Whatsapp is a chat application that provides instant messaging services to its users. It is one of the most used mobile applications on the planet connecting over 2 billion users in 180+ countries.

**REQUIREMENTS**

**FUNCTIONAL REQUIREMENTS**

1. System should support one to one chat.
2. System should support group chat.
3. Should support sending video, images.

**NON FUNCTIONAL REQUIREMENTS**

1. Low latency
2. High Available
3. System should be scalable and efficient.

**Optional requirements**

1. Sent, delivered and read receipt of messages
2. Show last seen of the user
3. Push notification

**ESTIMATION AND CAPACITY CONSTRAINTS**

Daily active user : 1billion

Each user daily send message : 20 messages

Daily send messages : 1B\*20 = 20 billion messages per day

Avg 1 msg storage size = 100byte

Daily message storage size = 20 \* 100 = 2TB/day

Daily media send = 2

1 media avg size = 100KB

Daily media storage size = 2\*100\*1B = 200TB/day

Total storage required per day = 202TB/day

**Data Model Desgin**

**User(**userId, name, email)

**UserGroup (**Id, userId, GroupId)

**Group (**GroupId, groupName, MessageId)

**userChat (**id, userId, ChatId)

**Messages (**Id, userId, chatId, groupId, type, content, timestamp)

**Chats** (Id, messageId)

We have the following tables:

**users**

This table will contain a user's information such as name, phoneNumber, and other details.

**messages**

As the name suggests, this table will store messages with properties such as type (text, image, video, etc.), content, and timestamps for message delivery. The message will also have a corresponding chatID or groupID.

**chats**

This table basically represents a private chat between two users and can contain multiple messages.

**users\_chats**

This table maps users and chats as multiple users can have multiple chats (N:M relationship) and vice versa.

**groups**

This table represents a group between multiple users.

**users\_groups**

This table maps users and groups as multiple users can be a part of multiple groups (N:M relationship) and vice versa.

### **What kind of database should we use?**

While our data model seems quite relational, we don't necessarily need to store everything in a single database, as this can limit our scalability and quickly become a bottleneck.

We will split the data between different services each having ownership over a particular table. Then we can use a relational database such as [PostgreSQL](https://www.postgresql.org) or a distributed NoSQL database such as [Apache Cassandra](https://cassandra.apache.org/_/index.html) for our use case.

**API Design**

1. Get all chats or groups : getAll(userId)
2. Get messages : getMessage(userId, chatId)
3. SendMessage : sendMessgae(userId, targetUserId, message)
4. Join group : joinGroup(userId, groupId)
5. Leave group : leaveGroup(userId, groupId)

**High Level Design**

**Services Required:** userService, chatService, Presence Service, Media Service, Notification Service

**Real Time Messaging :**

**Pull model**

The client can periodically send an HTTP request to servers to check if there are any new messages. This can be achieved via something like [Long polling](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events#long-polling).

**Push model**

The client opens a long-lived connection with the server and once new data is available it will be pushed to the client. We can use [WebSockets](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events#websockets) or [Server-Sent Events (SSE)](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events#server-sent-events-sse) for this.

### **Last seen**

To implement the last seen functionality, we can use a [heartbeat](https://en.wikipedia.org/wiki/Heartbeat_(computing)) mechanism, where the client can periodically ping the servers indicating its liveness. Since this needs to be as low overhead as possible, we can store the last active timestamp in the cache as follows:

| **Key** | **Value** |
| --- | --- |
| User A | 2022-07-01T14:32:50 |
| User B | 2022-07-05T05:10:35 |
| User C | 2022-07-10T04:33:25 |

### **Notifications**

Once a message is sent in a chat or a group, we will first check if the recipient is active or not, we can get this information by taking the user's active connection and last seen into consideration.

If the recipient is not active, the chat service will add an event to a [message queue](https://karanpratapsingh.com/courses/system-design/message-queues) with additional metadata such as the client's device platform which will be used to route the notification to the correct platform later on.

