COMP90015 Projet2 2024S1 Report

Dinyuan Wu 1538073

Introduction

A distributed shared whiteboard system allows multiple users to interact and collaborate in real-time by drawing shapes, adding text, and erasing content on a shared canvas. This report details the design and implementation of such a system, focusing on key aspects including System architecture, communication protocol, concurrency management, and GUI implementation.

System Architecture

The central server system architecture for the distributed shared whiteboard application involves a single server managing all the state and coordination among multiple clients. This architecture ensures that the whiteboard's state is consistent and synchronized across all clients. The central server handles user management, drawing operations, messaging, and permissions.

Components

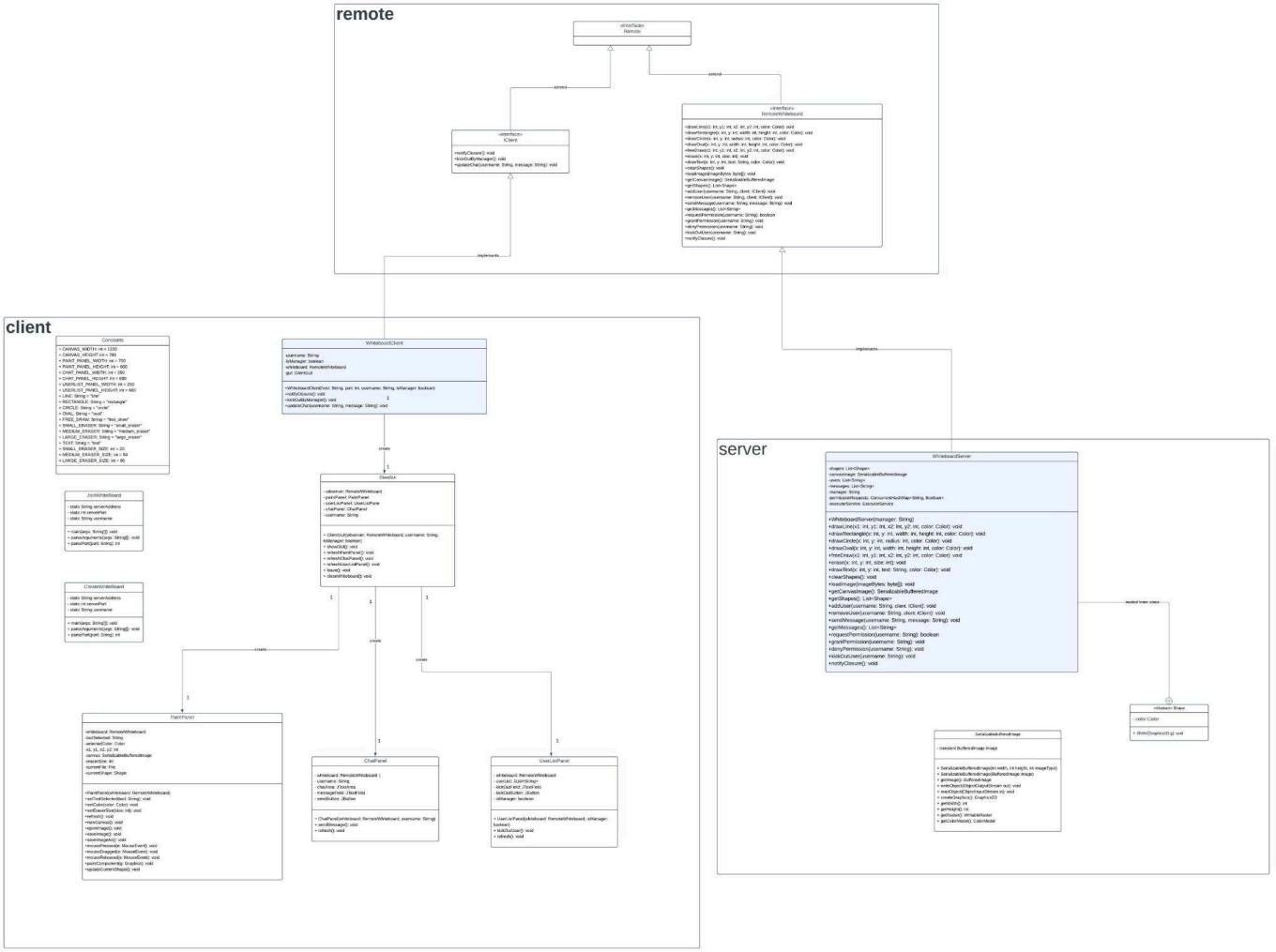
1. Central Server:
    - WhiteboardServer: This is the main server class that implements the `RemoteWhiteboard` interface. It handles all the remote method invocations from clients.
    - User Management: The server manages the list of connected users, including adding and removing users, and broadcasting user list updates.
    - Drawing Operations: The server maintains the current state of the whiteboard, including all shapes and drawings. It processes drawing commands from clients and updates the whiteboard state accordingly.
    - Chat Management: The server handles chat messages sent by users and broadcasts these messages to all clients.
    - Permissions and Roles: The server manages permissions for joining the whiteboard and handles role-specific actions such as kicking out users or closing the whiteboard.

