Deliverable #2

SE 3A04: Software Design II – Large System Design

Tutorial Number: T02
Group Number: G2
Group Members: Chengze Zhao, Ganghoon (James) Park, Jack Walmsley, Luna Aljammal, Pranav Kalsi, Samih Dalvi

1 Introduction

1.1 System Purpose

The document offers a comprehensive overview of the *SecureChat* application's architecture, depicting the high-level design of the system at large as well as intricate design aspects of its constituent subsystems. This ensures that all involved parties have a clear and unified understanding of the high-level design.
The document is designed to serve the informational needs of the internal stakeholders of *SecureChat*, encompassing but not limited to the board of directors, the employees, and the members of the company at large. Prior familiarity with *SecureChat's* SRS as well as some technical knowledge in regard to the KDC is recommended for seamless understanding of this document.

1.2 System Description

The *SecureChat* system follows a Model-View-Controller (MVC) style architecture, with the Account Management, Key Distribution Center, and Message Management subsystems as the repository style design architecture. The MVC style architecture allows for the synchronization of multiple views with the same data set and model, allowing for ease of updating the data and user interface. This is the optimal choice as showing different messaging views, such as a page with different chats, and the chats themselves would be displayed as different views within the system.

The subsystems were chosen to follow the Repository style architecture, as a means of allowing concurrent access to the data stores, which will be repositories maintaining the keys, account, and message details. This allows for data stability when updating large data within the repositories.

1.3 Overview

Section 2 contains the analysis class diagram for the *SecureChat* application, providing a visual representation of the system's structure and the relationships between the different components. Section 3 of the document demonstrates the architectural design of the system, its sub-systems, justification for the choice, alternatives that were considered, as well as a brief discussion on the subsystems. Finally, section 4 presents the Class Responsibility Cards (CRC), which portray the classes, their responsibilities, and their interaction with other classes.

