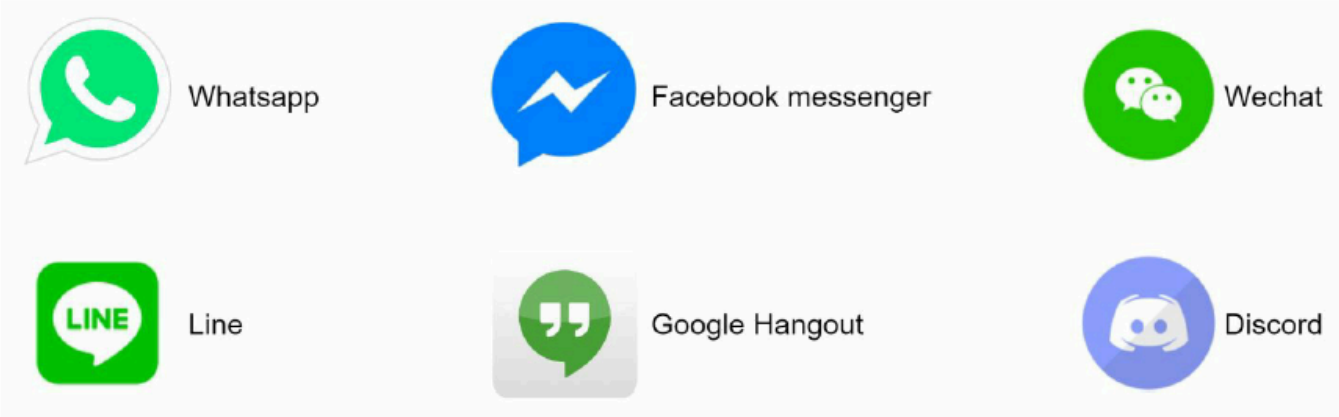
Chapter 12 Design a Chat System



Chat system = chat app in this chapter, not chat bot
Exact requirements: A chat app performs different functions for different people. E.g., individual chat vs group chat. Important to explore the feature requirements

Step 1 understand problem and establish design scope

Type of chat app to design

- one-on-one chat focused apps: Facebook Messenger, WeChat, and WhatsApp
- office chat apps focus on group chat: Slack, Teams
- game & large group chat interaction (low voice chat latency): Discord

Questions to clarify

- 1 on 1 vs group chat
- mobile and/or web
- scale of the app
- group chat member limit
- features to support, online indicator, voice, video, attachment, etc
- msg size
- e2e encryption requirement
- chat history storage time

In the chapter, we focus on designing a chat app like Facebook messenger, with an emphasis on the following features:

- 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

It is also important to agree on the design scale. We will design a system that supports 50 million DAU.

ChatGPT suggested:

1. Requirements Gathering

- **Functional Requirements**:
  - Real-time messaging (text, media, voice notes)
  - Message history and storage
  - End-to-end encryption (optional, but common in secure messaging apps)
  - Group chats
  - Online/offline status indicators
  - Notifications and read receipts
- **Non-Functional Requirements**:
  - Low latency
  - High availability
  - Scalability (handle millions of concurrent users)
  - Consistency and reliability

High level design and get buy-in

client - server communication: clients don't communicate directly with each other, each client connects to a **chat service**.

Chat service supports

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

Figure 12-2

Network Protocols

- one or more network protocols when a client connects the chats service
- Requests are initiated by the client for most client/server applications.
- **Sender side**
  - sends a msg to receiver via chat services: use **time-tested HTTP protocol**
  - keep-alive header: allow client to maintain a persistent connection with the chat service, and reduce TCP handshakes
- **Receiver side**
  - HTTP is client-initiated, not trivial to send msg from the server
  - **simulate a server-initiated connection**: polling, long polling, and WebSocket

Polling

Approach:

- The client (**receiver**) periodically asks the server if there are messages available

- close connection each time, no matter if there are new msgs from server

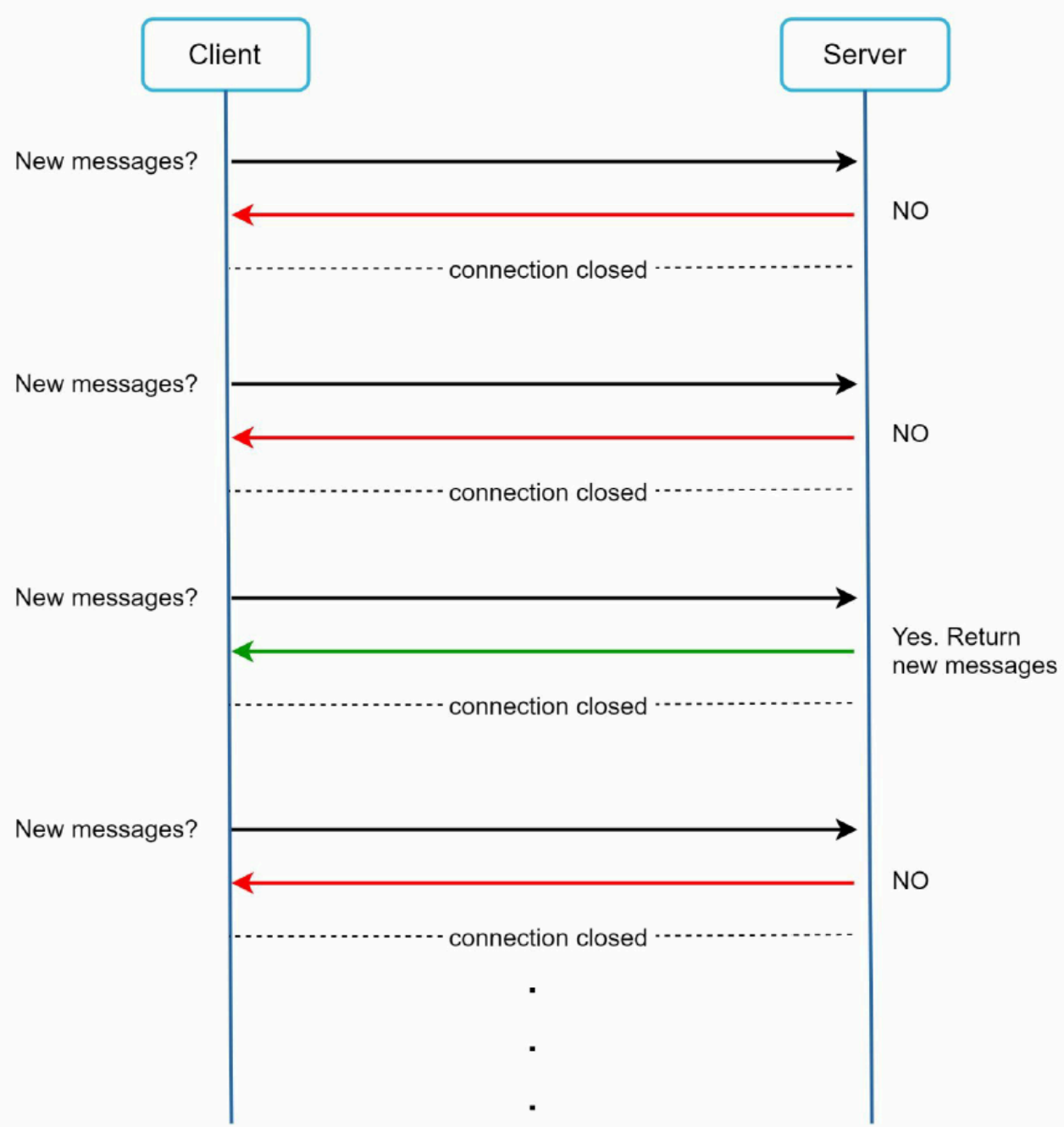


Figure 12-3

Drawbacks:

- Polling could be costly based on frequency - could consume precious server resources to answer a question that offers no as an answer most of the time

Long polling

Approach:

- Client **holds the connection open** until there are actually **new messages available or a timeout** threshold has been reached

- Received new messages, close connection
- Send another request to the server, restarting the process

