Deliverable #2 Template

SE 3A04: Software Design II – Large System Design

Tutorial Number: T03
Group Number: G06
Group Members:

- Virochaan Ravichandran Gowri
- Alex Yoon
- Noah Goldschmied
- Krish Dogra
- Leo Vugert

IMPORTANT NOTES

- Please document any non-standard notations that you may have used
  - *Rule of Thumb*: if you feel there is any doubt surrounding the meaning of your notations, document them
- Some diagrams may be difficult to fit into one page
  - Ensure that the text is readable when printed, or when viewed at 100% on a regular laptop-sized screen.
  - If you need to break a diagram onto multiple pages, please adopt a system of doing so and thoroughly explain how it can be reconnected from one page to the next; if you are unsure about this, please ask about it
- Please submit the latest version of Deliverable 1 with Deliverable 2
  - Indicate any changes you made.
- If you do NOT have a Division of Labour sheet, your deliverable will NOT be marked

1 Introduction

This section should provide an brief overview of the entire document.

1.1 Purpose

State the purpose and intended audience for the document.

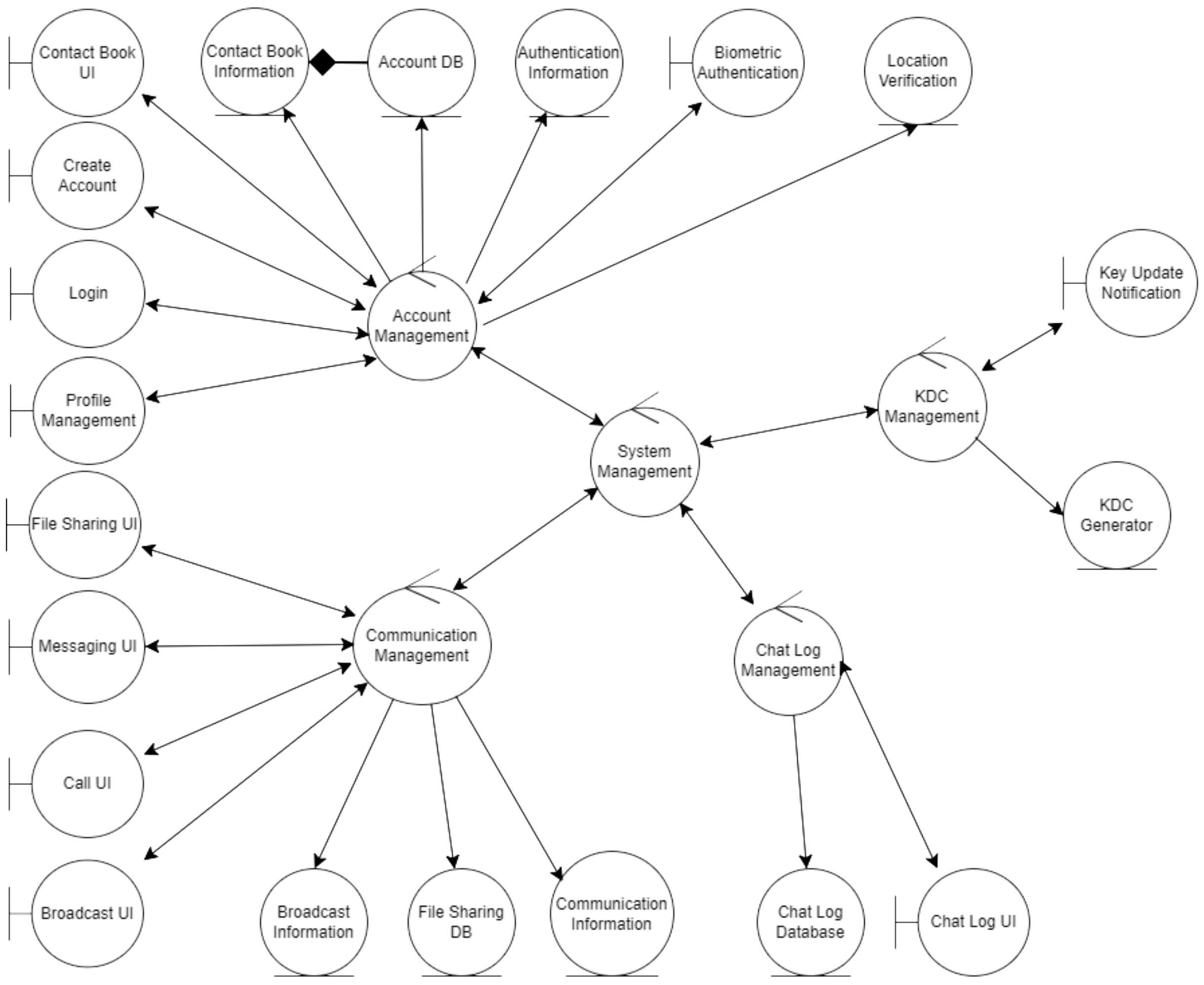
1.2 System Description

Give a brief description of the system. This could be a paragraph or two to give some context to this document.