2 Analysis Class Diagram

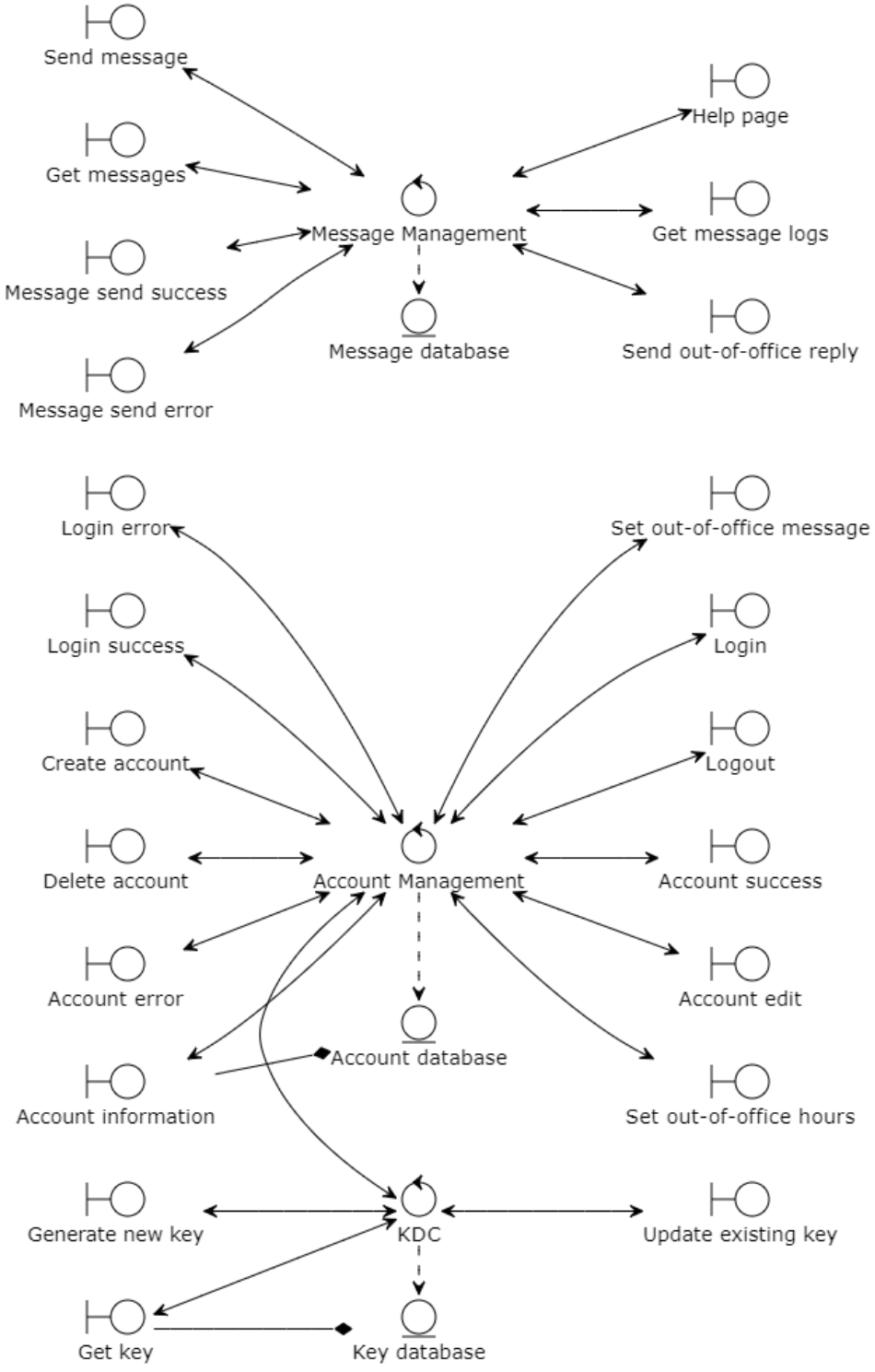

Figure 1. Analysis Class Diagram

3 Architectural Design

3.1 System Architecture

SecureChat will integrate the MVC (Model-View-Controller), specifically MVC-II, with the repository architecture style. The view module would represent the front-end development of the application whereas the controller and model module would be back-end, managing all user inputs, core functionality, and data. In this setup, the controller will register with the model module and initialize the view module. The view module will then register with the model module while the controller updates the model module, which notifies itself to update the view module and the controller. Moreover, the controller and the view module will provide a complete layer of the user interface along with user interactions. It will manage user input requests, such as selecting desired output displays and handling all initialization, instantiations, and registrations of other modules within the MVC system. The model module, on the other hand, will handle databases and data logic within the application. It will also notify the controller module of any updates to data and relay the information to the view module so that it can update the interface accordingly.

Within the controller, three subsystems were defined:

Subsystem	Purpose	Architectural Style
Account Management	Handle login, account creation, and account deletion	Repository
Key Distribution Center	Generate, store, and send keys for user communications	Repository
Message Management	Receive, send, and log user messages	Repository

Table 1. Subsystems within the system

The relationships between subsystems are defined in section 3.2.

The model modules for our system will interact with three databases, one for account information, one for message history log, and one for the keys. Within the account database, it will include information about the user, such as name, date of birth, employeeID, company email, etc. It will also include the key. The message history log database will include all chat history along with their respective timestamps.

The system architecture we chose for this project is a combination of the MVC-II and repository architectural styles. The reason for selecting MVC-II is because the system will need to support multiple synchronized views with a volatile GUI with frequency changes in data. It is suitable for interactive applications, such as SecureChat. This architecture also focuses on look-and-feel, which will make it more user-friendly and easier to use. In addition, it makes it easier to add new interfaces or changes to the existing system, making it more effective for development. Thus, this allows the system to be flexible and adaptable to updating the interface views. There are also many MVC frameworks and support documentations available. Considering the project's time constraint, MCV-II was more suitable. MVC-II separates the controller and the view modules, allowing separation of concerns and making the interaction between modules more stable. Moreover, MCV-II allows high cohesion and low coupling.

