Design WhatsApp Messenger: System Design Interview Question

What is Whatsapp?

WhatsApp is a social messaging platform that enables users to send messages to one another. It has become an integral part of many people's daily lives, but how does it work? What are the principles behind its design and functioning?

In this blog, we will answer these fundamental questions and explore the basics of how WhatsApp works. We will also discuss the generic architecture of WhatsApp, which can serve as a foundation for designing other chat applications.

Key Requirements

WhatsApp is a highly scalable system that is accessed frequently by a large number of users around the world, so it is essential that it is designed efficiently in order to remain reliable and operational. Therefore, it is important to identify the key requirements of the system.

Some of the basic requirements for WhatsApp include:

- The ability to support one-on-one conversations.
- The ability to show last seen and message acknowledgement (sent, delivered, and read).
- Media support (images/videos) and end-to-end encryption.

Let’s figure out the capacity estimation of our required service.

Capacity Estimation

Our goal is to build a highly scalable platform that can support a large volume of traffic. To do this, we need to consider various factors, such as

the number of messages and users, peak traffic levels, and data storage requirements.

Let's assume the following:

- 10 billion messages are sent per day by 1 billion users.
- At peak traffic, there are 700,000 active users per second.
- At peak traffic, there are 40 million messages per second.
- On average, each message has 160 characters, resulting in 1.6 TB (10B * 160) of data per day.
- The system is expected to be in service for 10 years, requiring approximately 6 PB (10 * 1.6B * 365) of storage.
- The application will consist of multiple microservices, each performing a specific task. Let's assume that the latency for sending a message is 20 milliseconds, and that each server can handle 100 concurrent connections. Based on these assumptions, we would need a fleet of 8000 (40M * 20ms / 100) servers to support the chat service.

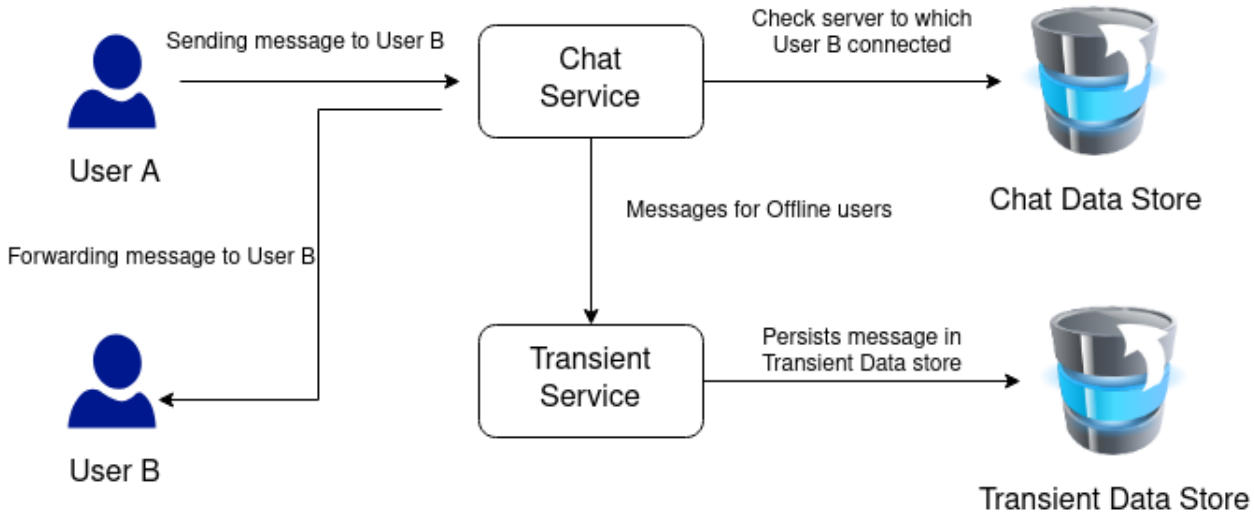
High Level Design

In the WhatsApp system, there are two primary services that form the core of the system: the chat service and the transient service. The **chat service** manages all traffic related to online messages sent by users. This includes handling incoming messages, sending messages to recipients, and maintaining the status of messages (such as whether they have been sent, delivered, or read).

The **transient service**, on the other hand, deals with traffic when the user is offline. This includes storing messages that are sent to a user while they are offline and delivering them once the user comes back online. It also handles other tasks related to offline users, such as managing the "last seen" status and message acknowledgement.

