Design WhatsApp

ASHISH PRATAP SINGH

MAY 01, 2025 • PAID

Share

Nearly everyone uses a **chat application** to send messages and stay connected.

With over **2.5 billion active users** and more than **100 billion messages** exchanged daily, **WhatsApp** is the world's most popular messaging app.

But what does it take to build such a platform that can connect billions of people in real-time across the globe?

In this article, we will dive into high-level design of building such a **scalable ch application**.

1. Requirement Gathering

Before diving into the design, lets outline the functional and non-functional requirements.

1.1 Functional Requirements

- Support **1:1 real-time messaging** between users.
- Show **online/offline** status and **last seen time** of users.
- Show **message delivery status** (sent, delivered, read)
- Allow users to share **images**, **videos**, and **audio clips**.

- Support **group conversations** with up to 100 members.
- Send **Push notifications** for new messages if the receiver is offline.
- **Store** and **retrieve** chat history for each user.

1.2 Non-Functional Requirements

- **Scalability**: Handle millions of concurrent users.
- **High Availability**: Ensure minimal downtime and resilience to server failures.
- **Low Latency**: Deliver messages in real-time with minimal delay.
- **Reliability**: Ensure messages are not lost.

2. Capacity Estimation

Let's assume the following traffic characteristics for our chat application:

> **Total Users**: Assume 1 billion registered users.
>
> **Daily Active Users** (DAU): Around 500 million users actively use the app each da
>
> **Peak Concurrent Connections**: Approximately 50 million users connected at pe
> times.

Average Messages per Day: If each active user sends an average of 10 messages dai
this results in 5 billion messages per day.

Storage Requirements (for Messages):

Assuming each message is around 1 KB.

- **Daily Storage**: 1 KB × 5 Billion messages = 5 TB

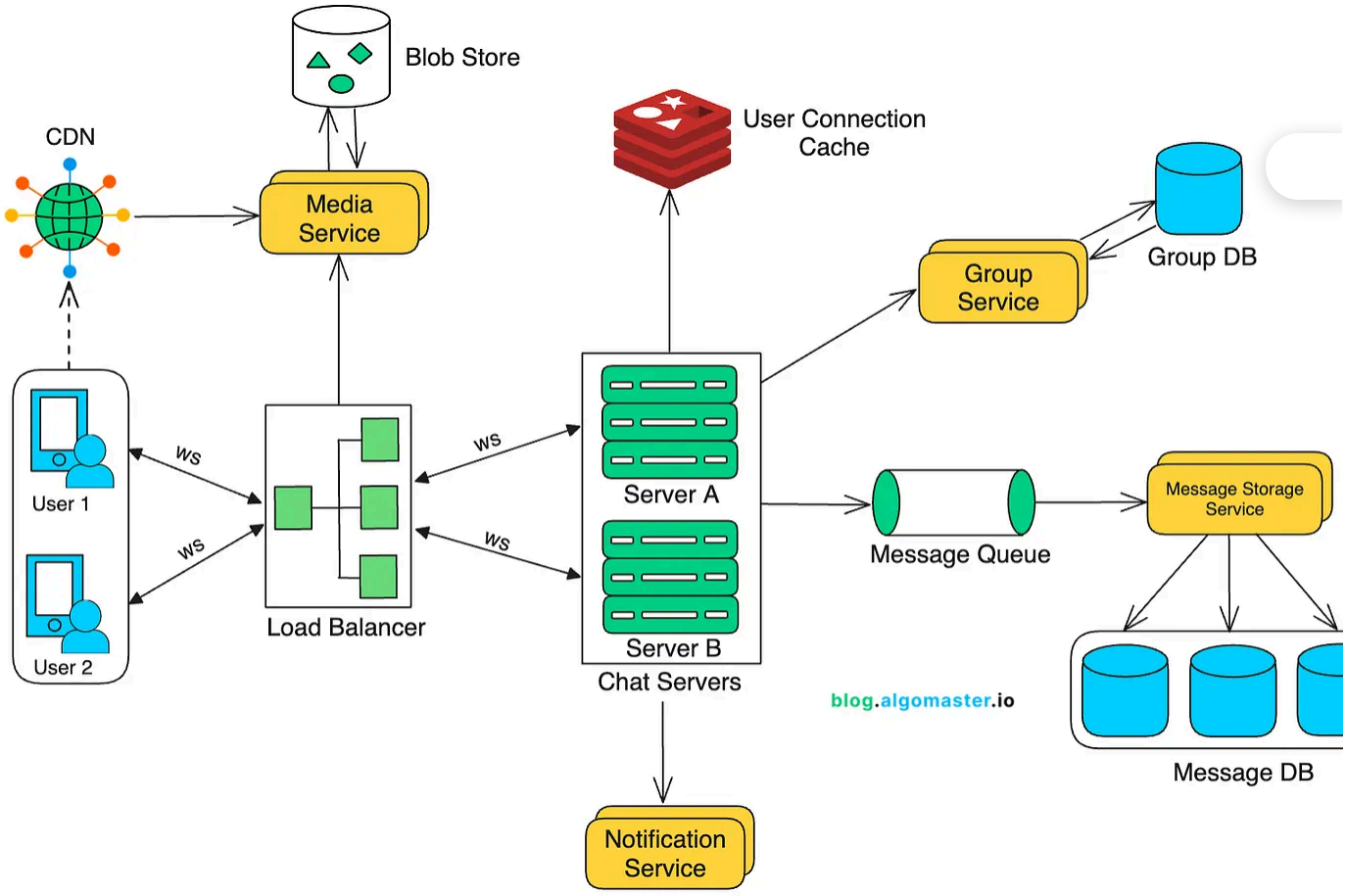
- **Annual Storage**: 365 × 5 TB ≈ 1.8 PB (Petabytes)

