TED UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

CMPE 492 - Senior Project Low-Level Design Report

Submission Date

13.10.2024

Group Members:

Bartu Özen **22804498074**

Elif Nazlı Böke **11863076002**

Gülce Ayşe Döker **16483072168**

Toygar Yurt **4012670916**

Table of Contents

1. Introduction

Communication represents one of the most fundamental aspects of human interaction, facilitating the exchange of emotions and information between individuals. Furthermore, the diversity of languages spoken globally represents a significant challenge to communication between individuals with different native languages.

The main purpose of this system is to remove the limiting linguistic barriers that affect individuals' real-time communication. Polyglot Chat aims to ease text messaging and voice calls in different languages through a synchronous translation system. It is basically a system which translates the written or spoken language in real-time implementing LLM, eliminating the need to switch between external translation apps and providing a comfortable communication environment for individuals.

The purpose of this report is to provide a detailed description of the low-level architectural design of the Polyglot Chat. The report includes Object design trade-offs, Interface documentation guidelines, Engineering standards, Definitions, acronyms, and abbreviations, Packages, and Class Interfaces.

1.1 Object design trade-offs

1.1.1 Usability vs. Functionality

In our application, the primary focus is usability. We want users to be able to access the app effortlessly and start using it immediately. To achieve this, we’ve minimized initial friction—such as extensive sign-ups or complex onboarding processes. The user interface will be designed for simplicity, with clear navigation that users can understand instantly. While additional functionality may offer more features, we intentionally avoid overwhelming users with too many options to keep the experience quick and straightforward. As a result, we prioritize essential features that are easy to access and use over a more feature-rich, potentially complex system.

1.1.2 Efficiency vs. Accuracy

In our app, speed is critical, especially when dealing with real-time data updates and user interactions. We favor an efficient solution that can process tasks swiftly, even if it results in some trade-offs in precision. For instance, when displaying data or responding to user actions, it’s more important that the system remains responsive and fluid, even if some background processes or calculations are approximations rather than exact. This ensures users experience minimal delay and smooth operation, which is essential for maintaining engagement in dynamic use cases.

1.1.3 Security vs. Usability

Balancing security and usability is a key challenge. To protect user data and ensure a secure environment, we have implemented measures like session timeouts and data encryption. However, these precautions do require small sacrifices in convenience. For example, users may need to re-authenticate after a period of inactivity to prevent unauthorized access. While this may slightly interrupt the user experience, we believe it’s necessary to safeguard the integrity of the system and the users' privacy.

1.1.4 Reliability vs. Compatibility

Reliability is the cornerstone of our platform. We aim to provide a stable experience, ensuring that the app performs consistently across use cases without unexpected crashes or downtime. While we could expand compatibility to more platforms or environments, this could introduce complexity and increase the likelihood of bugs or performance issues. By focusing on a more selective range of supported platforms, we can ensure a more robust and dependable user experience. Reliability is our primary concern, as any instability could detract from user satisfaction and trust.

1.2 Interface documentation guidelines

1.2.1 Server

The interfaces for each subsystem (User Management, Translation, Communication, etc.) are documented with specific endpoints, input parameters, and expected outputs. The server-side will be built using .NET for backend management and Python for LLM integration.

1.2.2 Client

The Android app will be written in Kotlin. Therefore, syntax in class interfaces is similar to Kotlin's syntax. In Kotlin, property types, parameter types, method return types are written at the end separated by a colon.

In the framework that will be used for creating the user interface, screens are written as methods. Therefore, they are not mentioned in class interfaces.

1.3 Engineering standards

The software documentation follows IEEE standards for requirements and interface design. This ensures that the report is structured, including clear sections for purpose, scope, definitions, and specific requirements for functionality, performance, and quality. Unified Modeling Language diagrams are used to model the software architecture.

1.4 Definitions, acronyms, and abbreviations

API: Application Programming Interface

Channel: A tool that lets different parts of the app send messages to each other

LLM: Large Language Models

StateFlow: A way for the app to hold onto data and keep sending updates whenever it changes

ViewModel: A part of the app that stores data and communicates with the backend

2. Packages

2.1 Server

Translation Service: For real time translation of text and voice.

NotificationService: For notifications of incoming and outgoing messages.

Entity: Sets the entities of conversation, message, user and userConversation.

Data: Configures the database context and defines how the entities are structured and related to each other.

User Management: Manages user registration or authentication.

2.2 Client

RequestManagers: Responsible for handling network operations, such as making HTTP and handling errors.

