**DESIGN A Chat System**

### Understanding Problem and Establishing Design Scope:

Query1: What kind of Chat app shall we design? 1on1 or group based?

-> It should support both 1on1 and group chat

Query2: Is this a mobile app? Or a web app? Or both?

-> Both

Query3: What is the scale of this app? A startup or massive scale?

-> It should support 50 million daily active users(DAU)

Query 4: What features are important for the chat app? Can it support attachment?

-> 1 on 1 chat, group chat, online indicator. The system only supports text messages.

Query5: Is there a message size limit?

-> Yes, text length should be less than 100,000 characters long.

Query6: How long shall we store the chat history?

-> Forever

Query7: For group chat, what is the group member limit?

-> A maximum of 100 people.

Feature to support:

• A one-on-one chat with low delivery latency

• Small group chat (max of 100 people)

• Online presence

• Multiple device support. The same account can be logged in to multiple accounts at the same time.

• Push notifications

### Back of the Envelope Estimation:

50 million DAU = 50,000,000 = 5\*10^7 users daily.

Assuming the average number of messages from one person each day is 10.

So, Daily messages = 5\* 10^8.

Let average message length be 100 characters = 5\*10^10 character message

Which is 5\*10^10 bytes ~ 50 GB

Peak Data usage per day is 100GB

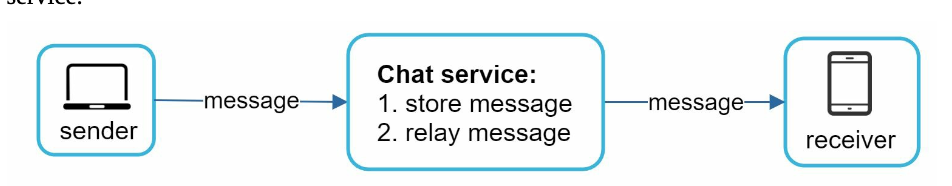
Taking 5x of this data for notification, user details etc. 500GB

For 1 year 365\*500GB = 10^6GB~ 1PB  
For 10 years 10 PB.

### High Level Design Propositions and approaches:

The chat service must support the following functions:

1. Receive messages from other clients
2. Find the right recipients for each message and relay the message to the recipients.
3. If a recipient is not online, hold the messages for that recipient on the server until she is online.

