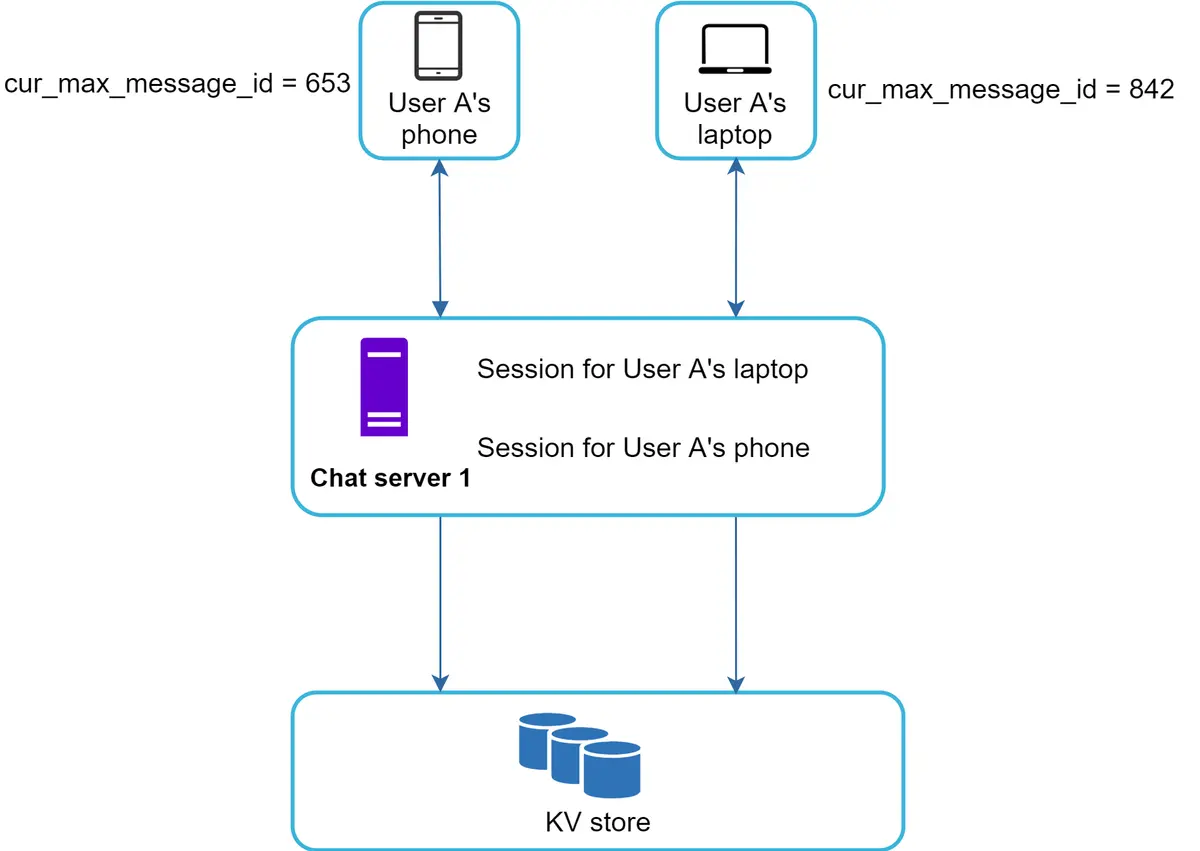


Figure 13

In Figure 13, user A has two devices: a phone and a laptop. When User A logs in to the chat app with her phone, it establishes a WebSocket connection with Chat server 1. Similarly, there is a connection  
  
 between the laptop and Chat server 1.

Each device maintains a variable called *cur\_max\_message\_id*, which keeps track of the latest message ID on the device. Messages that satisfy the following two conditions are considered as news messages:

* The recipient ID is equal to the currently logged-in user ID.
* Message ID in the key-value store is larger than *cur\_max\_message\_id*.

With distinct *cur\_max\_message\_id* on each device, message synchronization is easy as each device can get new messages from the KV store.  
  
  
**How does group chat work?**

**Small group chat flow**

In comparison to the one-on-one chat, the logic of group chat is more complicated. Figures 12-14 and 12-15 explain the flow.