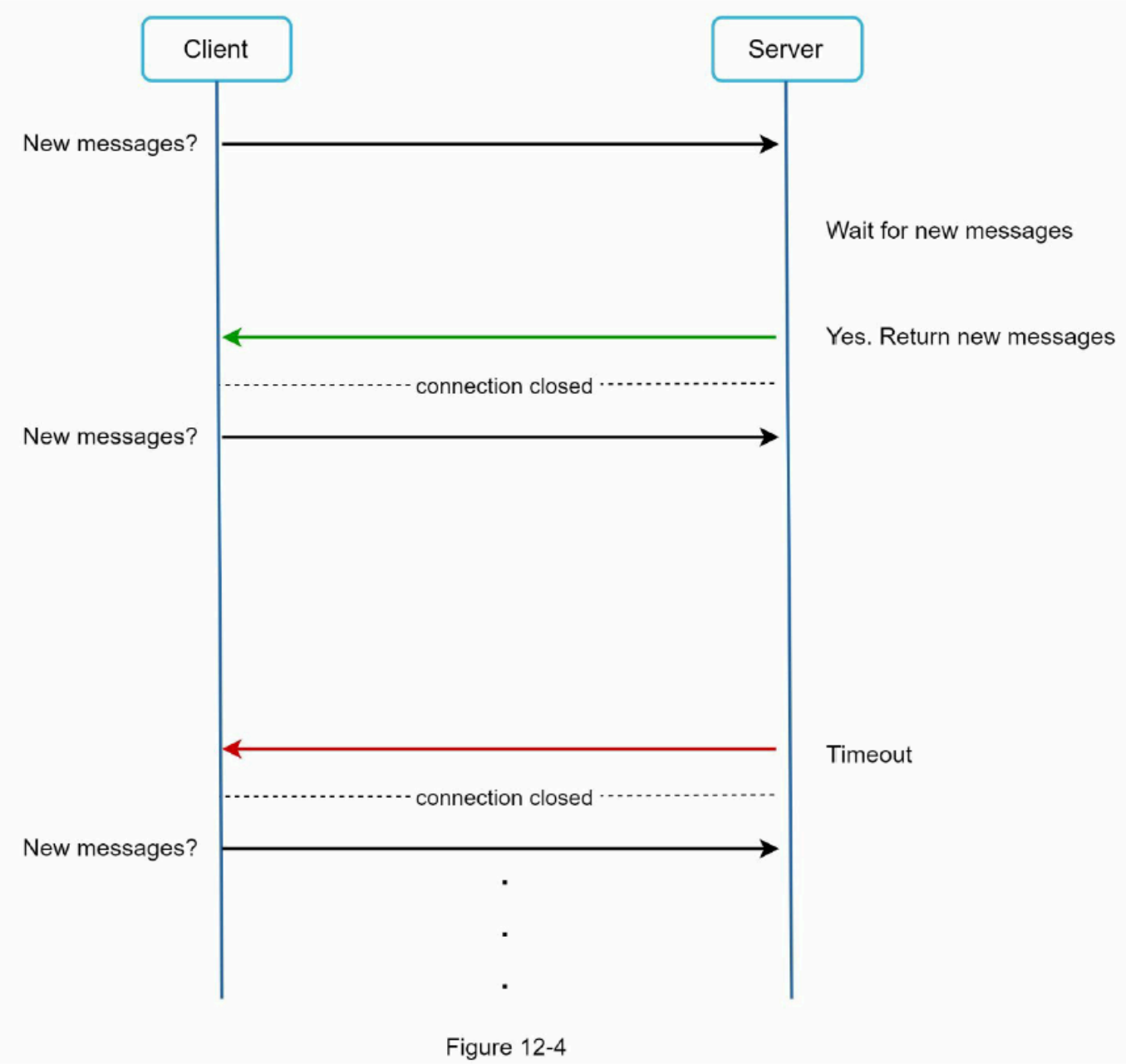


Figure 12-4

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. - similar to polling

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.

ChatGPT: A WebSocket is a communication protocol that provides full-duplex (two-way) communication channels over a single, long-lived connection. Unlike the HTTP request-response model, where the client has to continually send requests to receive updates (polling), WebSockets keep the connection open, allowing both the client and server to send data in real time without repeated requests

how long will the websocket be alive? what's the closing criteria?

Approach:

- 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. because they use port 80 or 443 which are also used by HTTP/HTTPS connections.

Here’s how it works:

1. **Establishing a WebSocket Connection**: The client (usually a browser) sends an HTTP request to the server with a special header indicating it wants to establish a WebSocket connection. If the server supports WebSockets, it responds with an acknowledgment, and the connection upgrades to a WebSocket. Once established, the connection remains open until it’s explicitly closed by either the client or the server.
2. **Full-Duplex Communication**: After the connection is open, both the client and the server can send messages to each other at any time. This two-way, event-driven communication makes WebSockets very efficient for real-time applications, like live chat apps, multiplayer games, financial tickers, or collaborative platforms (e.g., document editing).
3. **Data Format**: WebSocket messages are typically lightweight and can contain text (often in JSON format) or binary data, making them versatile for many kinds of real-time data.
4. **Closing the Connection**: The WebSocket connection can be closed by either the client or server. This closing procedure is also more graceful than HTTP, as it allows both sides to complete any remaining messages before shutting down.