2. Clients:
    - WhiteboardClient: Each client instance connects to the central server and interacts with it to perform various operations like drawing, messaging, and receiving updates.
    - Client GUI: The graphical user interface for the client, which includes the whiteboard canvas, chat panel, and user list panel.

System Workflow

1. Client Connection:
    - Clients connect to the server using Java RMI (Remote Method Invocation).
    - Clients request permission to join the whiteboard. The server processes these requests and grants or denies access.
2. Drawing Operations:
    - Clients send drawing commands (e.g., draw line, draw rectangle) to the server.
    - The server updates the whiteboard state and broadcasts the updated state to all connected clients.
3. Messaging:
    - Clients send chat messages to the server.
    - The server broadcasts these messages to all clients, ensuring everyone sees the same chat history.
4. User Management:
    - The server maintains a list of connected users and their roles (manager or regular user).
    - The server handles adding and removing users, and updates the user list for all clients.
5. Permission Management:
    - The server manages permission requests from clients wanting to join the whiteboard.
    - The manager can grant or deny these requests.
6. Session Management:
    - The manager can close the whiteboard session, which will notify and disconnect all clients.

Class Diagram



Advantages of Central Server Architecture

I chose this central server architecture for the following reasons:

1. Consistency and Synchronization: A central server ensures that the state of the whiteboard is consistent across all clients. Any changes made by one client are immediately reflected to all other clients. The server handles all updates, ensuring synchronization and avoiding conflicts that can arise in a decentralized architecture.
2. Simplified Client Logic: Clients are simpler and lighter as they only need to communicate with the central server and do not need to manage the state or communicate directly with each other. The server offloads complex tasks such as managing the whiteboard state, handling permissions, and broadcasting updates.
3. Centralized Control and Management: The server can enforce rules and permissions, ensuring that only authorized users can join and interact with the whiteboard. The manager has full control over the session, including the ability to kick out users or close the session, enhancing security and management.
4. Easier Maintenance and Updates: Maintenance and updates are easier to implement since changes need to be made only on the server side. Clients can be simpler and do not require frequent updates, reducing the complexity of maintaining multiple client versions.

Communication Protocol

The communication protocol in the distributed shared whiteboard system is based on Java Remote Method Invocation (RMI). RMI allows objects to communicate across a network, invoking methods on remote objects as if they were local. This section introduces how RMI is used in the system, the structure of communication, and the sequence of interactions between clients and the central server.

Components of RMI Communication

1. Remote Interfaces:
    - These define the methods that can be invoked remotely. In our system, `RemoteWhiteboard` and `IClient` are the primary remote interfaces.
    - Methods declared in these interfaces throw `RemoteException` to handle network-related issues.

2. Remote Objects:
    - These are the implementations of the remote interfaces. The server provides the implementation of `RemoteWhiteboard`, while each client provides an implementation of `IClient`.
    - Remote objects are exported to the RMI runtime, making them available to receive incoming remote calls.
3. RMI Registry:
    - The registry is a simple server-side name service that allows clients to obtain a reference to a remote object.
    - The server registers its remote objects with the RMI registry, associating them with names. Clients look up these names to obtain remote references.

