System Design Problems

People Tech Group Week 4(L1 Batch 2)

Name: Shrestha Sai Laxmi Yadavilli
Email: shrestha.yadavilli22@gmail.com

Table of Contents

1. Introduction

System design is a crucial aspect of software engineering that involves defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. Two significant system design problems are discussed in this document: **Designing a URL Shortener** and **Designing a Chat Application**.

1.1 Problems Covered

This document explores two essential problems in system design:

1. **Designing a URL Shortener System**: This problem addresses the need for a service that converts long URLs into shorter, more manageable versions. It is discussed how to generate short URLs, retrieve long URLs from these short versions, and handle redirections seamlessly.
2. **Designing a Chat Application**: This problem focuses on creating a system that facilitates real-time messaging between users. The key functional and non-functional requirement, various components involved, and the system design process is covered which is necessary for developing an efficient chat application.

1.2 Objective of Week's Task

The objective of this week's task is to learn about these two system design problems in detail and document our learnings. By analyzing the design aspects of a URL shortener and a chat application, the aim to gain insights into effective system architecture and the considerations needed to build scalable and robust software solutions.

2. Designing a URL Shortener System Design

The primary challenge of a URL shortening service is to generate shorter aliases for long URLs, enabling easy sharing and readability. For example, a long URL such as https://example.com/very-long-url could be shortened to something like https://tinyurl/abc123.

The system needs to perform two key functions:

1. **Generate Short URLs from Long URLs**: Convert long URLs into shorter, unique versions.
2. **Retrieve Long URLs from Short URLs**: When a short URL is accessed, it should redirect to the original long URL.

2.1 Functional and Non-functional Requirements

Functional Requirements:

- **Generate Short URL:** The system must accept a long URL as input and return a shortened URL.
- **Redirect to Long URL**: When a short URL is accessed, the system must retrieve the corresponding long URL and redirect the user.
- **Collision Handling**: The system must avoid generating duplicate short URLs for different long URLs.

Non-functional Requirements:

- **Low Latency**: The system must process requests with minimal delay to ensure quick URL shortening and redirection.
- **High Availability**: The system must be available at all times to handle requests without downtime.
- **Scalability**: The system should scale efficiently to support increasing traffic and URL generation.

2.2 Generate Short URLs from long URLs

To build an efficient URL shortener, the system should be both deterministic and optimized for performance, avoiding duplicate mappings and minimizing latency. For example, hashing functions are often used to ensure the same long URL always maps to the same short URL. However, the design must balance hashing efficiency and collision avoidance.

2.2.1 Hashing

Deterministic Hashing: The system should use a hashing function that generates the same output for the same long URL. This ensures that each long URL maps consistently to the same short URL every time, providing predictability. However, using the entire hash (e.g., up to 256 characters) would defeat the purpose of a short URL. Full-length hashes can be very long (sometimes up to 256 characters). To keep the short URL compact and meaningful, we should avoid placing the entire hash in the shortened link. For example, a short URL like "https://tinyURL.com/<from_hash>" may become unnecessarily long.

- **Truncate Hashes**: Instead of using the entire hash, we can extract only the first or last few characters (e.g., 4-5 characters). However, this introduces the possibility of collisions since different long URLs may share the same truncated hash. Collisions occur when two different long URLs map to the same short URL.
- **Handling Collisions**: To prevent collisions, before storing the URL, the system should check if the generated short URL already exists in the database. If it does, it means there's a conflict. In such cases, we append a pre-determined string (or modify the long URL slightly) and recompute the hash to generate a new short URL. This process repeats until we find a unique short URL. The number of characters used (e.g., 4-5 or 8-9) depends on the expected volume of URLs to be shortened. Larger systems may require more characters.

