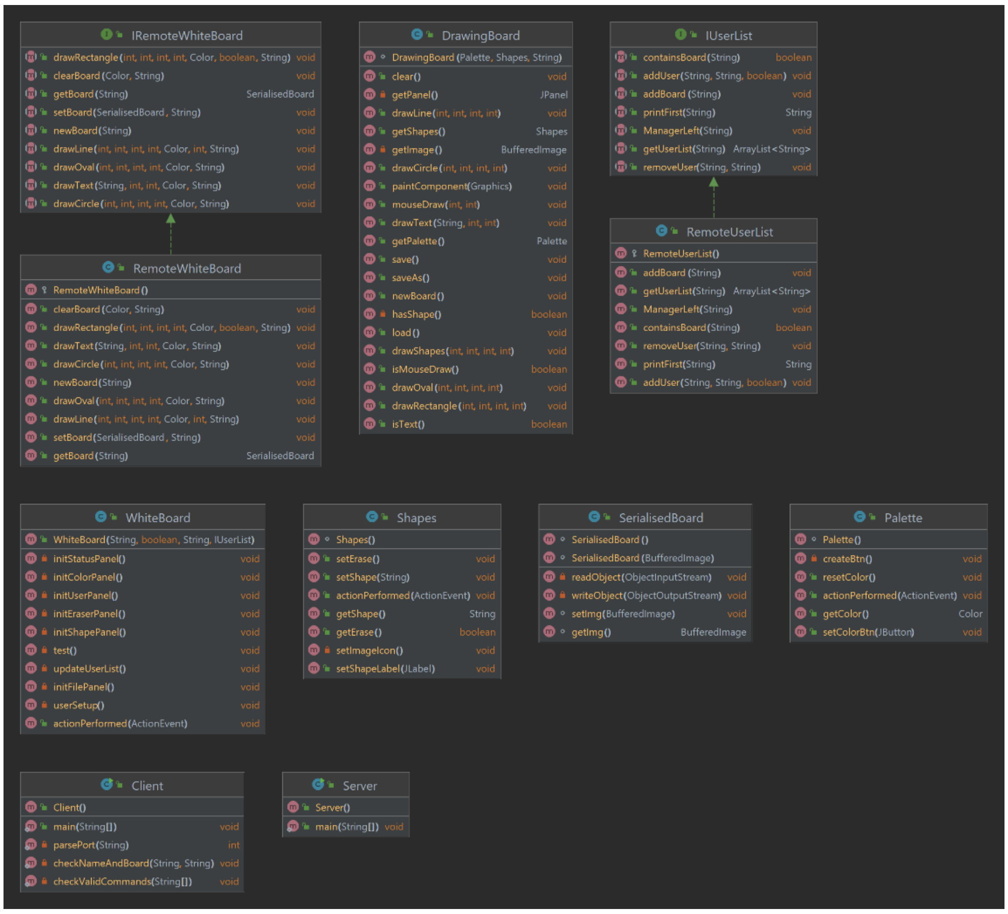
Distributed Systems Assignment 2

Deng-Yuan You 1337450

System architecture



IRemoteWhiteBoard
drawRectangle(int, int, int, int, Color, boolean, String) void
clearBoard(Color, String) void
getBoard(String) SerialisedBoard
setBoard(SerialisedBoard, String) void
newBoard(String) void
drawLine(int, int, int, int, Color, int, String) void
drawOval(int, int, int, int, Color, String) void
drawText(String, int, int, Color, String) void
drawCircle(int, int, int, int, Color, String) void
RemoteWhiteBoard
RemoteWhiteBoard()
clearBoard(Color, String) void
drawRectangle(int, int, int, int, Color, boolean, String) void
drawText(String, int, int, Color, String) void
drawCircle(int, int, int, int, Color, String) void
newBoard(String) void
drawOval(int, int, int, int, Color, String) void
drawLine(int, int, int, int, Color, int, String) void
setBoard(SerialisedBoard, String) void
getBoard(String) SerialisedBoard
DrawingBoard
DrawingBoard(Palette, Shapes, String)
clear() void
getPanel() JPanel
drawLine(int, int, int, int) void
getShapes() Shapes
getImage() BufferedImage
drawCircle(int, int, int, int) void
paintComponent(Graphics) void
mouseDraw(int, int) void
drawText(String, int, int) void
getPalette() Palette
save() void
saveAs() void
newBoard() void
hasShape() boolean
load() void
drawShapes(int, int, int, int) void
isMouseDraw() boolean
drawOval(int, int, int, int) void
drawRectangle(int, int, int, int) void
isText() boolean
IUserList
containsBoard(String) boolean
addUser(String, String, boolean) void
addBoard(String) void
printFirst(String) String
ManagerLeft(String) void
getUserList(String) ArrayList<String>
removeUser(String, String) void
RemoteUserList
RemoteUserList()
addBoard(String) void
getUserList(String) ArrayList<String>
ManagerLeft(String) void
containsBoard(String) boolean
removeUser(String, String) void
printFirst(String) String
addUser(String, String, boolean) void
WhiteBoard
WhiteBoard(String, boolean, String, IUserList)
initStatusPanel() void
initColorPanel() void
initUserPanel() void
initEraserPanel() void
initShapePanel() void
test() void
updateUserList() void
initFilePanel() void
userSetup() void
actionPerformed(ActionEvent) void
Shapes
Shapes()
setErase() void
setShape(String) void
actionPerformed(ActionEvent) void
getShape() String
getErase() boolean
setImageIcon() void
setShapeLabel(JLabel) void
SerialisedBoard
SerialisedBoard()
SerialisedBoard(BufferedImage)
readObject(ObjectInputStream) void
writeObject(ObjectOutputStream) void
setImg(BufferedImage) void
getImg() BufferedImage
Palette
Palette()
createBtn() void
resetColor() void
actionPerformed(ActionEvent) void
getColor() Color
setColorBtn(JButton) void
Client
Client()
main(String[]) void
parsePort(String) int
checkNameAndBoard(String, String) void
checkValidCommands(String[]) void
Server
Server()
main(String[]) void

