CSC 431 - Spring 2025

Bridge Chat

Software Architecture Specification (SAS)

Sophia Mantegari	(sam52464@miami.edu, (904) 772-5804)
Justin Bonner	(bjb63041@miami.edu, (443) 721-6184)
Andrea Venti Fuentes	(aav66@miami.edu, (954) 952-2239)

Version History

Version	Date	Author(s)	Change Comments
1.0	Mar 19	Justin Bonner	Initialization
1.1	Mar 30	Andrea Venti	Bug Fixes, figures added
1.5S	Apr 8	Sophia	Changes to design, presentation made
2.0	Apr 14	Justin Bonner	More figures and tables added
2.1	Apr 15	Andrea Venti	Translation Service API Details added, organization

Table of Contents

Version History **2**

Table of Contents **3**

Table of Tables **4**

Table of Figures **5**

1. System Analysis 6

1.1 System Overview 6

1.2 System Diagram: Bridge Chat Architecture 6

1.3 Actor Identification 6

1.4 Design Rationale 7

1.4.1 Architectural Style 7

1.4.2 Design Pattern(s) 7

1.4.3 Framework 7

1.4.4 Framework Diagram 8

2. Functional Design 9

2.1 Sequence Diagram: Message Translation Flow 9

2.1.1 Login Authentication and Flow 9

2.1.2 Messaging & Translation Flow 9

2.1.3 Session Termination 10

2.2 Sequence Diagram: Message Translation Flow 10

3. Structural Design 11

3.1 Class Diagram Concepts (High-Level) 11

3.2 Class Diagram Figure 12

4. Non-Functional Requirements 13

5. Translation Service API Details 14

5.1 Translation API & LibreTranslate Integration 14

Table of Tables

Table No.	Title	Page
Table 1	Version History	2
Table 2	Table of Contents	3

Table of Figures

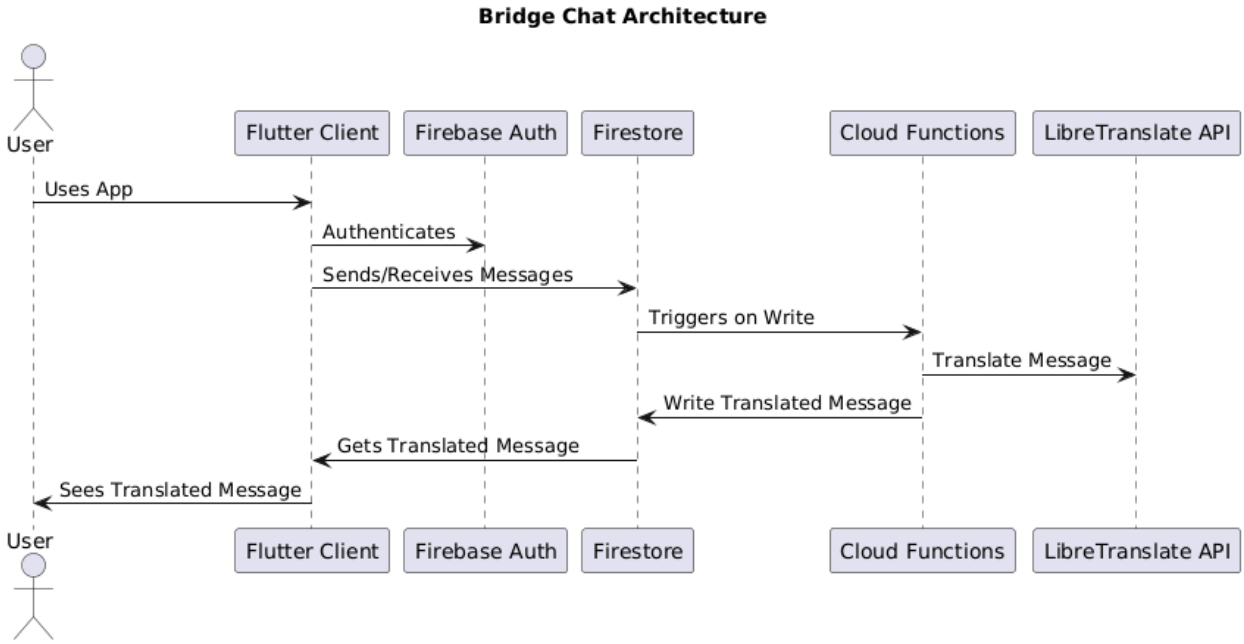
Figure No.	Title	Page
Figure 1	System Diagram: Bridge Chat Architecture	6
Figure 2	Framework Diagram	8
Figure 3	Sequence Diagram: Message Translation Flow	10
Figure 4	Class Diagram Figure	12