PersistentStorage: A utility for saving and retrieving key-value data, providing get and put methods for storage operations.

Models: Define the data structure for objects like Contact, ChatMessage, and User.

ViewModels: Provide the logic for screens, connecting the user interface with the underlying data and services. Each ViewModel manages state, handles requests, and communicates events through channels.

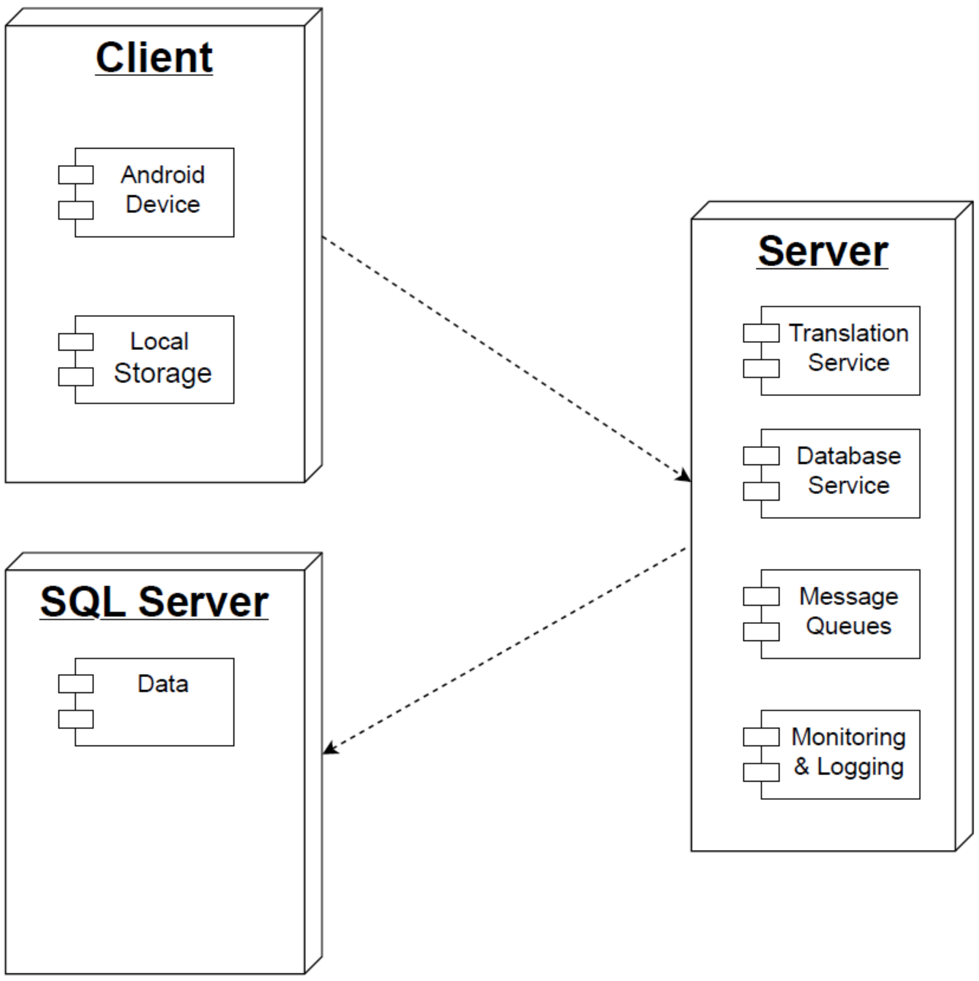

Figure 1: UML deployment diagram showing client-server communication.

3. Class Interfaces

3.1 Server

Entity

- Class: Conversation
  - Properties
    - int Id { get; set; }
    - ICollection<Message> Messages { get; set; }
    - ICollection<UserConversation> Participants { get; set; }

- Class: Message
  - Properties
    - int Id { get; set; }
    - string Content { get; set; }
    - string TranslatedContent { get; set; }
    - DateTime SentAt { get; set; } = DateTime.Now
    - bool IsTranslated { get; set; } = false
    - int SenderId { get; set; }
    - User Sender { get; set; }
    - int ConversationId { get; set; }
    - Conversation Conversation { get; set; }

- Class: User
  - Properties
    - int Id { get; set; }
    - string UserName { get; set; }
    - string PasswordHash { get; set; }
    - ICollection<UserConversation> Conversations { get; set; }
    - ICollection<Message> Messages { get; set; }