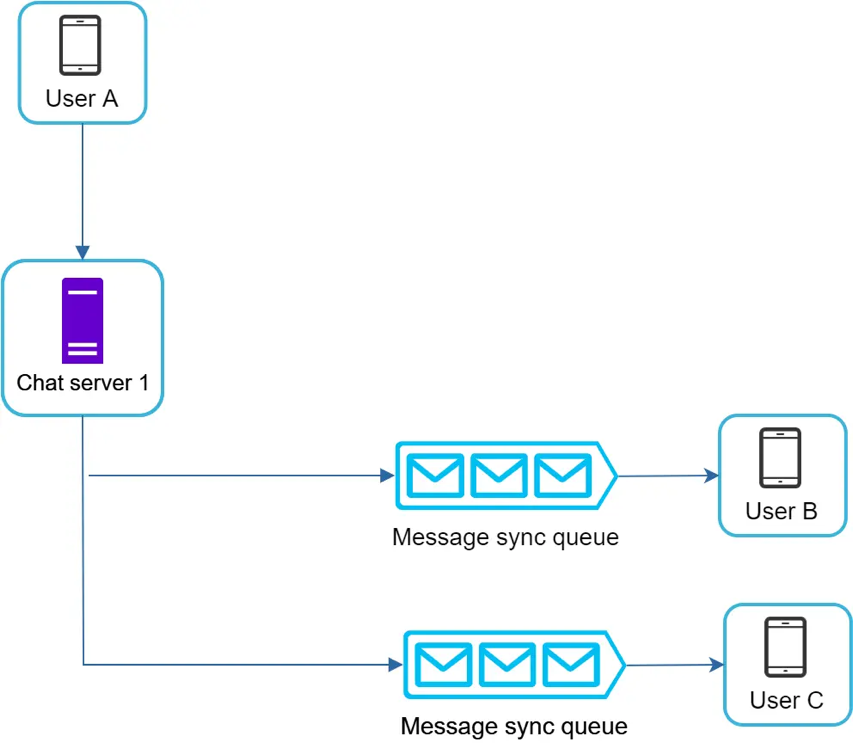


Figure 14  
  
Figure 14 here

Figure 14 explains what happens when User A sends a message in a group chat. Assume there are 3 members in the group (User A, User B and user C).   
  
 First, the message from User A is copied to each group member’s message sync queue: one for User B and the second for User C. You can think of the message sync queue as an inbox for a recipient. This design choice is good for small group chat because:

* it simplifies message sync flow as each client only needs to check its own inbox to get new messages.
* when the group number is small, storing a copy in each recipient’s inbox is not too expensive.

WeChat uses a similar approach, and it limits a group to 500 members [8]. However, for groups with a lot of users, storing a message copy for each member is not acceptable.

On the recipient side, a recipient can receive messages from multiple users. Each recipient has an inbox (message sync queue) which contains messages from different senders. Figure 15 illustrates the design.