Bandwidth Estimation (for Real-Time Communication):

With 10 million users connected concurrently during peak times.

- **Average Bandwidth per Connection**: Assuming an average of 10 KB/s per connection, we would need a total of 100 GB/s of bandwidth to support real-tin messaging at peak usage.

3. High-Level Design



Visualized using Multiplayer

Chat Servers

The **Chat Servers** manage a large number of concurrent connections, facilitate real-time communication, and ensure that messages are delivered efficiently between us with minimal latency.

To support seamless two-way messaging, a protocol like **WebSockets**—designed fo native bidirectional communication between clients and servers—is ideal. (We'll de into this in more detail later.)

Load Balancer

The **load balancer** efficiently distributes incoming traffic from users across multipl instances of chat servers and user-facing services, such as the media service.

Here's how the connection is established between the user and chat servers via the load balancer:

- **Initial Connection**: The client initiates an HTTP(S) request to set up a Web connection. This request passes through the load balancer, which routes it to a appropriate chat server based on factors like user's location and the load-balancing algorithm in use (e.g., round-robin, least connections).
- **Connection Upgrade**: Once the request reaches the selected server, the connection is upgraded from HTTP to WebSocket, establishing a persistent, bidirectional WebSocket link between the client and the selected chat server vi the load balancer.
- **Session Persistence**: To ensure that the client remains connected to the same c server, load balancer uses **sticky** sessions. This can be achieved through **IP hashing**, where the load balancer consistently routes the user to the same serve based on the hash of their IP address.

Alternative with Service Discovery

An alternative approach is to use service discovery, which enables users to connect directly to chat servers.

In this case, users first connect to the service discovery layer to identify the chat ser they should connect to and then establish a WebSocket connection directly with tha server.

User Connection Cache

The **User Connection Cache** is a fast, in-memory cache (e.g., Redis) that stores each user's active connection details, such as the chat server they're connected to and the `last_active` timestamp.

User ID	Server ID	Last Active (Timestamp)
101	srv-1	2024-11-05 14:32:10
102	srv-3	2024-11-05 14:31:50
103	srv-2	2024-11-05 14:32:08

Clients periodically send heartbeat signals to their connected server, and each heartbeat updates the user's `last_active` timestamp in the cache.

This setup enables efficient support for `online/offline` status and `last seen` functionality.

