COMP90015 Distributed Systems

Assignment 2: Distributed Shared Whiteboard

MUHAMMAD DANIEL FAHMI 197325

The University of Melbourne

May 17, 2024

1. Problem Context

The Distributed Shared Whiteboard System addresses the need for interactive, real-time collaboration tools in remote settings by enabling multiple users to draw and interact on a shared virtual canvas. This system tackles significant challenges such as maintaining concurrency, ensuring synchronization across clients, and efficiently managing network communications to minimize latency. Additionally, it incorporates robust user management features that differentiate between regular users and managers, with managers having enhanced controls over session management. Utilizing Java Remote Method Invocation (RMI), the system facilitates remote operations across different machines, ensuring that user interactions are synchronized and consistent regardless of the user's location. The design aims to be user-friendly, adaptable across various devices, and supportive of an intuitive interface that accommodates a range of interactive tools for drawing and text input, making it suitable for a variety of collaborative scenarios in educational and professional contexts.

2. How to Run

To run the distributed shared whiteboard application using the provided JAR files (createwhiteboard.jar, joinwhiteboard.jar, and whiteboardserver.jar), you need to follow a straightforward process. First, ensure you have the Java Development Kit (JDK) version 8 or higher installed, along with the necessary JavaFX libraries if you are using JDK 11 or higher. Begin by starting the server. Open a terminal or command prompt, navigate to the directory containing the whiteboardserver.jar file, and run the command java -jar whiteboardserver.jar <serverIP> <serverPort>. This command will initialize the server and automatically start the RMI registry on the specified port. The default port is 1099.

Next, to launch the manager client, open another terminal or command prompt, navigate to the directory with the createwhiteboard.jar file, and execute java -jar createwhiteboard.jar <serverIP> <serverPort> <managerUsername>, replacing

<serverIP> with the server's IP address, <serverPort> with the port number (default is 1099), and <managerUsername> with a unique username for the manager. To run a participant client, open additional terminals or command prompts for each participant, navigate to the directory with the joinwhiteboard.jar file, and run java -jar joinwhiteboard.jar <serverIP> <serverPort> <username>, substituting the placeholders with the appropriate server IP, port number, and unique usernames for each participant.

For example, if the server is running on localhost, with the default port 1099, and you want to use manager1 and user1 as usernames, you would start the server with java -jar whiteboardserver.jar localhost 1099, the manager client with java -jar createwhiteboard.jar localhost 1099 manager1, and a participant client with java -jar joinwhiteboard.jar localhost 1099 user1. Ensure the server is running before clients attempt to connect and use unique usernames to avoid conflicts. By following these steps, you can successfully set up and run the distributed shared whiteboard application, enabling real-time collaboration among multiple users.

3. System Components

3.1. Server

For this project, all the codes are implemented using JDK version 20 and IntelliJ IDEA. Maven is used as build and dependency management tool. JavaFX 22.0.1 library is widely used for implementing various features in this project.

The server components of the Distributed Shared Whiteboard system focus on the SessionManager and WhiteboardServer classes, essential for managing user sessions and whiteboard state. The SessionManager maintains active and pending users, handles user registration, and distinguishes between regular participants and the manager. It ensures real-time updates by synchronizing the whiteboard state and broadcasting drawing actions via methods like canvasAction(). It also controls access through approveClient() and refuseClient(). The WhiteboardServer sets up the RMI environment, binding the SessionManager to a specific port to handle incoming RMI calls, and initializes the RMI registry, enabling network accessibility. These components together ensure robust session management and seamless client interaction, maintaining the shared drawing environment's integrity and consistency.

3.2. Client

The client components of the Distributed Shared Whiteboard system, primarily comprising the ClientController, WhiteboardClient, and GUI classes like CreateWhiteBoard and JoinWhiteBoard, enable user interaction with the shared whiteboard. The ClientController mediates between the user interface and server, managing user registration, drawing commands, and server updates, including user approval via setApproved(). It distinguishes between manager and regular user roles. The WhiteboardClient establishes the RMI connection, handling remote method execution from the server's WhiteboardInterface. GUI classes extend WhiteboardApp to customize the JavaFX application for different roles, setting up stages and scenes based on user privileges. These components work together to provide a responsive, real-time drawing experience, chat communication, and session management through an intuitive graphical interface.