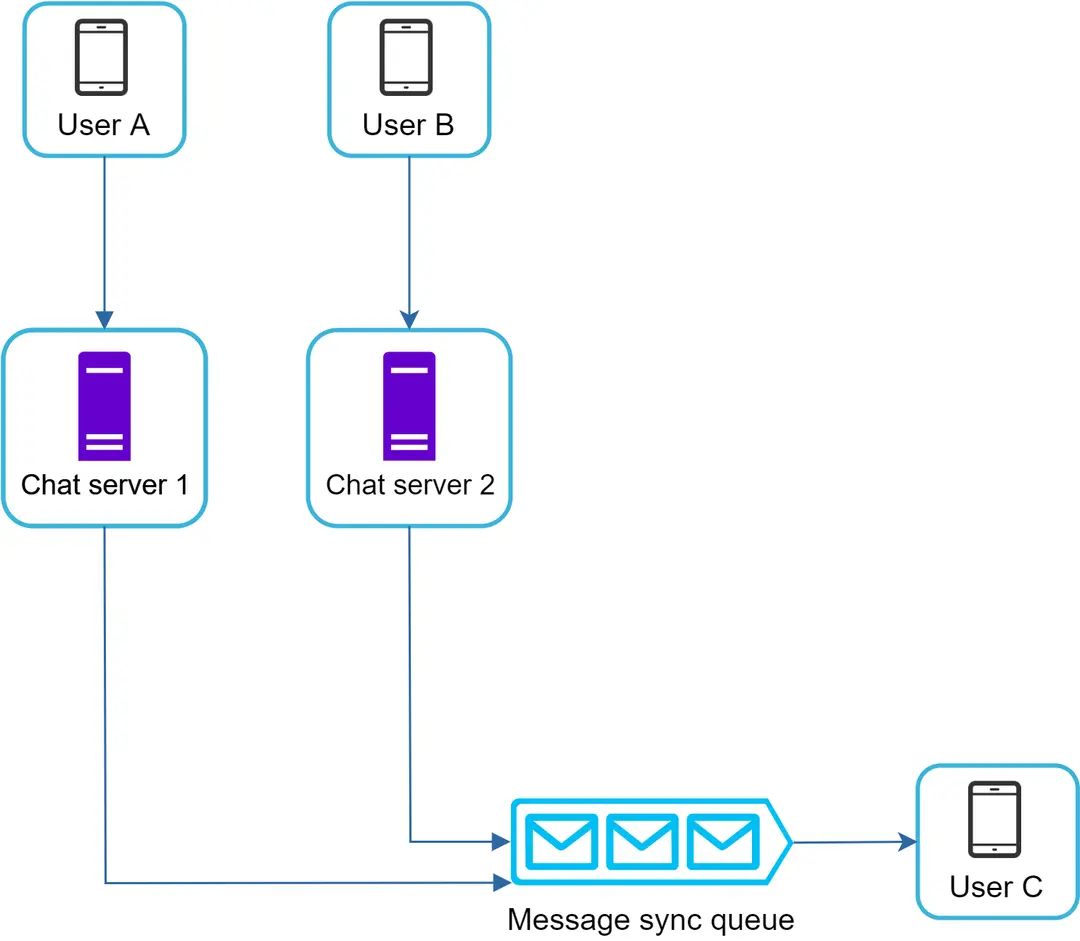


Figure 15

**Online presence**

An online presence indicator is an essential feature of many chat applications. Usually, you can see a green dot next to a user’s profile picture or username. This section explains what happens behind the scenes.

In the high-level design, presence servers are responsible for managing online status and communicating with clients through WebSocket. There are a few flows that will trigger online status change. Let us examine each of them.

**User login**

The user login flow is explained in the “Service Discovery” section. After a WebSocket connection is built between the client and the real-time service, user A’s online status and *last\_active\_at* timestamp are saved in the KV store. Presence indicator shows the user is online after she logs in.

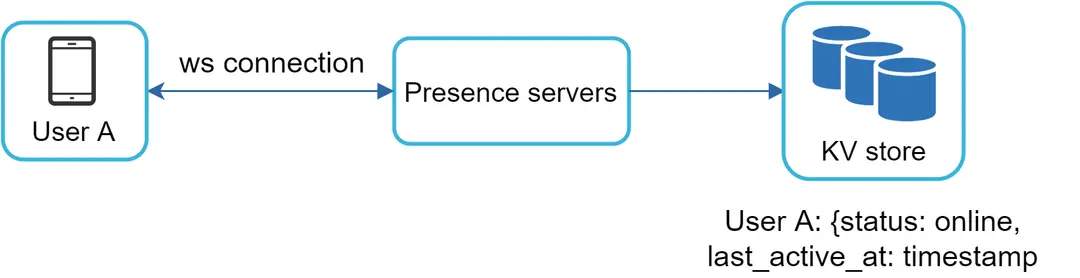


Figure 16

**User logout**

When a user logs out, it goes through the user logout flow as shown in Figure 17. The online status is changed to offline in the KV store. The presence indicator shows a user is offline.  
  
The kv store is like cassandra here

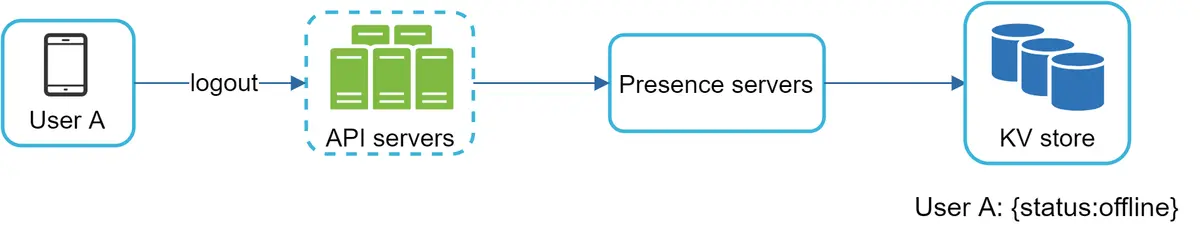


Figure 17

**User disconnection**

When a user disconnects from the internet, the persistent connection between the client and server is lost. A naive way to handle user disconnection is to mark the user as offline and change the status to online when the connection re-establishes.