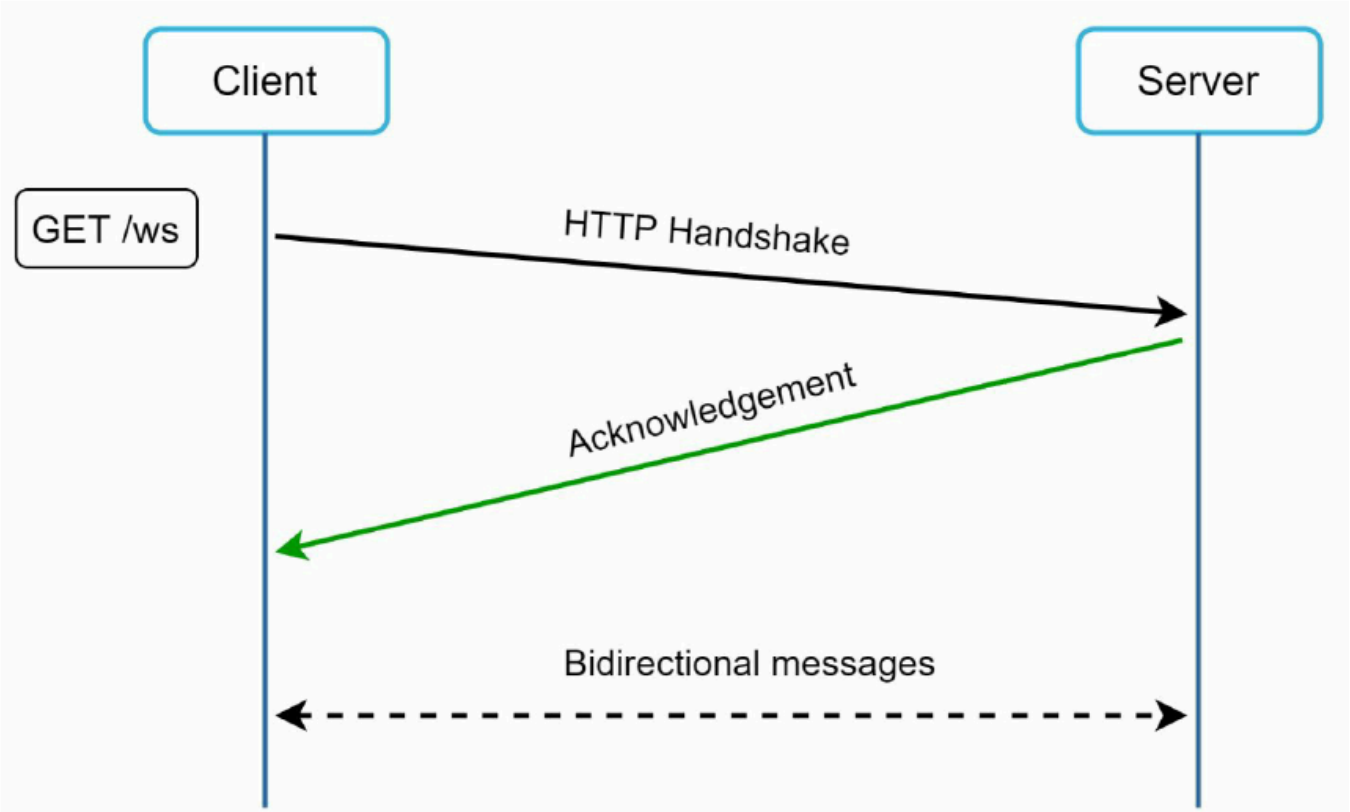


Figure 12-5

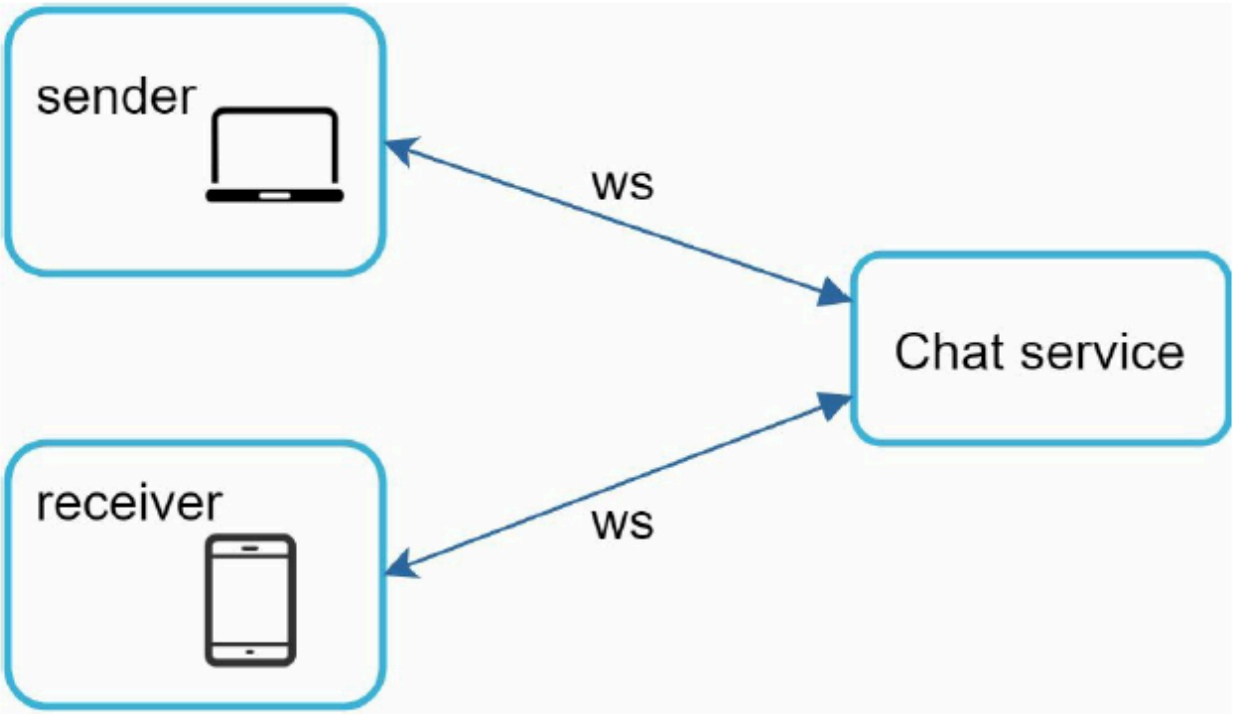


Figure 12-6

- WebSocket is bidirectional: can be also used for sending. By using WebSocket for both sending and receiving, it simplifies the design and makes implementation on both client

and server more straightforward.
- WebSocket connections are persistent: efficient connection management is critical on the server-side

WebSockets are ideal for applications **needing real-time communication between a client and server**, where frequent updates are necessary without the overhead of repeated HTTP requests.

High level design

- WebSocket was chosen as the main communication protocol between the client and server for its bidirectional communication, everything else does not have to be WebSocket.
- most features (sign up, login, user profile, etc) of a chat application could use the traditional request/response method over HTTP

3 major categories of chat systems:

- stateless services
- stateful services, and
- third-party integration

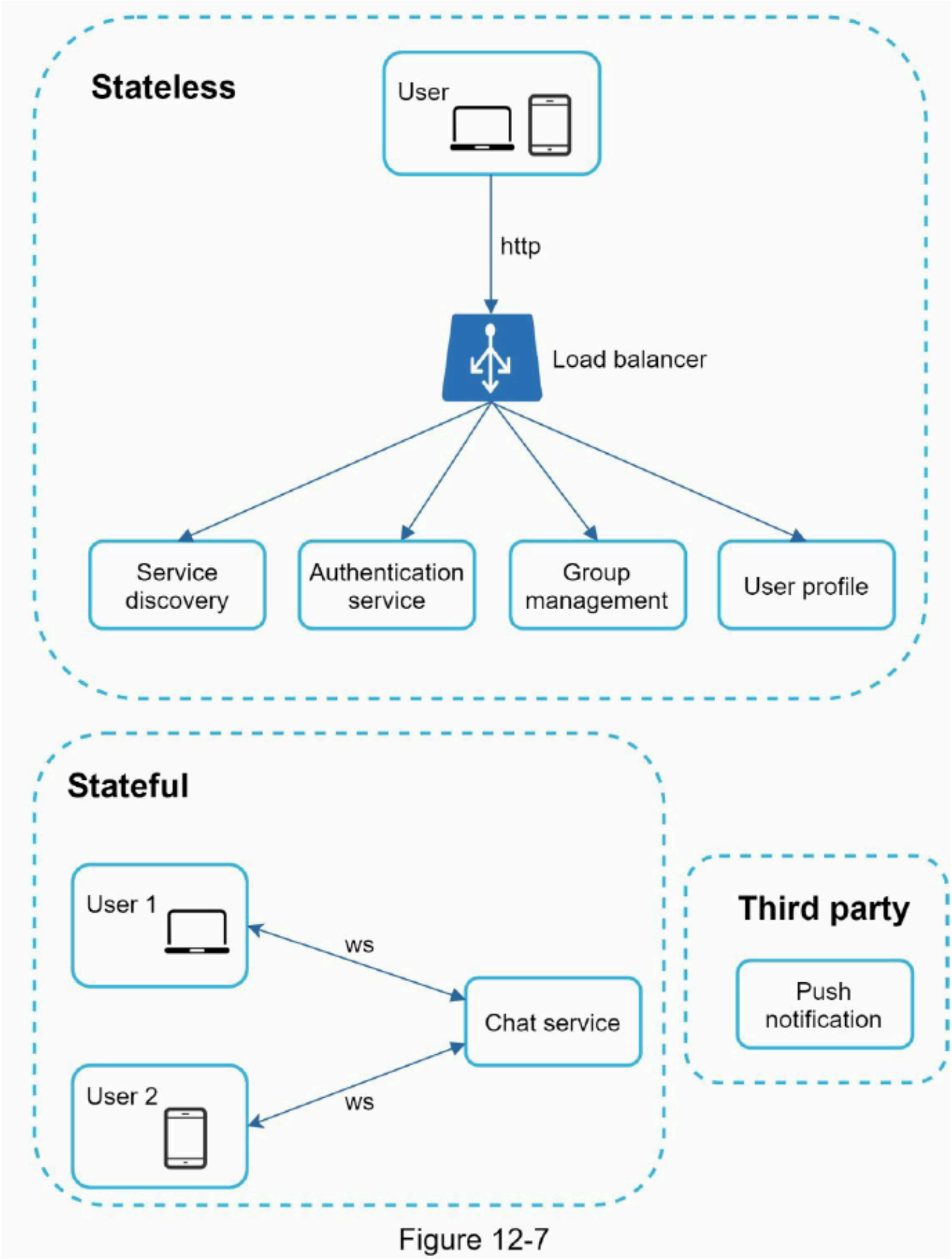


Figure 12-7

Stateless Services

- traditional **public-facing** request/response services

- used to manage the login, signup, user profile, etc (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.
- These services can be monolithic or individual microservices. No need to build many of these by ourselves, there are services in the market that can be integrated easily
- Will dive deep in 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.
- each client maintains a persistent network connection to a chat server.
- service discovery coordinates closely with the chat service to avoid server overloading

Third-party integration

- push notification is the most important third-party integration: 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.
- 1 M users * 10K memory for each user connection = 10G
- Though it can be fit into one server and we can start with single server design, we should consider scalable design and let interviewer know

combining the considerations above for a high-level design:

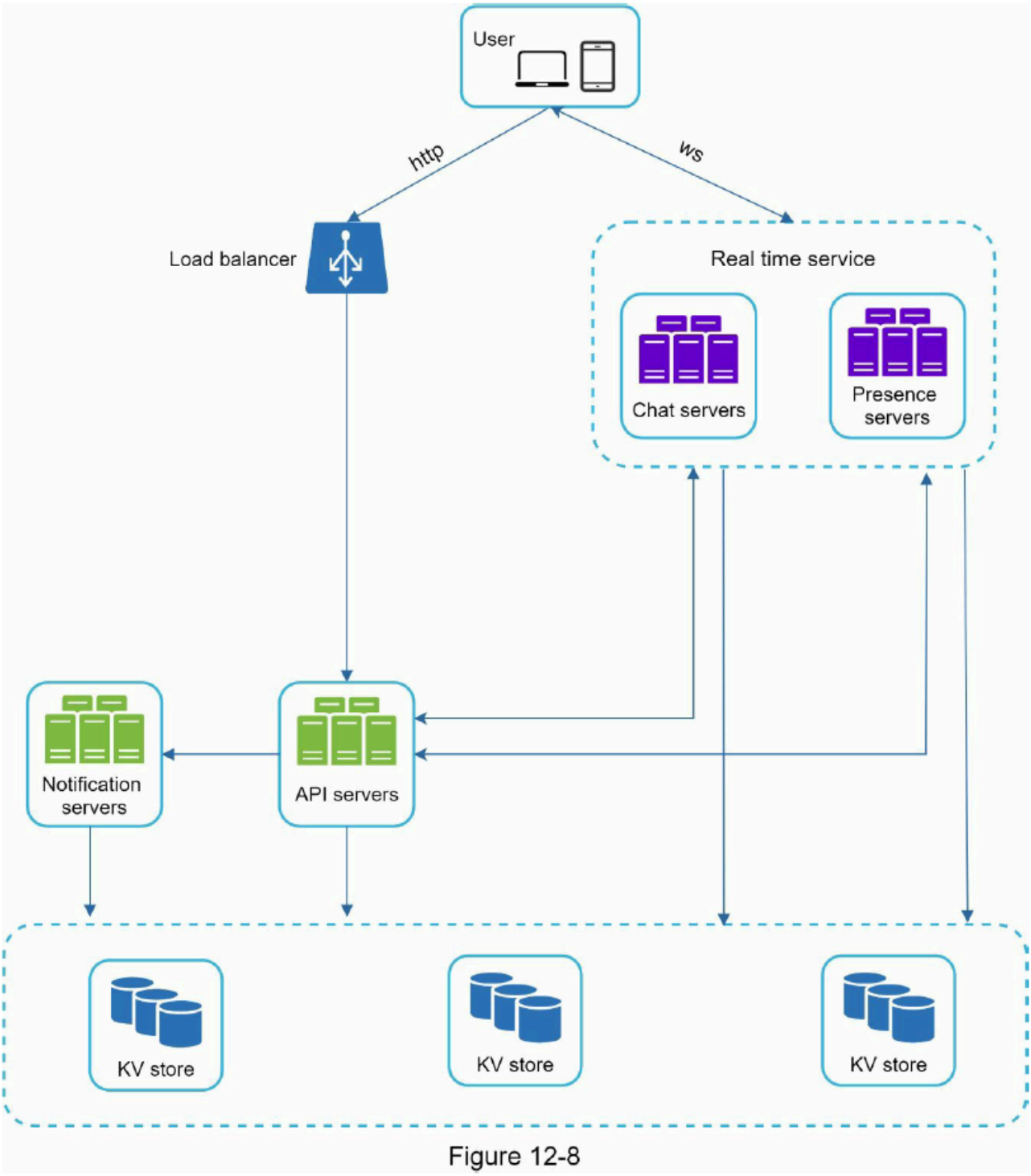


Figure 12-8

- The client maintains **a persistent WebSocket connection to a chat server for real-time messaging**
- 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.