Here's a summary of how these two services work together to ensure that messages are delivered efficiently and reliably:

- When a user sends a message, the chat service is responsible for checking whether the recipient is online. If the recipient is online, the chat service delivers the message immediately.
- If the recipient is offline, the chat service passes the message to the transient service. The transient service stores the message in a separate storage until the recipient comes online.
- When the recipient comes online, the transient service delivers the stored message to the chat service, which then sends it to the recipient.



High Level API Design

The messaging system described will have two main APIs for sending and viewing messages. These APIs will use the REST (representational state transfer) architecture. REST is a popular choice for designing APIs because it is lightweight, easy to understand, and can be implemented using the standard HTTP protocol.

Sendmessage (fromUser, toUser, clientMetaData, message)

This API is used for sending messages from one user to another.

1. fromUser: The user who is sending the message.
2. toUser: The user who is receiving the message.
3. clientMetaData: Metadata about the client, such as information about

the device or platform being used.
4. message: The actual message being sent.

Conversation(userId, offset, messageCount, TimeStamp)

This API displays conversations in a thread, similar to the view you see when you open WhatsApp. To avoid fetching too many messages at once, the API retrieves a limited number of messages for a single user in each call. The offset and message count parameters help control how many messages are retrieved.

Parameters:

- userId: a unique identifier for the user
- offset: used to retrieve previous messages
- messageCount: the number of messages to be displayed
- TimeStamp: the last time the messages were updated

How features like last seen, single tick, and double tick work?

The acknowledgment service is responsible for implementing these features by continuously generating and checking acknowledgement responses. Based on the responses received, the service determines how to proceed.

Single Tick: When a message from User A reaches User B, the server sends an acknowledgement signal to confirm that the message has been sent.

Double Tick: After the server sends a message to User B through the appropriate connection, User B will send an acknowledgement back to the server to confirm receipt of the message. The server will then send another acknowledgement to User A, which will be displayed as a double tick.

Blue Tick: When User B reads the message, they will send another

acknowledgement to the server to confirm that they have read it. The server will then send another acknowledgement message to User A, which will be displayed as a blue tick.

Last Seen Feature: This feature relies on the **heartbeat mechanism**, which sends continuous messages to the server every 5 seconds. The server uses these messages to maintain a table of the last-seen status of various users, which any other user can easily retrieve to check their last-seen status.