**Why can’t we mark the user as offline and change status online when the connection reestalblishes?**

What’s the solutino to this?

Using a heartbeat mechanism here

However, this approach has a major flaw. It is common for users to disconnect and reconnect to the internet frequently in a short time. (user in a tunnel) And the above is important here

Using heartbeat mechanism. Periodically, an online client sends a heartbeat event to presence servers. If presence servers receive a heartbeat event within a certain time, say x seconds from the client, a user is considered as online. Otherwise, it is offline.

A screenshot of a video game

Description automatically generated  
  
Figure 18 here:

In Figure 18, the client sends a heartbeat event to the server every 5 seconds. After sending 3 heartbeat events, the client is disconnected and does not reconnect within x = 30 seconds (This number is arbitrarily chosen to demonstrate the logic). The online status is changed to offline.  
  
**Online status fanout**

How do user A’s friends know about the status changes?   
  
 Figure 19 explains how it works. Presence servers use a publish-subscribe model, in which each friend pair maintains a channel.   
  
 When User A’s online status changes, it publishes the event to three channels, channel A-B, A-C, and A-D. Those three channels are subscribed by User B, C, and D, respectively. Thus, it is easy for friends to get online status updates. The communication between clients and servers is through real-time WebSocket.

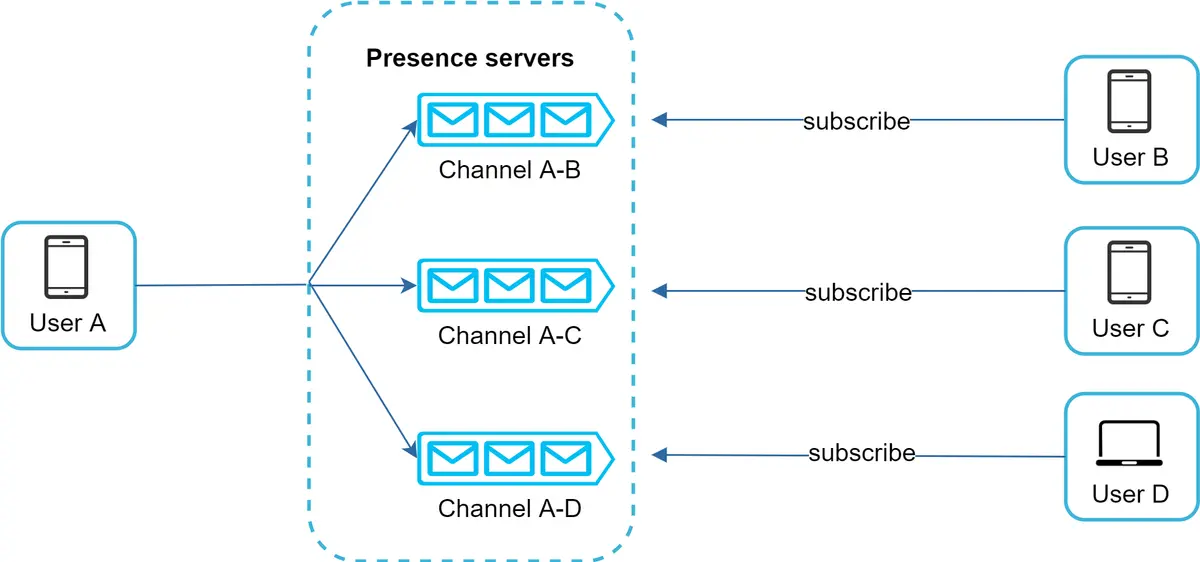


Figure 19

The above design is effective for a small user group. For instance, WeChat uses a similar approach because its user group is capped to 500. For larger groups, informing all members about online status is expensive and time consuming. Assume a group has 100,000 members. Each status change will generate 100,000 events. To solve the performance bottleneck, a possible solution is to fetch online status only when a user enters a group or manually refreshes the friend list.

**Step 4 - Wrap up**

In this chapter, we presented a chat system architecture that supports both 1-to-1 chat and small group chat. WebSocket is used for real-time communication between the client and server. The chat system contains the following components: chat servers for real-time messaging, presence servers for managing online presence, push notification servers for sending push notifications, key-value stores for chat history persistence and API servers for other functionalities.