- key-value store is used to store chat history. When an offline user comes online, she will see all her previous chat history.

Storage

The right type of database to use: relational databases or NoSQL databases?

T**wo types of data exist in a typical chat system**.

- generic data, such as user profile, setting, user friends list. These data are stored in robust and reliable relational databases.
- unique to chat systems: chat history data. important to understand the read/write pattern.
  - data is enormous for chat systems. Facebook messenger and Whatsapp process 60 billion messages a day.
  - Only recent chats are accessed frequently
  - Users might use features that require random access of 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

Recommending key-value stores:

- Key-value stores allow easy horizontal scaling.
- Key-value stores provide very low latency to access data.
- Relational databases do not handle long tail [3] 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, and Discord uses Cassandra

Data Models

The most important data is message data

1 on 1 chat message table

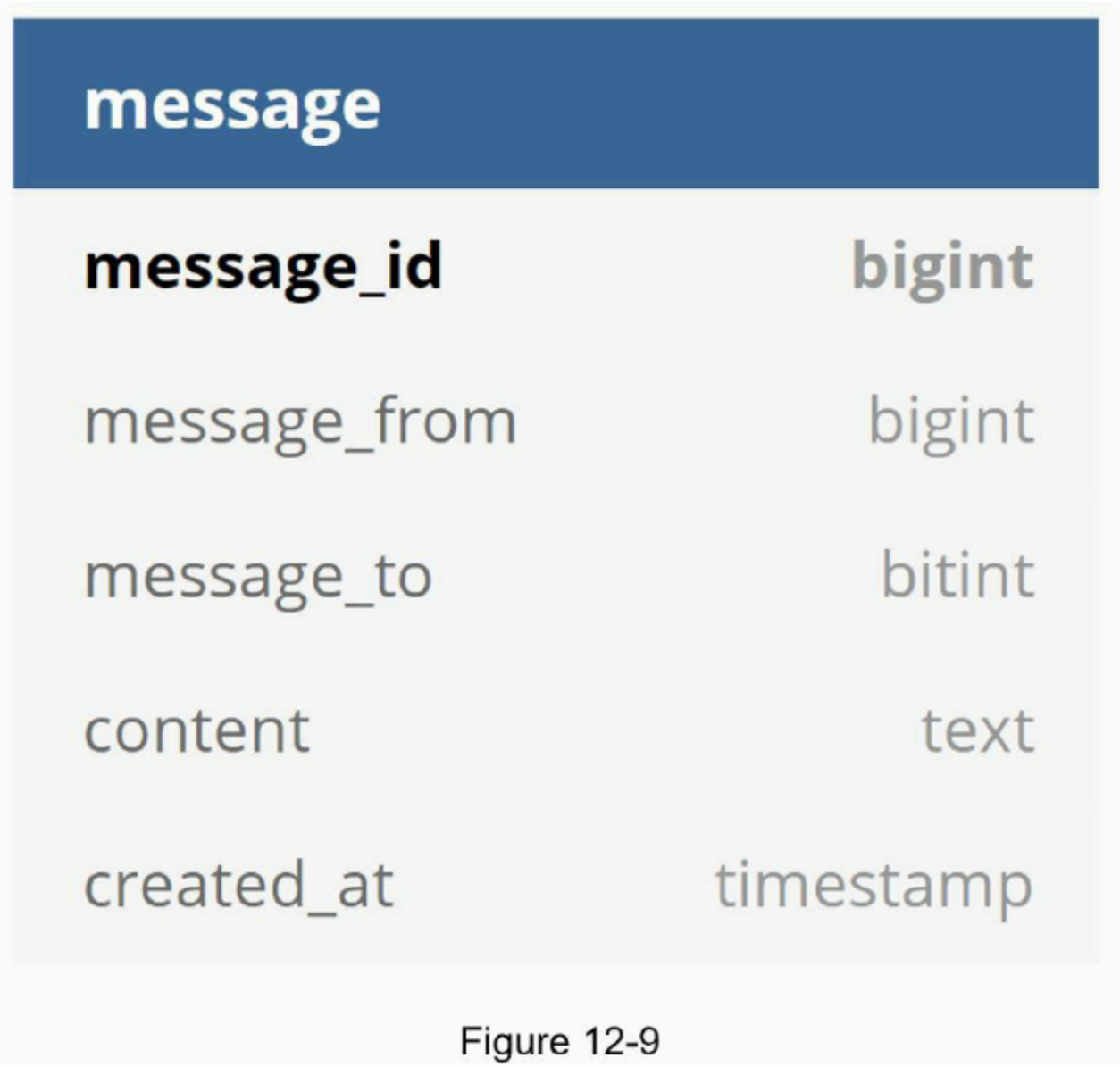


Figure 12-9

- primary key: message_id

Group chat message table

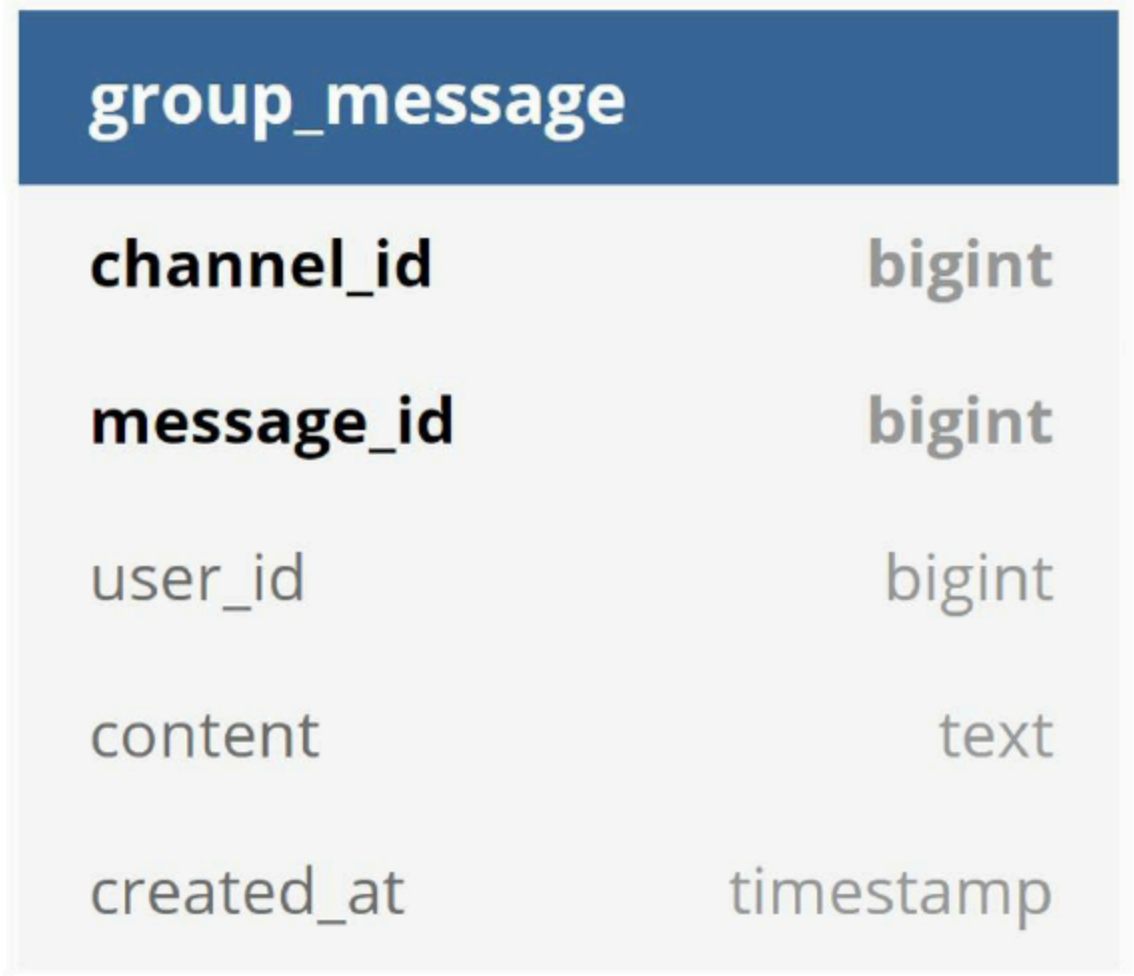


Figure 12-10

- composite primary key: (channel_id, message_id).
- partition key: channel_id. Because all queries in a group chat operate in a channel

Message ID

Requirements: Message_id carries the responsibility of ensuring the **order** of messages:

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

Solution:

- “auto_increment” keyword in MySql? - NoSQL databases usually do not provide such a feature.
- use a global 64-bit sequence number generator like Snowflake (Chapter 7)
- use local sequence number generator for group: maintaining message sequence within one-on-one channel or a group channel is sufficient. This approach is easier to implement

in comparison to the global ID implementation

Design Deep Dive

service discovery, messaging flows, and online/offline indicators worth deeper exploration.