1.3 Overview

Describe what the rest of the document contains and explain how the document is organised (e.g. ”In Section 2 we discuss...in Section 3...”).

2 Analysis Class Diagram



3 Architectural Design

This section should provide an overview of the overall architectural design of your application. Your overall architecture should show the division of the system into subsystems with high cohesion and low coupling.

3.1 System Architecture

The architectural design of the VANKL secure chat application is based on the Interaction-Oriented Architecture style, which uses the Model-View-Controller (MVC) pattern and incorporates some elements of the Repository Architecture Style. By combining these architectural paradigms together, the system will have a well-structured framework for managing real-time communication, user interactions, and data storage.

There a 3 primary subsections within the system that utilize their own distinct architecture style but still exist within the overall larger architecture style:

Subsystem	Purpose	Architectural Style
Account Management	Create, Login and Manage Account Information and Functions	Repository
Communication Management	Provides communication functionality between different agents and users.	Repository
KDC Management	Provides encryption and decryption functionalities	Pipe and Filter

The subsystems employ the Repository and the Pipe and Filter architecture styles and the relationship and functionality of the subsystems are further defined in section 3.2.

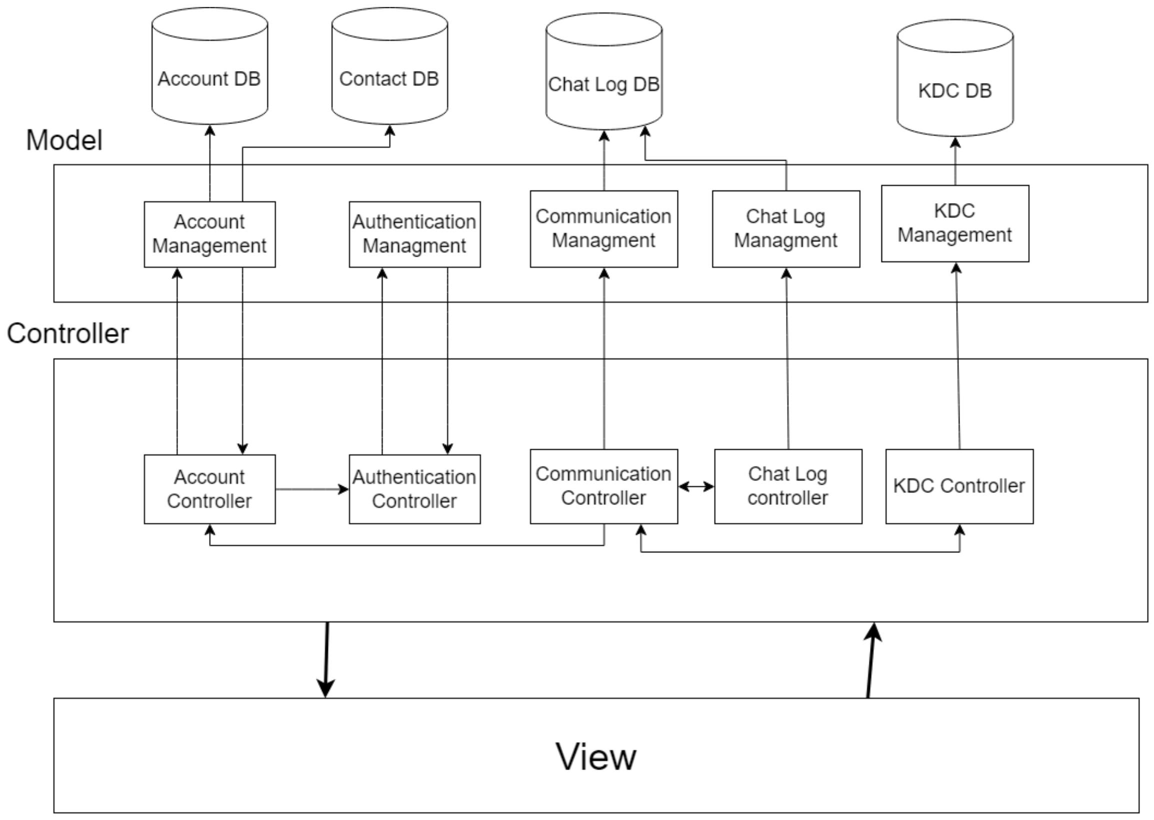
We also include 4 databases on the model level, an account database, a contacts database, a KDC database and a Chat Log database.