If you have extra time at the end of the interview, here are additional talking points:

* Extend the chat app to support media files such as photos and videos. Media files are significantly larger than text in size. Compression, cloud storage, and thumbnails are interesting topics to talk about.
* End-to-end encryption. Whatsapp supports end-to-end encryption for messages. Only the sender and the recipient can read messages. Interested readers should refer to the article in the reference materials [9].
* Caching messages on the client-side is effective to reduce the data transfer between the client and server.
* Improve load time. Slack built a geographically distributed network to cache users’ data, channels, etc. for better load time [10].
* Error handling.
* The chat server error. There might be hundreds of thousands, or even more persistent connections to a chat server. If a chat server goes offline, service discovery (Zookeeper) will provide a new chat server for clients to establish new connections with.
* Message resent mechanism. Retry and queueing are common techniques for resending messages.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

The above makes sense here:

And that’s step # 1 here

**What r the failover scenario here?**

**When a server instance is marked as unavailable or detected as unhealthy,** the reassignment of devices to other available servers can be handled in the following steps:

A white board with writing on it

Description automatically generated

1. **Health Monitoring:** The health monitoring component periodically checks the health of server instances. If a server is found to be unhealthy or unresponsive, it triggers the reassignment process.
2. **Device Disconnection:** The connection manager receives a notification that the server instance is unavailable. It initiates the disconnection process for all devices connected to the unhealthy server.
3. **Reassignment Algorithm:**The connection manager uses a reassignment algorithm to determine the new server instance to which the devices should be reassigned. The algorithm can consider factors like server availability, load balancing, and minimizing network latency.
4. **Device Reconnection:**The connection manager sends instructions to the devices to reconnect to the newly assigned server instance. This can be done by notifying the devices through push notifications or by providing updated connection details to the devices.
5. **State Transfer:**If necessary, the connection manager coordinates with the new server instance to transfer any relevant state or session information for the devices being reassigned. This ensures a seamless transition for the connected devices.
6. **Connection Establishment:** The devices receive the reconnection instructions and establish a new WebSocket connection with the newly assigned server instance. The devices update their connection details accordingly.

**High level design here**

**Building the chat system design here**

Desigining the database table for 1-1 chat, for group chat, with contacts, adding attachments for messages. And then also some other parts of this

**To handle millinos of ysers, 1 websocket will not be enough here, needs more for sure her.**

A diagram of a software system

Description automatically generated

**What does message service do?**

It acts as an interface to the database for other services interacting with the databases. It stores and retrieves messages from the database and deletes them after a specific amount of time

**What does the websocket manager do?**

1. It figures out which websocket server each user is connected to

**What happens if a user A wants to send a mesasge to a user B?**

Now, if user A wants to send a message to user B. Since we have a number of WebSocket servers these users can be connected to different ones. So how does this work:

1. User A communicates with the corresponding WebSocket server to which it is connected.
2. The WebSocket server associated with user A identifies the WebSocket to which user B is connected via the WebSocket manager. If user B is online, the WebSocket manager responds back to user A’s WebSocket server that user B is connected with its WebSocket server.
3. At the same time, the WebSocket server sends the message to the message service and it is stored in the database (in case user B is offline). So the eviction policy for the message to be processed will be first-in-first-out (which makes perfect sense in this case). And when the messages are delivered to the receiver, they are deleted from the database.
4. Now, user A’s WebSocket server has the information that user B is connected with its own WebSocket server. Both users communicate with the WebSocket manager to find each other’s WebSocket server.
5. If user B is offline, messages are kept in the database. Whenever they become online, all the messages intended for user B are delivered via push notification. Otherwise, these messages are deleted permanently after 30 days.

**How does user A send data to user B here?**

A diagram of a service

Description automatically generated

when a user say user a wants to send a message to user B he sends a request to the messaging service with the ID of user B now before this user a establishes a persistent connection to the messaging service via websocket protocol because like we discussed earlier it's a bi-directional connection now in a traditional HTTP via session service and sends the message accordingly

**How would a session service work?**

so how does session service works? whenever a user connects to the messaging service, the messaging service tells the session service in which websocket handler the user has established the connection which is stored in a database more

**What happens if user is offline here? This is an important point here**

A diagram of a service

Description automatically generated

likely a nosql database since you don't have any scope for relations between the data later this information is used to send messages to the other end now what if the user B is offline in such cases we need to temporarily store the message to deliver it

**Using the relay service:**

lately message service forwards this message to the relay service in the occasion when the user is offline, the relay service will store the unsigned messages the from and the two user ID in a database like Cassandra.