2.2.2 System Components

1. **Client (Web Browser)**: The client is the front-end interface where users submit long URLs. This client sends an HTTP POST request to the backend, asking for a short URL. The request is directed to a load balancer.
2. **Load Balancer**: The load balancer routes the request to one of several available API servers, ensuring balanced traffic distribution. This prevents all user requests from overloading a single API server.
3. **API Server**: Multiple API servers (horizontal scaling) handle the requests. Upon receiving a request, the API extracts the long URL from the request body. It checks the database to see if this long URL already has a corresponding short URL. If it does, the system returns the existing short URL to the user. If no existing mapping is found, the system hashes the long URL to generate a 5-6 character hash, ensuring it's unique by checking for collisions in the database.
4. **Database**: The system stores mappings between long and short URLs in a simple relational database (e.g., MySQL). The table consists of three main columns:
   a. **ID**: Auto-incremented unique identifier for each record.
   b. **Long URL**: The original long URL input by the user.
   c. **Short URL**: The generated short URL that maps to the long URL.

2.2.3 Performance Optimization

Users are more likely to access recently created URLs, so recent long-to-short URL mappings can be cached. This reduces database lookups, improving response times.

- **Redis Cache**: Redis (or any other caching solution) can store recent URL mappings. Once a new URL is written to the database, it's also written to Redis as a key-value pair (long URL as key, short URL as value).
- **Time to Live (TTL)**: Each cache entry can have a TTL (e.g., 1-2 hours). If the user doesn't access the short URL within this time, the system can safely expire the cache entry, deferring future requests to the database.

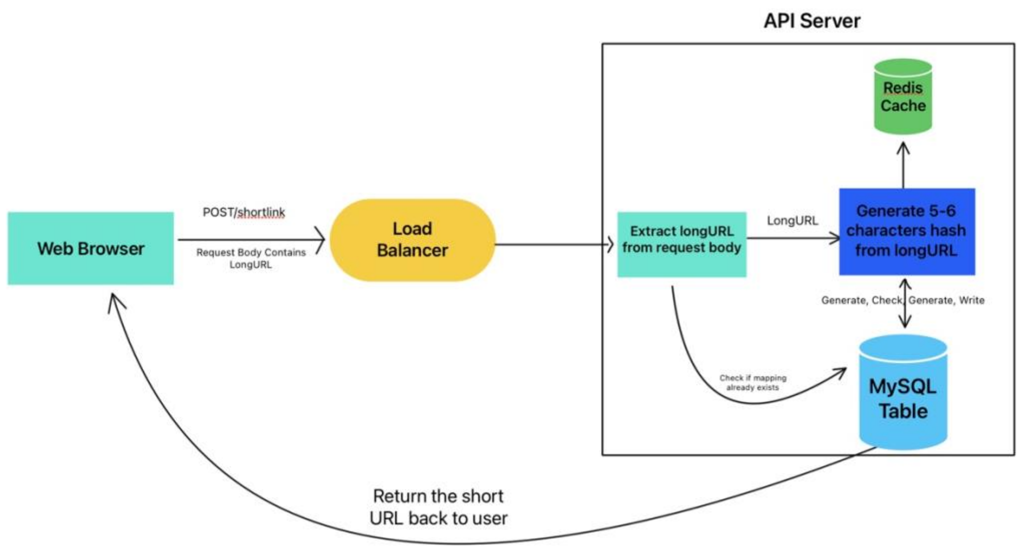
API Server
Redis Cache
Web Browser
POST/shortlink
Request Body Contains LongURL
Load Balancer
Extract longURL from request body
LongURL
Generate 5-6 characters hash from longURL
Generate, Check
Generate, Write
Check if mapping already exists
MySQL Table
Return the short URL back to user

Figure 1: Generate ShortURLs from LongURLs, Source: [1]

2.3 Retrieve Long URLs from Short URLs

This system is to retrieve long URLs from the system after a user feeds short URLs to the browser.

2.3.1 System Components

1. **Client (Web Browser)**: A user submits a request to retrieve the original long URL by accessing the short URL. The browser sends a **GET** request to the load balancer with the short URL in the request.
2. **Load Balancer**: It forwards the request to the appropriate API server, which then handles the logic for retrieving the long URL.
3. **API Server**:
    a. The API server first extracts the **hash** (unique part) from the short URL.
    b. **Check in Cache**: The server first checks in **Redis Cache** for the hash. If it finds the hash there, it immediately retrieves the long URL and redirects the user to the long URL.
    c. **Check in Database**: If the hash isn't found in Redis, the server looks it up in the **database** (e.g., MySQL). Once found, the server redirects the user to the long URL and may update the cache for future requests.