Firstly, the Model component utilizes the Repository Architecture Style, acting as the repository for the system's data storage and retrieval. This component manages the application's data entities, such as user profiles, chat messages, and metadata, all stored within a centralized repository. As a result, the repository pattern strengthens the application's data integrity, promotes efficient backup and restore operations, and provides a scalable and reusable structure for data management. Secondly, the View component is in charge of rendering the system's user interface and presenting all data retrieved from the Model component. It interconnects with both the Model and the Controller components in order to display up-to-date notifications/alerts, real-time chat conversations, and accurate chat history logs. Lastly, the Controller component plays the role of an intermediary between the View and Model components by handling user input and controlling the flow of data. Specifically, the Controller takes advantage of the Model's capabilities (including the Repository pattern) to process incoming user requests, update the mutually exclusive data, and trigger the corresponding updates in the View.

The rationale behind choosing the MVC Architecture Style is threefold - clear separation of concerns, ease of testing, and adaptability to changing requirements. One of the key benefits in using the MVC architecture is that it separates the application space into Model, View, and Controller, promoting modularity and maintainability. For example, when a user sends a message, the Controller component manages the input processing, the Model component then updates the relevant data (i.e. message content, sender, timestamp), and finally, the View component dynamically updates the chat's interface, thus enforcing a clear separation of concerns. Having a clear separation of concerns is essential in software development, as it allows developers to divide a complex system into independent modules, each assigned a specific responsibility, encouraging modular reusability for other projects. Additionally, an MVC architecture supports ease of testing for quality assurance, due to the separation of concerns characteristic. This separation allows software testers to isolate and test individual components independently, for example, unit tests can focus on the business logic within the Model, while integration tests can verify the interactions between the View and Controller. Lastly, using the MVC Style offers adaptability to changing requirements, by permitting flexibility in changing the user interface based on new security requirements or user preferences. For instance, if the development team decides to add Multi-Factor Authentication (MFA) as an extra layer of security, each component will be individually modified so as to provide the new MFA functionality, without compromising the overall structure.

The rationale behind choosing the Repository Architecture Style for certain subsystems is twofold - transaction support and scalability optimization. An important feature of the Repository pattern is that it supports atomic transactions linearly, which ensures data integrity during security-critical operations. For example, if a user chooses to update their profile or even send multiple messages (not necessarily simultaneously), the Repository will guarantee that these operations are atomic. This essentially means that if any operation were to fail (e.g. due to a security violation), then the system will rewind back to the previous state, maintaining data consistency and integrity. Furthermore, the Repository Architecture supports optimized data storage and retrieval, which enhances the scalability and performance aspects of the system. As the number of registered users increases, the repository is optimized to handle increased user profiles and message volumes. Security measures, including secure encryption and data control, are implemented efficiently in the repository to sustain performance levels, even as demand increases.

The Pipe and Filter Architecture Style was chosen specifically for the KDC since it provides efficient processing of data with a high throughput and flexibility. Since we are expecting many messages at the same time throughput is a crucial factor to consider and the pipe and filter architecture allows for easy scaling of throughput through concurrency. It also provides simplicity by sectioning different parts of the KDC into filters for encryption and decryption. Finally it also allows for reusability and room to expand by simplify modifying or adding more pipes and filters.



The Presentation-Abstraction-Control (PAC) Architecture was considered for its support of multiple agents and loose coupling, but it was ultimately rejected due to its complexity in agent management and communication. Due to the loose coupling between agents, it is a challenge when determining how many agents are required for optimal system performance. This pitfall combined with its high interdependence would lead to issues in managing user interactions, especially when user populations dynamically change.

Additionally, the Process-Control Architecture was evaluated as a potential architecture style but was also rejected, since it is designed for embedded systems and low-level control, rendering it irrelevant to the high-level interactions and data management requirements of the chat application.