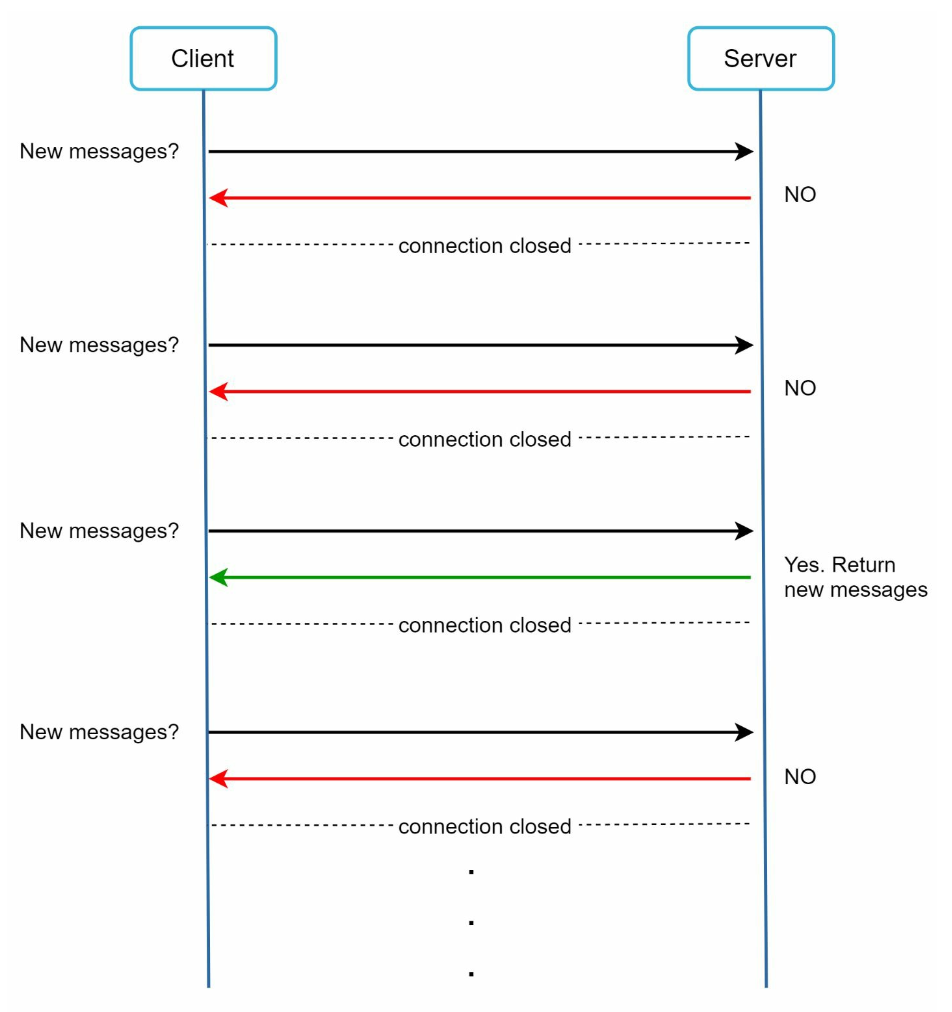


When a client intends to start a chat, it connects the chats service using one or more network protocols. For a chat service, the choice of network protocols is important.

Requests are initiated by the client for most client/server applications. This is also true for the sender side of a chat application. When the sender sends a message to the receiver via the chat service, it uses the time-tested HTTP protocol, which is the most common web protocol. In this scenario, the client opens a HTTP connection with the chat service and sends the message, informing the service to send the message to the receiver. The keep-alive is efficient for this because the keep-alive header allows a client to maintain a persistent connection with the chat service. It also reduces the number of TCP handshakes. HTTP is a fine option on the sender side, and many popular chat applications such as Facebook used HTTP initially to send messages.   
Since HTTP is client-initiated, it is not trivial to send messages from the server. Over the years, many techniques are used to simulate a server-initiated connection: polling, long polling, and WebSocket.

**Polling**

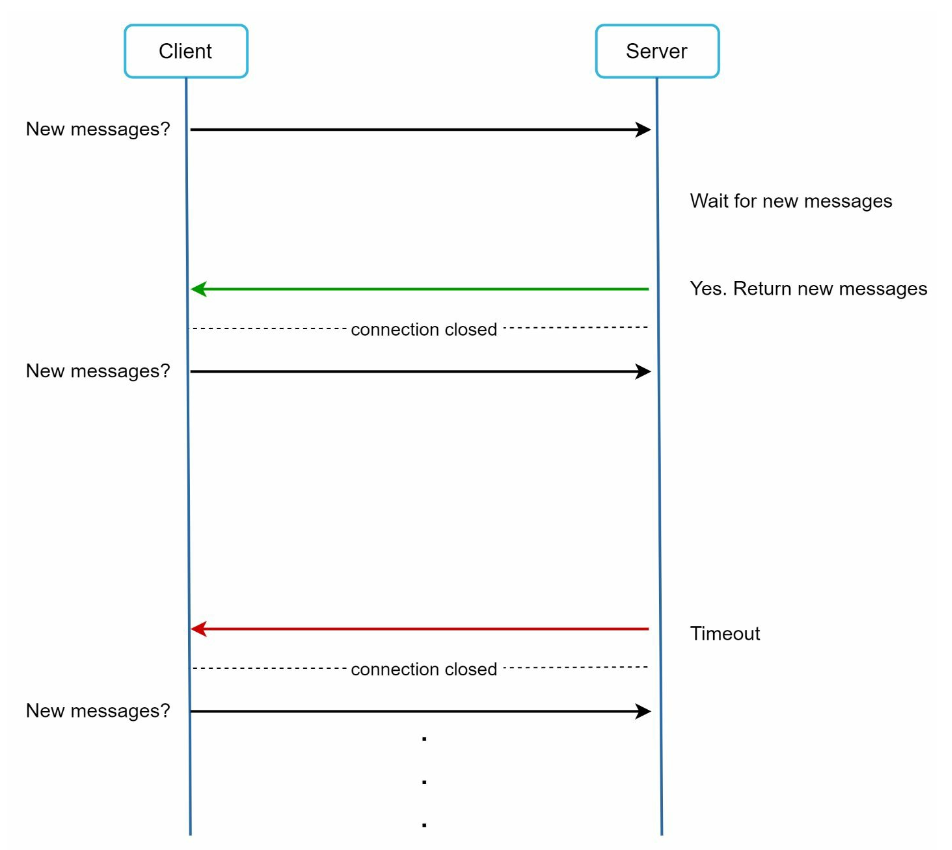
Polling is a technique in which the client periodically asks the server if there are messages available. **Depending on polling frequency, polly could be costly.** It could consume precious server resources to answer a question that offers no as an answer most of the time.



**Long polling**

Because polling could be inefficient, the next progression is long polling.

In long polling a client holds the connection open until there are actually new messages available or a timeout has been reached. Once the client receives new messages, it immediately sends another request to the server, restarting the process. Long polling has a few drawbacks:

* Sender and receiver may not connect to the same chat server. HTTP based servers are usually stateless. If you use round robin for load balancing, the server that receives the message might not have a long-polling connection with the client who receives the message.
* A server has no good way to tell if a client is disconnected.
* It is inefficient. If a user does not chat much, long polling still makes periodic connections after timeouts.