Message Formats

1. Drawing Messages: contain information about the shape being drawn, its coordinates, dimensions, and color.

    drawLine(x1: int, y1: int, x2: int, y2: int, color: Color)
    drawRectangle(x: int, y: int, width: int, height: int, color: Color)
    drawCircle(x: int, y: int, radius: int, color: Color)
    drawOval(x: int, y: int, width: int, height: int, color: Color)
    freeDraw(x1: int, y1: int, x2: int, y2: int, color: Color)

2. Chat Messages: include the username of the sender and the message content.

    sendMessage(username: String, message: String)
    getMessages() -> List<String>

3. User Management Messages: These messages manage users joining, leaving, and being kicked out of the session.

    addUser(username: String, client: IClient)
    removeUser(username: String)
    getUserList() -> List<String>

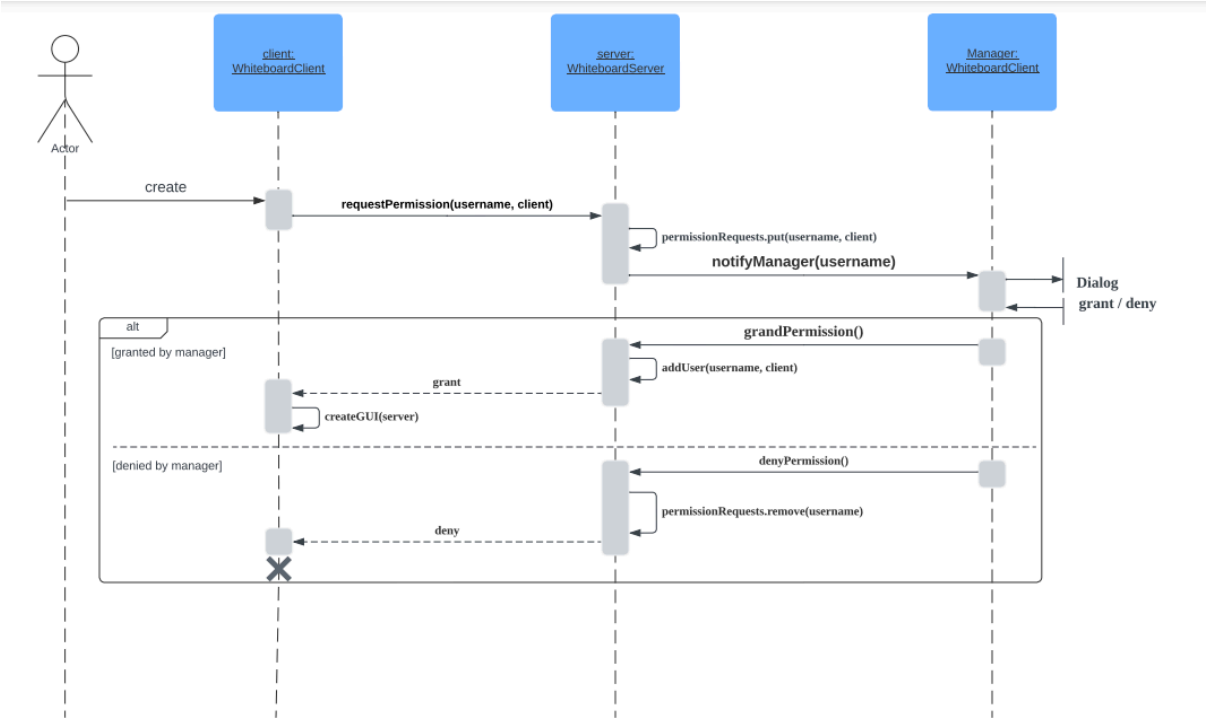
4. Permission Management Messages: handle requests for permission to join the whiteboard.

   requestPermission(username: String, client: IClient)
   grantPermission(username: String, client: IClient)
   denyPermission(username: String)