If the difference between the current time and the `last_active` timestamp is wit a defined threshold (e.g., 3 seconds), the user is shown as online; otherwise, they are

marked as offline.

Notification Service

The Notification Service is responsible for delivering real-time notifications to use especially when they are offline or not actively using the application.

When a user is offline, the chat server forwards the message to the Notification Service.

To enhance efficiency, the chat server can send this message to a message queue rat than directly interacting with the Notification Service and waiting for a response.

The Notification service integrates with external push notification providers like **Firebase Cloud Messaging** (**FCM**) and **Apple Push Notification Service** (**APNS**) to deliver messages as push notifications to offline users.

Message Queue

The Message Queue is a distributed, high-throughput queue (e.g., Kafka, RabbitMC that temporarily stores messages before they are consumed by the Message Storage Service.

By acting as an intermediary, the Message Queue decouples message storage from real-time message handling on chat servers, reducing latency and enhancing the scalability of the application.

Message Storage Service

The **Message Storage Service** is responsible for the reliable storage, fast retrieval, a efficient archival of chat messages.

It consumes incoming messages from the **Message Queue** and persists them in the **Message DB** for efficient storage and retrieval.

Message DB

The **Message DB** stores all chat messages in a reliable and efficient manner, ensuri users can access past messages.

This database is designed for **high-write** throughput and efficient retrieval (e.g., **Cassandra**) to handle the large volume of messages in real-time chat applications.

Group Service

The **Group Service** is responsible for handling all group-related functionalities, including creating groups, updating group details and managing group membershi

When a message needs to be delivered to a group conversation, the Chat Servers qu the Group Service to retrieve the current list of group members.

Group DB

The **Group DB** stores and retrieves all data associated with group chats, including group IDs, member lists, admin roles, and group setting.

Media Service

The **Media Service** handles the uploading and management of multimedia content, such as images, videos, and audio files.

It securely stores media files in a blob storage system, while maintaining metadata- such as file type, size, and upload timestamps—in a separate database for easy acce and organization.

By offloading media storage from the main chat servers, the Media Service reduces bandwidth usage on the chat servers and enhances overall app performance.

Blob Store

The **Blob Store** is the storage backend for a chat application's multimedia content, including images, videos, audio files, and documents.

It's designed to handle large volumes of media content while ensuring fast, secure, reliable access.

The Media Store typically leverages cloud-based object storage solutions (such as Amazon S3, Google Cloud Storage, or Azure Blob Storage) that provide high durabi scalability, and cost-effectiveness.

CDN

To reduce latency when uploading or downloading multimedia content, files are distributed to locations geographically closer to users via a **Content Delivery Netw (CDN)**.

When a user shares a multimedia file, the client application uploads it directly t CDN, storing it in a location close to the recipient.

Instead of sending the file itself, the client sends the file's URL to the chat server as part of the message, allowing other users to download and access the content quick and efficiently from the nearest CDN location.

