COMP90015 2022 SM2 Project2
Distributed Systems

Distributed Shared White Board Report

Author: Yvonne Tao

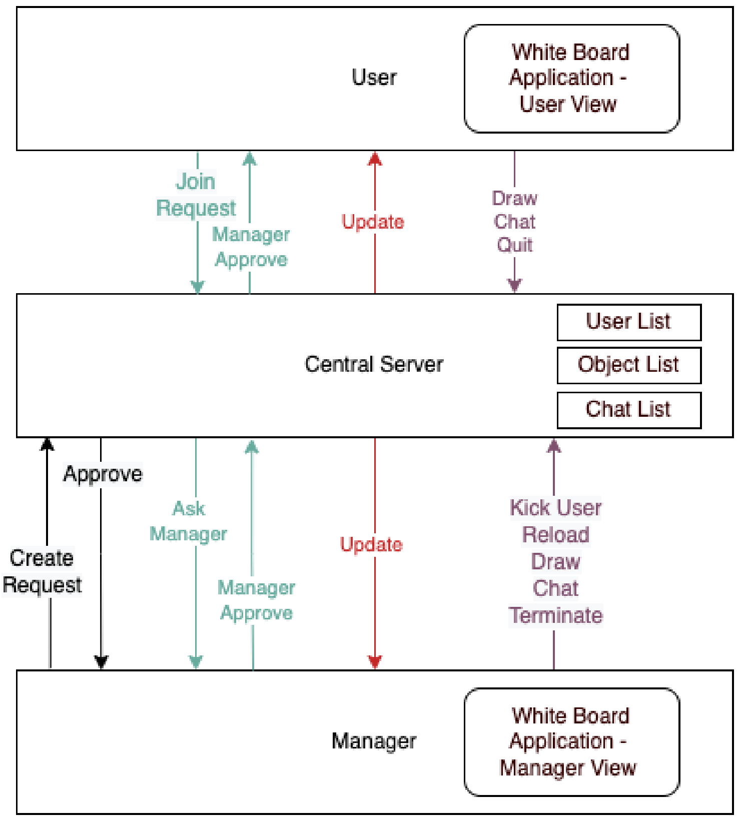
Student number: 1183577

Author: Haiyao Yan

Student number: 1073969

System Architecture

The following diagram shows the general architecture of our distributed shared whiteboards, which allow multiple users to draw simultaneously on a canvas.



The main component contains:

1. A central server that manages all the system state. The server keeps three lists (user list, object list and chat list), and keep them up to date with every client changes so that it can broadcast any changes to all other users.
2. User who requires manager's approval to join the white board, afterward being able to draw shapes on to the white board, and send messages to chat window, and choose to leave the Application.
3. A white board manager who has all the ability as a normal user, plus a few higher privileges such as kick out user, open a new white board, reload a saved white board and terminate the application for all users.

This architecture connects all client nodes directly to a central server, who will broadcast any updates it receives to all clients.

Considering the high demand on concurrency and consistency of white board application, the choice of using a single central server will make sure all users are drawing on the same white board as there is only one source of truth. Besides, users will have better experience with less latency time for any remote update from other users.

One of the disadvantages of such design, however, is its inability to scale after a certain limit as the server can only have a finite number of ports open for connections. It is also very vulnerable to DOS attack, and once the server is down, it causes the single point of failure as there is no other server up for sending/receiving responses and requests.

Communication Protocols

The communication between server and clients are through sockets using the TCP protocol. Specifically, the server will establish a server socket awaiting, and accepts all connection that come in from clients. It will server each single connection with a separate thread, keeps the connection open until the user decides to quit or the manager terminate the whole application.

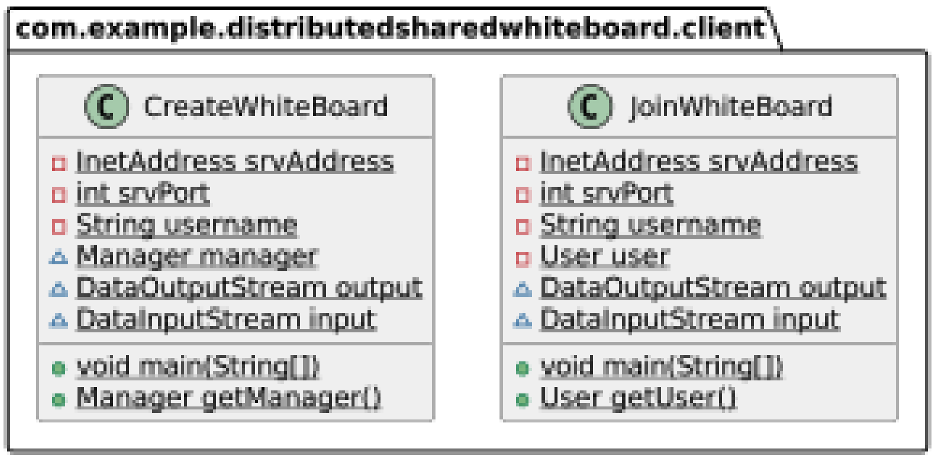
The main reasons of choosing socket for communication is that it has easy access to centralized data, which is exactly what the system requires. Compared with Remote Method Invocation (RMI), Socket connection allows us to format Messages in a custom way. In addition to that, the communication required for this white board application is nothing more than Strings and numbers (for passing username, chat box message, object coordinates, width, heights, class names, etc.), using Socket is sufficient. It’s also more effective as it produces less network traffic.

