# COMP90015: Distributed Systems

# Assignment 2 Report

**Name: Sanya Shrivastava**
**Student ID: 1678877**

## 1. Introduction:

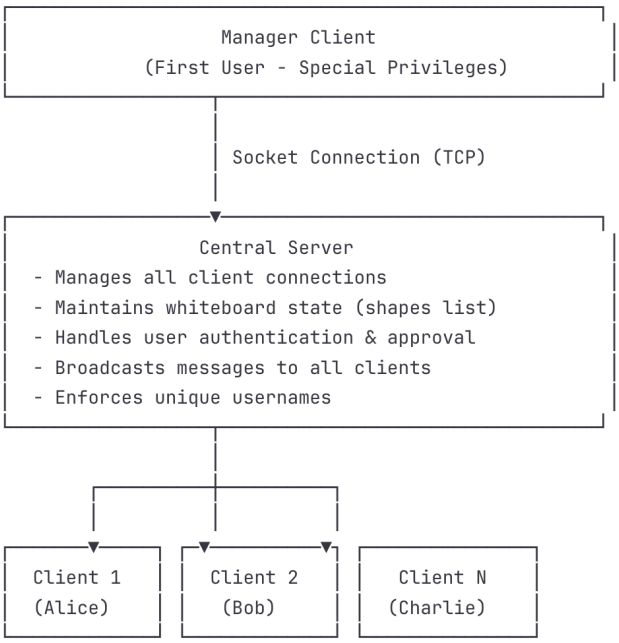
This project implements a distributed collaborative whiteboard system, enabling multiple users to draw and interact in real time. The system is managed by a designated manager, who sets up and oversees the whiteboard, while other users can join the session after receiving the manager's approval.

The platform includes a variety of features: core tools such as shape drawing, freehand sketching, an eraser, text input, color selection, stroke size adjustment, and real-time state synchronization. It also offers advanced capabilities like an integrated chat, participant approval and removal by the manager, file operations, and the option to close the board.

## 2. System Architecture

### 2.1 Overall Architecture

The system implements a centralized client-server architecture with the following components:




### 2.2 Component Description

**Server Component:**

- Listens on a designated port for incoming client connections.
- Launches a separate ClientHandler thread for each connected client.
- Maintains shared state, including the board's shapes, connected users, and pending join requests.
- The first user to connect is automatically assigned as the Manager.
- All other users must be approved by the manager before gaining full access.

**Client Component:**

- Connects to the server via TCP socket.
- Provides a GUI for drawing, chatting, and viewing connected users.
- Sends drawing actions, chat messages, and commands to the server.
- Updates the local GUI based on server notifications.
- Manager clients have additional privileges, including approving join requests, removing users, and performing file operations.

**Entry Point (WhiteboardMain):**

- Determines whether the application runs in CreateWhiteBoard or JoinWhiteBoard mode.
- Managers start the server first and then connect the client.
- Regular users connect directly to an existing server.

## 3. Communication Protocols and Message Formats

### 3.1 Communication Protocol

The system uses TCP sockets to ensure reliable, ordered delivery of messages. All messages are Java objects serialized and sent via ObjectOutputStream and ObjectInputStream.

**Connection Flow:**

1. The client establishes a TCP connection to the server.
2. The client sends a JOIN_REQUEST with its username.
3. The server handles the request as follows:
4. The first user is immediately assigned as the Manager.
5. All other users require the Manager's approval.
6. After approval, the server sends a FULL_STATE message to synchronize the whiteboard.
7. Once synchronized, the client and server exchange messages bidirectionally.