Service discovery

- ***Service discovery primary role**: recommend the best chat server for a client based on the criteria like geographical location, server capacity, etc
- **Apache Zookeeper** 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.

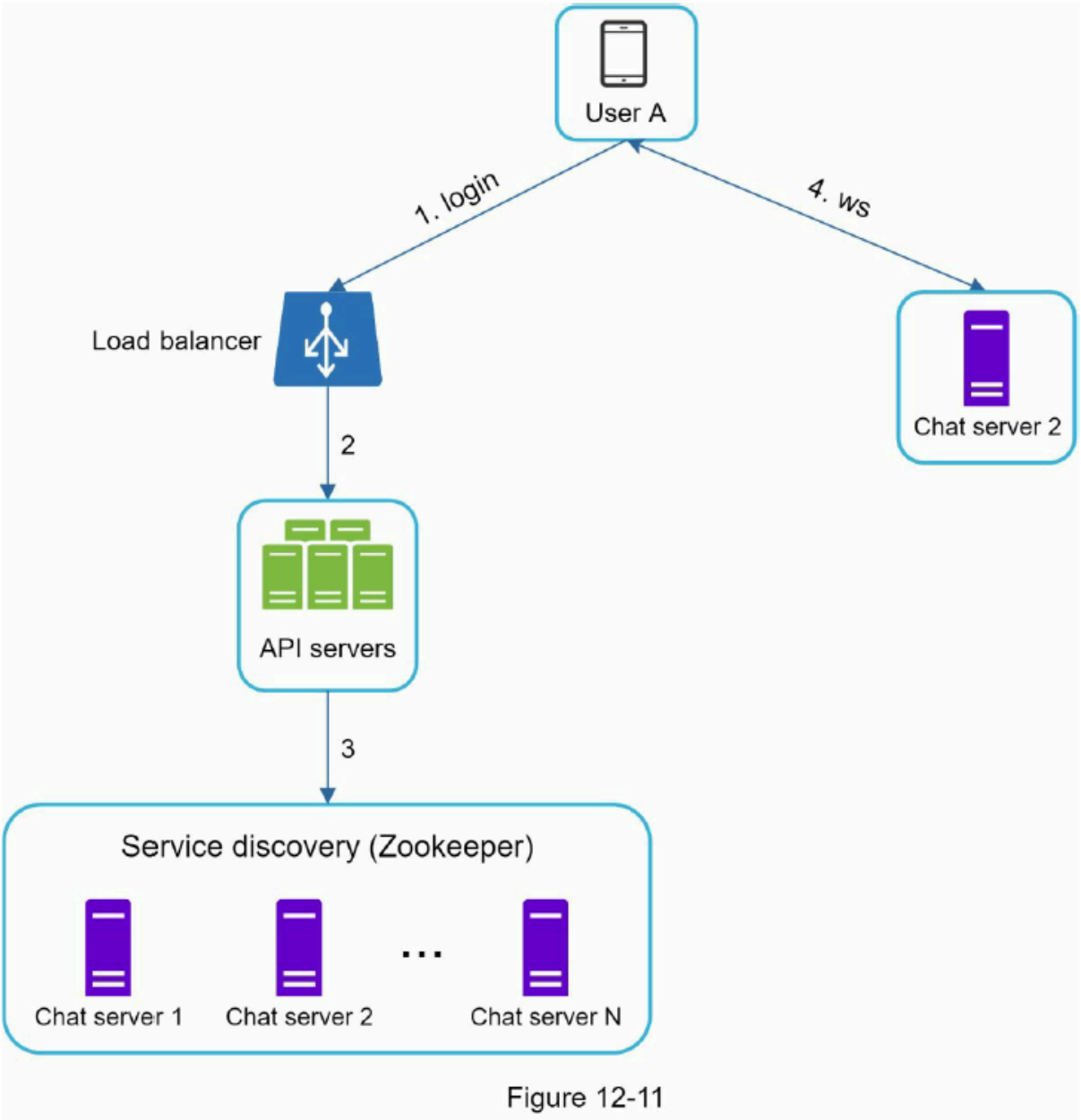


Figure 12-11

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

Explore 1 on 1 chat flow, message synchronization across multiple devices and group chat flow.

1 on 1 chat flow

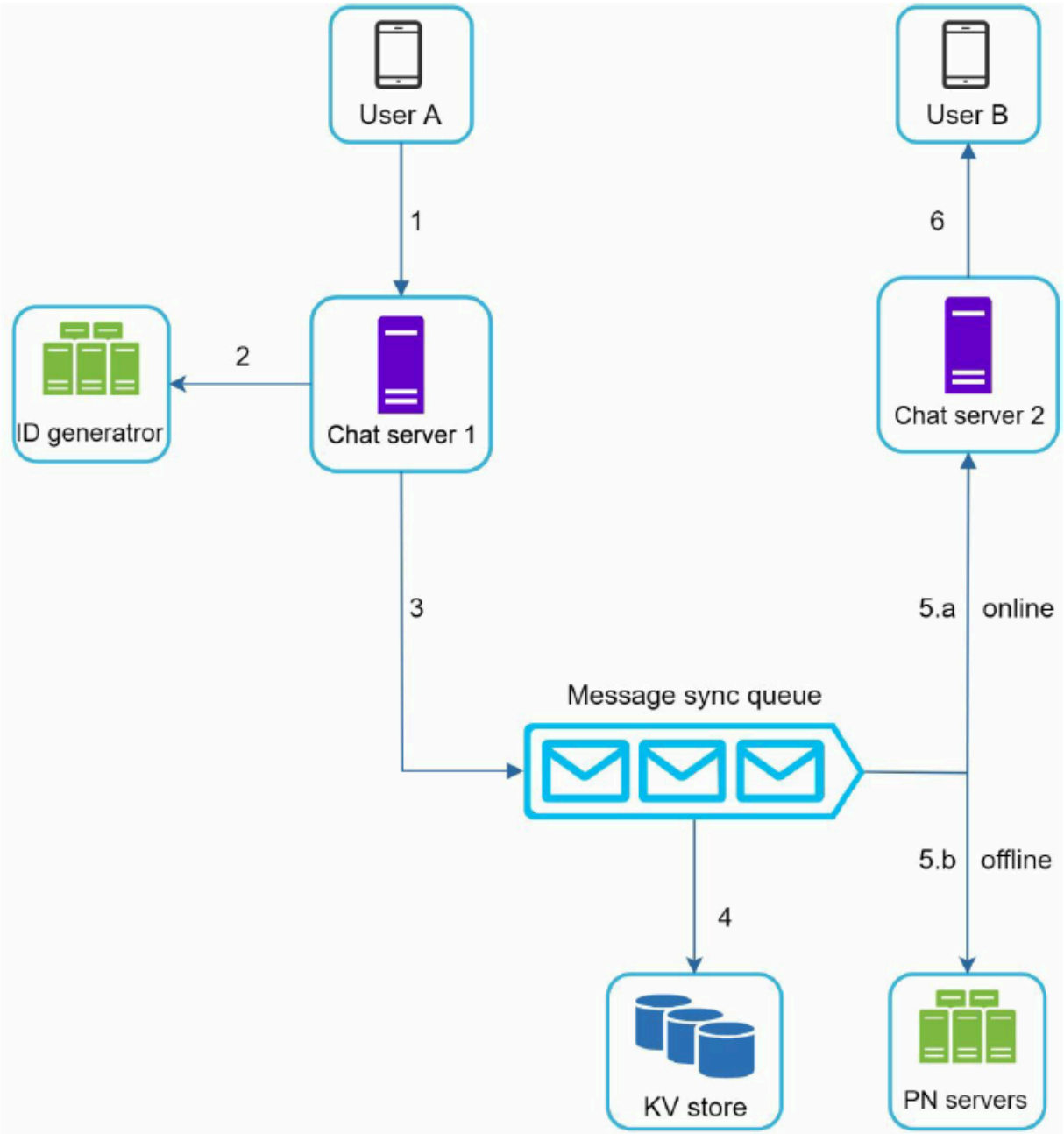


Figure 12-12

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

Message synchronization across multiple devices

deal with the msg sync problem for multiple devices for the same user.