3.3. Remote Interface

The common components of the distributed whiteboard application, specifically the remote interfaces ClientCallback and WhiteboardInterface, are essential for client-server communication. ClientCallback includes methods like updateUserList, updateCanvas, notify, receiveMessage, and deregister, which allow clients to receive real-time updates and notifications from the server. WhiteboardInterface provides methods for clients to invoke on the server, such as canvasAction, clearCanvas, registerUser, sendMessage, kickUser, createNewBoard, openBoard, saveBoard, closeBoard, approveClient, refuseClient, and deregisterCallback. These interfaces enable drawing, user session management, and whiteboard state handling, ensuring an interactive and synchronized experience. Utilizing Java RMI, they abstract network communication complexities, maintaining a consistent state across multiple users and enhancing collaboration.

3.4. Drawing

The drawing components of the distributed whiteboard application manage core functionalities related to drawing and user interactions on the canvas. The CanvasManager class oversees the entire drawing process on a JavaFX canvas, handling various drawing tools, managing user input, and ensuring drawing actions are correctly propagated to the server. It maintains a list of drawing actions for redrawing during canvas resizing or state synchronization. The CanvasManager collaborates with the CommandListener interface, enabling seamless communication between the canvas and other application parts. Drawing tools like FreeDrawTool, LineTool, CircleTool, RectangleTool, OvalTool, and EraserTool implement the DrawingTool interface, standardizing drawing operations. Each tool class encapsulates the logic for rendering specific shapes or performing actions like freehand drawing and erasing. Additionally, the TextTool facilitates text input on the canvas. These components provide a robust framework for managing drawing operations, ensuring real-time, collaborative interaction on the whiteboard.

3.5. User Interface (UI)

The UI components of the distributed whiteboard application create an intuitive and interactive user experience, seamlessly integrating collaboration functionalities. The UIManager class manages the layout and interaction of GUI elements, including the menu bar, tool palette, canvas area, chat window, and user list. The menu bar supports file operations and user session management, aligning with the manager's privileges. The tool palette offers various drawing tools, color pickers, and text input fields. The WhiteboardApp class, extending JavaFX’s Application, serves as the entry point, handling the application's lifecycle and resource management. The CanvasManager manages the canvas area, enabling users to draw shapes, freehand lines, and text. The chat window and user list in the sidebar facilitate communication and display active participants. Together, these components provide a cohesive, user-friendly interface supporting the application's rich interactive features.