### 3.2 Message Format

- All messages follow a common structure defined in the Message class:

```
public class Message implements Serializable {
  public final MessageType type;    // Message category
  public final Object payload;      // Message data
  public final String from;         // Sender username
}
```

- MessageType is an enumeration defining 15 possible message categories:

  JOIN_REQUEST, JOIN_RESPONSE, MANAGER_APPROVAL_REQUEST, MANAGER_APPROVAL_RESULT, FULL_STATE, DRAW, CHAT, USER_LIST_UPDATE, KICK, KICKED, FILE_OPEN, NEW_BOARD, CLOSE_BOARD, SHUTDOWN, DRAWING_ACTIVITY

### 3.3 Message Types

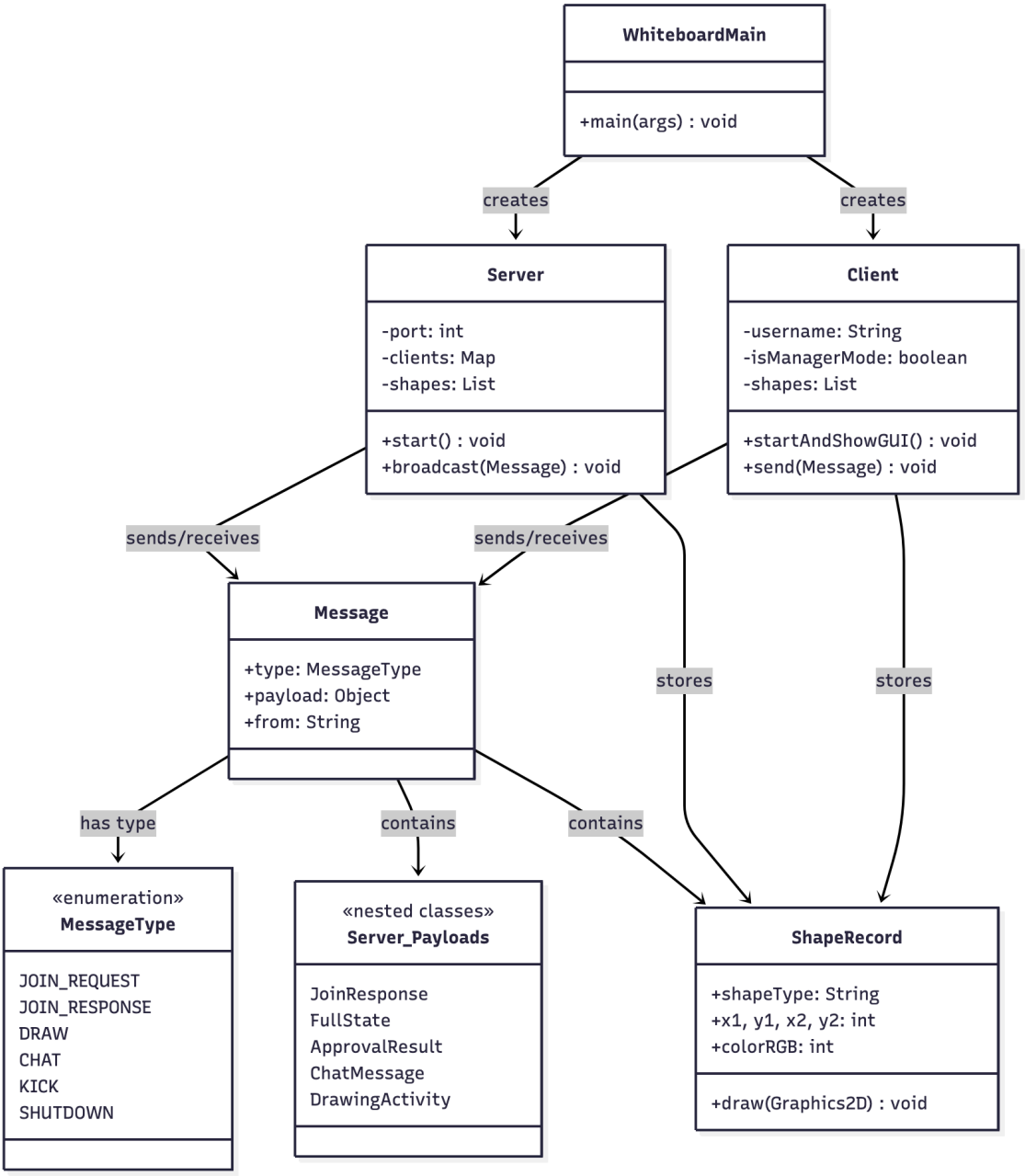
| Message Type | Direction | Payload Type | Purpose |
|---|---|---|---|
| JOIN_REQUEST | Client → Server | String (username) | Request to join whiteboard |
| JOIN_RESPONSE | Server → Client | JoinResponse | Approval/rejection result |
| MANAGER_APPROVAL_REQUEST | Server → Manager | String (username) | Ask manager to approve user |
| MANAGER_APPROVAL_RESULT | Manager → Server | ApprovalResult | Manager's approval decision |
| FULL_STATE | Server → Client | FullState | Complete whiteboard sync |
| DRAW | Client ↔ Server | ShapeRecord | New shape drawn |
| CHAT | Client ↔ Server | ChatMessage | Text message |
| USER_LIST_UPDATE | Server → Clients | List<String> | Updated user list |
| KICK | Manager → Serve | String (username) | Kick user command |
| KICKED | Server → Client | String | Notification of being kicked |
| FILE_OPEN | Manager → Server | FullState | Load whiteboard from file |
| NEW_BOARD | Manager → Server | null | Clear whiteboard |
| CLOSE_BOARD | Manager → Server | null | Shutdown system |
| SHUTDOWN | Server → Clients | String | Server shutdown notification |
| DRAWING_ACTIVITY | Client ↔ Server | DrawingActivity | User started/stopped drawing |