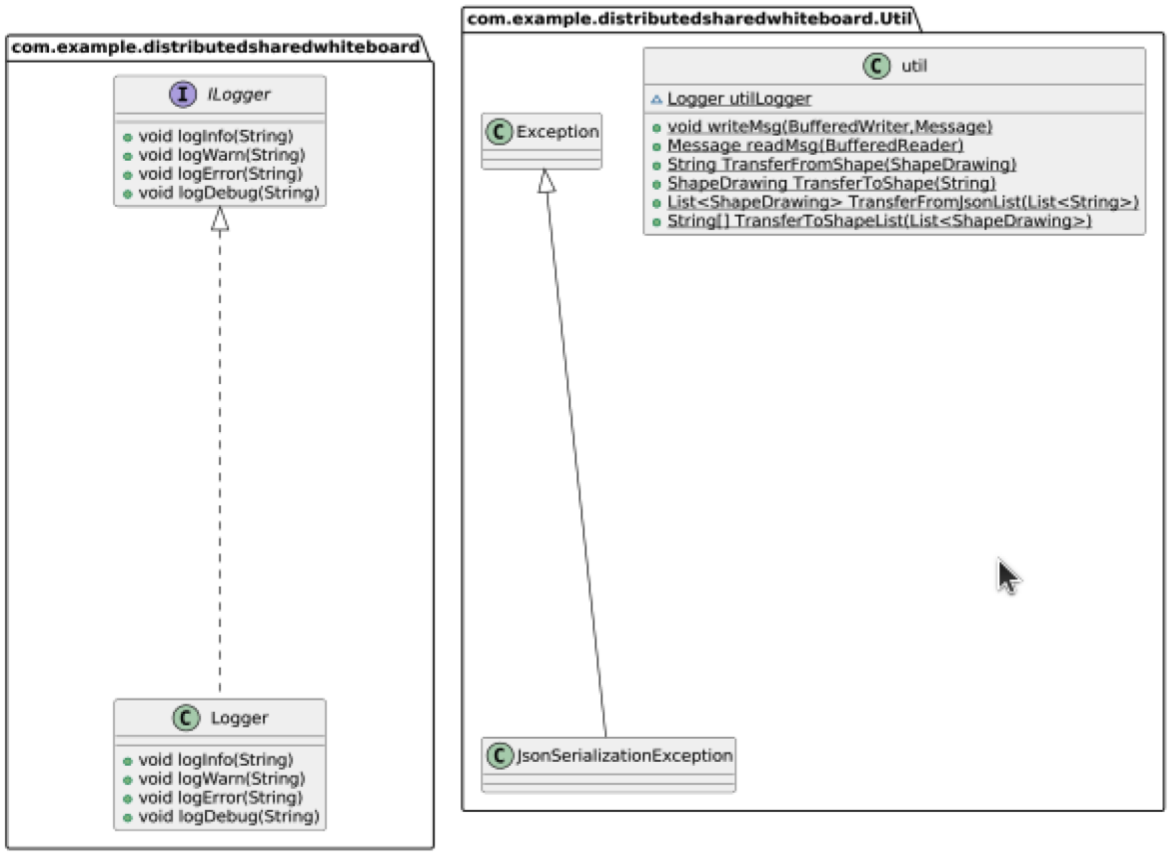
The reasons we choose TCP as exchange protocols over UDP is its reliability. Often in between the transport, the segments may get lost on its way to the destination, using TCP protocol ensures each segment reaches to the receiver so that users see the same white board, which is somewhat important for application like shared white board.