2.3.2 Redirection

The response from the server redirects the user to the original long URL. There are two types of redirection:

1. **301 Permanent Redirect**: This type of redirect allows the browser to cache the response. As a result, the next time a user attempts to access a specific short URL, the browser already knows the corresponding long URL and will directly navigate to it without calling the server for redirection. The advantage of this approach is that the server will only receive a request once per short URL, as the browser retains the long URL in its cache. However, this also means that you won't be able to track how many times users access the URL since the server isn't hit with subsequent requests.
2. **302 Temporary Redirect**: In this case, the user instructs the browser to reach out to the server, which responds with the long URL. The browser does not cache this mapping, meaning that each time the user tries to access the short URL, it will query the backend server again. The downside of this method is that the server is contacted every time someone attempts to use the short URL. On the positive side, this allows you to track every user interaction with your short URLs, making it easier to monitor usage and engagement with your product. However, it also increases the number of requests your server must handle.

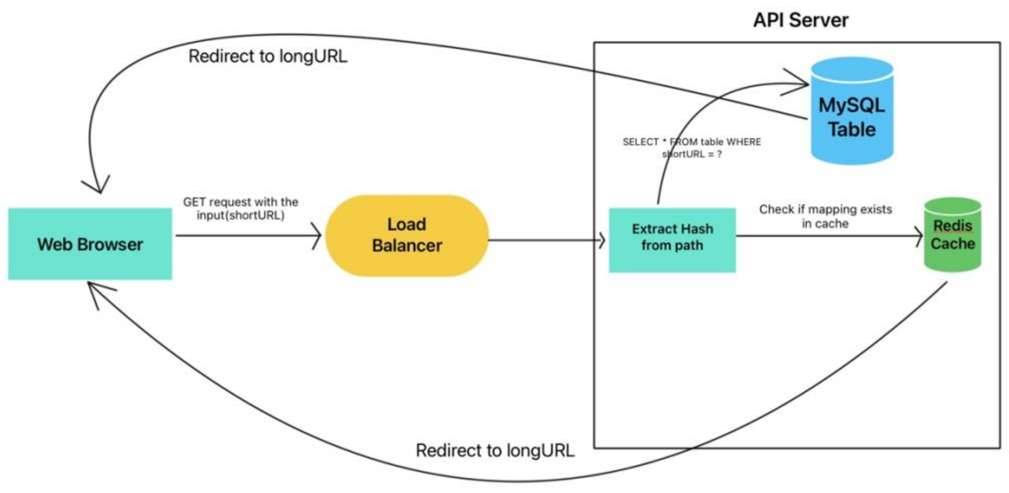

Figure 2: Retrieve LongURLs from ShortURLs, Source: [1]

3. Designing a Chat Application System Design

The problem statement is to design a chat application. It involves creating a system that enables real-time communication between users through messaging. The application must support one-on-one and group chats, handle multimedia file exchanges, and ensure reliable message delivery. Key challenges include maintaining low latency, high availability, and data security while scaling to accommodate increasing user demand.

3.1 Functional and Non-functional Requirements

Functional Requirements

1. **User Registration and Login:** Users need to create an account or log in using credentials like an email and password or social media accounts (e.g., Google or Facebook).
2. **One-to-One Messaging:** Users should be able to send private, real-time text messages to one another. The app should save these messages so users can access them later.
3. **Group Chats:** The app should let users create or join group chats where multiple people can exchange messages in real-time.
4. **Message Delivery Notifications:** The app should show whether a message has been sent, delivered, or read by the recipient.
5. **Multimedia Support:** Users should be able to send multimedia files such as images, videos, and voice messages.
6. **User Status:** The app should display the online or offline status of users and also the last seen or activity timestamps for each user.
7. **Message Storage:** Messages should be saved in a database for future reference and users should be able to retrieve these messages.
8. **Push Notifications:** The app should notify users of new messages or activities even when they are not actively using the app.