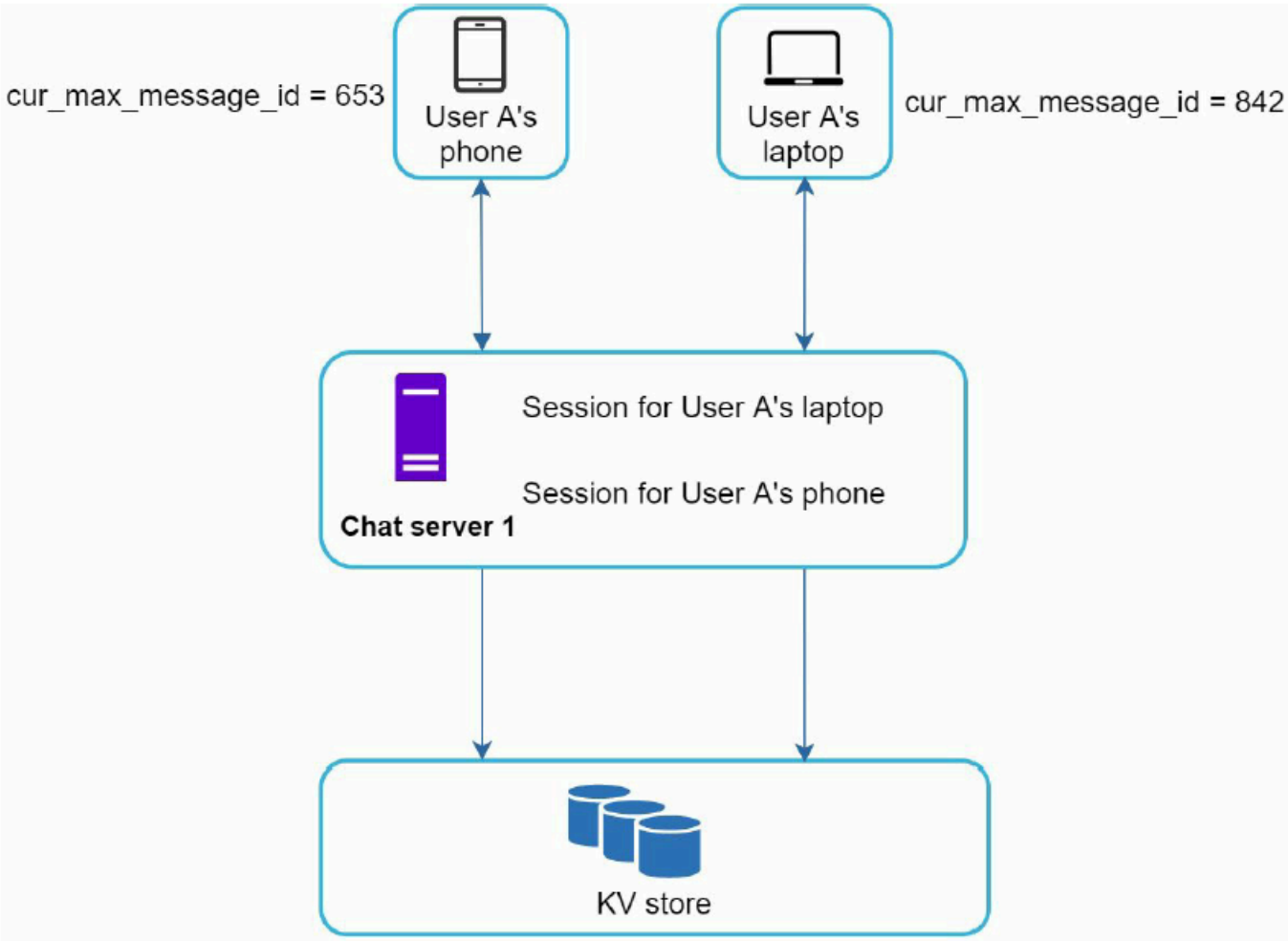


Figure 12-13

- When User A logs in to the chat app with her phone, it establishes a WebSocket connection with Chat server 1.
- there is a connection between the laptop and the **same** Chat server 1
- Each device maintains a cur_max_message_id, which keeps track of the latest message ID on the device.
- 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 to deal with the situation when a user send to herself?**
- **How many/recent msg are loaded into the other device?**

Small group chat flow

User A sends a message in a group chat:

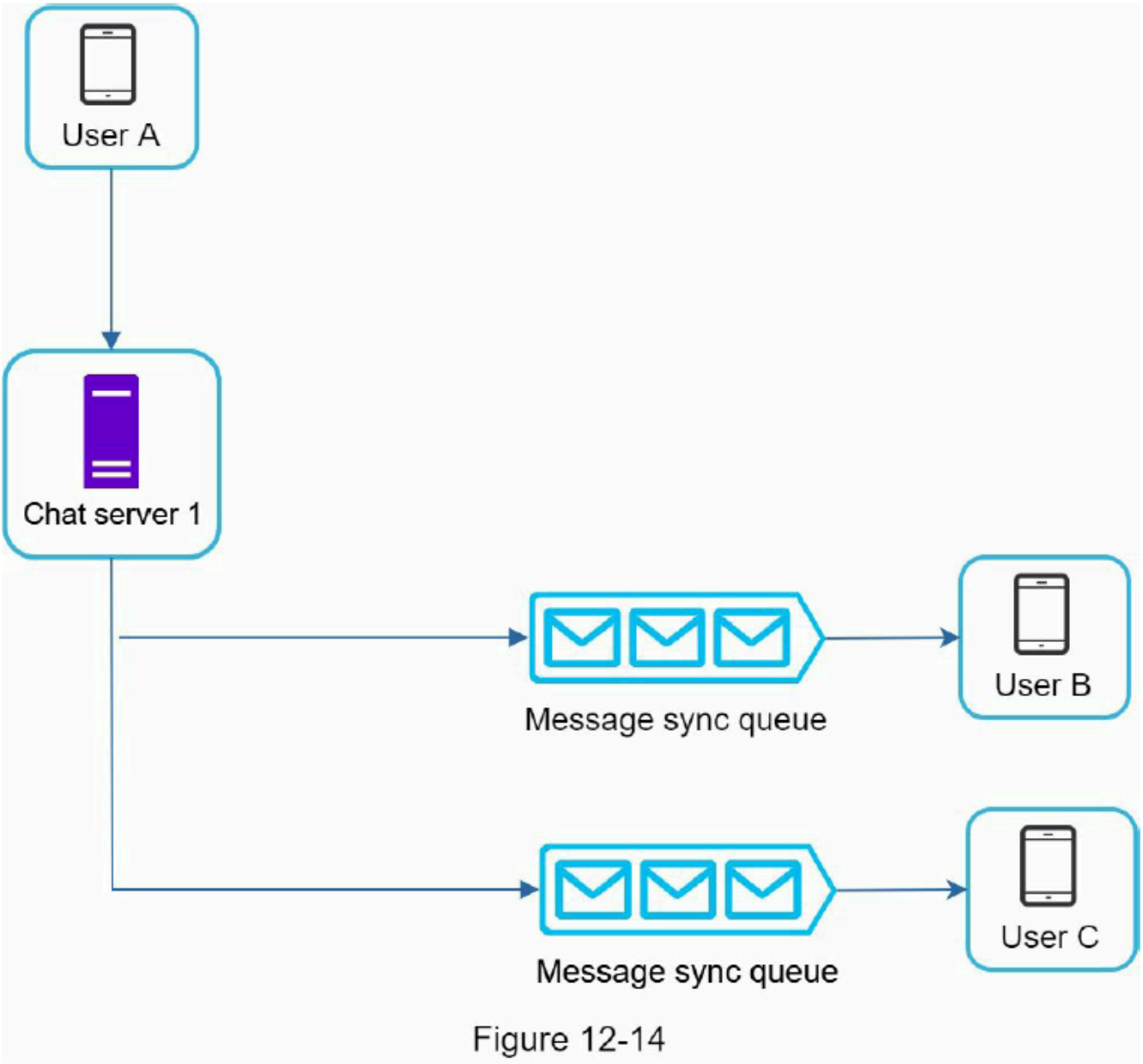


Figure 12-14

- The message from User A is copied to each group member's message sync queue (inbox for a recipient)
- This design choice is good for small group chat
  - 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. However, groups with a lot of users, storing a message copy for each member is not acceptable

- **What's the approach for large group?**

A recipient can receive messages from multiple users:

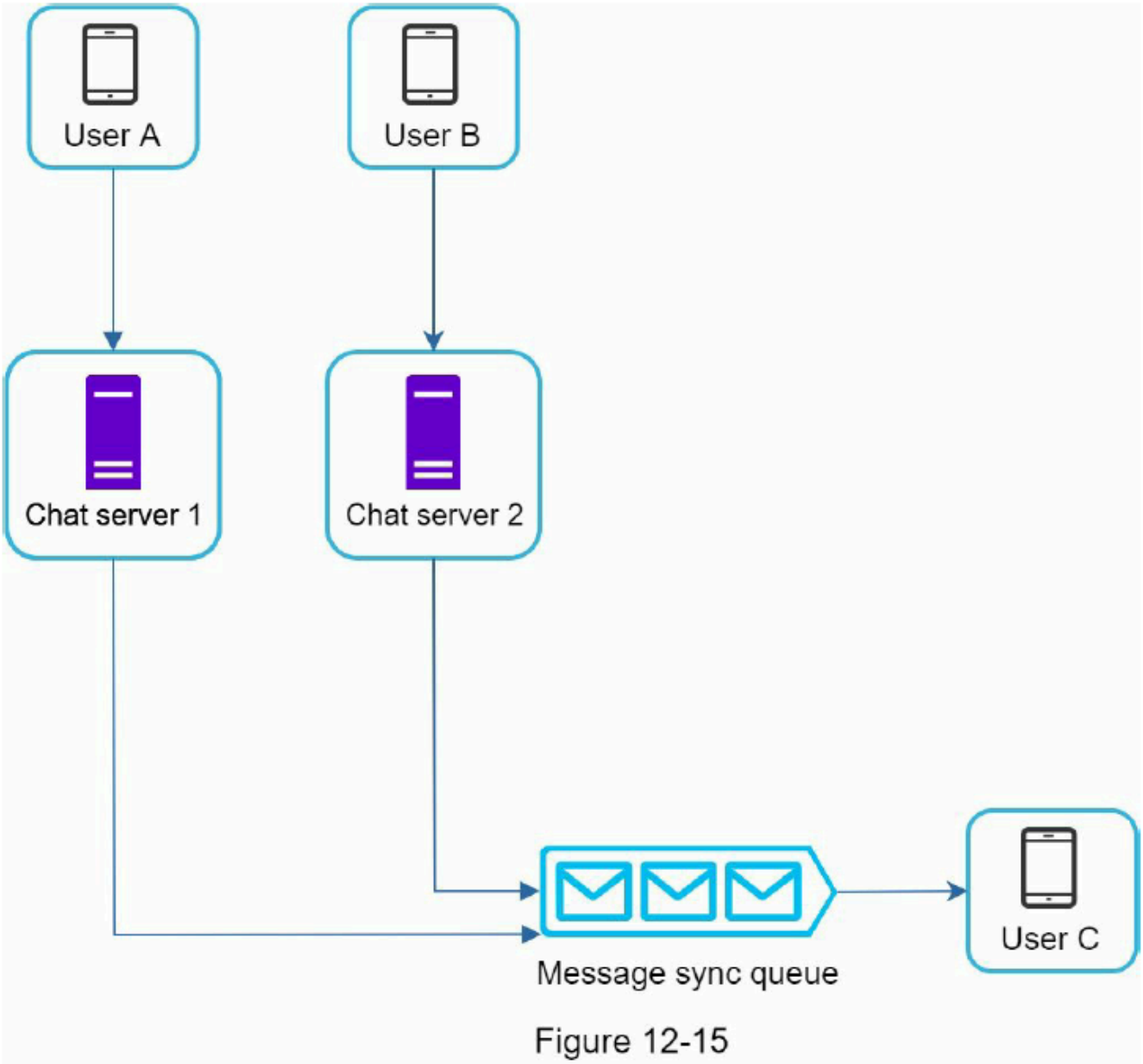


Figure 12-15

Each recipient has an inbox (message sync queue) which contains messages from different senders.

Online presence

- Usually, you can see a green dot next to a user's profile picture or username
- presence servers are responsible for managing online status and communicating with clients through WebSocket
- A few flows that will trigger online status change: user login, user logout, user disconnection,

User login

- user login via Service Discovery.
- 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.