The Blackboard Architecture Style was also considered but quickly ruled out, since it involves multiple agents contributing to a shared data structure (the blackboard) without any explicit control over the other. As a result, Blackboard possesses many issues in decision-making termination and synchronization, which is vital in a secure chat application. Furthermore, software testers have a tough time when it comes to debugging, as the lack of a clear execution path means it is not favorable to ensure timely and deterministic message delivery.

Another architectural pattern that was taken into account was the Batch Sequential Architecture, however was excluded, based on the fact that batched data processing is not typically suited for real-time interactions inherent in a chat application. Since the chat messages would be processed in batches at fixed intervals rather than in real-time, it would not be suitable for a dynamic chat environment with multiple users, lacking an interactive interface and low throughput.

3.2 Subsystems

We have 3 primary subsystems that are contained within the overall system which provide distinct functionalities and which also have their own distinct architectural styles for implementation. Our Three distinct subsytems include: Account Management, Communication Management Service, and KDC Managment.

The Account Management subsystem provides users with ability to create and login to their account and is responsible for ensuring that only authorized users account access the system by interfacing with the external apis. It also manages users contacts and interfaces with the communication management service to allow for communication and the authentication management for authentication management. It utilizes the repository architecture style.

The Communication Management Service provides users with the ability to contact other authorized user by sending and recieving messages. It also allows for the sending and recieving of files through file transfer, the reporting of messages and recieving of announcement board posts. It interacts with the authentication managment to ensure the authentication of users, the KDC Management for encryption and decryption of messages and the chat log managment and adheres to the Repository architecture style.

The KDC Management provides the encryption and decryption functions for the system. It interacts with the communication module to ensure a secure communication process and utilizes a pipe and filter architecture style.

4 Class Responsibility Collaboration (CRC) Cards

Class Name: System Management (Controller)	
Responsibility:	**Collaborators:**
Knows Account Management Knows Communication Management Knows Chat Log Management Knows KDC Management	Account Management Communication Management Chat Log Management KDC Management

Class Name: Communication Management (Controller)	
Responsibility:	**Collaborators:**
Knows System Management	System Management
Knows File Sharing UI	File Sharing UI
Knows Call UI	Call UI
Knows Broadcast UI	Broadcast UI
Knows Broadcast Information	Broadcast Information
Knows File Sharing DB	File Sharing DB
Knows Communication Information	Communication Information
Knows Communication DB	Communication DB

Class Name: File Sharing UI (Boundary)	
Responsibility:	**Collaborators:**
Knows Communication Management	Communication Management
Handles the User Interface for file sharing	
Handles the encryption and decryption of files	

Class Name: Messaging UI (Boundary)	
Responsibility:	**Collaborators:**
Knows Communication Management	Communication Management
Handles the User Interface for messaging	
Handles the encryption and decryption of messages	

Class Name: Call UI (Boundary)	
Responsibility:	**Collaborators:**
Knows Communication Management Handles the User Interface for calls Handles the encryption and decryption of calls	Communication Management

Class Name: Broadcast UI (Boundary)	
Responsibility:	**Collaborators:**
Knows Communication Management Handles the User Interface for broadcasts Handles the encryption and decryption of broadcasts	Communication Management

Class Name: Broadcast Information (Entity)	
Responsibility:	**Collaborators:**
Knows Communication Management Knows what users belong to broadcasts Knows which users can message in broadcasts	Communication Management

Class Name: File Sharing DB (Entity)	
Responsibility:	**Collaborators:**
Knows Communication Management Knows what users sent which files Knows which users can view individual files Knows where all files that have been sent are stored	Communication Management

Class Name: Communication Information (Entity)	
Responsibility:	**Collaborators:**
Knows Communication Management Knows what users belong to which chat Knows which users can message in which chat	Communication Management

Class Name: Chat Log Management (Controller)	
Responsibility:	**Collaborators:**
Knows System Management Knows Chat Log Database Knows Chat Log UI	System Management Chat Log Management Chat Log UI

Class Name: Chat Log Database (Entity)	
Responsibility:	**Collaborators:**
Knows Chat Log Management Knows what user sent each message Knows the identifiers of users and the date, time, and content of message	Chat Log Management

Class Name: Chat Log UI (Boundary)	
Responsibility:	**Collaborators:**
Knows Chat Log Management Handles presentation of Chat Logs	Chat Log Management