### 3.4 Payload classes

- JoinResponse
- FullState
- ShapeRecord
- ChatMessage
- DrawingActivity

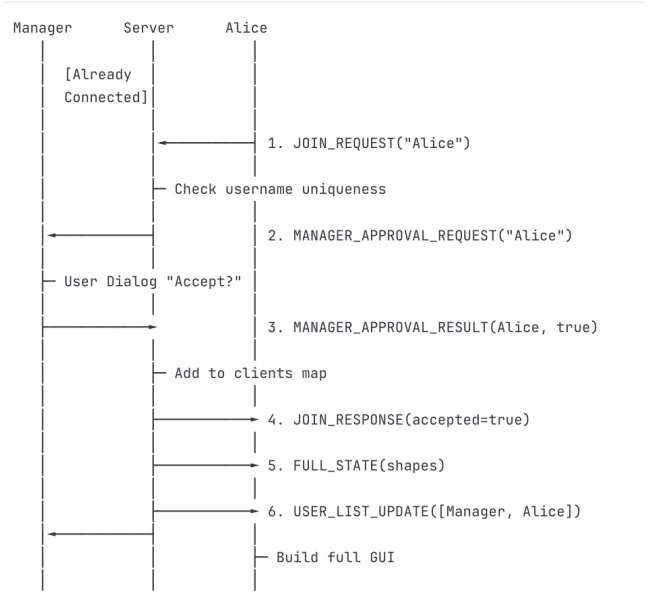
# 4. Design Diagrams

## 4.1 Class Diagram




## 4.1 Interaction Diagram

This diagram shows how Alice joins and gets approved by Manager




# 5. Implementation details

## 5.1 Server Implementation

**1. Single-threaded connection acceptance with multi-threaded client handling:**

- The server runs a main thread dedicated to listening for incoming client connections.
- Each time a new client connects, a separate ClientHandler thread is created to manage communication with that client.
- This approach allows the server to support multiple clients interacting with the system at the same time.

**2. Centralized state management:**

- The server maintains the authoritative state, including the shared list of shapes.
- To ensure thread safety and avoid concurrency issues, it uses `Collections.synchronizedList()` for managing shared data and a `ConcurrentHashMap` for tracking connected clients.