Non-functional Requirements

1. **Low Latency(Real-time Chat messages):** The app should ensure real-time communication with minimal delay in message delivery.
2. **High availability:** The app should be always available to ensure that the users can access the service 24/7 without interruptions. This can be achieved with load balancing and multi-region deployment.
3. **Scalability:** The system should be able to handle a growing number of users and messages. Scaling horizontally can support high traffic.
4. **Security:** The app should use encryption so that only the sender and recipient can read the messages. Authentication should be secure to prevent unauthorized access.
5. **Data Consistency:** Messages should be reliably delivered, even if the network is slow or there's a system error. The data should eventually be the same across all systems.
6. **Fault Tolerance:** The system should handle failures without losing data or crashing. If part of the system fails, it should recover gracefully.
7. **Performance:** The app should be able to handle a lot of messages quickly and keep working efficiently, even with many users.

3.2 System Components

1. **Client-Side Applications (Mobile/Web):** The part of the app that users interact with. This could be a mobile app or a web app. Technologies like React Native or Flutter can be used to build these.
2. **Authentication Service:** This handles signing up, logging in, and making sure only authorized users can access the app. Tokens like JWT (JSON Web Tokens) can be used for security.
3. **WebSocket/Real-Time Communication Server:** WebSockets enable real-time communication between the user's device and the server. This allows messages to be sent and received instantly.
4. **Message Queue (Broker):** A message queue system (e.g., Kafka or RabbitMQ) helps manage the flow of messages between users and ensures messages are delivered even if the app is under heavy load.
5. **Database:** The database stores user data and chat messages. NoSQL databases like MongoDB are often used because they are fast and can scale well.
6. **Media Storage Service:** For storing larger files like photos or videos, a service like Amazon S3 or Azure Blob Storage can be used.
7. **Notification Service:** Services like Firebase or Apple Push Notifications send alerts to users when they get new messages.
8. **Load Balancer:** A load balancer ensures that traffic is distributed across multiple servers, preventing any one server from becoming overloaded.
9. **CDN (Content Delivery Network):** A CDN speeds up the delivery of images, videos, and other content by storing copies in different locations around the world.
10. **Analytics and Monitoring:** Tools like Google Analytics help track how the app is performing and allow developers to monitor and fix issues quickly.

3.3 System Design Process

1. **User Registration and Authentication:** Start by creating a system for users to register and log in securely. You can use standard sign-up methods (like email/password) or social logins (like Google or Facebook). After login, generate a secure session token that identifies the user.
2. **Real-Time Communication:** Use WebSockets to enable instant messaging between users. WebSockets maintain an open connection between the server and client, allowing for real-time message delivery. If the chat app grows, you can use message queues like Kafka to manage the communication between users, so no messages are lost.
3. **Storing Data:** Store all chat messages in a database, including metadata like who sent the message, at what time, and whether the message has been read. Use databases like MongoDB to store this data efficiently. If you need to store lots of data across different locations, consider using a distributed database to ensure high availability.
4. **Handling Media Files:** When users send media files (images, videos, etc.), the app should upload these to a cloud storage service like AWS S3 or Azure Blob. The app will store only the links to these files in the chat messages, so users can download them when needed.
5. **Push Notifications:** Set up a push notification system so that users receive alerts for new messages even when they aren't using the app. Use services like Firebase Cloud Messaging (for Android) or Apple Push Notification Service (for iOS).
6. **Managing Message Flow:** Use message brokers like Kafka or RabbitMQ to handle large amounts of messages at a time. This ensures the app can deliver messages even under heavy use and makes the system more reliable.
7. **Load Balancing and Scaling:** Add a load balancer to distribute user traffic across multiple servers, ensuring no single server becomes overloaded. This will help the system perform well even when user activity spikes.
8. **Consistent Hashing:** Consistent hashing is a method that helps distribute data evenly across a cluster of servers, ensuring that only relevant information is stored in each gateway.
9. **Making the App Resilient to Failures:** Design the system to handle failures gracefully. If one part of the system goes down, others should keep working. This might include making backups of the database or replicating services across multiple regions to ensure reliability.
10. **End-to-End Encryption:** To ensure privacy, encrypt all messages on the user's device before sending them to the server, so only the intended recipient can read them. This makes it impossible for anyone else (including the server) to access the message content.
11. **Monitoring and Logging:** Set up monitoring tools to track how well the app is performing. Keep logs of system activities to help diagnose problems and improve the app.
12. **User Presence (Online/Offline Status):** Use WebSockets to show whether a user is online or offline in real time. The server tracks user activity to determine their status.
13. **Testing and Deployment:** Thoroughly test the app to ensure everything works as expected. Perform unit tests, integration tests, and user tests. When the app is ready, deploy it to the cloud (like AWS or Azure) and set up automatic scaling for when the app grows.