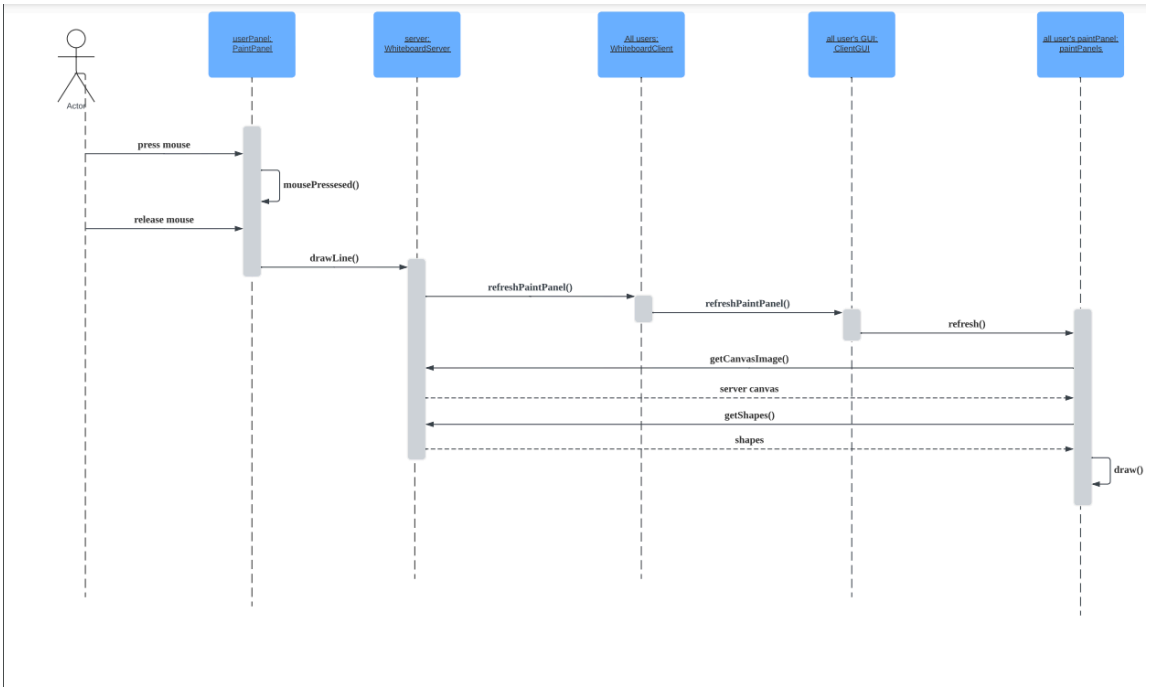
Sequence Diagrams

The sequence diagrams illustrate the communication flows in this system.