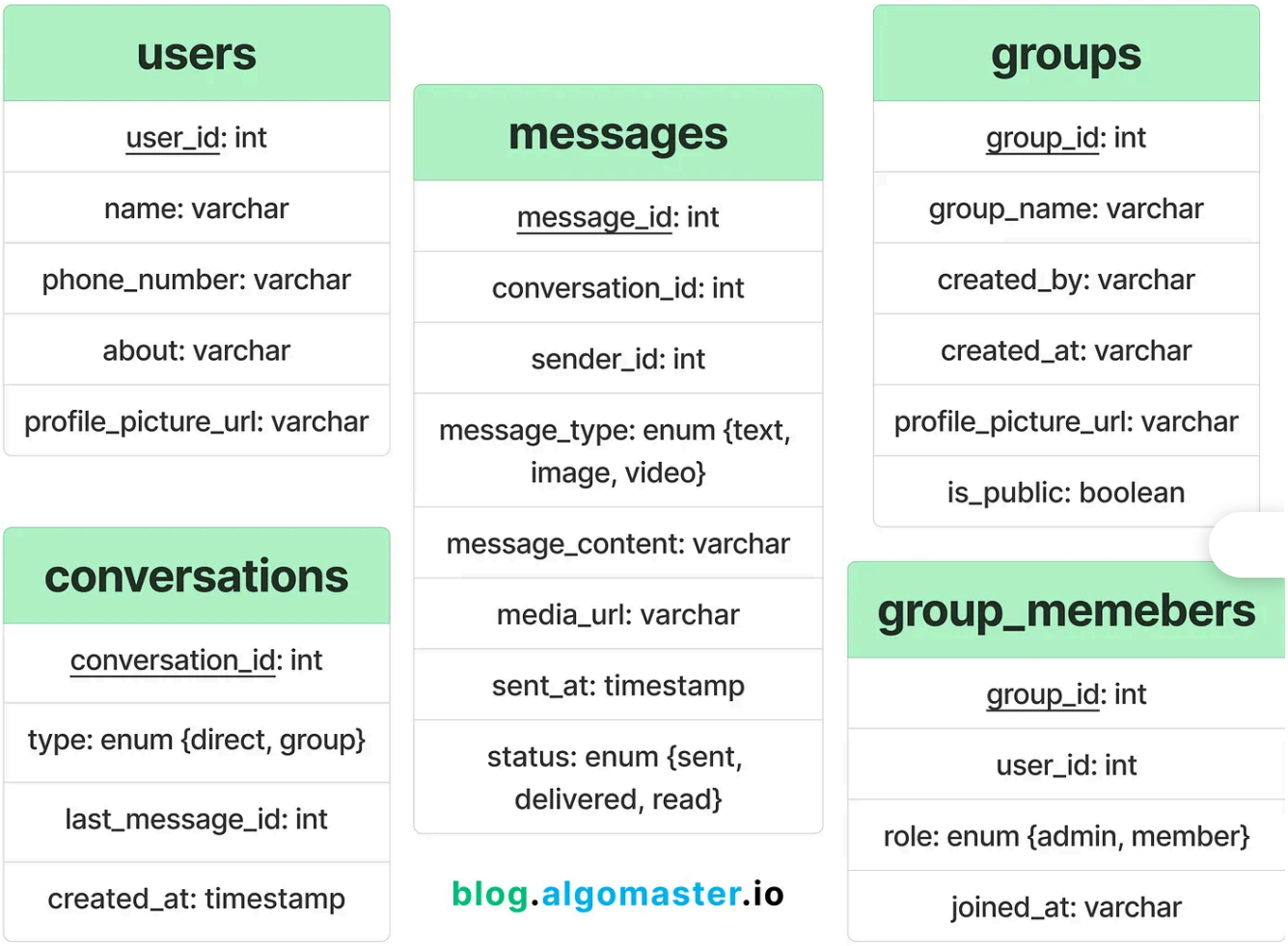
Once files are uploaded to the CDN, the Media Service retrieves them and stores th in a blob store for long-term storage.

This approach reduces the load on chat servers, minimizes latency, and significantl improves media delivery speed for users.

4. Database Design

For a chat application, the database needs to handle core entities like **users**, **messag** and **groups**.

Here's a breakdown of a database design that could support a scalable and efficient chat application.



To store the user, group and conversations data, we can use a SQL database like PostgreSQL.

For message data, it's preferred to use a NoSQL database like Cassandra due to hig write throughput.

For media files, an object store like AWS S3 provides scalable and secure storage.

5. Deep Dive into Key Components

5.1 Why Choose WebSockets over HTTP?

To understand why WebSockets are ideal for real-time messaging, let's examine oth potential solutions and their limitations:

Polling

In polling, the client periodically sends HTTP requests to the server to check for ne messages.

Drawbacks: Polling can be resource-intensive, especially with high polling frequen Since the server responds with "no new messages" most times, this approach can ac substantial overhead and waste server resources.

Long Polling

In long polling, the client holds an open connection with the server until a new message is available or a timeout occurs. When the server has new data, it responds and the client immediately re-establishes the connection, restarting the process.

While this reduces the need for repeated requests as in standard polling, long pollir has several limitations:

- **Connection Overhead**: Each message exchange requires re-establishing the connection, creating significant overhead and putting a heavy load on servers a network resources.
- **Resource Consumption**: The server must maintain many open connections, consuming memory and capacity even without active data exchange.