1. System Analysis

1.1 System Overview

Bridge Chat is a unified messaging platform that integrates communication services such as iMessage, WhatsApp, Discord, and GroupMe into a single interface. It aims to streamline communication by enabling users to send and receive messages from multiple platforms in one place. Key features include real-time AI translation, end-to-end encryption, chat categorization, and notifications. The system architecture follows modular design with a focus on scalability, usability, and security.

1.2 System Diagram: Bridge Chat Architecture



1.3 Actor Identification

- **User**: Registers, sends messages, views translations.

- **Firebase** Auth: Handles login and account security.

- **Firestore**: Stores user data, conversations, and messages.

- **Cloud Functions**: Executes backend logic (e.g., translation).

- **LibreTranslate API**: Provides language translation.

1.4 Design Rationale

1.4.1 Architectural Style

Client-Server + Event-Driven Architecture: Bridge Chat is architected as a client-server model using Flutter clients that interact with Firebase services over the web. Event-driven architecture is utilized through Firebase Cloud Functions, which react to message events (e.g., triggering translation when a message is written to Firestore).

1.4.2 Design Pattern(s)

Facade Pattern: Abstracts away external APIs (LibreTranslate) into unified translation handlers.

Observer Pattern: Real-time updates and message streaming rely on Firestore's pub/sub-like capabilities.

MVVM/MVC Style Separation: UI logic (Widgets), service layers (chat_service.dart, auth_service.dart), and models are cleanly separated.