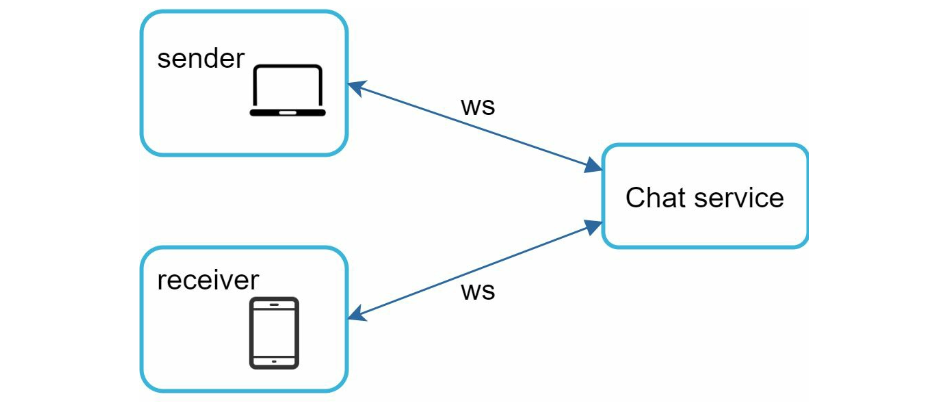
**WebSocket**

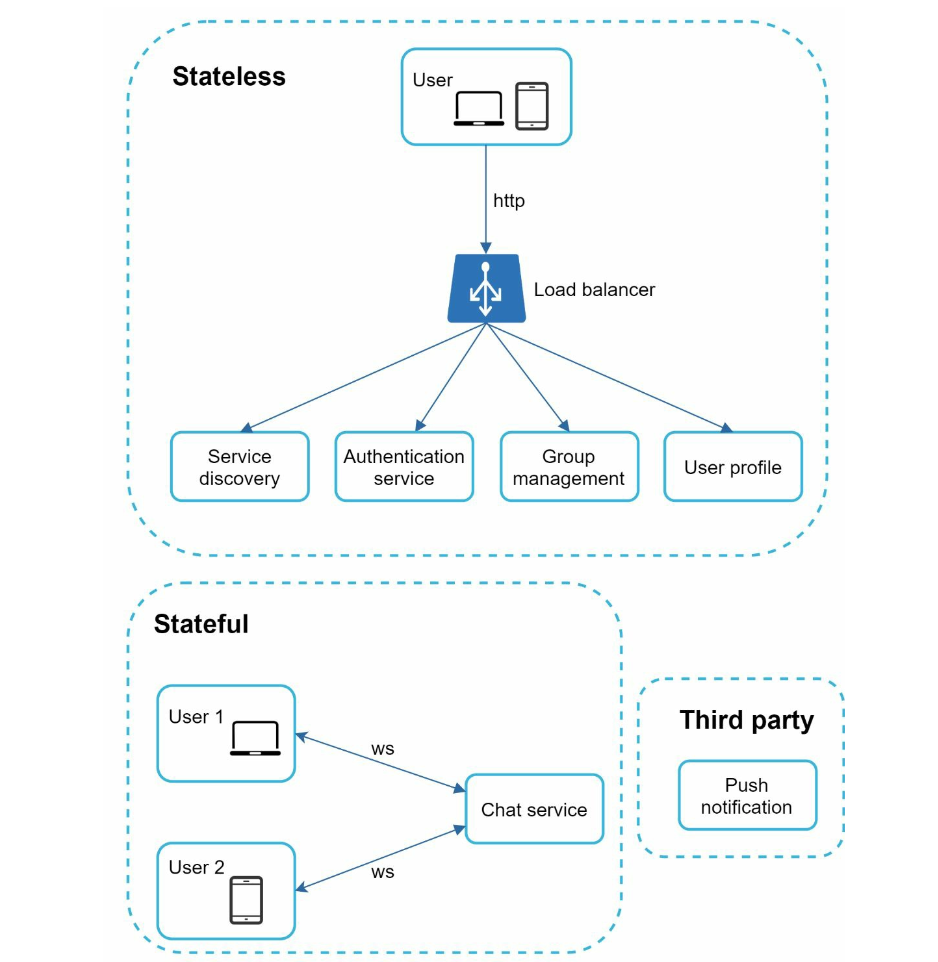
WebSocket is the most common solution for sending asynchronous updates from server to client.

WebSocket connection is initiated by the client. It is bi-directional and persistent. It starts its life as a HTTP connection and could be “upgraded” via some well-defined handshake to a Websocket connection . Through this persistent connection a server could send updates to a client. WebSocket connections generally work even if a firewall is in place

### 

By using WebSocket for both sending and receiving, it simplifies the design and makes implementation on both client and server more straightforward. Since WebSocket connections are persistent, efficient connection management is critical on the server-side.