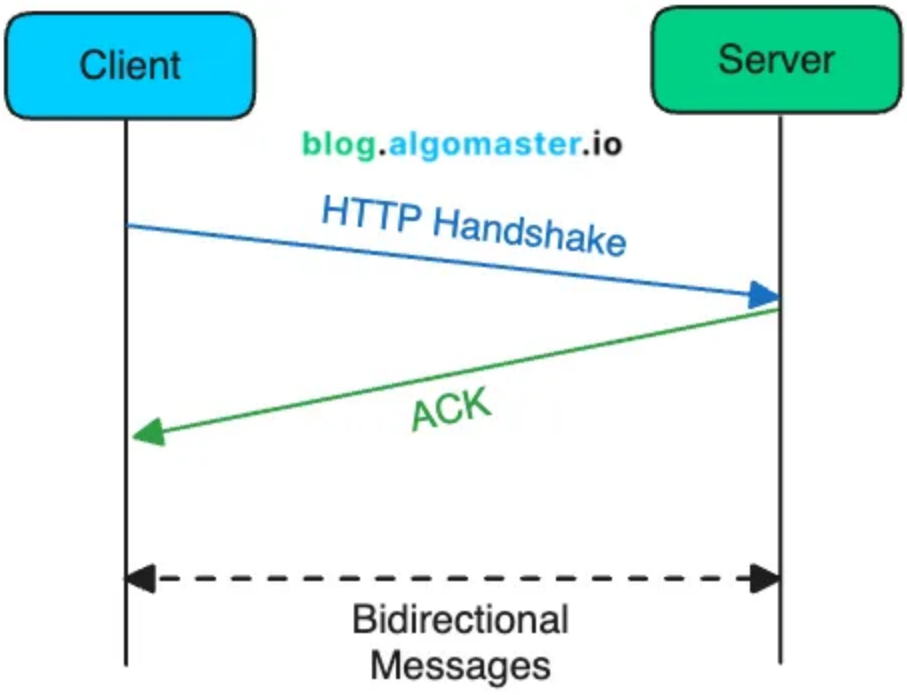
- **Latency Issues**: If the timeout is long, messages may be delayed. If short, frequ resets add the same overhead as standard polling.

Overall, long polling's connection and resource demands make it less suited for rea time chat applications.

WebSockets

WebSockets on the other hand eliminate the need for repeated HTTP handshakes, headers, and responses, reducing overhead and enhancing performance.

The client and server establish a connection once, and this connection stays open f the entire chat session, enabling seamless data transfer.