3.3.1 One-to-One Chat

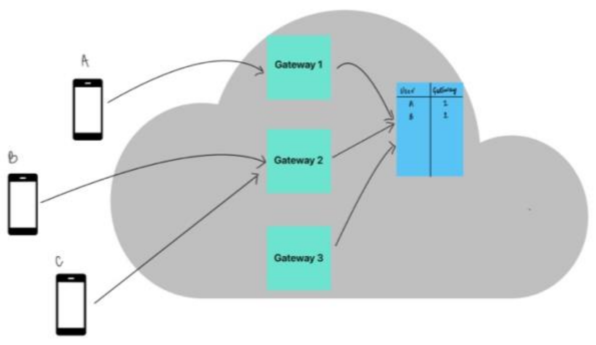

Figure 3: User-to-box Mapping, Source: [3]

The system begins when a user installs the Chat application on their cell phone and connects to the Chat application's cloud through a component known as a **gateway**. This gateway serves as an intermediary, allowing

users to communicate using external protocols while Chat's internal services may utilize different, more efficient communication methods. The main purpose of the gateway is to handle the necessary security features, minimizing the overhead that comes from larger HTTP headers when communicating internally, as these internal connections are generally more secure.
Once connected, when Person A sends a message to Person B, the gateway plays a crucial role in routing that message. It needs to know which server box Person B is connected to, requiring a **user-to-box mapping** system. This means that the gateway must store information indicating which user is connected to which server box. For example, if there are three boxes, the gateway must be aware that User A is on box 1 and User B is on box 2. However, maintaining this mapping directly on each server box can be inefficient because it consumes memory and creates duplicate data across the servers. Moreover, as this information frequently changes due to users connecting and disconnecting, it leads to unnecessary complexity and resource consumption.

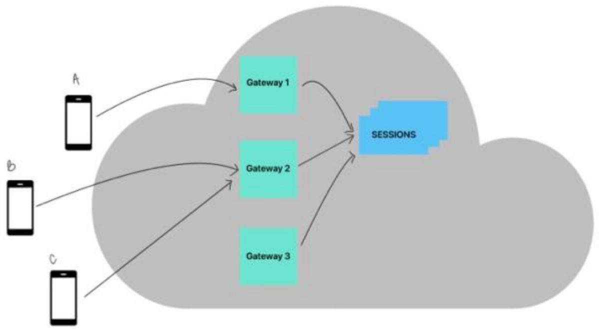

Figure 4: Sessions Microservice, Source: [3]

To address these issues, a **sessions microservice** is used. This service is responsible for maintaining the mapping of which users are connected to which boxes. It decouples the connection management from the gateway, allowing the gateway to focus solely on message routing. When User A sends a message to User B, the gateway simply forwards the request to the sessions service, which acts like a router. It determines which box User B is connected to and then routes the message accordingly. However, since traditional protocols like HTTP only allow clients to send requests to servers and not the other way around, another method is needed for real-time communication.

To achieve real-time messaging, the system uses **WebSockets**. Unlike HTTP, WebSockets allow for peer-to-peer communication, meaning messages can be sent directly between users without the need for repeated requests. When User A's message reaches the gateway, it can be sent to the sessions service, which then routes it to the appropriate box for User B. Once User B receives the message, they send an acknowledgment back to the gateway, confirming receipt. This acknowledgment is processed by the sessions service, which then notifies User A that their message has been delivered. Furthermore, if User B opens the chat and marks the message as read, the same communication flow is followed to ensure that User A receives a read receipt, completing the messaging lifecycle.

3.3.2 Last Seen or Online Feature

Another feature is to track the last seen status or 'Online' status of users in a messaging application, which can become complex at a large scale with millions of users. The primary requirement is for User B to know when User A was last online. Instead of having the server repeatedly ask User A for their status, which would be inefficient, the server can maintain this information independently. Therefore, User B will only interact with the server to inquire about User A's last seen status.