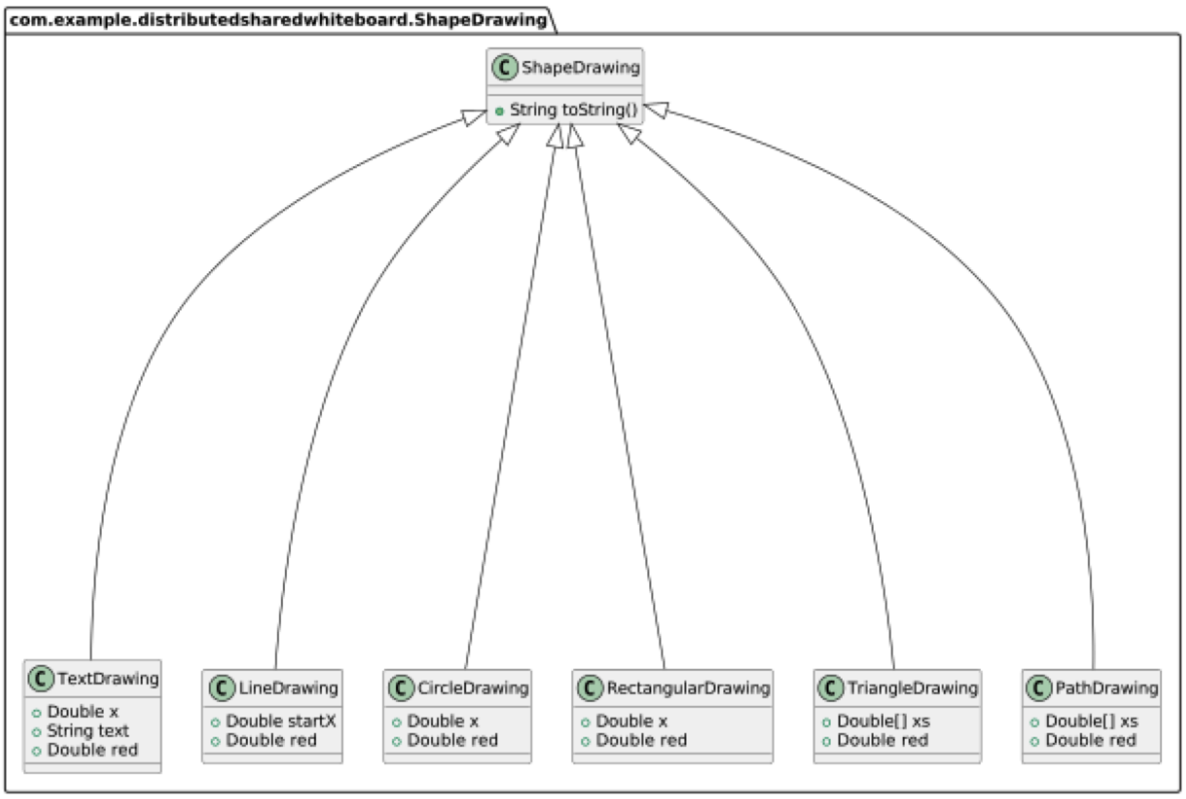
Message Formats

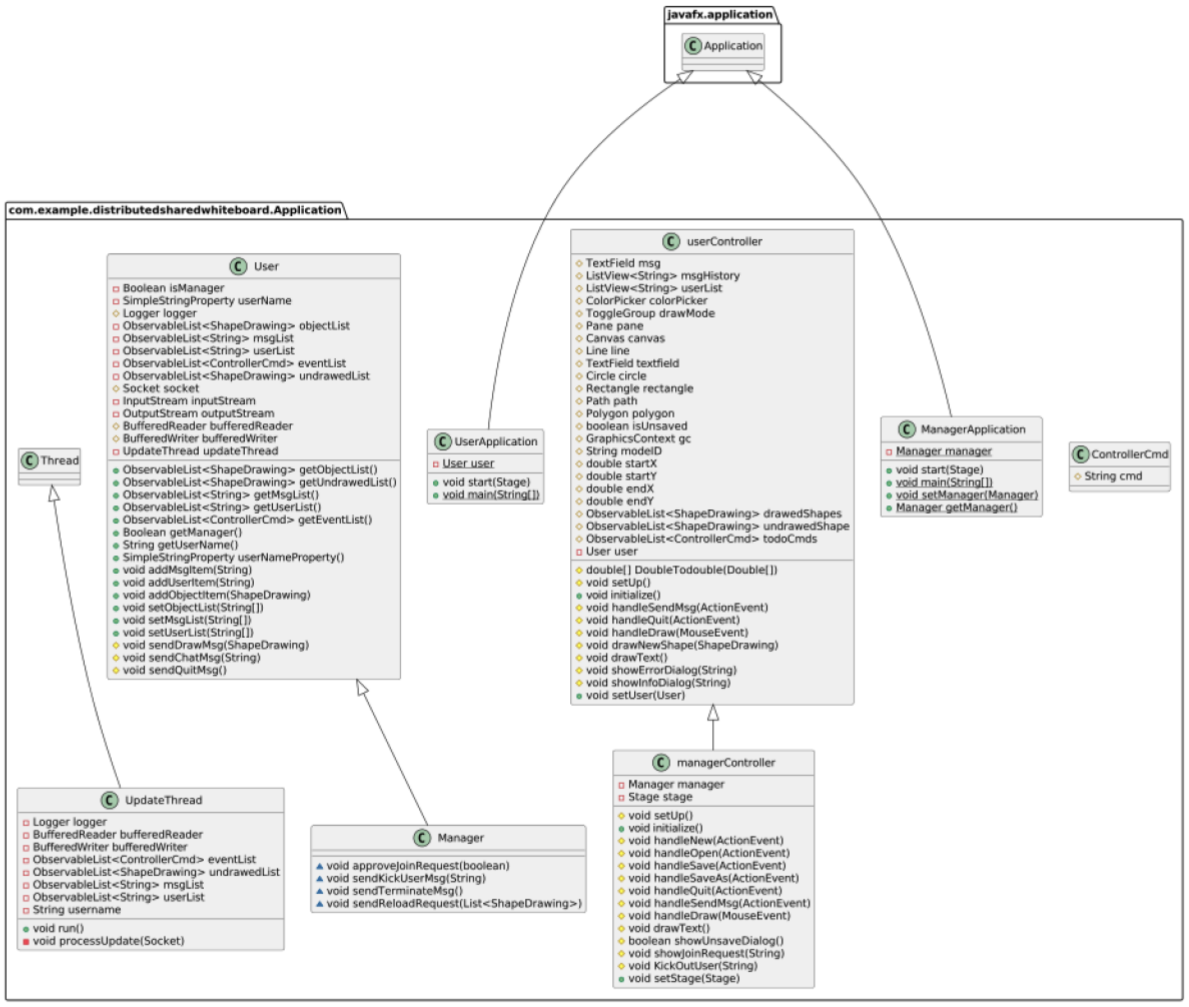
All messages are serialized to JSON serializable object before transform and then deserialize back to strings after transform using message factory. This will ensure the correctness of each message, also allows transmission of more complex objects such as a list of shapes a user draw on the white board.

