Omegle-Like Chat Application - Use Cases & Architecture

1. Overview

This document outlines the use cases and architecture of a real-time chat application similar to Omegle. The system uses WebSockets for communication, Redis for user queuing, and a clean architecture for maintainability and scalability.

2. Technologies Used

- Programming Language: Go (Golang)

- WebSockets: Gorilla WebSockets

- Data Storage: Redis (Priority Queue)

- Architecture: Clean Architecture, Domain-Driven Design (DDD)

- Deployment: Docker, Kubernetes

3. System Design

3.1 Key Components

1. **User Connection Management**: Handles WebSocket connections and stores active users.

2. **Queue Management**: Redis-based waiting queue for efficient pairing.

3. **Pairing Algorithm**: Fetches two users, removes them from queue, and stores the active chat session.

4. **Skip/Disconnect Handling**: Ensures smooth reconnections and prevents users from getting stuck.

4. Use Cases

4.1 User Connects

1. User initiates a WebSocket connection.

2. A unique userID is generated based on the user's IP.

3. The user is added to Redis waiting queue.

4. If another user is available, they are paired immediately.

4.2 Pairing Users

1. System picks two users from the waiting queue.
2. Stores their pairing information in Redis.
3. Sends WebSocket messages notifying users of their new chat partner.

4.3 Skip/Next Chat

1. User sends a 'skip' command.
2. Their current chat session is removed.
3. Both users are returned to the waiting queue.
4. The pairing process restarts.

4.4 User Disconnects

1. User closes the WebSocket connection.
2. System removes their chat session.
3. If the partner is still online, they are returned to the queue.
4. The disconnected user is fully removed.

4.5 Message Exchange

1. User sends a message over WebSocket.
2. System looks up the paired user from Redis.
3. Message is forwarded in real-time.
4. If the recipient is disconnected, the sender is notified.

5. Conclusion

This architecture ensures scalability, efficiency, and maintainability using a clean architecture approach. Redis allows real-time queuing, WebSockets enable seamless communication, and the system is designed to handle high concurrency while ensuring a smooth user experience.