  
**High Level Design**

Most features(sign up, login, user, profile, etc) of a chat application could use the traditional request/response method over HTTP.

**Stateless Service**

Stateless services are traditional public-facing request/response services, used to manage the login, signup, user profile, etc. These are common features among many websites and apps.

Stateless services sit behind a load balancer whose job is to route requests to the correct services based on the request paths.

The one service that has an important factor for chat systems is service discovery. Its primary job is to give the client a list of DNS host names of chat servers that the client could connect to.

**Stateful Service**

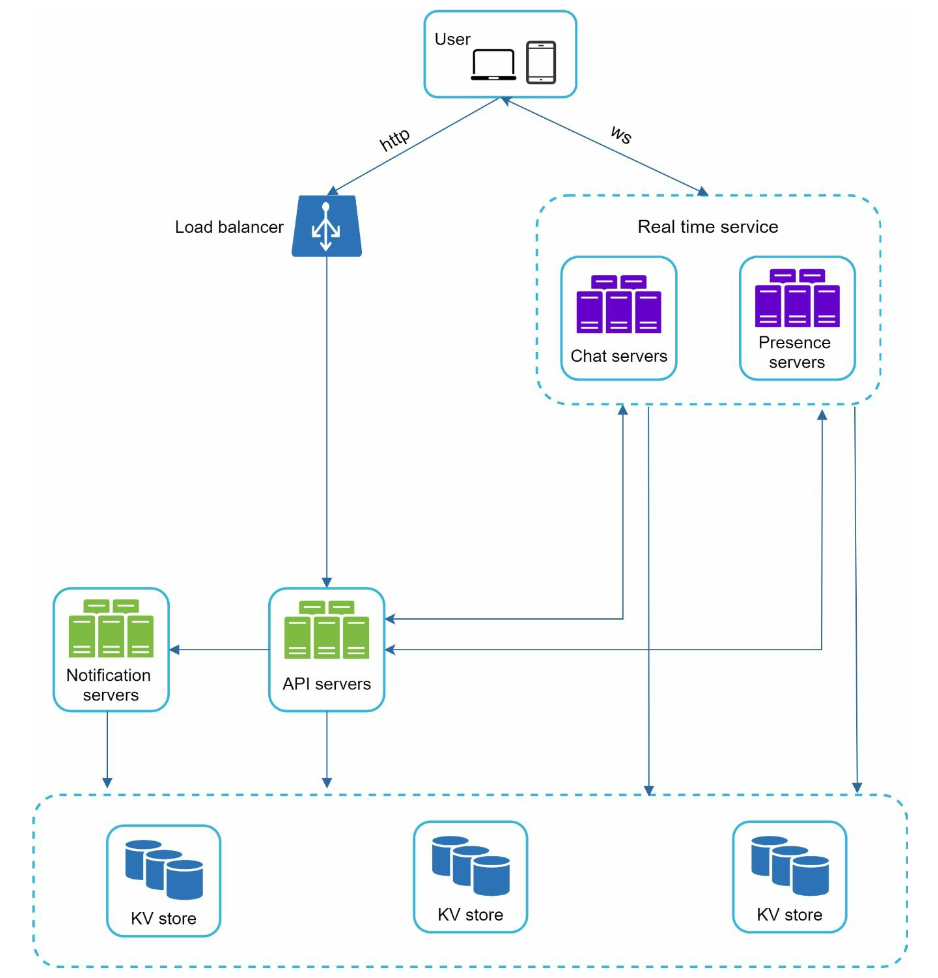
The only stateful service is the chat service. The service is stateful because each client maintains a persistent network connection to a chat server. In this service, a client normally does not switch to another chat server as long as the server is still available. The service discovery coordinates closely with the chat service to avoid server overloading.

**Third Party integration**

For a chat app, push notification is the most important third-party integration. It is a way to inform users when new messages have arrived, even when the app is not running.

**Scalability**

The number of concurrent connections that a server can handle will most likely be the limiting factor. In our scenario, at 1M concurrent users, assuming each user connection needs 10KB of memory on the server, it only needs about 10GB of memory to hold all the connections on the box.



* Chat servers facilitate message sending/receiving.
* Presence server 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.

**Storage**

Data layer usually requires some effort to get it correct. An important decision we must make is to decide on the right type of database to use: relational databases or NoSQL databases?

Two types of data exist in a typical chat system:

1. The first is generic data, such as user profile, setting, user friend list. These data are stored in robust and reliable relational databases. Replication and sharding are common techniques to satisfy availability and scalability requirements.
2. The second is unique chat systems: chat history data.