**3. Manager-first access model:**

- The first user to connect is automatically assigned as the manager.
- Until a manager is present, other clients cannot access or modify the whiteboard, ensuring controlled coordination and access management.

## 5.2 Client Implementation

**1. Approval-based GUI rendering:**

- When a new user connects, they initially see a “Waiting for approval” screen.
- The complete whiteboard interface is only displayed once the client receives a JOIN_RESPONSE message with accepted = true.
- This ensures that only approved users can interact with the drawing interface, preventing unauthorized access.

**2. Event-driven updates:**

- The client runs a dedicated listener thread to handle incoming messages from the server.

- To maintain thread safety and prevent UI freezes, SwingUtilities.invokeLater() is used for all GUI updates.
- This event-driven approach allows smooth, non-blocking interaction for the user.

**3. Local optimistic drawing:**

- When a user draws a shape, it is immediately added to the local list for instant feedback.
- The shape is then sent to the server, which broadcasts it to all connected clients.
- This approach provides a more responsive and seamless drawing experience for users.

### 5.3 Concurrency Management

Server side:
To ensure thread-safe operation and prevent data inconsistencies, several synchronization mechanisms are used:

- synchronized(Server.this): protects critical sections during manager assignment and user admission.
- Collections.synchronizedList(): ensures safe concurrent access to the shared shapes list.
- ConcurrentHashMap: manages client connections and pending approval requests without explicit locking.
- synchronized(out): secures writes to each client's ObjectOutputStream, avoiding message conflicts.

Client-side:
Similar synchronization strategies are applied to maintain thread safety and smooth GUI rendering:

- synchronized(shapes): ensures safe access to the shapes list when repainting the whiteboard.
- synchronized send(): prevents race conditions during message transmission via ObjectOutputStream.
- SwingUtilities.invokeLater(): handles all GUI updates from the listener thread safely on the event dispatch thread.

### 5.4 File operations (Manager only):

- Save / Save As: serialises List<ShapeRecord> to file.
- Open: loads a saved list of shapes and broadcasts it as FILE_OPEN.
- New: clears board and broadcasts empty FULL_STATE.
- Close: broadcasts SHUTDOWN and terminates all clients.

## 6. New innovations

### 6.1 Real-time Drawing Activity Indicator

A visual status label displays messages such as "[Username] is drawing..." whenever a user is

actively drawing. This feature enhances collaboration by keeping all participants aware of others' actions in real time, reducing confusion in multi-user environments.

### 6.2 Unique Username Enforcement

To maintain clear user identification, the server validates each new username against both active and pending user lists. This prevents duplicate usernames and ensures that all actions on the whiteboard are correctly attributed, without requiring manual ID generation.

### 6.3 Approval-based Access Control

The system implements a two-phase GUI rendering process. Unapproved users are shown a waiting screen with all drawing tools disabled until the manager approves their access. This mechanism prevents unauthorized interaction and strengthens overall session security.

### 6.4 Graceful Degradation

The system is designed to handle client disconnections smoothly. If a regular user disconnects, only their session is affected. However, if the manager disconnects, the server coordinates a controlled shutdown and sends notifications to all connected clients.

### 6.5 Manager-Only File Operations

Role-based permissions restrict critical file operations such as New, Open, and Save, exclusively to the manager. This approach preserves data integrity and avoids potential conflicts caused by simultaneous file modifications from multiple users.

## 7. Conclusion:

The distributed whiteboard system showcases the following capabilities:

- A fully functional client-server architecture supporting real-time collaboration among multiple users.
- Instant synchronization of the shared drawing state, ensuring that all participants view updates simultaneously.
- Enhanced usability features including colour selection, stroke control, text insertion, and integrated chat functionality.
- Manager-based session management with controls for user approval, removal, file saving/loading, and session termination.
- A modular and extensible design, enabling future enhancements such as undo/redo functionality, additional shape types, and persistent data storage.