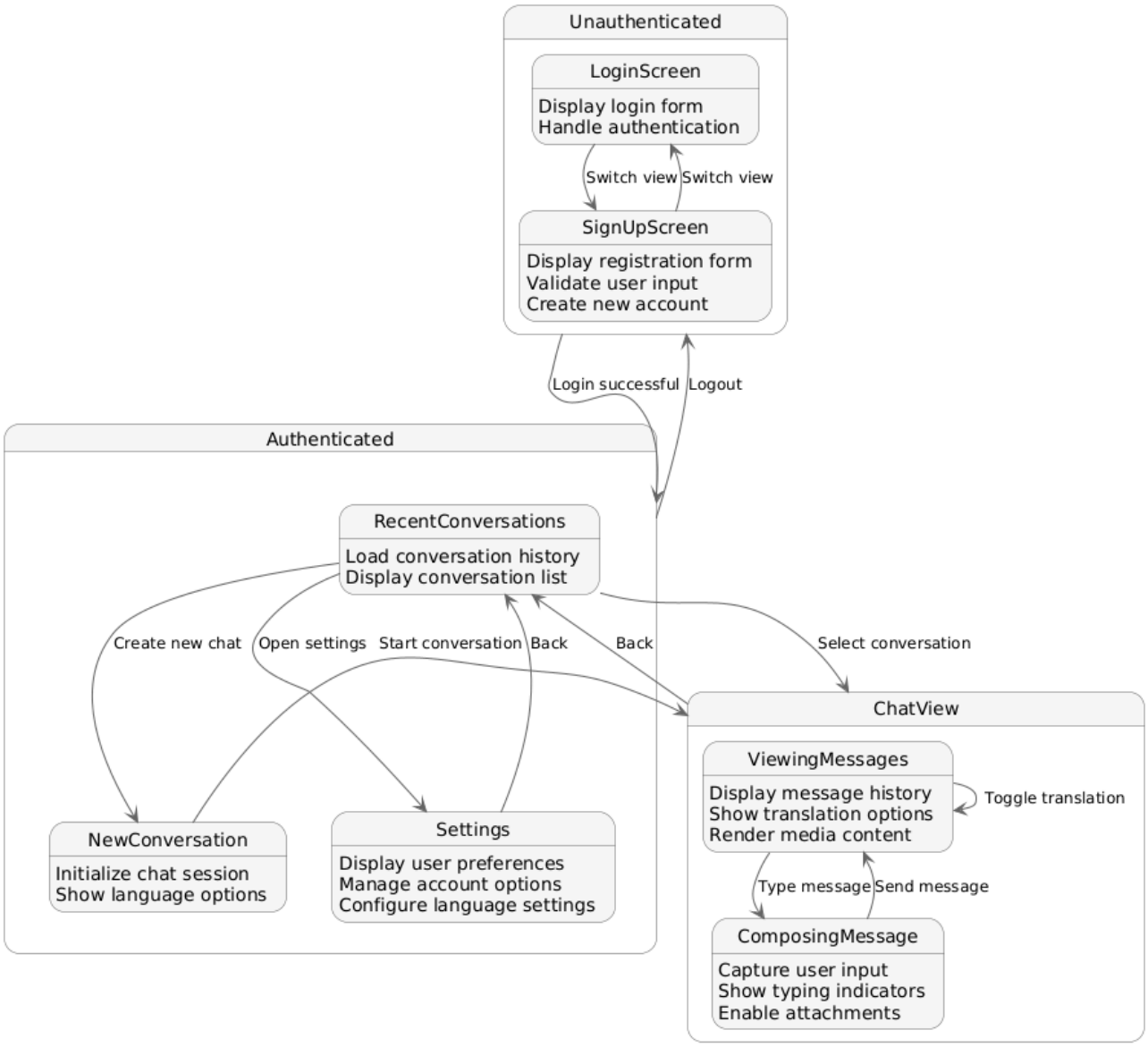
1.4.3 Framework

Flutter: For a cross-platform frontend experience (Web, iOS, Android, Desktop).

Firebase (Auth, Firestore, Functions, Hosting): For real-time backend, serverless logic, and web hosting.

LibreTranslate API: For privacy-preserving machine translation (open source), hosted separately, and no costs associated with it.

1.4.4 State Diagram



2. Functional Design

2.1 Sequence Diagram: Message Translation Flow

The following diagram illustrates the step-by-step message flow within Bridge Chat, from user login to message translation and delivery. This functional breakdown reflects the dynamic behavior of the system during typical usage.

2.1.1 Login Authentication and Flow

- When a user opens the app, the Message Client initiates a login request with credentials.
- The Authentication service verifies credentials and either returns session data (on success) or sends back an authentication error.
- If login fails, the user is prompted to retry or cancel.
- On successful authentication, the client requests and receives a user list from the Database.