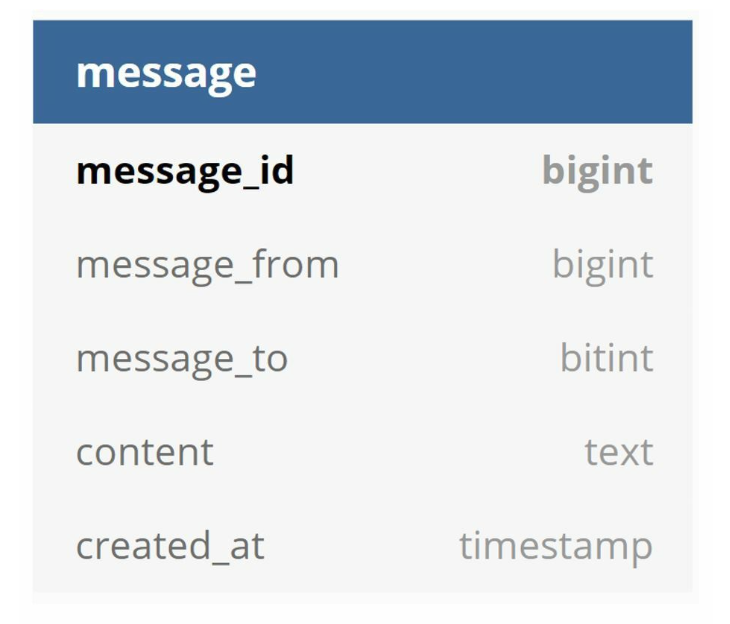
* The amount of data is enormous for a chat system. A previous study reveals that FB messenger and WhatsApp process 60 billion messages a day.
* Only recent chats are accessed frequently. Users do not usually look up for old chats.
* Although very recent chat history is viewed in most cases, users might use features that have unique random access to data, such as search, view your mentions, jump to specific messages, etc. These cases should be supported by the data access layer.
* The read to write ratio is about 1:1 for 1 on 1 chat apps.

We recommend key-value stores for the following reasons:

* Key-value stores allow easy horizontal scaling.
* Key-value stores provide very low latency to access data.
* Relational databases handle long tails of data well. When the indexes grow large, random access is expensive.
* Key-value stores are adopted by other proven reliable chat applications. For example, both Facebook messenger and Discord use key-value stores. Facebook messenger uses HBase [4], and Discord uses Cassandra [5].

**Message table for 1 on 1 chat**

Message table for 1 on 1 chat. Primary key is message\_id which helps to decide the message sequence. We cannot rely on created\_at because two messages can be created at the same time.



**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.



**Message ID**

Message\_id carries the responsibility of ensuring the order of messages. To ascertain the order of messages, message\_id must satisfy the following two requirements:

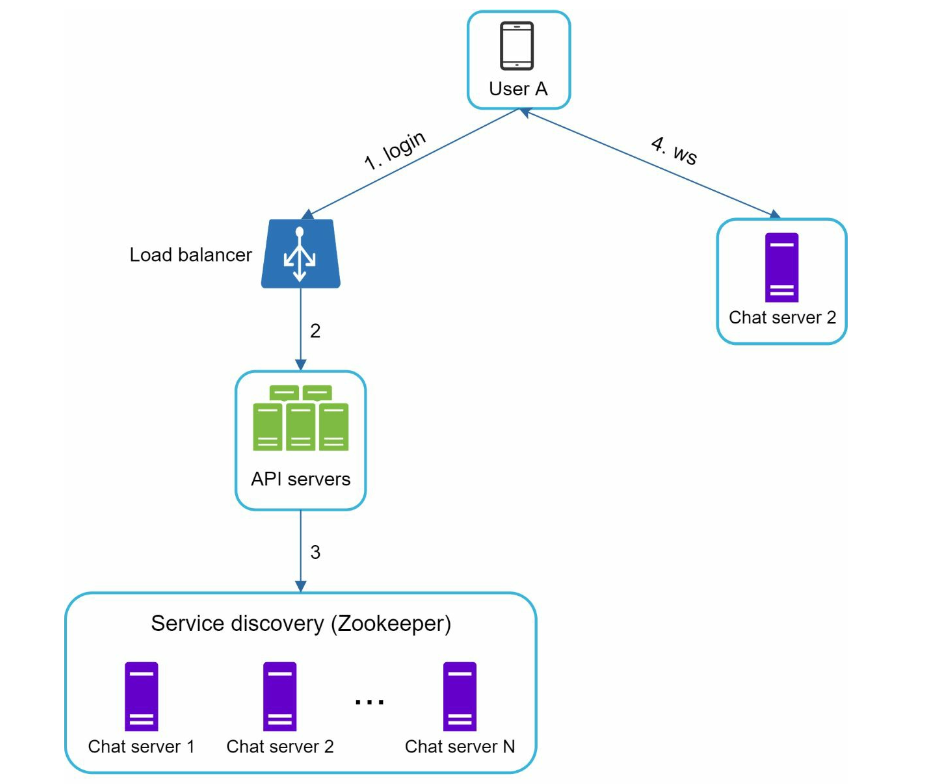
• IDs must be unique.  
• IDs should be sortable by time, meaning new rows have higher IDs than old ones.