4. Design Specification

4.1. Client Package

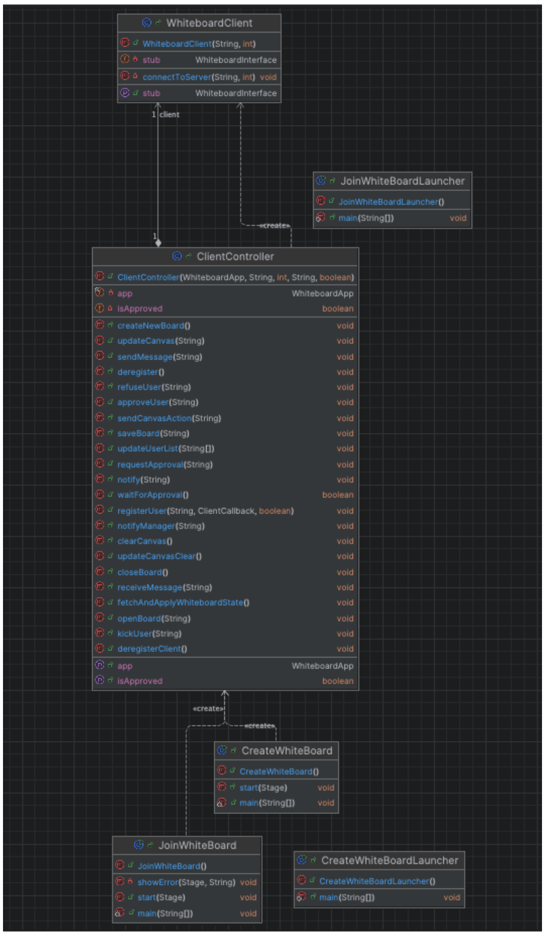


Figure 1: Client Package Class Diagram

Figure 1 shows how classes in the client package defined and interacts with each other. In the client package, the classes interact in a coordinated flow to enable user interaction with the distributed whiteboard. When the application starts, either CreateWhiteBoard or JoinWhiteBoard is launched, extending WhiteboardApp and setting up the JavaFX stage for managers or participants. These classes instantiate ClientController, which handles user registration and session management. ClientController establishes a connection to the server via WhiteboardClient, which connects to the server's RMI registry and accesses remote methods from WhiteboardInterface. During execution, ClientController processes user actions, such as drawing commands and chat messages, sending these to the server and receiving updates. It updates the GUI accordingly, ensuring real-time synchronization. The ClientController also handles user approvals and session termination, maintaining consistent state across all clients. This flow ensures a seamless, interactive experience for users in the shared whiteboard environment.

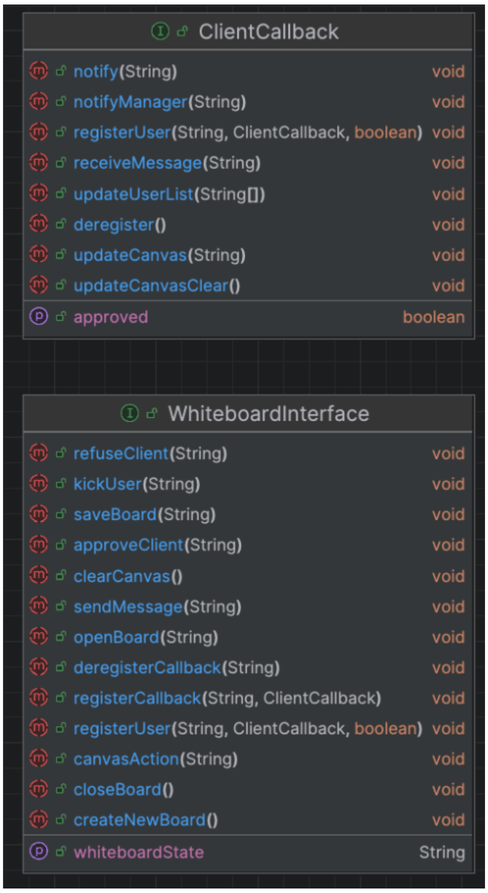
4.2. Common Package (Remote Interface)

In the client package, execution begins with either CreateWhiteBoard or JoinWhiteBoard, which extend WhiteboardApp to set up the JavaFX stage. These classes instantiate ClientController, responsible for user registration and session management. ClientController uses WhiteboardClient to connect to the server's RMI registry, invoking remote methods via WhiteboardInterface. During execution, ClientController handles user actions such as drawing and messaging, sending these to the server and processing updates from the server. The GUI is updated in real-time through ClientController, ensuring synchronization across clients. This interaction flow enables users to seamlessly create, join, and interact with the shared whiteboard.

The inter-process communication (IPC) format and protocol used in the project is Java Remote Method Invocation (RMI) as tabulated in Table 1. RMI allows methods to be called from one Java virtual machine to another, facilitating communication between the client and server components of the distributed whiteboard application. Figure 2 depicts the remote interfaces used for two-way communication between client and server.

Aspect	**Description**
Protocol	Java Remote Method Invocation (RMI)
Communication Type	Synchronous
Transport Layer	TCP/IP
Serialization	Java Object Serialization
Security	SSL/TLS (optional, not configured)
Components Involved	Client (**ClientController**, **WhiteboardClient**), Server (**SessionManager**)
Remote Interface	**WhiteboardInterface, ClientCallBack**
Servant Implementing Interface	**SessionManager, ClientController**
Key Methods	**canvasAction()**, **clearCanvas()**, **registerUser()**, **sendMessage()**, etc.
Registry	Java RMI Registry

Table 1: Java RMI Protocol



ClientCallback

notify(String)	void
notifyManager(String)	void
registerUser(String, ClientCallback, boolean)	void
receiveMessage(String)	void
updateUserList(String[])	void
deregister()	void
updateCanvas(String)	void
updateCanvasClear()	void
approved	boolean

WhiteboardInterface

refuseClient(String)	void
kickUser(String)	void
saveBoard(String)	void
approveClient(String)	void
clearCanvas()	void
sendMessage(String)	void
openBoard(String)	void
deregisterCallback(String)	void
registerCallback(String, ClientCallback)	void
registerUser(String, ClientCallback, boolean)	void
canvasAction(String)	void
closeBoard()	void
createNewBoard()	void
whiteboardState	String