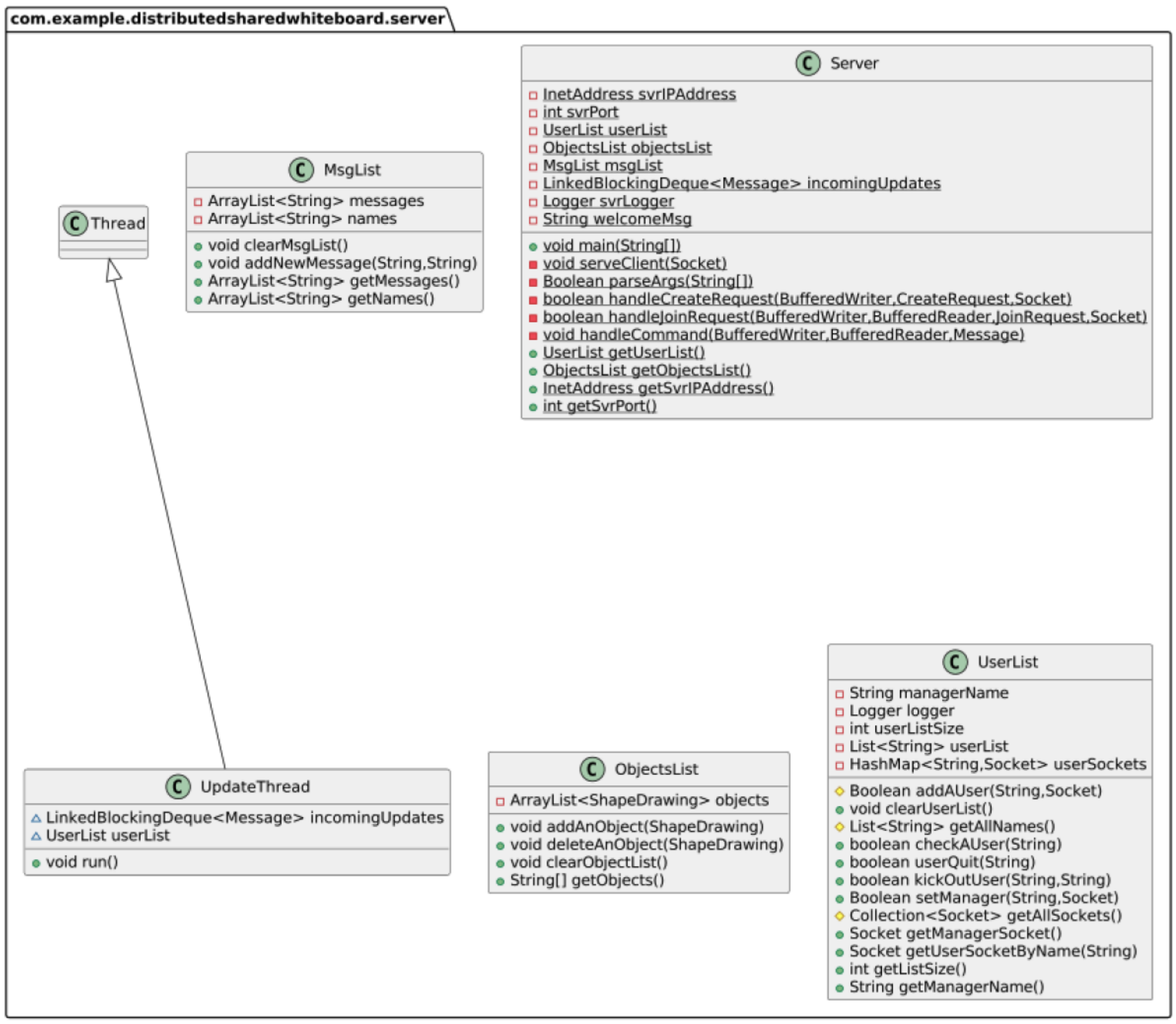
Design Diagrams (UML)

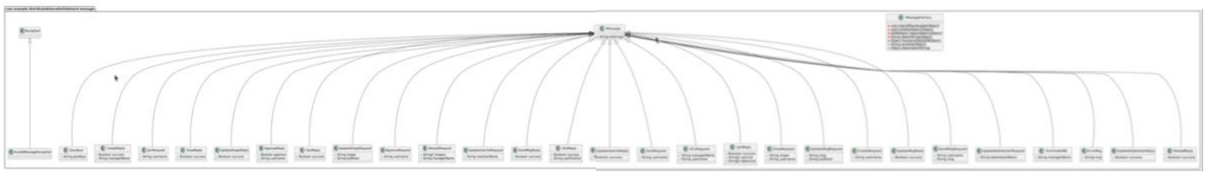



com.example.distributedsharedwhiteboard
ILogger
void logInfo(String)
void logWarn(String)
void logError(String)
void logDebug(String)
Logger
void logInfo(String)
void logWarn(String)
void logError(String)
void logDebug(String)
com.example.distributedsharedwhiteboard.Util
util
Logger utilLogger
void writeMsg(BufferedWriter,Message)
Message readMsg(BufferedReader)
String TransferFromShape(ShapeDrawing)
ShapeDrawing TransferToShape(String)
List<ShapeDrawing> TransferFromJsonList(List<String>)
String[] TransferToShapeList(List<ShapeDrawing>)
Exception
JsonSerializationException