Visualized using Multiplayer

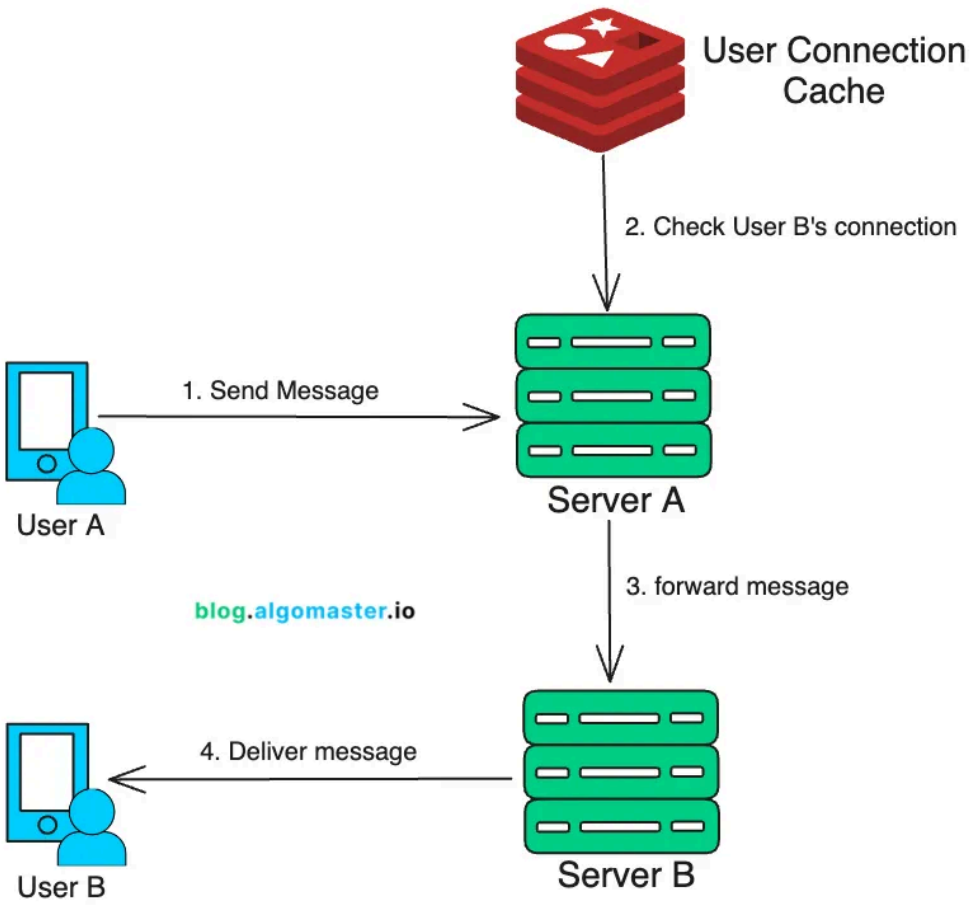
This persistent connection makes WebSocket ideal for real-time communication, where both the client and server need to exchange data frequently and in a timely manner.

5.2 Real-time Message Delivery

Establishing the Connection

When a user opens the chat app, it initiates a WebSocket connection with one of th chat servers. This persistent connection remains open throughout the chat session, enabling real-time communication between the client and server without requiring repeated HTTP requests.

Sending Messages



Visualized using Multiplayer

Once connected, when User A sends a message to User B, the message travels throu User A's WebSocket connection to the server managing that connection (Server A).

Server A then looks up the **user connection cache** to determine whether User B is online and, if so, which server currently holds User B's connection.

- **If User B is online**: Server A forwards the message to Server B, which delivers i User B via their open WebSocket connection.
- **If User B is offline**: Server A sends the message to the **notification service**, wh triggers a push notification to notify User B of the new message.

5.3 Message Delivery Indicators

WebSockets enable real-time status updates for messages (e.g., “message sent,” “message delivered,” “message read”), providing users with instant feedback on message states.

Message Sent

When User A sends a message, it is transmitted over their WebSocket connecti the server handling their connection (Server A).

1. Server A receives the message, pushes it to the message queue for storage, and sends an acknowledgment back to User A.
2. Upon receiving this acknowledgment, User A’s app updates the message status “sent.”

If User A is offline when attempting to send a message, the message won’t be sent until they are back online. The message remains in a pending state on User A's dev until it reconnects and successfully sends the message to Server A.

Message Delivered

Once the User B receives the message, it sends an acknowledgment to Server B.

- Server B sends a delivery acknowledgment to Server A.

- Server A sends the message status “delivered” to the message queue for permanent storage and then relays this update to User A’s app, which reflects t message as “delivered.”

If User B is offline, Server A will not receive an acknowledgment of delivery from Server B, so the message remains in the “sent” state for User A until User B reconnects.

When User B comes online, the client app sends the updates to Server B, at which point it sends a “delivered” acknowledgment to Server A. User A’s app is then upda to reflect the “delivered” status.