### **Read receipts**

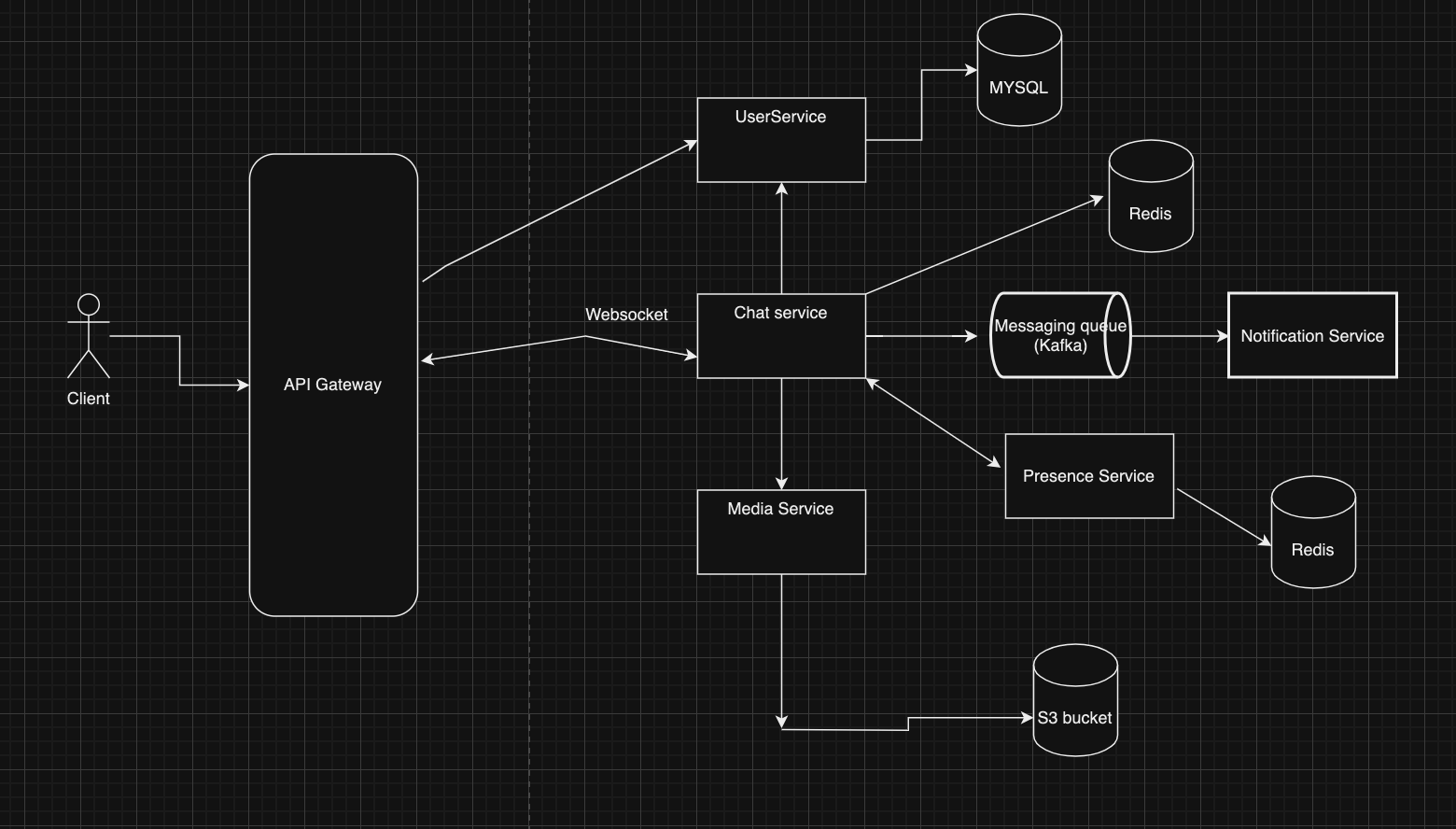
Handling read receipts can be tricky, for this use case we can wait for some sort of [Acknowledgment (ACK)](https://en.wikipedia.org/wiki/Acknowledgement_(data_networks)) from the client to determine if the message was delivered and update the corresponding deliveredAt field. Similarly, we will mark message the message seen once the user opens the chat and update the corresponding seenAt timestamp field.

In the WhatsApp system, a combination of databases and data storage solutions is typically used to serve different purposes:

1. **Key-Value Stores:** Key-value stores like Redis or Amazon DynamoDB are commonly used to store real-time message data and metadata. These databases are highly scalable and offer low-latency access, making them suitable for managing chat conversations and message history.
   * Real-Time Messaging: Key-value stores can be used to store real-time message data, including the sender, receiver, timestamp, and message content.
   * Message Status: To track the delivery and read status of messages, key-value stores can efficiently store metadata associated with each message.
2. **Chats and Conversations:** For managing conversations and chat history, a combination of relational databases and NoSQL databases may be used.
   * Relational Databases: User data, user profiles, and other structured data can be stored in a relational database like PostgreSQL or MySQL.
   * NoSQL Databases: NoSQL databases like MongoDB can be used to store chat conversation history, media messages, and other semi-structured data.
3. **Media Storage:** To handle media messages (images, videos, etc.), a separate storage solution is required. Cloud-based object storage like Amazon S3 or Google Cloud Storage is often used for efficient and scalable media storage.
4. **Caching Solutions:** Caching is crucial to improve the performance of the messaging system. In-memory caching solutions like Redis can be used to cache frequently accessed data, such as user profiles and chat metadata, reducing database load and response times.
5. **Message Queues:** Message queues like Apache Kafka or RabbitMQ can be used to handle asynchronous communication and data processing in the system. They are useful for managing real-time event streams and ensuring data consistency across different components.

The choice of the database for WhatsApp's system design depends on the specific requirements, read and write patterns, and the scale of the platform. Using a combination of databases allows WhatsApp to achieve a balance between data consistency, scalability, high availability, and low-latency access, ensuring a seamless messaging experience for its vast user base.

1. **Chat Service:** The Chat Service is responsible for storing real-time message data and handling chat conversations between users.
   * **Database Choice:** Key-Value Store (e.g., Redis) or a combination of Relational and NoSQL Databases.
2. Explanation: A Key-Value Store like Redis is often used to handle real-time message data due to its high performance and low-latency access. It can efficiently manage message metadata and status (e.g., delivered, read). For storing chat conversations and message history, a combination of Relational Databases (e.g., PostgreSQL or MySQL) and NoSQL Databases (e.g., MongoDB) may be used. Relational databases are suitable for structured user data and metadata, while NoSQL databases can handle semi-structured message data and media messages.
3. **Presence Service:** The Presence Service is responsible for tracking the online/offline status of users and their availability for real-time messaging.
   * **Database Choice:** Key-Value Store (e.g., Redis) or a separate User Database.
4. Explanation: A Key-Value Store like Redis can be used to maintain user presence data efficiently. Alternatively, a separate User Database (e.g., PostgreSQL or MySQL) can store user profiles and status information.
5. **Notification Service:** The Notification Service is responsible for sending push notifications to users for new messages and other activities.
   * **Database Choice:** NoSQL Database or Message Queues.
6. Explanation: The Notification Service may not require a traditional database. Instead, it can utilize NoSQL databases for storing temporary message status data or message queues (e.g., Apache Kafka or RabbitMQ) to manage notification delivery asynchronously.
7. **UserService:** The UserService is responsible for managing user data, authentication, and user-related operations.
   * **Database Choice:** Relational Database (e.g., PostgreSQL or MySQL).
8. Explanation: The UserService often deals with structured user data, user profiles, and authentication information. A relational database is a common choice to handle these requirements.
9. **Media Service:** The Media Service is responsible for storing and serving media messages (e.g., images, videos) sent by users.
   * **Database Choice:** Cloud-Based Object Storage (e.g., Amazon S3 or Google Cloud Storage).
10. Explanation: Cloud-based object storage solutions are preferred for efficient and scalable media storage. They can handle media files and serve them quickly to users, providing a seamless media sharing experience.



Let us identify and resolve bottlenecks such as single points of failure in our design:

* "What if one of our services crashes?"
* "How will we distribute our traffic between our components?"
* "How can we reduce the load on our database?"
* "How to improve the availability of our cache?"
* "Wouldn't API Gateway be a single point of failure?"
* "How can we make our notification system more robust?"
* "How can we reduce media storage costs"?
* "Does chat service has too much responsibility?"

To make our system more resilient we can do the following:

* Running multiple instances of each of our services.
* Introducing [load balancers](https://karanpratapsingh.com/courses/system-design/load-balancing) between clients, servers, databases, and cache servers.
* Using multiple read replicas for our databases.
* Multiple instances and replicas for our distributed cache.
* We can have a standby replica of our API Gateway.
* Exactly once delivery and message ordering is challenging in a distributed system, we can use a dedicated [message broker](https://karanpratapsingh.com/courses/system-design/message-brokers) such as [Apache Kafka](https://kafka.apache.org) or [NATS](https://nats.io) to make our notification system more robust.
* We can add media processing and compression capabilities to the media service to compress large files similar to WhatsApp which will save a lot of storage space and reduce cost.
* We can create a group service separate from the chat service to further decouple our services.

References

https://github.com/karanpratapsingh/system-design#whatsapp