Figure 2: Remote Interface Implementation

4.3. Server Package

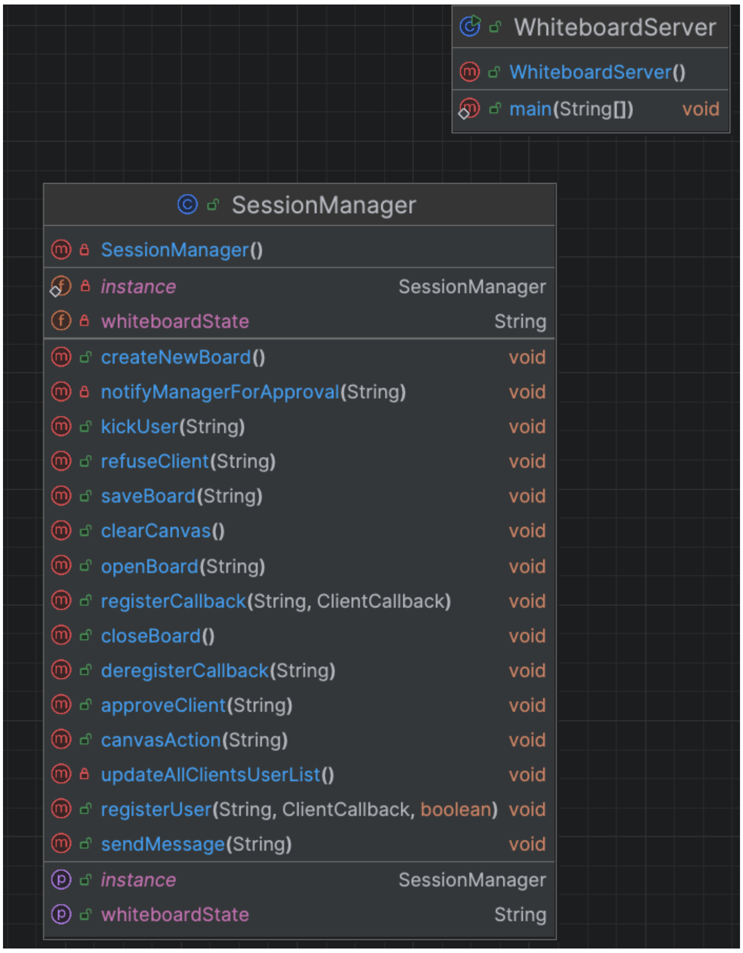


Figure 3: Server Package Class Diagram

In the server package, the classes SessionManager and WhiteboardServer are central to managing user sessions and the whiteboard state. Based on Figure 3, WhiteboardServer initializes the server environment, creating and binding the SessionManager instance to the RMI registry on a specified port, allowing it to accept incoming client connections. SessionManager implements the WhiteboardInterface, handling all server-side operations such as user registration, drawing actions, and session management. When a client connects, SessionManager processes registration via registerUser(), manages user approval with approveClient() or refuseClient(), and synchronizes drawing actions using canvasAction(). It also maintains the whiteboard state and broadcasts updates to all connected clients. This interaction ensures that user actions are consistently propagated and synchronized across the distributed whiteboard environment.