Message Read

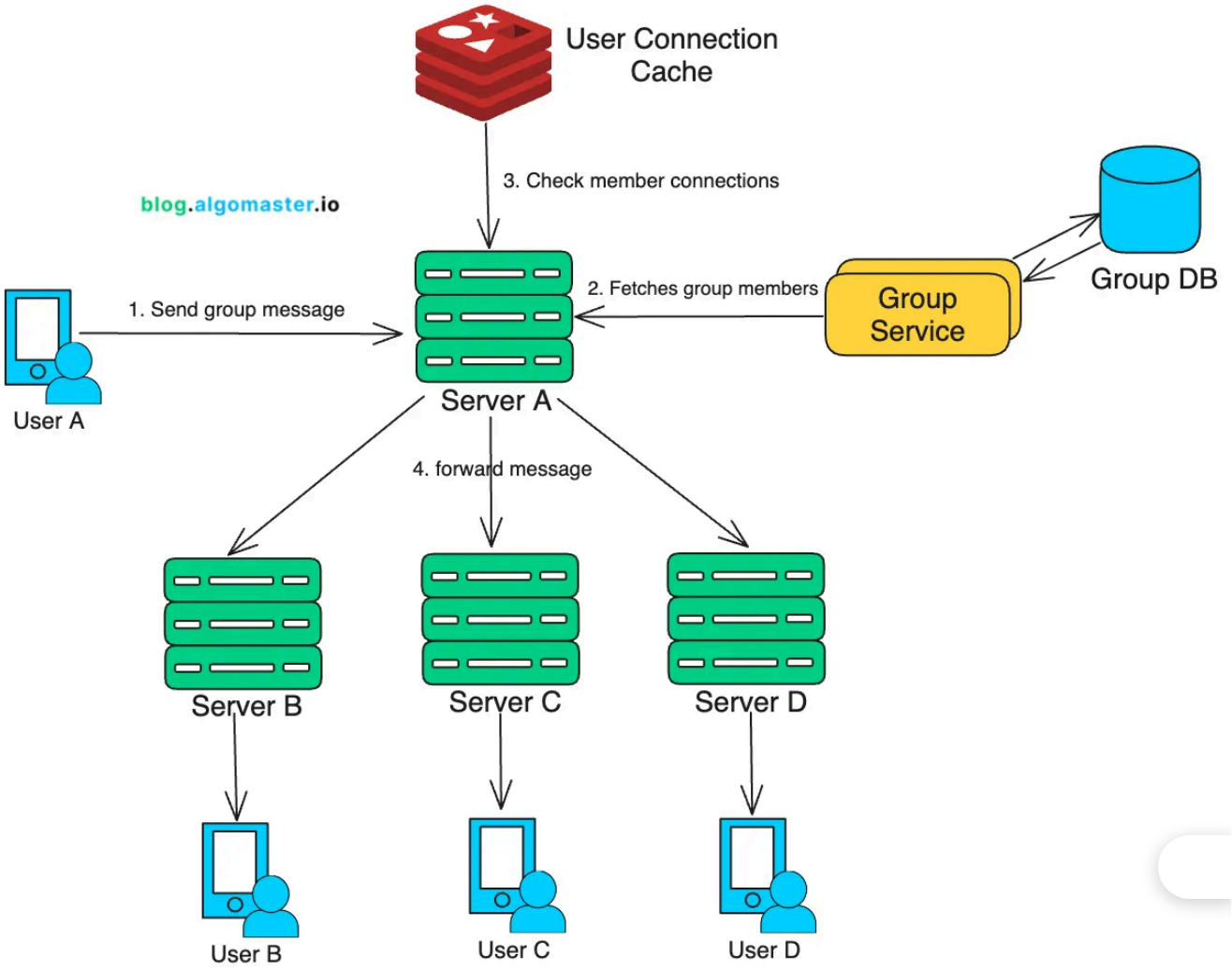
When User B opens the chat window and views the message, their app sends a “rea acknowledgment to Server B.

1. **Server B** logs this event in the message queue and forwards the “read” stat Server A.
2. **Server A** pushes this update to User A’s device, allowing User A’s app to displa the message as “read.”

If User B is offline, they cannot view the message, so it will not trigger a “read” stat When User B reconnects and opens the chat, their app will send a “read” acknowledgment to Server B.

The Server B logs the “read” status in the message queue and forwards it to Server A User A’s app then receives this update, marking the message as “read.”