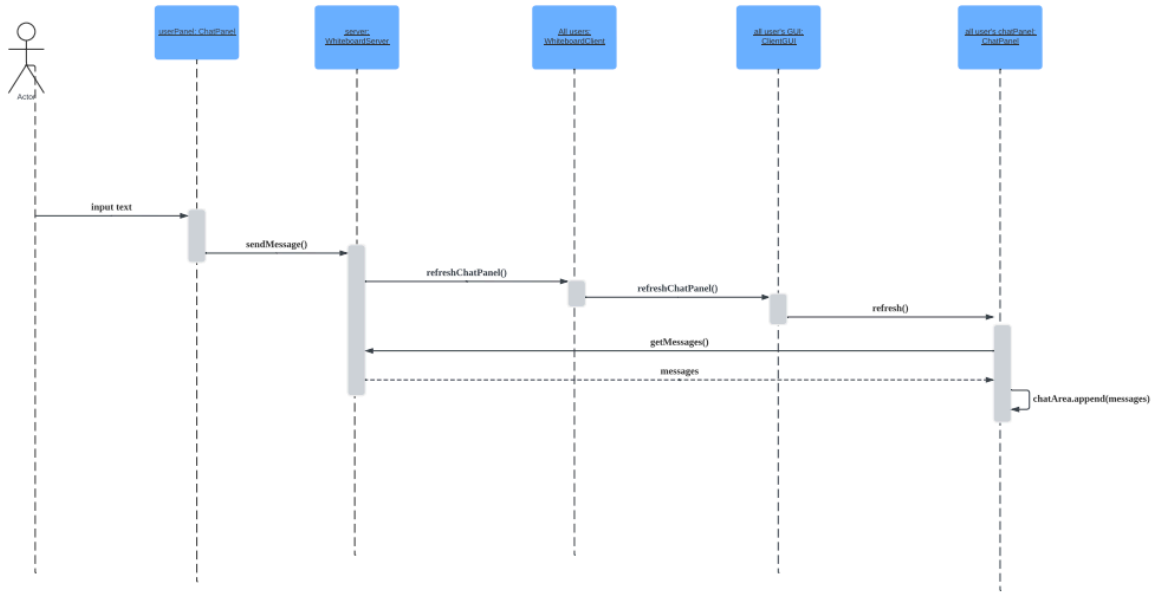
1. Join request



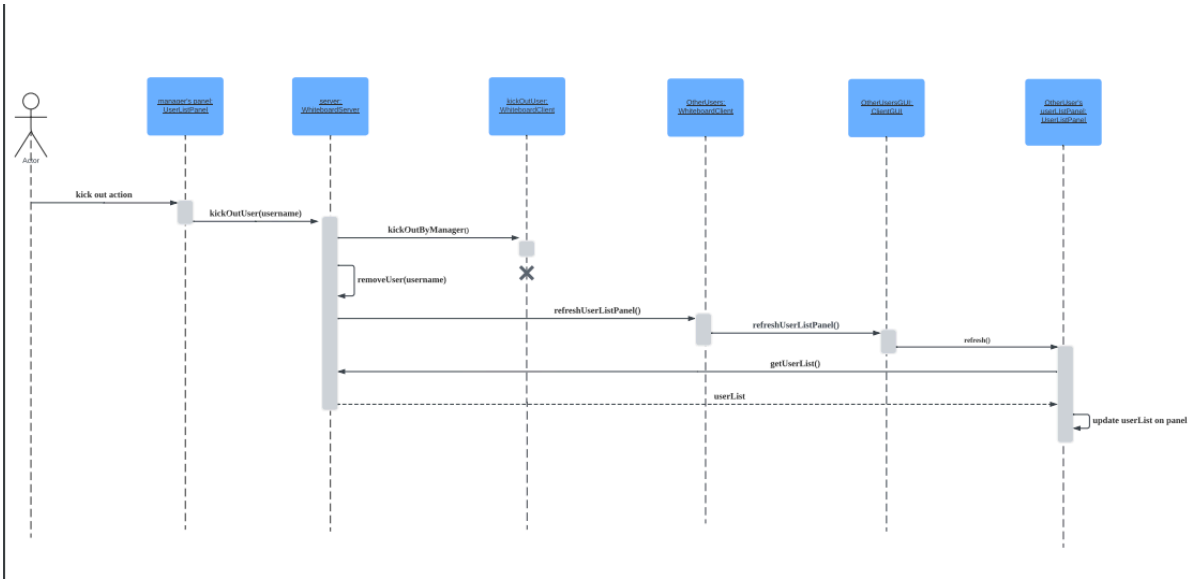
2. draw (here we use drawLine as example)



3. chat room



4. kick out user



Advantages of RMI

I choose use RMI for several reasons:

1. Simplicity: RMI abstracts the complexities of network communication, allowing developers to focus on application logic. Methods can be invoked on remote objects as if they were local, simplifying the development process.
2. Robustness: RMI handles the serialization and deserialization of objects, ensuring that complex data types can be transmitted over the network. Built-in exception handling (e.g., `RemoteException`) helps manage network failures gracefully.
3. Security: RMI can use Java's security manager to enforce access control policies, ensuring that only authorized clients can invoke methods on remote objects.

Concurrency Management

Dealing concurrency can be a challenge in this system, as there can be many users drawing and chatting at the same time, this part explains how concurrency is handled in the system.

Methods

1. Java RMI (Remote Method Invocation):
    - Java RMI handles concurrent remote method calls by creating a new thread for each incoming remote invocation. This allows multiple clients to interact with the server concurrently. RMI's built-in concurrency support simplifies the development of distributed applications by abstracting away the low-level threading details.
2. Synchronized Methods:
    - To ensure thread safety, methods in the `WhiteboardServer` class that modify shared resources are marked as `synchronized`. This includes methods for drawing shapes, adding or removing users, sending messages, and handling permission requests.
    - The `synchronized` keyword ensures that only one thread can execute a synchronized method on a particular object instance at a time. This prevents race conditions and ensures that the state of shared resources remains consistent.

Number of Threads per Peer

In this system, each peer (client) and the server manage their own threads. Here's a breakdown of the threading model:

1. Server:
    - The server uses one thread per remote method invocation. This is managed by the Java RMI runtime, which spawns a new thread for each incoming RMI request.
2. Client:
    - Each client has its main thread that interacts with the user interface.
    - When a client receives updates from the server (e.g., updates to the paint panel, chat messages, user list), these updates are handled in separate threads. This is achieved using the `SwingUtilities.invokeLater` method to ensure that updates to the GUI are performed on the Event Dispatch Thread (EDT), which is the standard way to perform GUI updates in Java Swing.

GUI Implementation

The graphical user interface (GUI) of the distributed shared whiteboard system is designed to provide a user-friendly experience for drawing, chatting, and managing users. The GUI is implemented using Java Swing, which offers a rich set of components for building interactive applications. This section introduces the structure and components of the GUI, highlighting how various features are implemented.