We chose the repository architecture style for the subsystems to streamline the exchange of data between the users and other system components. This architecture style allows the agents to initiate interactions in the system. The account management system follows the repository style since a user would actively log in to the system. As a result, the Account Database verifies the account information and proceeds to display error and success messages. This style is also applicable to the KDC subsystem, as the KDC server releases new keys after a user initiates a chat. In this case, the data store is passive, and awaits actions from the agents. The Message Management subsystem follows the repository style, as the actions must be taken by the agents sending the message first, which then calls the data store to log the chat. The overall system must be scalable to accommodate for more employees and chats as the company grows. Furthermore, the reusability of agents is a key feature that is required to allow users to participate in

multiple chats. Therefore, due to the large and complex information that is exchanged over multiple databases, and the importance of maintaining the integrity of the messages and account information the repository architecture style is optimal for the subsystems.

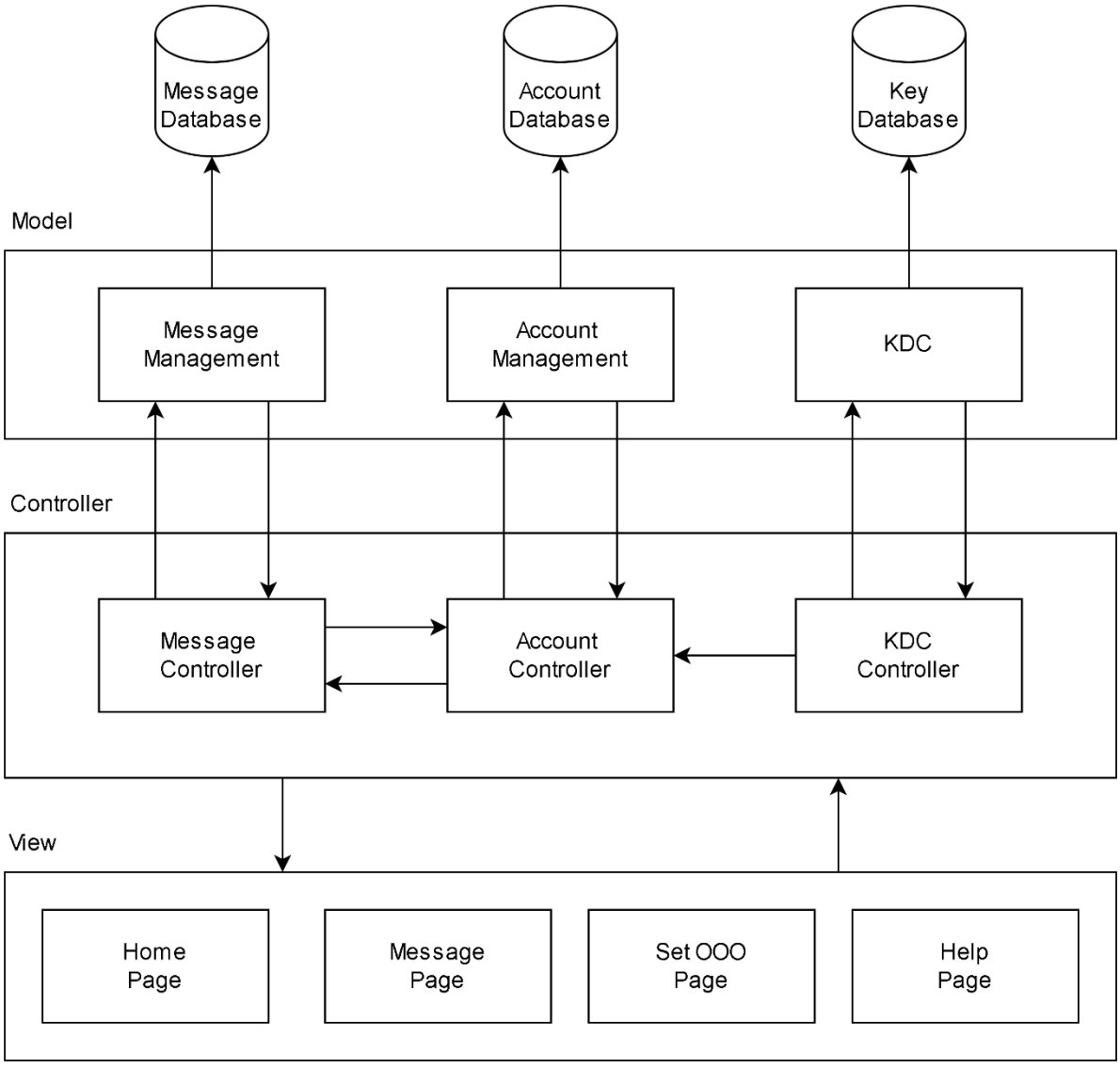

Figure 2. System Architecture

There were other design alternatives that we considered. These include blackboard, batch sequential, pipe and filter, process control, presentation-abstraction control (PAC), and master-slave architecture styles.

The blackboard architecture style was not used in this use case because the users are active agents whereas for blackboard, the data is active. The users are responsible for reading and writing to the repository. The whole premise of the chat app is around sending requests whether that be to get a key or send a message. Additionally, all the processes are independent meaning that one user's send chat action does not depend on another users' send chat action. The blackboard architecture style would be suited for a use case where the system is responsible for sending new data change or alerts to the users. Synchronization of multiple agents is also an issue for this architecture. In addition, there is tight dependency between blackboard and the knowledge source, which is unsuitable for this app.

The batch sequential architecture style does not make sense for this application as well as the data is not batched. This architecture would make more sense in applications where inter-submodule communication was done through files or some application along those lines. Additionally, systems with this architecture are seen with high latency which in turn means that a low a, these qualities are not suitable for a real-time messaging application. Especially in a high paced corporate environment, time is money and slowing down processes will hurt the organization's overall productivity.