4.4. Drawing Package

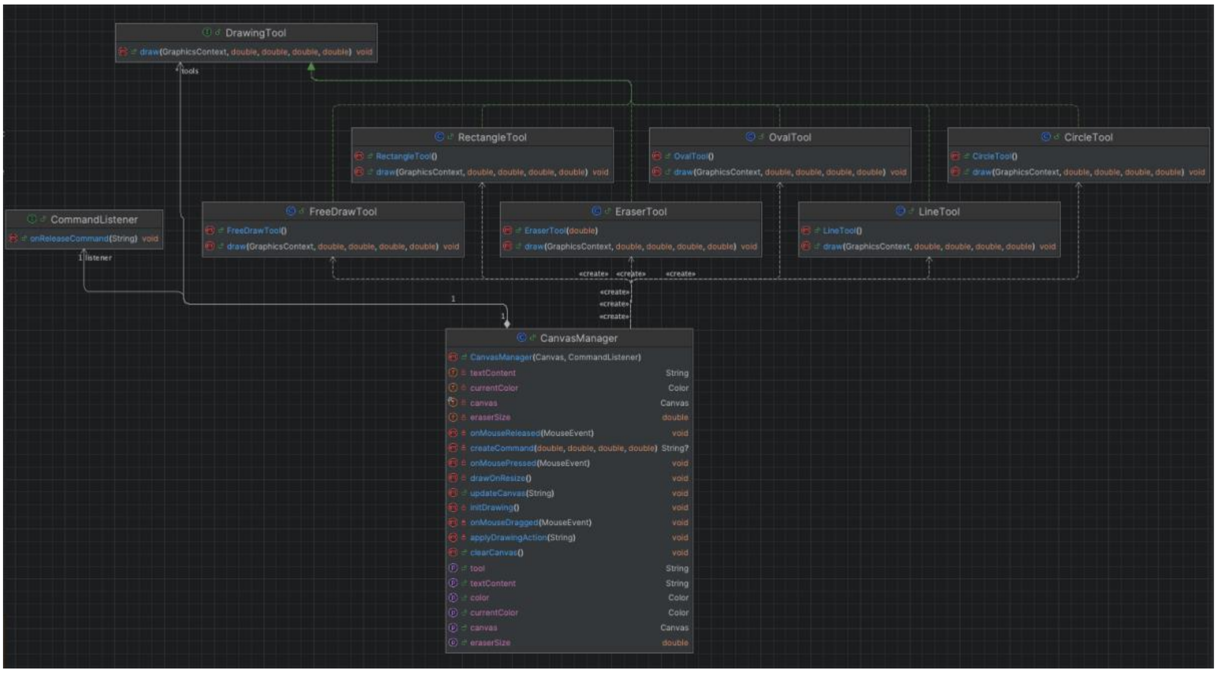

DrawingTool
draw(GraphicsContext, double, double, double, double) void
* tools
RectangleTool
RectangleTool()
draw(GraphicsContext, double, double, double, double) void
OvalTool
OvalTool()
draw(GraphicsContext, double, double, double, double) void
CircleTool
CircleTool()
draw(GraphicsContext, double, double, double, double) void
CommandListener
onReleaseCommand(String) void
1 listener
FreeDrawTool
FreeDrawTool()
draw(GraphicsContext, double, double, double, double) void
EraserTool
EraserTool(double)
draw(GraphicsContext, double, double, double, double) void
LineTool
LineTool()
draw(GraphicsContext, double, double, double, double) void
«create»
«create»
«create»
«create»
«create»
«create»
1
1
CanvasManager
CanvasManager(Canvas, CommandListener)
textContent String
currentColor Color
canvas Canvas
eraserSize double
onMouseReleased(MouseEvent) void
createCommand(double, double, double, double) String?
onMousePressed(MouseEvent) void
drawOnResize() void
updateCanvas(String) void
initDrawing() void
onMouseDragged(MouseEvent) void
applyDrawingAction(String) void
clearCanvas() void
tool String
textContent String
color Color
currentColor Color
canvas Canvas
eraserSize double

Figure 4. Drawing Package Class Diagram

In the drawing package, the CanvasManager class orchestrates the drawing process on the JavaFX canvas, managing user interactions and various drawing tools. It initializes event handlers for mouse actions, captures drawing commands, and communicates these actions to the server. CanvasManager interacts with the CommandListener interface, which handles drawing commands and ensures they are properly executed. The suite of drawing tools, including FreeDrawTool, LineTool, CircleTool, RectangleTool, OvalTool, and EraserTool, implements the DrawingTool interface, providing specific drawing functionalities. During execution, when a user selects a tool and performs a drawing action, CanvasManager invokes the appropriate DrawingTool method to render the shape on the canvas and then sends the command to the server for synchronization. This interaction ,as seen in Figure 4, ensures real-time, collaborative drawing across all clients.