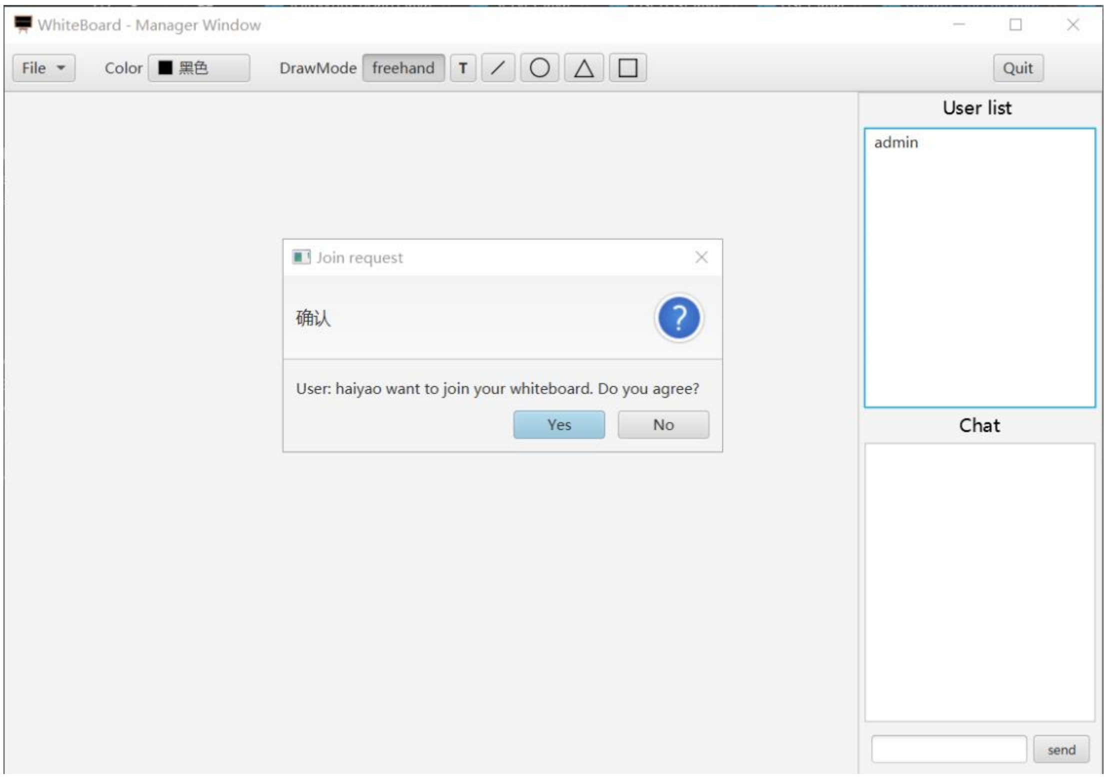

com.example.distributedsharedwhiteboard.ShapeDrawing
ShapeDrawing
String toString()
TextDrawing
Double x
String text
Double red
LineDrawing
Double startX
Double red
CircleDrawing
Double x
Double red
RectangularDrawing
Double x
Double red
TriangleDrawing
Double[] xs
Double red
PathDrawing
Double[] xs
Double red


javafx.application
Application
com.example.distributedsharedwhiteboard.Application
User
Boolean isManager
SimpleStringProperty userName
Logger logger
ObservableList<ShapeDrawing> objectList
ObservableList<String> msgList
ObservableList<String> userList
ObservableList<ControllerCmd> eventList
ObservableList<ShapeDrawing> undrawedList
Socket socket
InputStream inputStream
OutputStream outputStream
BufferedReader bufferedReader
BufferedWriter bufferedWriter
UpdateThread updateThread
ObservableList<ShapeDrawing> getObjectList()
ObservableList<ShapeDrawing> getUndrawedList()
ObservableList<String> getMsgList()
ObservableList<String> getUserList()
ObservableList<ControllerCmd> getEventList()
Boolean getManager()
String getUserName()
SimpleStringProperty userNameProperty()
void addMsgItem(String)
void addUserItem(String)
void addObjectItem(ShapeDrawing)
void setObjectList(String[])
void setMsgList(String[])
void setUserList(String[])
void sendDrawMsg(ShapeDrawing)
void sendChatMsg(String)
void sendQuitMsg()
Thread
UserApplication
User user
void start(Stage)
void main(String[])
userController
TextField msg
ListView<String> msgHistory
ListView<String> userList
ColorPicker colorPicker
ToggleGroup drawMode
Pane pane
Canvas canvas
Line line
TextField textfield
Circle circle
Rectangle rectangle
Path path
Polygon polygon
boolean isUnsaved
GraphicsContext gc
String modelD
double startX
double startY
double endX
double endY
ObservableList<ShapeDrawing> drawedShapes
ObservableList<ShapeDrawing> undrawedShape
ObservableList<ControllerCmd> todoCmds
User user
double[] DoubleTodouble(Double[])
void setUp()
void initialize()
void handleSendMsg(ActionEvent)
void handleQuit(ActionEvent)
void handleDraw(MouseEvent)
void drawNewShape(ShapeDrawing)
void drawText()
void showErrorDialog(String)
void showInfoDialog(String)
void setUser(User)
ManagerApplication
Manager manager
void start(Stage)
void main(String[])
void setManager(Manager)
Manager getManager()
ControllerCmd
String cmd
UpdateThread
Logger logger
BufferedReader bufferedReader
BufferedWriter bufferedWriter
ObservableList<ControllerCmd> eventList
ObservableList<ShapeDrawing> undrawedList
ObservableList<String> msgList
ObservableList<String> userList
String username
void run()
void processUpdate(Socket)
Manager
void approveJoinRequest(boolean)
void sendKickUserMsg(String)
void sendTerminateMsg()
void sendReloadRequest(List<ShapeDrawing>)
managerController
Manager manager
Stage stage
void setUp()
void initialize()
void handleNew(ActionEvent)
void handleOpen(ActionEvent)
void handleSave(ActionEvent)
void handleSaveAs(ActionEvent)
void handleQuit(ActionEvent)
void handleSendMsg(ActionEvent)
void handleDraw(MouseEvent)
void drawText()
boolean showUnsaveDialog()
void showJoinRequest(String)
void KickOutUser(String)
void setStage(Stage)