When User B queries the server about User A's last online time, the server checks a database table that stores user activity, specifically a timestamp indicating when User A was last active. To keep this timestamp accurate, the server must log User A's activities—such as sending or reading messages. Every time User A performs any action that indicates they are online, this activity is recorded, and the last seen timestamp is updated accordingly. It's essential to ensure that if User A was online just seconds ago, the status should reflect "online" rather than showing the exact time they were last active. For instance, if User A has been inactive for three seconds, User B should be informed that User A is online, rather than displaying that they were last seen three seconds ago. A threshold can be established for how recent an activity must be to consider the user as "online," potentially set to 10 or 15 seconds.

To efficiently manage this, a **Last Seen microservice** is introduced. This microservice tracks user activities and updates the last seen timestamp whenever User A sends a request to the gateway. However, not all requests made by User A signify user activity; some may be system-generated, such as requests for delivery receipts or notifications. Therefore, the application needs to differentiate between user activities and system requests. Each request sent from the client can include a flag indicating whether it is a user activity or an app-generated request. If it is an app request, it will not be sent to the Last Seen microservice, while user activity requests will trigger an update to the last seen timestamp. This approach ensures that User B can accurately query the last seen status of User A and determine whether they are currently online or when they were last active.

3.3.3 Group Messaging

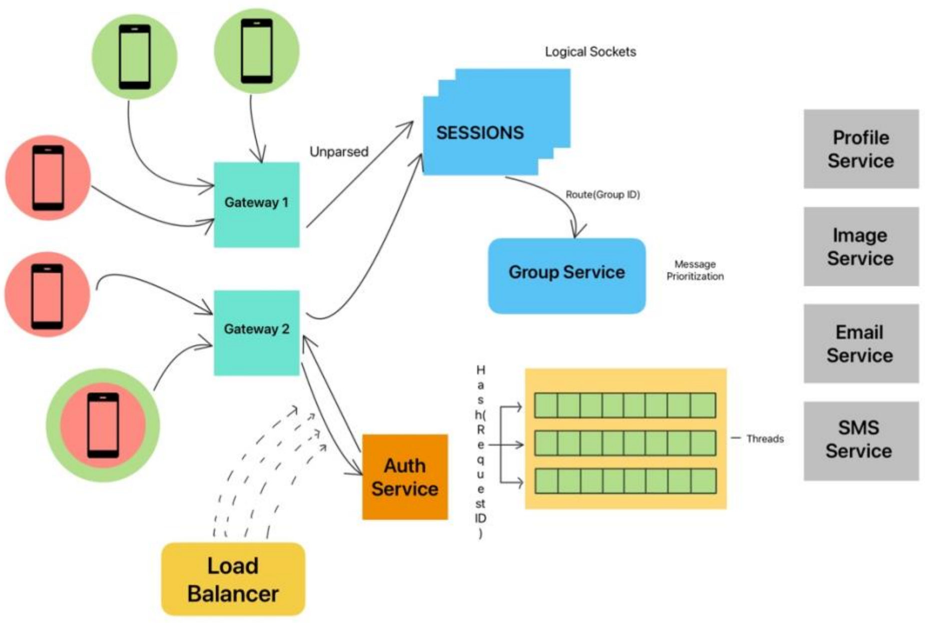

Figure 5: Group Messaging, Source: [3]

Overview:
An important feature of Chat application is the group messaging within the messaging application, focusing on how messages are distributed among users in different groups. Let's take an example where there are five users, represented in two distinct groups: the red group and the green group. When a user from the red group sends a message, it should be delivered to all other users within that same group. For instance, if a red user is connected to Gateway One, while other red users are connected to Gateway Two, the message needs to traverse through the appropriate gateways to reach all group members.

To facilitate this, the system employs a **Group Service** that manages information about group memberships separately from the Session Service. The Session Service is responsible for handling user sessions and message routing but can become overwhelmed if it tries to store all information related to group memberships. Instead, when the Session Service receives a message from a red user, it queries the Group Service to determine who else belongs to that group. The Group Service responds by providing the list of group members along with their user IDs. The Session Service then utilizes its database, which ideally contains cached information, to identify where each of these users is connected—specifically, which gateway they are using.