4.5. UI Package

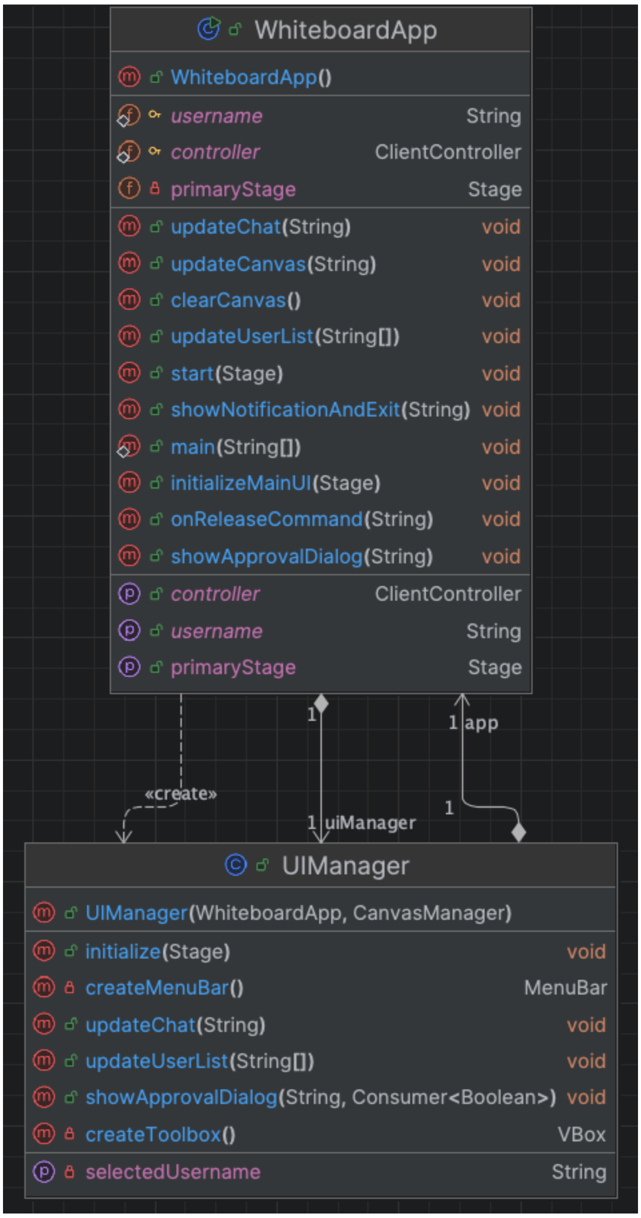
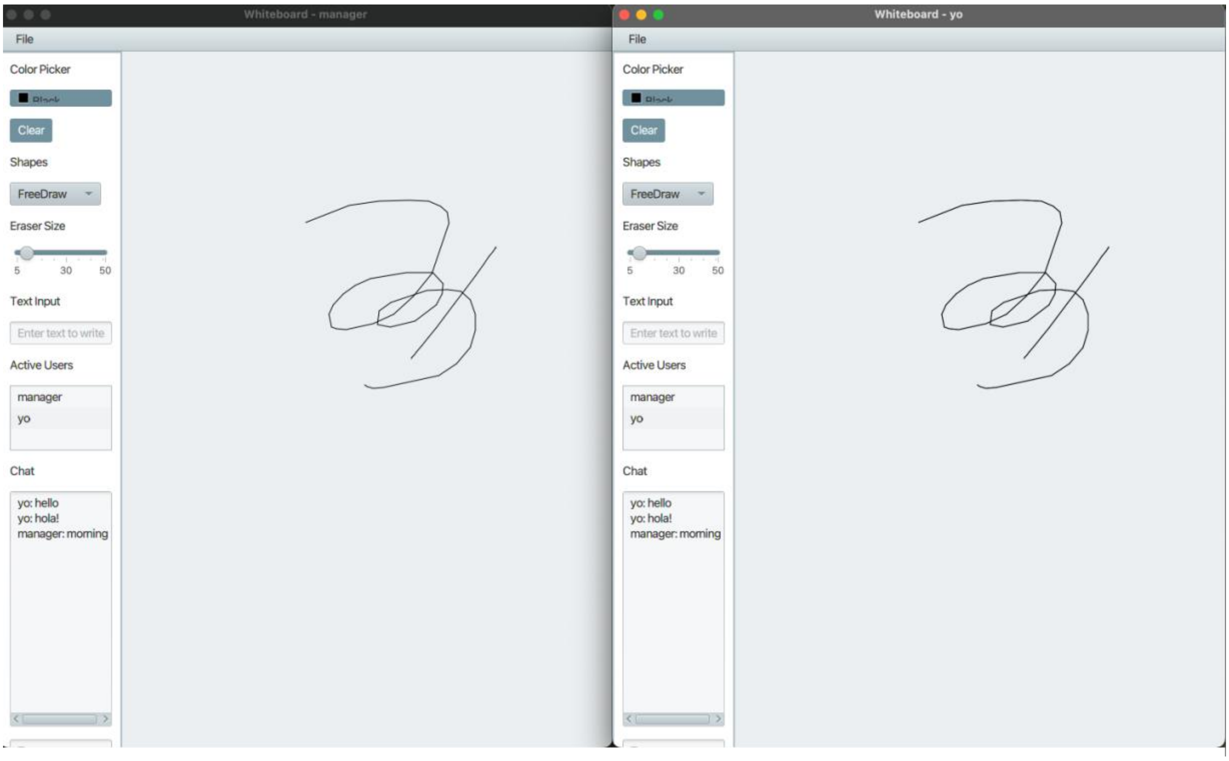