Approach1: Use a global 64-bit sequence number generator like Snowflake

Approach2: Final approach is to use a local sequence number generator. Local means IDs are only unique within a group. The reason why local IDs work is that maintaining message sequence within a one-on-one channel or group channel is sufficient. This approach is easier to implement in comparison to global ID implementation.

### Design Deep Dive:

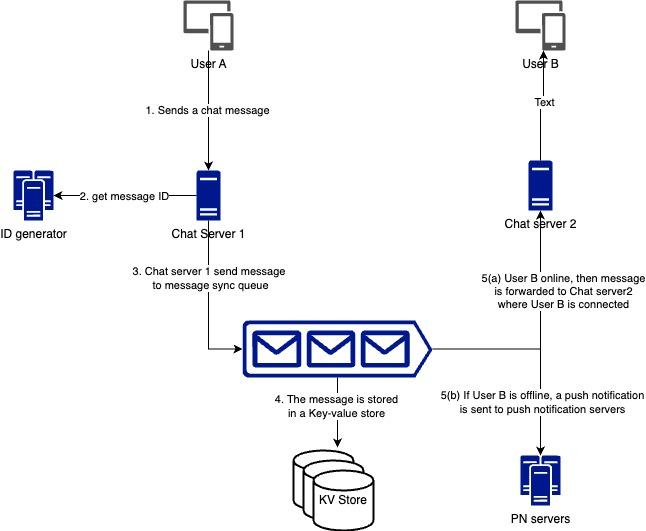
**Service Discovery:**

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

1. User A tries to login 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 Flow**

1. **1 on 1 chat flow**

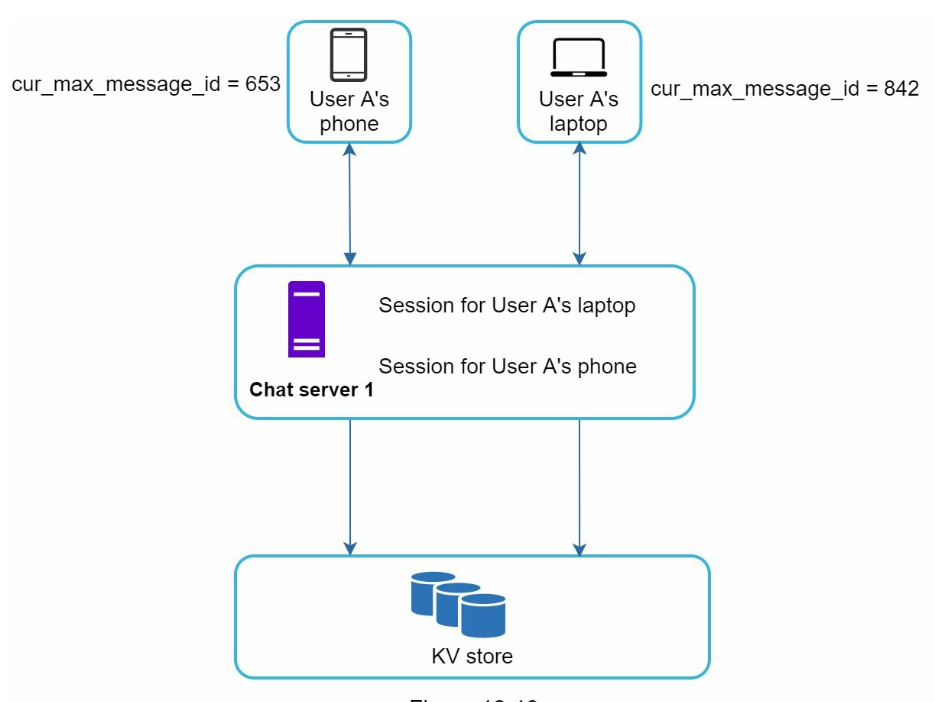
****

1. **Message synchronization across multiple devices**

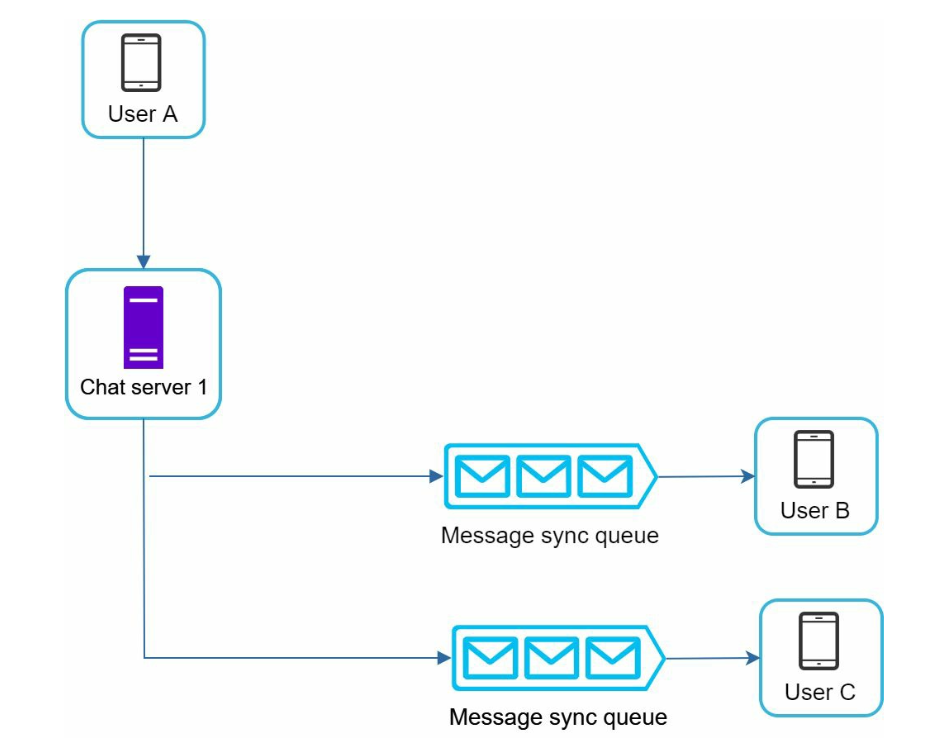