**Last seen service:**

This is used to indicate the online status of each user here.

last scene service is used to store the timestamp of each user this information is based on logging of each user activity the client-side application should be intelligent enough to identify the difference between the user activity and the application activity itself and sends a signal to the app server this information can also be used to show the online status of the users.

**Using an asset service**

Asset Services used to store and retrieve multimedia files in an object-based storage or a blob store like AWS S3 bucket and finally we have a group messaging service which is more like the messaging service except we need to publish the message to all the users associated with the same group ID this service will rely on the session service to identify the server to which each user is connected to now compared to other services group messaging service is fairly complicated and

**Maintaining a hash for each file**

When sending a file here, the asset service assigns an ID that’s communicated with the sender. The asset service also maintains a hash for each file to avoid duplication of content on the blob storage.

* + For example, if a user wants to upload an image that’s already there in the blob storage, the image won’t be uploaded. Instead, the same ID is forwarded to the receiver.

File can be downloaded using the same hash id here,

**How would user b know what image or asset to download?**

when a user a sends an image to another user B user a will upload the image to a server and get the image ID and then it will send the image ID to user B user B can now search the image and download the image from the server or another method is the image itself will be compressed on the device side

and sent to the asset service the asset service can take the message and find out the type of message once the asset service detects the format of the message as media or image it is stored in our blob store which in this case is an S3 bucket the links to the location of the media files can be stored in a SQL or nosql database with mapping to the user details now guys there is no right and wrong design as long you can clarify the requirements for the interviewer defined your design and make changes if any you are all set [Music

Database schema here

A diagram of a database schema

Description automatically generated

**Database section here**

apical contact number and then we'll have groups and each group will have its own group ID and the user ID of the user part of that group

then we have unsent messages with Fields message ID sent from ID sent to ID the content or the media URL say if you are sending messages such as video messages or PNG

files you send the location of that PNG file in the form of media URL and the timestamp you can also have a last scene table with a user ID and this timestamp and finally you have the sessions table which basically maintains mapping of user ID with a server ID what we discuss until now was how a personal chat works on a system like

WhatsApp but what about group chats for every group Creator will have a new group ID and this group ID will have a mapping to all the users who are in the group that is in the group mapping database

**Desigining for group chat here**

A diagram of a chat

Description automatically generated

now groups will behave a little different from users because websocket handlers won't keep track of groups it just tracks active users so when user a wants to send a message to Group 1 websocket

**Why add kafka here?**

1. Using producer here: Messaging service
2. Using consumer: the

Handler 1 gets in touch with message service message service will store the information to a queue or Kafka topic automation

**How does it work exactly here? What’s the total flow here?**

such as which user is sending word message to which group basically the message service will act as a Kafka producer and then we'll have group message Handler as our Kafka consumer which will be listening to the Kafka topics so whenever the message service posts a new message to Kafka topic that is user a is sending this message hello to group 1.

**group message Handler will query group service** to get all the users which are in group id1 after it gets the list of users who are supposed to get the message it now needs the respective list of machines those users are connected to which it will get from the websocket manager once it gets the list of all the machines the group message Handler will send messages to individual machines by talking to the respective websocket Handler again websocket Handler is a lightweight server which keeps an open bi-directional connection with all the active users and for whatever reason if the receiver is offline the message must be encrypted and stored in the servers database o

**What exactly happens?**

Since user A is connected to a WebSocket server, it sends a message to the message service intended for Group A.The message service sends the message to Kafka with other specific information about the group. The message is saved there for further processing. In Kafka terminology, a group can be a topic, and the senders and receivers can be producers and consumers, respectively.

Now, here comes the responsibility of the group service. The group service keeps all information about users in each group in the system. It has all the information about each group, including user IDs, group ID, status, group icon, number of users, and so on. This service resides on top of the MySQL database cluster, with multiple secondary replicas distributed geographically. A Redis cache server also exists to cache data from the MySQL servers. Both geographically distributed replicas and Redis cache aid in reducing latency.

The group message handler communicates with the group service to retrieve data of Group/A users. In the last step, the group message handler follows the same process as a WebSocket server and delivers the message to each user.