WhiteboardApp
WhiteboardApp()
username String
controller ClientController
primaryStage Stage
updateChat(String) void
updateCanvas(String) void
clearCanvas() void
updateUserList(String[]) void
start(Stage) void
showNotificationAndExit(String) void
main(String[]) void
initializeMainUI(Stage) void
onReleaseCommand(String) void
showApprovalDialog(String) void
controller ClientController
username String
primaryStage Stage
«create»
1
1 uiManager
1 app
1
UIManager
UIManager(WhiteboardApp, CanvasManager)
initialize(Stage) void
createMenuBar() MenuBar
updateChat(String) void
updateUserList(String[]) void
showApprovalDialog(String, Consumer<Boolean>) void
createToolbox() VBox
selectedUsername String

Figure 5. UI Package Class Diagram

In the UI package as shown in Figure 5, WhiteboardApp and UIManager are key classes that manage the user interface. WhiteboardApp, extending JavaFX's Application, serves as the entry point, initializing the JavaFX stage and setting up the primary UI layout. UIManager handles the layout and interaction of GUI elements, including the menu bar, tool palette, canvas area, chat window, and user list. During execution, WhiteboardApp sets up the main interface and delegates control to UIManager, which initializes and manages the different UI components. The CanvasManager from the drawing package is integrated to manage canvas interactions, while ClientController handles the communication with the server. UIManager ensures that user actions, such as drawing, messaging, and managing sessions, are seamlessly reflected in the GUI, providing a cohesive and interactive user experience. This setup allows for real-time collaboration and user management within the distributed whiteboard application.