However, a potential issue arises when dealing with larger groups. Messaging applications like WhatsApp typically limit group sizes to a maximum of 200 members. This limitation is important because sending a message to a large number of recipients simultaneously could overwhelm the system, particularly when handling messages for high-profile users with millions of followers. To avoid this, messages can be processed in batches or users can be encouraged to pull messages themselves to minimize server load. Therefore, setting a cap on group sizes to a reasonable number, such as 200, ensures that the system can effectively manage the delivery of messages while maintaining real-time communication. By relying on WebSockets, the sessions can send messages to the appropriate users without excessive delays or any strain on the server.

In Detail:

To go in more detail of the group messaging mechanism within the architecture, it’s essential to understand that a substantial number of users will be connecting to the gateways. This high volume of connections can lead to memory shortages. To mitigate this issue, we have separated out the **Session Service** from the gateways, which effectively reduces the memory footprint on the gateways themselves.

When a message is transmitted, it is typically sent over HTTP in a JSON format. However, it is not practical to parse and authenticate messages at the gateway level, as these tasks can burden the system and consume valuable resources. Instead, the goal is to offload as many responsibilities as possible from the gateways since they are primarily managing WebSocket connections, which require a higher memory allocation. Therefore, it is more efficient to send an **unparsed message** to the **Session Service** or any relevant service that can handle it.

A streamlined approach to processing unparsed messages is to route them through a dedicated **Parser/Unparser Microservice**. This microservice takes the unparsed message and converts it into a structured format that is more suitable for internal communication. For instance, if the internal protocol is something like Thrift (used by Facebook), this microservice will handle the parsing and transform the electronic message into a corresponding object in the programming language that is being used. The advantage of this design is that the gateways remain free of heavy lifting, allowing them to focus on maintaining active WebSocket connections while the Parser/Unparser Microservice handles the necessary transformations before routing the message to its final destination.

Another crucial aspect of the architecture is the management of **Group ID to User ID** mappings. Since one group can consist of multiple user IDs, it is vital to minimize duplication of information. This is where **consistent hashing** comes into play. Consistent hashing is a method that helps distribute data evenly across a cluster of servers, ensuring that only relevant information is stored in each gateway. By routing requests based on Group IDs, the system can efficiently determine which users belong to a specific group. This approach not only optimizes memory usage but also improves the overall performance and scalability of the messaging system.

The routing mechanism in the architecture is designed to handle potential failures effectively. In the event that the **Group Service** encounters an issue, such as being unable to send a message to the appropriate gateway, a strategy is needed for retrying the operation. A reliable solution for this is the use of **message queues**. Message queues play a crucial role in ensuring that messages are sent, regardless of whether they are delivered immediately or after a configurable delay—be it 10 seconds, 15 seconds, or any other duration that fits the application’s requirements.

When a message is placed into the message queue, it guarantees that the message will be processed, allowing for retries as necessary. For instance, if the message queue fails to send the message after a predefined number of attempts—say, five retries—it will inform the system of the failure. This failure notification is essential for maintaining transparency with the client, allowing the application to respond appropriately by informing the user that the attempt to send the group message has failed.

Once the **Group Service** successfully receives the message, it acknowledges receipt by sending a confirmation back to the **Session Service**. Subsequently, the **Session Service** communicates this status to the gateway, ensuring that the user who initially sent the message receives a visual indicator, such as a tick mark, confirming that their message has been sent.

4. Reference

1. [1] “Design a URL Shortener,” Youtube, [Online]. Available: https://www.youtube.com/watch?v=4fTWfFg91V0.
2. [2] “TinyURL System Design,” Youtube, [Online]. Available: https://www.youtube.com/watch?v=AVztRY77xxA.
3. [3] “Whatsapp System Design: Chat Messaging Systems for Interviews,” Youtube, [Online]. Available: https://www.youtube.com/watch?v=vvhC64hQZMk.
4. [4] “Designing a Chat Application: System Design,” Youtube, [Online]. Available: https://www.youtube.com/watch?v=Kz7hxN92aik
5. [2] “WhatsApp System Design | Large Scale Messenger or Chat like application | System Design Interview,” Youtube, [Online]. Available: https://www.youtube.com/watch?v=jN1pnBVIj7M.