5.4 Group Messages



Visualized using Multiplayer

1. When User A sends a message in a group chat, the message travels through Us A's WebSocket connection to the server managing that connection (Server A).
2. **Server A** queries the **Group Service** to retrieve a list of all active members in th group.
3. **Server A** checks the **user connection cache** to determine which group member are currently online and the specific servers they are connected to.
4. **For Each Online Member**:
    - If a member is connected to **Server A**, it delivers the message directly over existing WebSocket connection.

- If a member is connected to a different server (e.g., **Server B**), **Server A** forwards the message to **Server B**, which then delivers the message to the u over their WebSocket connection.

5. For offline group members, **Server A** sends the message to the **Notification Service**. The Notification Service triggers push notifications for each offline gr member, alerting them of the new message.

In a group chat, every message sent must be distributed (or "fanned out") to each group member.

As group size increases, so does the fan-out workload. For instance, in a group of 5(members, each message requires 500 individual message deliveries, which can quic overwhelm the server.

That's why, most chat applications put limits on the number of members a group ca have (WhatsApp currently has 1024).

Message Persistence:

To ensure that all group members can access the message history, **Server A** pushes message to the **Message Queue** for storage.

The **Message Storage Service** consumes the message from the queue and stores it i the **Message DB**, where it will be available for group members to retrieve later.

Acknowledgments and Status Updates

- As each online group member receives the message, their respective server sen delivery acknowledgments to **Server A** (or directly updates the status in the **Message DB** if needed).
- Once users open and view the message, read acknowledgments are similarly propagated back to **Server A** and stored for future reference, allowing User A t see which group members have received and read the message.

5.5 How to generate Message ID?

To retrieve recent messages efficiently, we can use a **time-based message ID.**

A time-based message ID typically combines a timestamp with a unique identifier f each message.

This allows us to order messages by timestamp, retrieve messages within a specific time range and support pagination ("load more") where you retrieve messages befor or after a specific timestamp without re-sorting the dataset.

A common format might include:

```
[Timestamp in milliseconds][Unique Sequence/Random Component]
```

- **Timestamp** (**in milliseconds**): The first part of the ID is the timestamp at w the message is created. This allows messages to be sorted chronologically by th creation time.
- **Unique Sequence or Randomized Component**: Appending a unique componer (like a random or incremental sequence) ensures each message ID is unique, ev if multiple messages are sent at the same millisecond.

For instance, a message sent at `2024-11-05 12:34:56.789` with a sequence `00` could have an ID: `2024110512345678900`1.

6. Address Key Issues and Bottlenecks

6.1 Chat Server Failure

In the event of a chat server failure, all clients connected to that server will lose the connection.

To recover, clients automatically attempt to reconnect, this time establishing a new connection with a different available server.

The load balancer continuously monitors the health of each chat server through regular health checks.

If a server goes down, the load balancer immediately stops directing traffic to it, ensuring new connections are routed to healthy servers only.

6.2 Sharding

To support horizontal scaling and efficient data access, we can implement sharding across different data types:

1. **User Data Sharding**: Shard user data based on `user_id`. This will allow us to distribute user records across multiple servers and enable us to scale as the use base grows.
2. **Message Data Partitioning**: Partition messages based on `message_id`, using a timestamp-based `message_id` to enable efficient time-based searches. This structure allows recent messages to be accessed quickly and older messages to located based on timestamp.

6.3 Optimizing Storage Cost

With large volumes of messages and multimedia content, optimizing storage costs i essential.

Here are some effective strategies:

1. **Compress Multimedia Files**: Compressing large files (e.g., images, videos) can reduce storage requirements and cuts costs significantly.
2. **Archive Older Messages**: Most users only access recent messages, which can b cached locally on their devices. Older messages can be moved to lower-cost, co storage (e.g., Amazon Glacier), reducing expenses while still allowing access if needed.
3. **Deduplicate Files**: Avoid storing multiple copies of identical files by implementing deduplication, which can save significant space when the same media is shared across multiple users or groups.
4. **Efficient Metadata Storage**: Store metadata (e.g., file type, size, timestamps) separately from the media itself to reduce the load on primary storage and mak searches faster and more efficient.

Thank you for reading!

I hope you have a lovely day!

See you soon,
Ashish