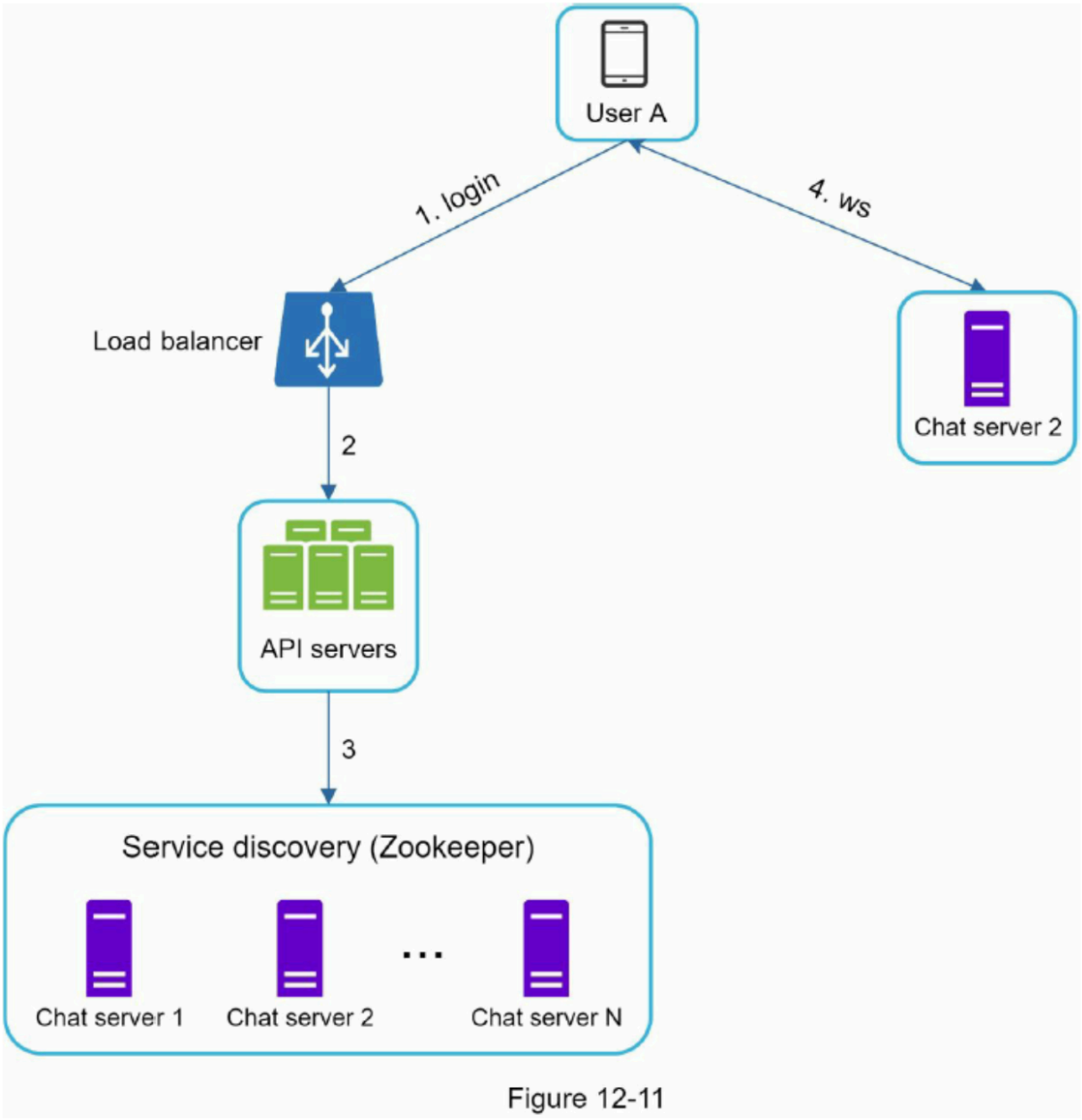


Figure 12-11

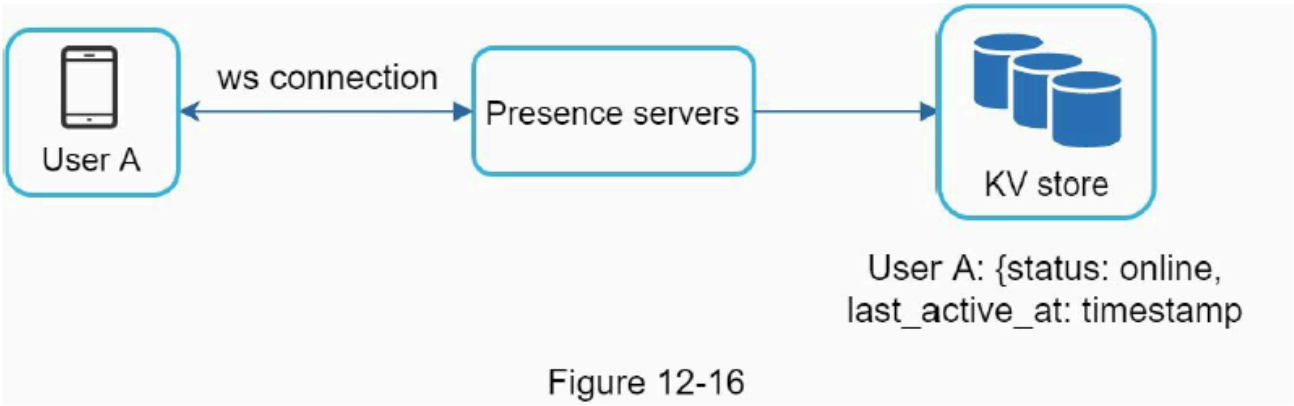


Figure 12-16

User logout

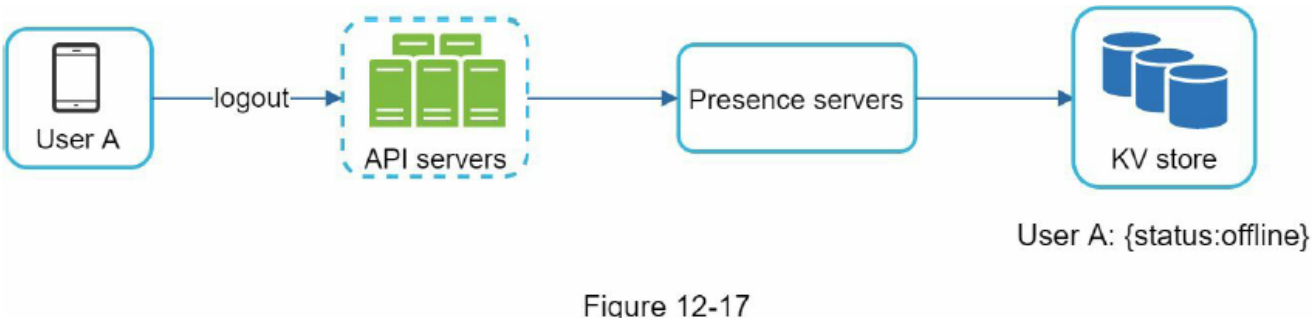


Figure 12-17

User disconnection

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

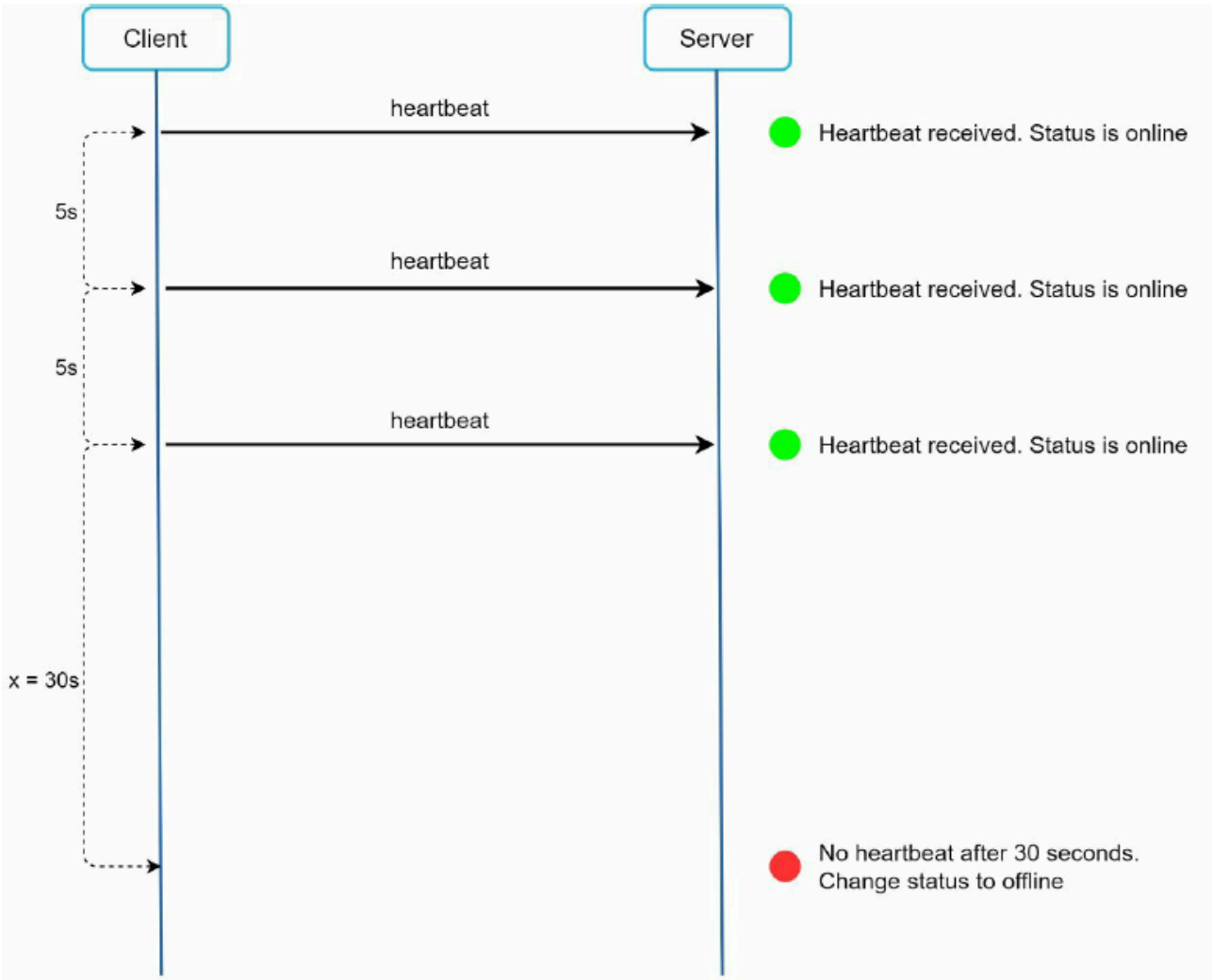


Figure 12-18

Online status fanout

How do user A's friends know about the status changes?

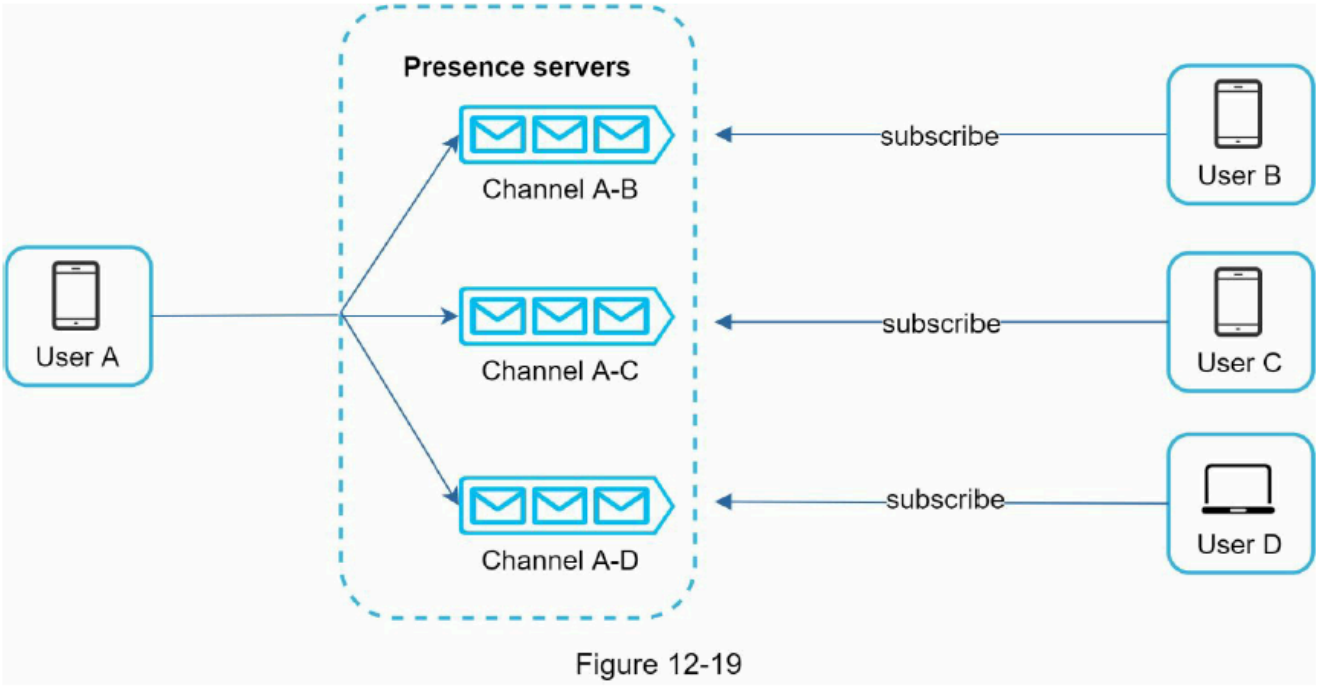


Figure 12-19

How it works:

- Presence servers use a publish-subscribe model: each friend pair maintains a channel
- When User A's online status changes, it publishes the event to all channels. Those channels are subscribed by the receivers
- The communication between clients and servers is through real-time WebSocket.

small vs large group:

- The above design is effective for a small user group.
- Performance bottleneck for large group: a possible solution is to fetch online status only when a user enters a group or manually refreshes the friend list.

Wrap up

Additional talking points:

- Extend the chat app to support media files such as voice, photos and videos.
  - Large size: storage, compression, thumbnails
  - cache
  - database
- End-to-end encryption for msgs: only the sender and the recipient can read messages.
- Caching messages on the client-side: is effective to reduce the data transfer between the client and server.
- Improve load time. e.g., geographically distributed network to cache users' data, channels, etc.

- Error handling
  - The chat server error. 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.
- Voice call and video call?
  - 1 on 1
  - group
- Message search
- Data retention policy
- Monitoring and logging