com.example.distributedsharedwhiteboard.server
Server
InetAddress svrIPAddress
int svrPort
UserList userList
ObjectsList objectsList
MsgList msgList
LinkedBlockingDeque<Message> incomingUpdates
Logger svrLogger
String welcomeMsg
void main(String[])
void serveClient(Socket)
Boolean parseArgs(String[])
boolean handleCreateRequest(BufferedWriter,CreateRequest,Socket)
boolean handleJoinRequest(BufferedWriter,BufferedReader,JoinRequest,Socket)
void handleCommand(BufferedWriter,BufferedReader,Message)
UserList getUserList()
ObjectsList getObjectsList()
InetAddress getSvrIPAddress()
int getSvrPort()
MsgList
ArrayList<String> messages
ArrayList<String> names
void clearMsgList()
void addNewMessage(String,String)
ArrayList<String> getMessages()
ArrayList<String> getNames()
Thread
UpdateThread
LinkedBlockingDeque<Message> incomingUpdates
UserList userList
void run()
ObjectsList
ArrayList<ShapeDrawing> objects
void addAnObject(ShapeDrawing)
void deleteAnObject(ShapeDrawing)
void clearObjectList()
String[] getObjects()
UserList
String managerName
Logger logger
int userListSize
List<String> userList
HashMap<String,Socket> userSockets
Boolean addAUser(String,Socket)
void clearUserList()
List<String> getAllNames()
boolean checkAUser(String)
boolean userQuit(String)
boolean kickOutUser(String,String)
Boolean setManager(String,Socket)
Collection<Socket> getAllSockets()
Socket getManagerSocket()
Socket getUserSocketByName(String)
int getListSize()
String getManagerName()