Main Components of the GUI

1. Main Frame (ClientGUI):
    - The main frame is the top-level container that houses all other GUI components.
    - It includes the whiteboard canvas, user list panel, chat panel, and toolbars for drawing and color selection.
2. Whiteboard Canvas (PaintPanel):
    - The canvas is where users draw shapes, text, and freehand drawings.
    - It supports various tools such as lines, rectangles, circles, ovals, and erasers of different sizes.
    - The canvas updates in real-time to reflect changes made by other users.
3. User List Panel (UserListPanel):
    - This panel displays the list of users currently connected to the whiteboard.
    - The manager can kick out users using this panel.
    - It is positioned on the left side of the main frame.
4. Chat Panel (ChatPanel):
    - The chat panel allows users to send and receive messages.
    - Messages are displayed in a text area, and a text field is provided for input.
    - It is positioned on the right side of the main frame.
5. Tool Panel and Color Panel:
    - The tool panel provides buttons for selecting different drawing tools (e.g., line, rectangle, free draw, text).
    - The color panel provides buttons for selecting different colors.
    - These panels are positioned at the top and bottom of the main frame, respectively.

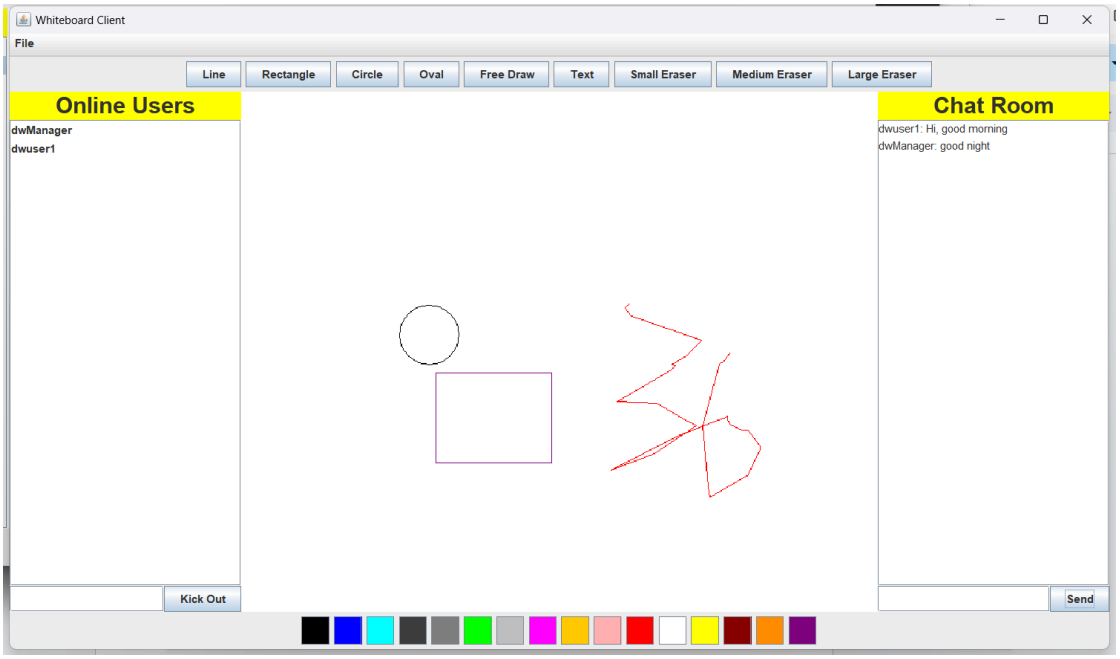
Integration and Interaction

- Event Handling: User interactions such as mouse events and button clicks are handled using listeners. For example, mouse events on the `PaintPanel`

trigger drawing actions, while button clicks on the tool and color panels update the selected tool and color.

- Real-Time Updates: The GUI updates in real-time based on changes from the server. Methods like `refreshPaintPanel`, `refreshChatPanel`, and `refreshUserListPanel` ensure that the latest state is displayed to the user.
- User Management: The `UserListPanel` allows the manager to manage connected users. The `kickOutUser` method sends a request to the server to remove a user, and the server notifies the user to disconnect.

The graph shows the GUI in use



Conclusion

The distributed shared whiteboard system employs a robust architecture centered on a single central server, ensuring consistent state management and seamless real-time collaboration. Java RMI facilitates efficient communication between clients and the server, enabling remote method invocations with structured message formats for drawing, messaging, and user management. Concurrency is handled effectively through synchronized methods, ensuring thread safety and data integrity. The GUI, implemented with Java Swing, provides an intuitive and responsive interface for users to draw, chat, and manage participants. This design balances simplicity, scalability, and security, making it an efficient solution for collaborative

applications. The system's modular architecture allows for easy maintenance and future enhancements, ensuring a reliable and user-friendly experience.