- Class: User Conversation
  - Properties
    - int UserId { get; set; }
    - User User { get; set; }
    - int ConversationId { get; set; }

        - Conversation Conversation { get; set; }

Data

- Class: ApplicationDBContext
    - Properties
        - DbSet<Conversation> Conversations { get; set; }
        - DbSet<Message> Messages { get; set; }
        - DbSet<User> Users { get; set; }
    - Methods
        - override void OnModelCreating(ModelBuilder modelBuilder)

Translation Service

- Class: TranslationService
    - Properties
        - supportedLanguages
        - translationAPI
    - Methods
        - translateText(text: String, sourceLang: String, targetLang: String): String
        - translateVoice(voice: file, sourceLang: String, targetLang: String): File
        - detectLanguage(text: String): String

Notification Service

- Class: Notification Service
    - Properties
        - List<notification>
    - Methods
        - sendNotification(userID: int, notificationText: String): Boolean
        - fetchNotification(userID: int): List<notification>

User Management

- Class: Auth

    - Properties
        - string userName
        - string email
        - string password
    - Methods
        - registerUser(email: String, userName: String, password: String): Token
        - loginUser(email: String, password: String): Token
- Class: Profile Management
    - Properties
        - int userID
    - Method
        - editProfile(userID: int)

3.2 Client

Managers

- **RequestManager**
    - **Properties:**
        - client: HttpClient
    - **Methods:**
        - get(path: String, parameters: List<String, Any>): HttpResponse
        - post(path: String, parameters: List<String, Any>): HttpResponse
        - login(email: String, password: String): Result<Unit>
        - register(username: String, email: String, password: String): Result<Unit>
- **PersistentStorage**
    - **Methods:**
        - <T> get(key: String) : T
        - <T> put(key: String, value: T)

Models

- **Contact**
    - **Properties:**
        - name: String
        - phoneNumber: String
- **ChatMessage**
    - **Properties:**
        - message: String
        - isSentByCurrentUser: Boolean
- **User**
    - **Properties:**
        - username: String
        - avatarUrl: String

ViewModels

LoginViewModel

- **Properties:**
    - requestManager: RequestManager
    - persistentStorage: PersistentStorage
    - eventChannel: Channel<Event>
- **Methods:**
    - login(email: String, password: String)
- **Subclasses:**
    - **Event**
        - **LoginSuccess**
        - **Error**
            - **Properties:**
                - message: String

RegisterViewModel

- **Properties:**
    - requestManager: RequestManager
    - persistentStorage: PersistentStorage
    - eventChannel: Channel<Event>
- **Methods:**
    - register(username: String, email: String, password: String)
- **Subclasses:**
    - **Event**
        - **RegisterSuccess**
        - **Error**
            - **Properties:**
                - message: String

ContactsViewModel

- **Properties:**
    - eventChannel: Channel<Event>
    - contacts: StateFlow<Contact>
- **Methods:**
    - updateContacts()
- **Subclasses:**
    - **Event**
        - **Error**
            - **Properties:**
                - message: String

ChatViewModel

- **Properties:**
    - requestManager: RequestManager
    - persistentStorage: PersistentStorage

    - eventChannel: Channel<Event>
    - userId: Int
    - recipientId: Int
    - messages: StateFlow<List<ChatMessage>>
    - translatedMessages: StateFlow<Map<Int, String>>
- **Methods:**
    - updateMessages()
    - retrieveTranslation(messageId: Int)
- **Subclasses:**
    - **Event**
        - **Error**
            - **Properties:**
                - message: String

ProfileViewModel

- **Properties:**
    - requestManager: RequestManager
    - eventChannel: Channel<Event>
    - userId: Int
    - user: User
- **Methods:**
    - updateUser()
- **Subclasses:**
    - **Event**
        - **Error**
            - **Properties:**
                - message: String

SettingsViewModel

- **Properties:**
    - requestManager: RequestManager

    - persistentStorage: PersistentStorage
    - eventChannel: Channel<Event>
    - user: User
- **Methods:**
    - updateUser()
    - setUserDetails()
- **Subclasses:**
    - **Event**
        - **Error**
            - **Properties:**
                - message: String

4. References

- https://www.acm.org/code-of-ethics
- https://www.lucidchart.com/pages/how-to-draw-a-deployment-diagram-in-uml#:~:text=Turn%20on%20the%20UML%20shape,instance%2C%20or%20a%20basic%20object.
- https://www.visual-paradigm.com/tutorials/how-to-draw-deployment-diagram-in-uml/
- https://app.diagrams.net/