Each device maintains a variable called cur\_max\_message\_id, which keeps track of the latest messageID on the device. Message that satisfy the following two conditions are considered as new messages:

* The recipient ID is equal to currently logged in user ID
* MessageID is the key-value store 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.

****

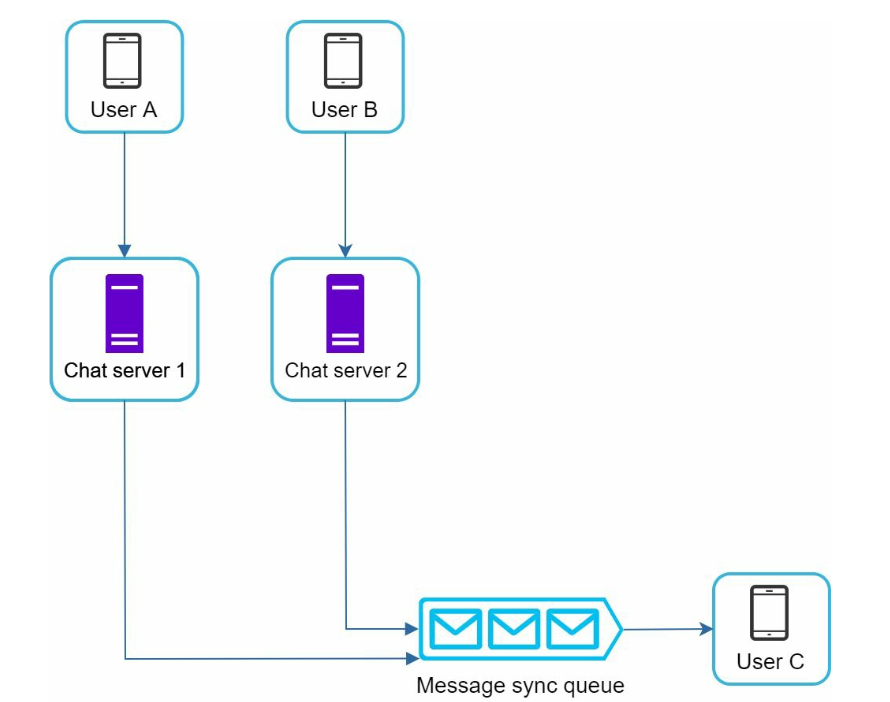
**Small Group Chat Flow:**



The above design is good for small group chat because:

* It simplifies 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.

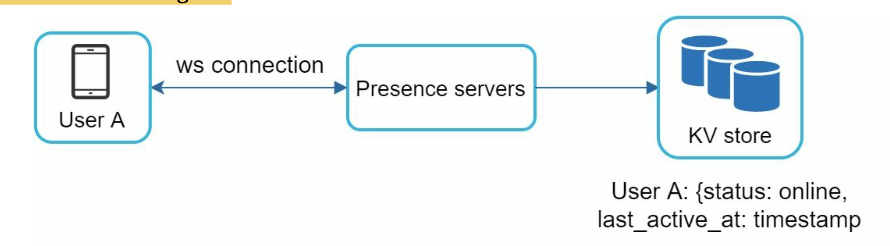
WeChat uses a similar approach, and it limits a group to 500 members. However, for groups with a lot of users, storing a message copy for each member is not acceptable.