The pipe and filter architecture style is suitable for application with data processing and data transformation such as graphics rendering. This architecture style involves using pipes between filters which perform some sort of operation on the data. Additionally, this architecture is not suitable for dynamic interactions. This whole app revolves around dynamic interaction which may be in the form of clicking a button or creating a request.

The process control architecture (PCA) is a style with a clear set use case which is embedded systems. SecureChat does not require data transformation, or any type of sensor input therefore making this architecture style unapplicable for this use case.

The PAC architecture style is similar to MVC but is based on a hierarchical system that is decomposed into many cooperating agents, where they each have three components: presentation, abstraction, and control. This system is most applicable for a distributed system that has agents with their own functionalities, data, and interactive interface. In other words, each agent has a specific role in a hierarchical structure. The reason this architecture style was not chosen was because of its increased complexity in the design and implementation of the system, its difficulty in determining the exact number of agents and their role due to loose coupling and high independence, its issues with overhead due to its method of communication, and the time constraint for this project.

The master-slave architecture also does not apply for this use-case as one node (master) controls the rest (slaves). The relationship between two users on the app or of the system and a user is that of equals. No “node” is controlling the behavior of other nodes. Instead, all user-user and user-system interaction is treated as equals.

3.2 Subsystems

Our system consists of three different subsystems, including Account Management, Key Distribution Center (KDC), and Message Management. The subsystems act as the controller class in the analysis class diagram and follow the principles of the repository style architecture within the MVC architecture of the system at large.

The Account Management subsystem is responsible for allowing the users to create an account, delete an account, logging in/out of the account, making modifications to the account (such as personal information). Additionally, it also enables the data to be stored in a database. This subsystem interacts with the KDC as well as the Message Management Subsystem to ensure correct functionality of the messaging app.