manager
yo

yo: hello
yo: hola!
manager: morning

manager
yo

yo: hello
yo: hola!
manager: morning

Figure 6: Manager (left), and Regular User (right) Whiteboard UI

4.6. Interaction Diagram

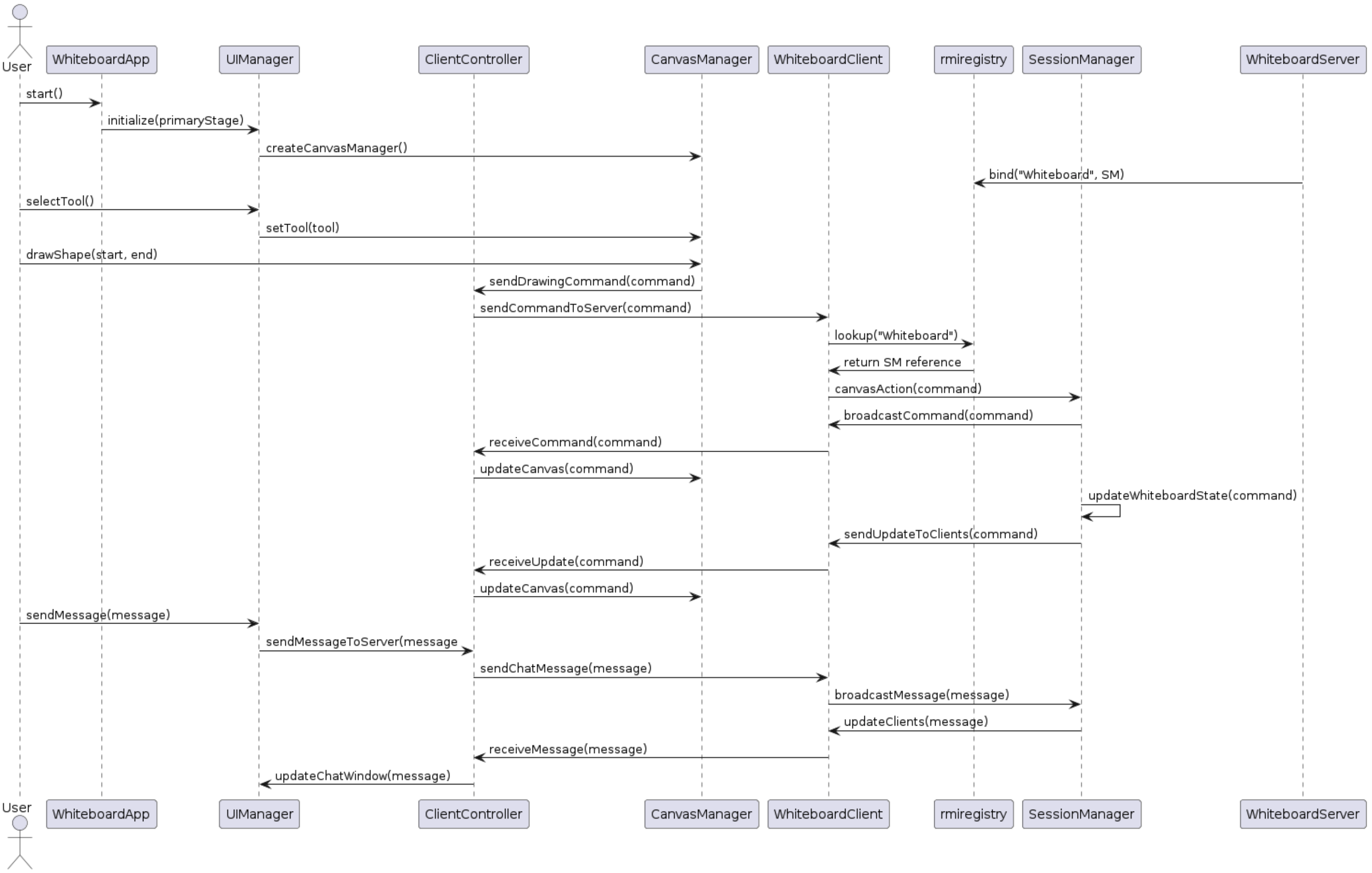


Figure 4: Interaction Diagram

5. New Innovations

5.1. Enhanced UI Design

The application features a well-organized and intuitive UI, managed by the UIManager class. The interface includes a user-friendly tool palette, color pickers, text input fields, and a responsive layout that adjusts dynamically to window resizing. These enhancements improve the overall user experience, making it easier for users to interact with the whiteboard.

5.2. Advanced Drawing Tools

In addition to basic shapes like lines, circles, rectangles, and ovals, the drawing tools include a FreeDrawTool for freehand drawing and an EraserTool with adjustable size, allowing users to erase specific parts of the canvas. These tools provide a richer set of drawing functionalities that enhance creativity and usability.

5.3. Real-time Collaboration

The implementation ensures that all users see real-time updates with minimal latency. The CanvasManager efficiently manages drawing actions and propagates them to the server, which then broadcasts these updates to all clients. This real-time collaboration feature is critical for maintaining a consistent and synchronized whiteboard experience across multiple users.

5.4. Session Management and User Roles

The implementation ensures that all users see real-time updates with minimal latency. The CanvasManager efficiently manages drawing actions and propagates them to the server, which then broadcasts these updates to all clients. This real-time collaboration feature is critical for maintaining a consistent and synchronized whiteboard experience across multiple users.

5.5. Integrated chat Feature

The chat window, managed by UIManager, allows users to communicate via text messages in real-time. This feature supports collaboration by enabling users to discuss their drawings, share ideas, and coordinate their efforts directly within the application.

5.6. Scalable Architecture

The use of Java RMI for remote method invocation provides a scalable and robust communication framework. This architecture allows the application to handle multiple clients simultaneously, ensuring that the system remains responsive and reliable even as the number of users grows.

5.7. Persistence and State Management

The application includes features for saving and loading whiteboard states, allowing users to persist their work and resume it later. This functionality is handled by the SessionManager class, which manages the saving and opening of whiteboard files, ensuring that all drawing actions are correctly restored.

5.8. Customizable Appearance

The inclusion of a style.css file allows for easy customization of the application's appearance. Users can modify the CSS to change the look and feel of the interface, catering to different aesthetic preferences and improving accessibility.