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. 

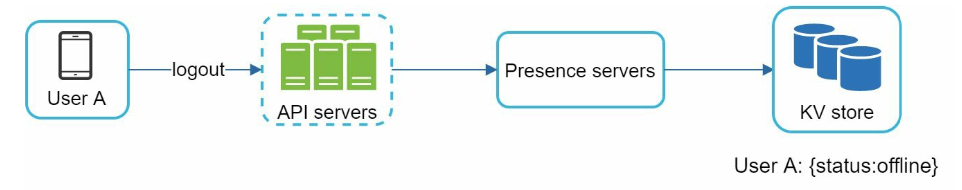
**Online presence**

Presence servers are responsible for managing online status and communicating with clients through WebSocket.

* **User login**

After a WebSocket connection is built between client and 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.

* **User logout**

When a user logs out, it goes through the user logout flow. The online status is changed to offline in the KV store. The presence indicator shows a user is offline.

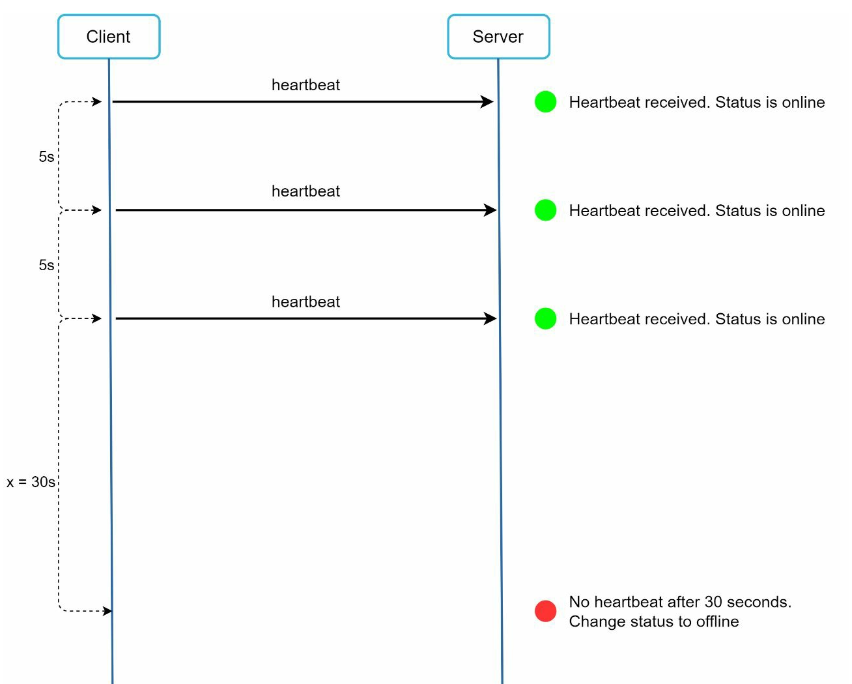
* **User disconnection**

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. However, this approach has a major flaw. It is common for users to disconnect and reconnect to the internet frequently in a short time.

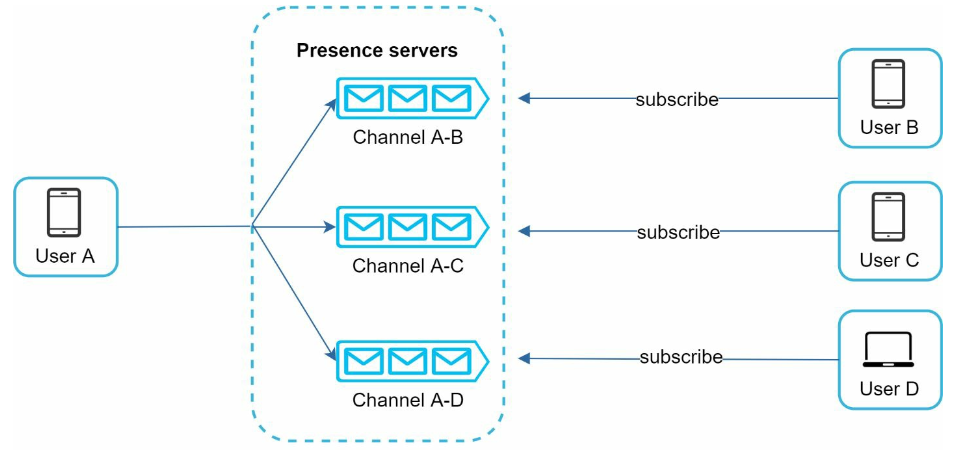
For example, network connections can be on and off while a user goes through a tunnel

Updating online status on every disconnect/reconnect would make the presence indicator change too often, resulting in poor user experience.

We introduce a heartbeat mechanism to solve this problem. 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.



* **Online status Fanout:**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.



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 10,000 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 list.

### Additional optimization:

• 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.

### 