The KDC subsystem is responsible for the generation, retrieval, and modification (update) of the keys. This follows the repository style architecture as it generates keys upon user login, suggesting that the users would be active agents. This subsystem interacts with the Account Management subsystem to generate account specific keys as well as Message Management to enable users to send messages.

The Message Management subsystem allows users to send and receive messages, generate out-of-office replies, and keeps a logbook of the chat. It interacts with both the Account Management and the KDC subsystems to control how messages are sent and received from different accounts.

4 Class Responsibility Collaboration (CRC) Cards

Class Name: Account Management Controller (Controller)	
Responsibility:	Collaborators:
Knows Login Knows Log Out Knows Login Error Knows Login Success Knows Create Account Knows Delete Account Knows Account Error Knows Account Success Knows Account Information Knows Account Edit Knows Account Database Knows Set OOO Hours Knows Help Page	Log In Log Out Log in Error Log in Success Create Account Delete Account Account Error Account Success Account Information Account Edit Account Database Set Out-Of-Office Hours Help Page

Class Name: Login (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Handles click-event of “log in” button	Account Management Controller

Class Name: Logout (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows employeeID (username) Handles click-event of “log out” button	Account Management Controller

Class Name: Login Error (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows employeeID (username) Handles incorrect log in event Knows Account Database Displays “Help” button	Account Management Controller Account Database Help Page

Class Name: Login Success (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows employeeID (username) Handles log in success event Knows Account Database	Account Management Controller Account Database

Class Name: Create Account (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Handles click-event of "Create Account" button	Account Management Controller

Class Name: Delete Account (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows employeeID (username) Handles click-event of "Delete Account" button	Account Management Controller

Class Name: Account Error (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Handles incorrect credentials event (create account) Handles time-out event	Account Management Controller

Class Name: Account Success (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Handles successful events for creating an account	Account Management Controller

Class Name: Account Information (Entity)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows Account Database Knows employeeID (username) Knows Password Knows DOB (date of birth) Knows company email Knows department	Account Management Controller Account Database

Class Name: Account Edit (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows employeeID (username) Knows password Knows DOB Knows company email Handles click-event of “Save Information” button	Account Management Controller

Class Name: Account Database (Entity)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows Account Information	Account Management Controller Account Information Key Distribution Center

Class Name: Set OOO hours (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows Message Management Controller Knows employeeID Handles set hours of OOO hour event Handles set OOO message event Handles click-event of “Set OOO hours” button	Account Management Controller Message Management Controller

Class Name: Message Management Controller (Controller)	
Responsibility:	Collaborators:
Knows Send Messages Knows Get Messages Knows Message Send Success Knows Message Send Error Knows Help Page Knows Get Message Logs Knows Send OOO Reply	Send Messages Get Messages Message Send Success Message Send Error Help Page Get Message Logs Send out-of-office Reply

Class Name: Send Messages (Boundary)	
Responsibility:	Collaborators:
Knows account information Knows Message Management Controller Handles message sending events	Message Management Controller

Class Name: Get Messages (Boundary)	
Responsibility:	Collaborators:
Knows account information Message Management Controller Handles message receiving events	Message Management Controller

Class Name: Message Send Success (Boundary)	
Responsibility:	Collaborators:
Message Management Controller Handles successful sending of messages	Message Management Controller

Class Name: Message Send Error (Boundary)	
Responsibility:	Collaborators:
Message Management Controller Handles sending failures or errors	Message Management Controller

Class Name: KDC Controller	
Responsibility:	Collaborators:
Knows Generate New Key Knows Get Key Knows Update Existing Key	Generate New Key Get Key Update Existing Key

Class Name: Generate New Key (Boundary)	
Responsibility:	Collaborators:
Knows KDC Controller	KDC Controller

Class Name: Key Database	
Responsibility:	Collaborators:
Knows Keys Knows KDC Controller	KDC Controller Get Key

Class Name: Get Key (Boundary)	
Responsibility:	Collaborators:
Knows Key Database Knows KDC Controller	KDC Controller