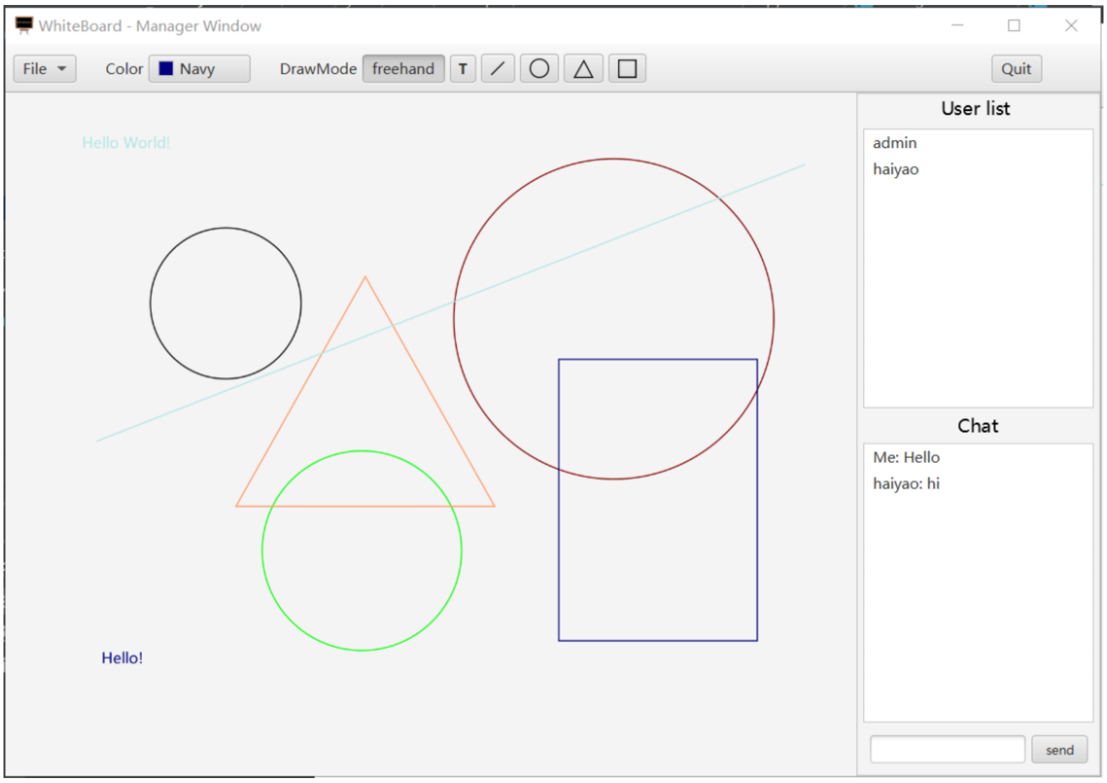


GUI Designs



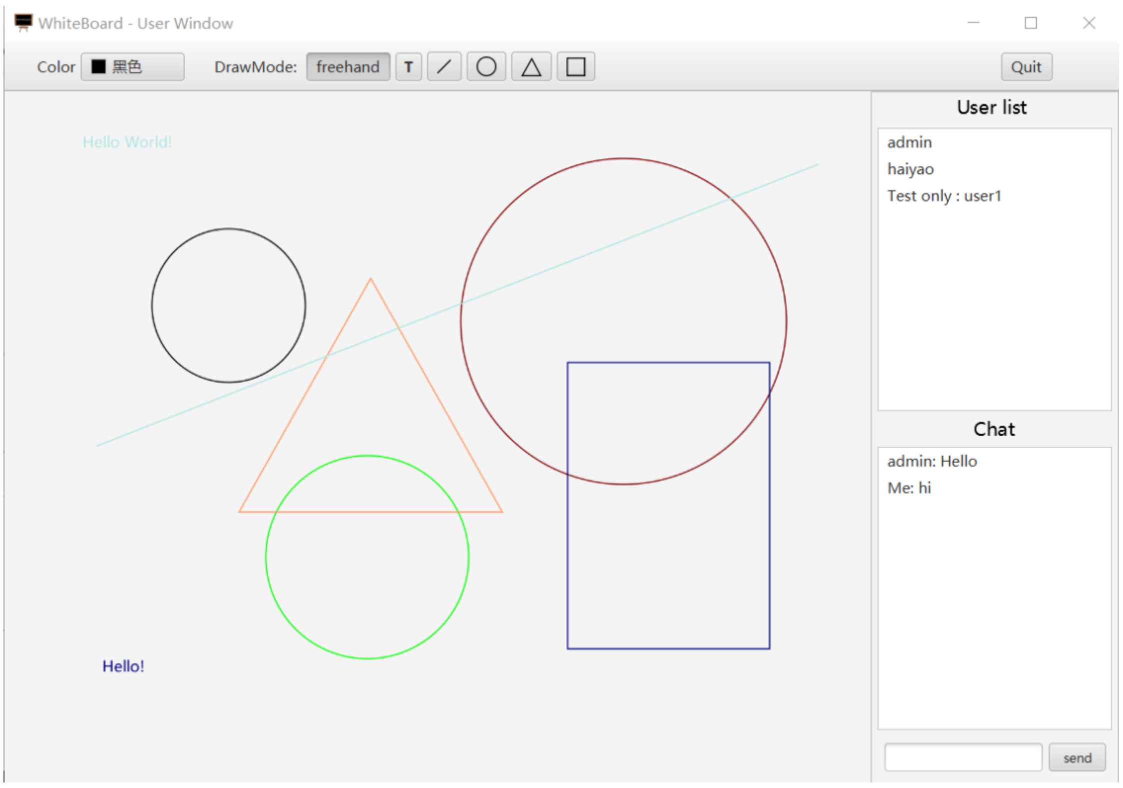
User: haiyao want to join your whiteboard. Do you agree?