Design of Key Features

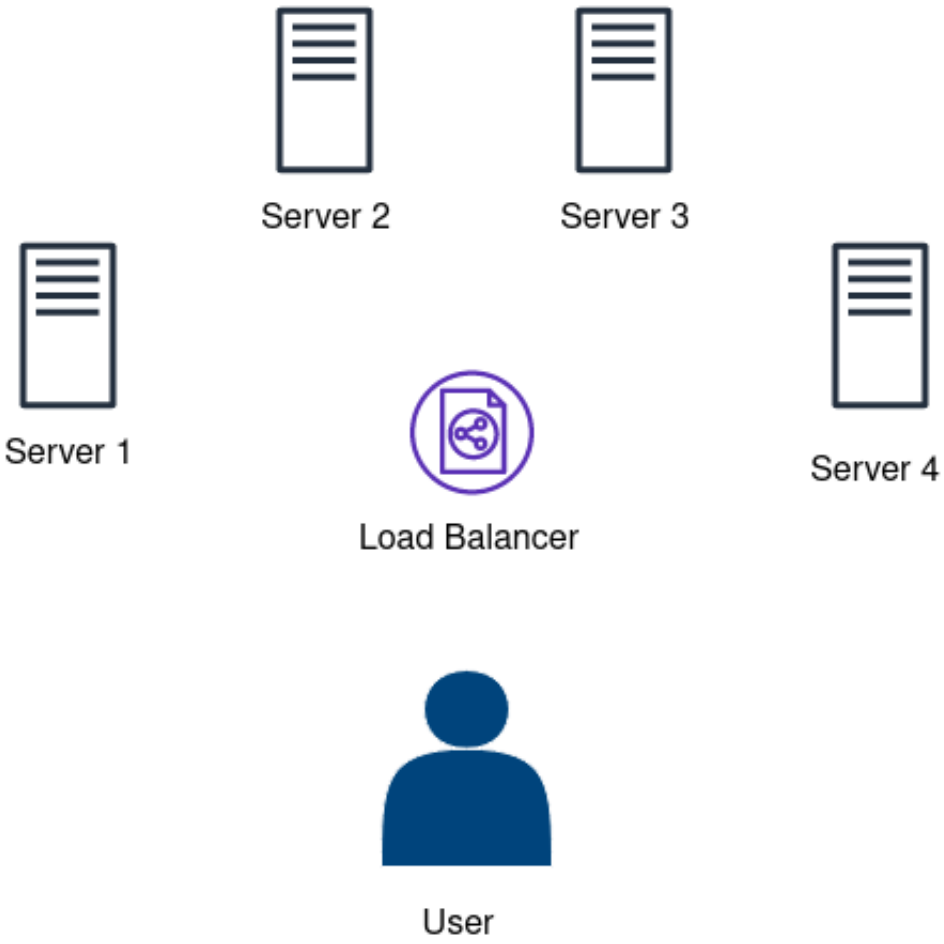
One-to-One Communication

This is an essential component of the chat service that allows one user to send messages to another user. Here's how it works:

Suppose Alice wants to send a message to Bob. The message is sent to the chat server that Alice is connected to. Alice receives an acknowledgement from the chat server that the message has been sent. The chat server then asks the data storage to fetch information about the chat server that Bob is connected to. The chat server of Alice then forwards the message to the chat server of Bob, and the message is delivered to Bob using a push mechanism. Bob then sends an acknowledgement back to the chat server of Alice, which informs Alice that the message has been delivered. If Bob reads the message again, a new acknowledgement is sent to Alice to confirm that the message has been read.

User Activity Status

This figure shows a mechanism for maintaining a connection between the client and the server. A connection is established between the server and the client using web sockets, which create a bidirectional connection. Heartbeats are sent through these connections to monitor the user's activity status.



End-to-End Encryption

End-to-End encryption is a feature that ensures that only the communicating users can read the messages. This is achieved through the use of public keys shared among all the users participating in the communication.

For example, suppose Alice and Bob are communicating with each other in a channel. Both Alice and Bob have each other's public keys, and they each have their own private key that is not shared. When Alice wants to send a message to Bob, she encrypts the message using Bob's public key. This message can only be decrypted using Bob's private key, which he keeps to himself. Similarly, Alice can only decrypt messages sent by Bob. In this way, only Alice and Bob can read each other's messages, and the server acts only as a mediator in the process.

Bottlenecks

All systems are vulnerable to failures, and it is important to have measures in place to handle such situations. In order to handle a large volume of traffic, the service must remain active and fault-tolerant to handle any bottlenecks that may arise. In our service, the chat and transient servers are essential components, and it is necessary to address any challenges that may arise in their operation.

- **Chat Server Failure:** The chat server is the core component of our system and is responsible for delivering messages to users who are online. It maintains connections with users, so if it fails, it will affect the entire architecture. There are two ways to handle the chat server's failure: transferring the TCP connections to another server, or allowing users to automatically initiate a new connection in case of a connection loss.
- **Transient Storage Failure:** The transient storage component is also prone to failures, which can affect the entire service. If this component fails, it can result in the loss of messages that were in transit to offline users. To prevent this, we can replicate each user's temporary storage to ensure that no messages are lost. When the user comes back online, the replica can be used to process their messages. If the original server becomes available again, the original and replica instances of the user's transient storage are merged to create a unique store for storing messages.

Optimizations

Latency: To provide a smooth and better customer experience, the messenger service must be real-time. This means that latency needs to be minimized, which can be achieved by using caching to store frequently queried data. A distributed cache like Redis can be used to cache user activity status and recent chats in memory. This helps to improve the performance of the service.

Availability: It is important for our service to remain available as much as possible. To ensure that our system is fault-tolerant, we can store multiple

copies of transient messages. If any message is lost, it can be easily retrieved from its replicas. This helps to maintain the availability of the system and prevent any disruption in service.

Further Requirements

The current version of our system only supports a few basic features, but it can easily be extended to support group chats, video and phone calls, and the ability to view each other's status or stories. Additionally, we can also extend the system to allow for payments or transactions. All of these additional features require advanced concepts that are beyond the scope of this blog. We will cover these functionalities in the second part of this blog.

Thanks to Suyash Namdeo for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!