Class Name: Update Existing Key (Boundary)	
Responsibility:	Collaborators:
Knows Key Database Runs automatically at specified intervals	KDC Controller

Class Name: Send OOO Hours Reply (Boundary)	
Responsibility:	Collaborators:
Knows Account Management Controller Knows Message Management Controller Knows employeeID Handles sending OOO hour message event	Account Management Controller Message Management Controller

Class Name: Help page (Boundary)	
Responsibility:	Collaborators:
Knows FAQs Knows Account Management Controller Handles “contact support” button event	Account Management Controller

Class Name: Get message logs (Boundary)	
Responsibility:	Collaborators:
Knows Message Database Knows Message Management Controller	Message Management Controller Message Database

A Division of Labour

Samih Dalvi

- Introduction
  - Section 1.1: Purpose
  - Section 1.3 Overview
- Section 3.1 Architecture
  - Discussed with the team why we should pursue MVC and repository architecture styles
  - Discussed why we should not use other architectural styles especially blackboard and PAC
- Section 3.2 subsystems
  - Helped identifying subsystems
  - Talked about the different subsystems (Account Management, KDC, and Message management)
    - Their responsibilities/purpose
    - Their interactions with the different subsystem
- Did the CRCs for:
  - Create Account
  - Delete Account
  - Account Error
  - Account Success
  - Account Information
  - Account Edit
  - Account Database
  - Help Page (along with Pranav, Luna, and James)
  - Get Message Logs
- Met with group to edit the document for deliverable submission
  - Provided some feedback to group members on some of the sections such as the analysis class diagram and how we can improve it

Pranav Kalsi

- Section 3.1: System architecture
  - Helped with decision of architectural style and how it would be beneficial to the application at large
  - Provided justifications for chosen architecture styles
  - Provided justifications for architectural styles that were not pursued:
    - Blackboard
    - Master-Slave
    - Batch-Sequential
    - Pipes and Filter
    - Process Control
- Section 4: CRC:
  - Helped with CRC for Help page (along with Samih, Luna, and James)
  - Helped with the CRC for Get Message Logs

Luna Aljammal

- Introduction
    - Section 1.2: System Description
- Section 3.1 Architecture
    - Discussed with the team which architecture styles to use (MVC II and Repository)
    - Identified discrepancies in the other architectural styles that would not match our product
    - Explained the reasoning behind using the repository architectural style for the subsystems (KDC, Account Management, and Message Management)
- Completed the following CRCs:
    - Log In
    - Log Out
    - Log In Error
    - Log In Success
    - Create Account
    - Help Page (along with Samih, Pranav, and James)
- Met with group to edit the document for deliverable submission
    - Identified some discrepancies in the collaborators of some of the CRCs
- Participated in the team editing sessions

Chengze Zhao

- Section 3.1: Discuss about which System Architecture should be used
- Section 4: Class Responsibility Collaboration (CRC) Cards
    - Message Management Controller
    - Send Messages
    - Get Messages
    - Message Send Success
    - Message Send Error
- Brainstormed architectural styles and CRC components
- Helped identify subsystems
- Participated in group discussions
- Helped review final document

Ganghoon (James) Park

- Section 3.1: Identified and explained the overall architecture of our system
- Section 3.1: Created structural architecture diagram showing the relationship among the subsystems
- Section 3.1: Explained elimination of PAC
- Section 4: Created CRC for Set OOO Hours
- Section 4: Created CRC for Send OOO Reply
- Brainstormed architectural styles and CRC components
- Formatted document
- Joined team meetings to edit and review the document
- Participated in the team editing session
- Contributed to team discussions and suggested ideas

Jack Walmsley

- Section 2: Analysis Class Diagram
- CRC Card for KDC Controller, Get Key, Update Existing Key, and Generate New Key
- Section 3.1: group discussion over architecture choice
- Section 3.2: discussed subsystem choice
- Brainstormed architectural styles and CRC components
- Helped identify subsystems
- Participated in group discussions
- Helped edit final document