Screen shot of prompt a User Join Request dialog to the Manager



Me: Hello
haiyao: hi

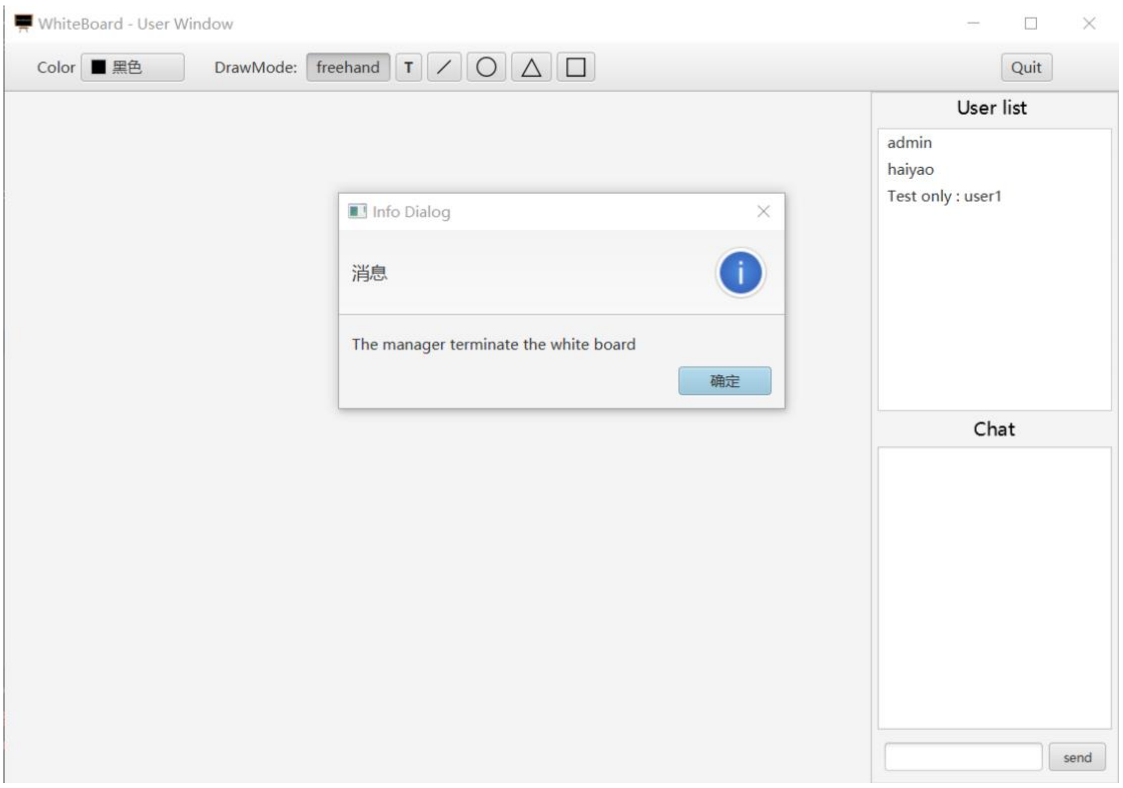
Screen shot of Manager window



admin
haiyao
Test only : user1

admin: Hello
Me: hi

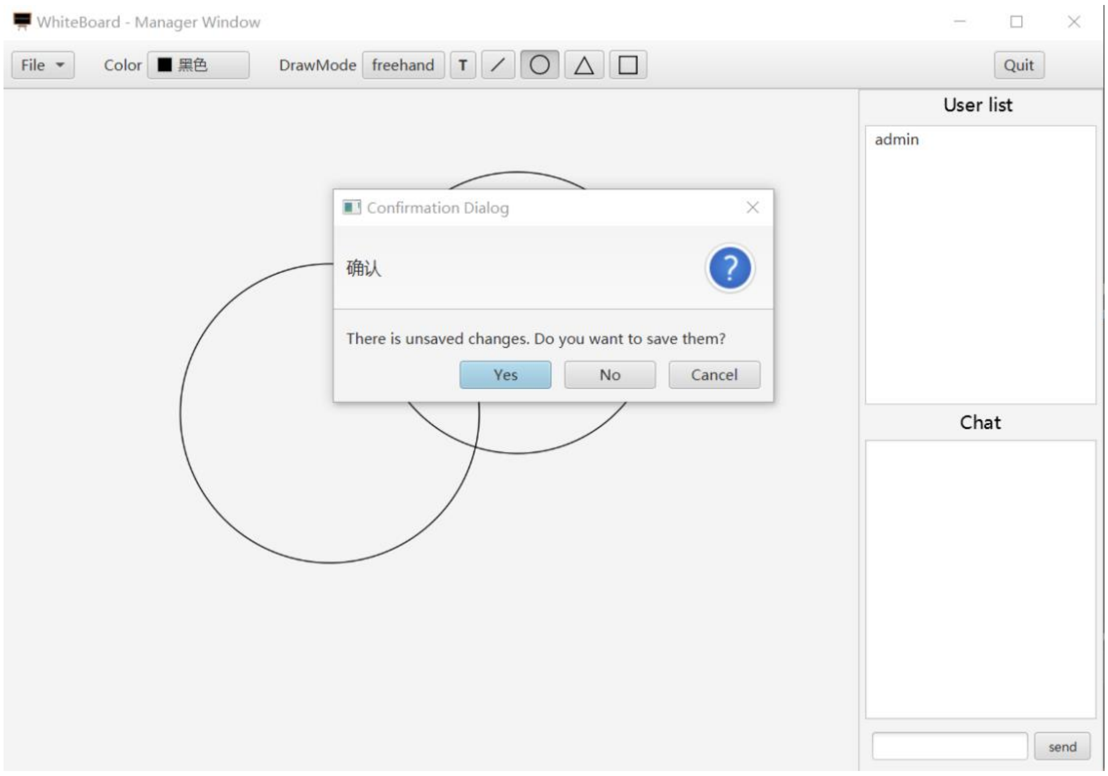
Screen shot of User window



admin
haiyao
Test only : user1

The manager terminate the white board

Screen shot of users receiving a notice when the manager terminates the white board



There is unsaved changes. Do you want to save them?

Screen shot of manager click Quit with unsaved changes

Implementation details

As shown in the above design diagrams, the whole project is composed of Client, Application, Server, Drawing shapes, Messages for socket communication and some utility methods under Util.

After setting up the central server, the CreateRequest and JoinRequest are the two starting points for a manager and users to launch their respective applications. It will first set up socket connections between the user and the server, save user input and initialize a user, then launch the GUI. The GUI thread will keep listening to the user's activities in the application and send updates to the central server. The server then replies to the user if this update has been successfully recorded, and broadcast the update to all users, who actively listens for update messages from server on another thread, then render the changes to their GUI.

The application contains three parts: the Controller, View and Model. The controller defines how the application would react, such as triggering an event when a button being clicked or displaying confirmation dialog when receiving update from server. The model works as a data container, where relevant information about users is stored and updated, it encapsulates all the business logic of communication with the central server. The view shows all the UI changes to the users, making the white board application working as a shared canvas.

The white board user interface is designed using JavaFx Scene Builder, and is saved in FXML file format that can be combined with the Java project by binding the UI to the application’s logic. Users can select from different colours and drawing mode from the top tool bar, including free-hand drawing, circles, triangle, rectangular and line mode. Specifically, the controller will record any drawings on the drawing pane, save each individual drawings as list of objects and pass to the model for communication. This will allow the users easily transmit the drawings across all users, also makes it possible for the manager to save and reload the white board into and from a list of objects in TXT file.