2.1.2 Messaging & Translation Flow

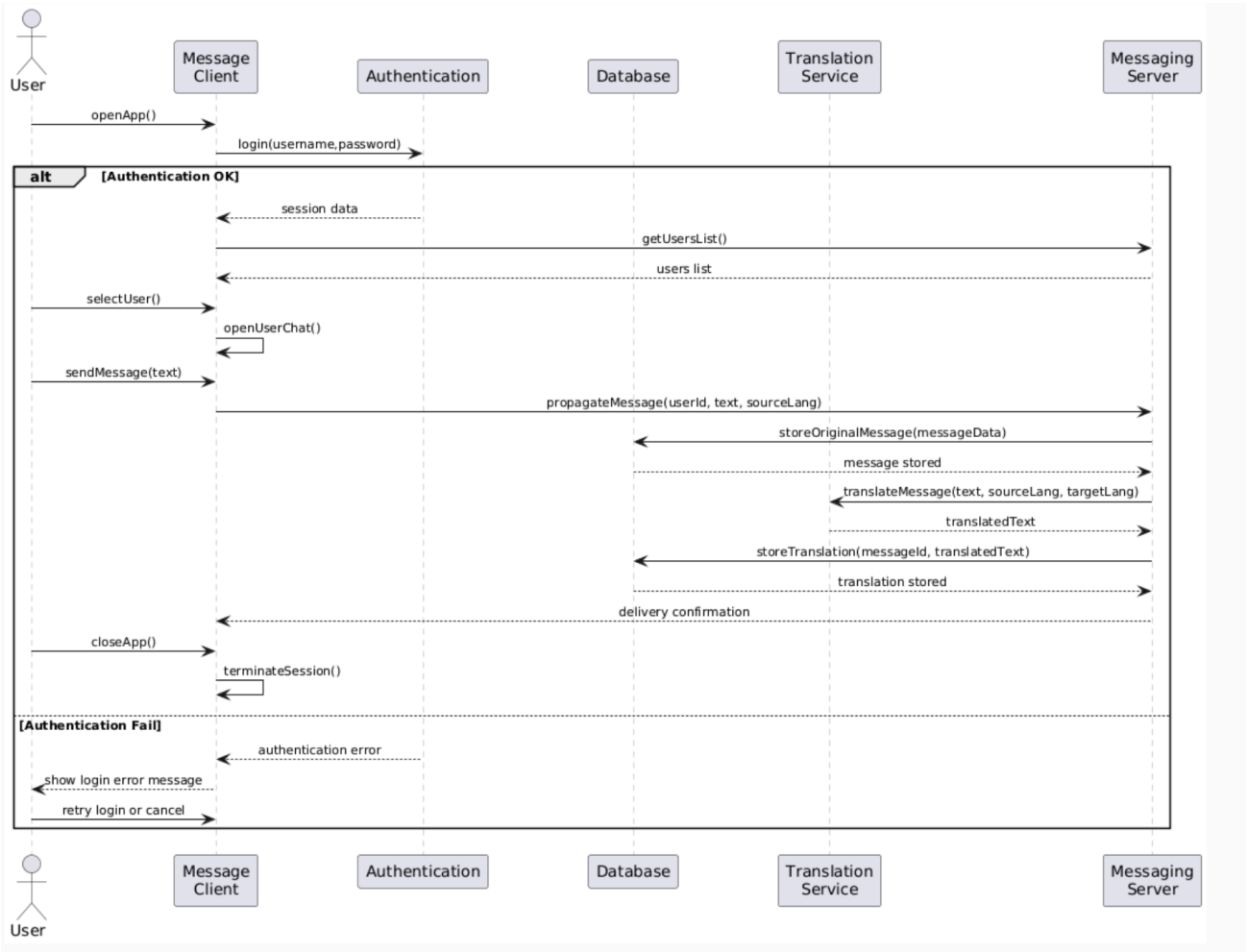
- The user selects a recipient, which opens the relevant chat view.
- Upon sending a message, the client propagates it along with metadata (e.g., source language) to the Messaging Server.
- The server first stores the original message via the Database service.
- A Cloud Function then invokes the Translation Service, which performs language translation using LibreTranslate.
- The translated message is stored in the database, linked to the original message.
- Once both the original and translated texts are saved, the Messaging Server confirms delivery back to the client.

2.1.3 Session Termination

- When the user closes the app, a termination request is sent, closing the session cleanly.

This functional flow leverages Firebase Authentication for login, Firestore for message and user storage, and external translation APIs triggered by serverless functions. Real-time data syncing ensures that messages and translations are reflected near-instantaneously across devices.

2.2 Sequence Diagram: Message Translation Flow



3. Structural Design

3.1 Class Diagram Concepts (High-Level)

User:

- id, email, username, preferredLanguage

Message:

- senderId, originalText, translations, timestamp, sourceLang

Conversation:

- participants[], isGroup, groupName, messages[]

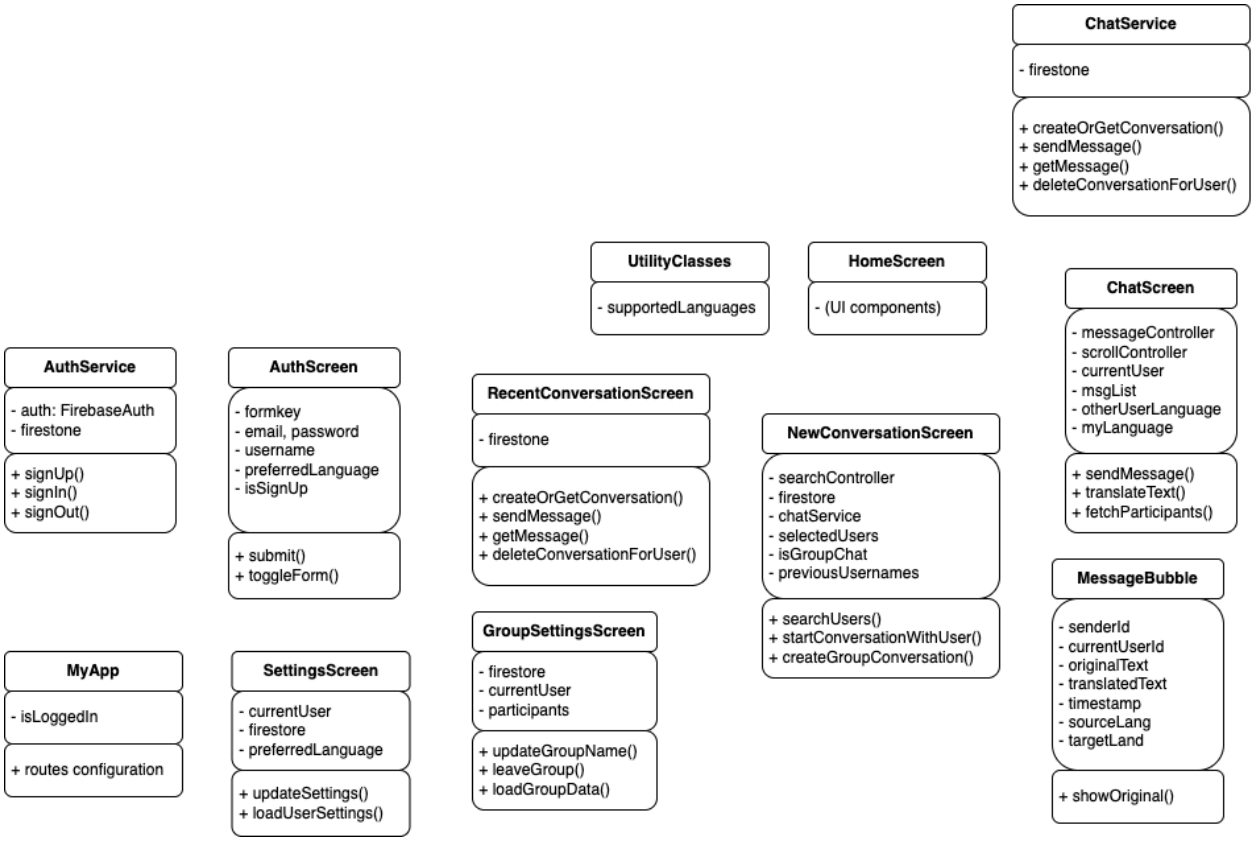
AuthService:

- signUp(), signIn(), signOut()

ChatService:

- createOrGetConversation(), sendMessage(), getMessages()

3.2 Class Diagram Figure



Note: There is no inheritance in our classes

4. Non-Functional Requirements

Performance: Uses Firestore's low-latency syncing for real-time messaging.

Scalability: Firebase Hosting + Firestore scale automatically with usage.

Security: Firebase Auth, Firestore rules, secret-managed API access.

Usability: Responsive UI using Flutter + day/night message grouping + clean onboarding UX.

Maintainability: Modular codebase with services, separation of logic, CI workflows via GitHub Actions.

5. Translation Service API Details

5.1 Translation API & LibreTranslate Integration

- **API Used:**
    - Bridge Chat uses the LibreTranslate API, an open source machine translation engine that enables real-time translation of messages across different languages.
- **Cost Considerations:**
    - **LibreTranslate is Free:**
        - The core advantage of using LibreTranslate is that it's free and open source. When you self-host LibreTranslate, you avoid per-character or per-request charges that are often associated with paid translation services like Google Translate.
    - **Self-Hosting vs. Paid Services:**
        - You can self-host the LibreTranslate instance on your own server, which means you'll only need to cover the hosting costs (if any) rather than paying for every translation request. For small-scale or hobby projects, using a community-provided instance might be sufficient and completely free.
- **Advantages Over Google Translate:**
    - **Cost Efficiency:**
        - Google Translate is typically available through paid APIs, which charge based on usage (characters or requests). In contrast, LibreTranslate eliminates these variable costs.
    - **Privacy and Data Control:**
        - LibreTranslate is a self-hosted solution that ensures user data remains private and under your control. This aligns well with Bridge Chat's emphasis on secure and private communication.

- **Open Source & Community Driven:**
    - As an open source project, LibreTranslate allows for customization, auditing of code for security, and avoids vendor lock-in, giving you more flexibility in how you manage and integrate translation features.

- **Connection Details:**
    - **Simple Integration Using URL:**
        - The connection to LibreTranslate is established by a simple HTTP request to an API endpoint. In the code, the URL is specified for the hosted translation service as a constant.