Fig1. Class diagram

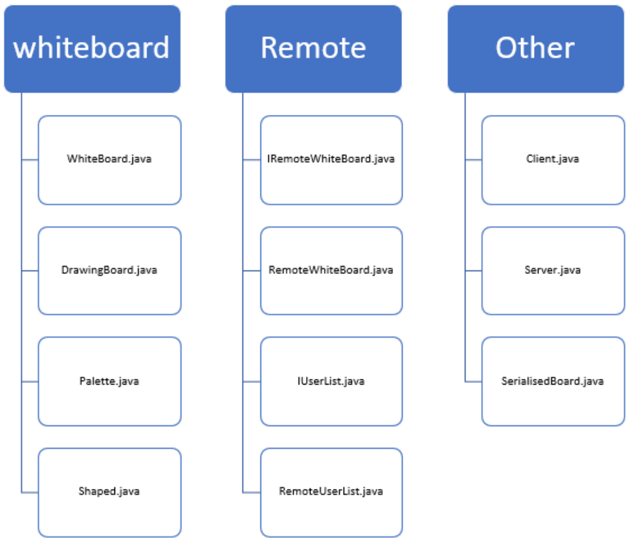


Fig2. Classes of the project

Fig1 and Fig2 above show the class of the project. From Fig2, classes of the project can be divided into 3 main categories.

- WhiteBoard

There are four classes related to the whiteboard GUI. **WhiteBoard.java** contains the whole GUI element as shown in Fig3 below. The upper part of the GUI is the toolbar, the bottom left part is the userlist part display the name of the user along with all users joining the same whiteboard and the bottom right part is the whiteboard for drawing, represented by **DrawingBoard.java**. Apart from File, Eraser and Status, Colors and Shapes in the toolbar are represented by **Palette.java** and **Shapes.java**, respectively.

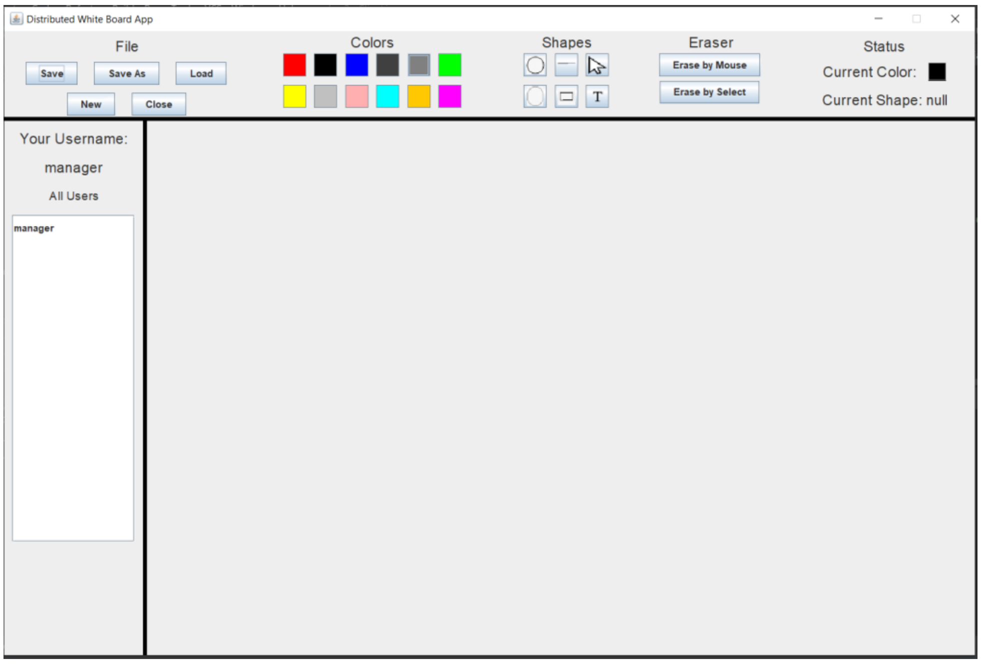


Fig3. Whiteboard GUI

- Remote

There are four classes dedicated for RMI, **IUserList.java** and **IRemoteWhiteBoard.java** are two interfaces which define remote methods. **RemoteUserList.java and RemoteWhiteBoard.java** implement the above two interfaces and provide detailed implementation for methods defined in the interface.

- Other

In addition, there are 3 other classes. **Server.java** binds remote objects to RMI registry and **Client.java** invokes the whiteboard GUI in which looks up remote objects from RMI registry as shown in Fig4 below.

Lastly, **SerialisedBoard.java** implements Serializable interface on BufferedImage class which allows the image to be transmitted through the network. When clients are drawing any shape or text, they are actually drawing on the serializable bufferedImage and when they get the drawing, they also get the serializable bufferedImage, so that it's easier to save the image as .png and .jpg format and load it back to the GUI if needed.

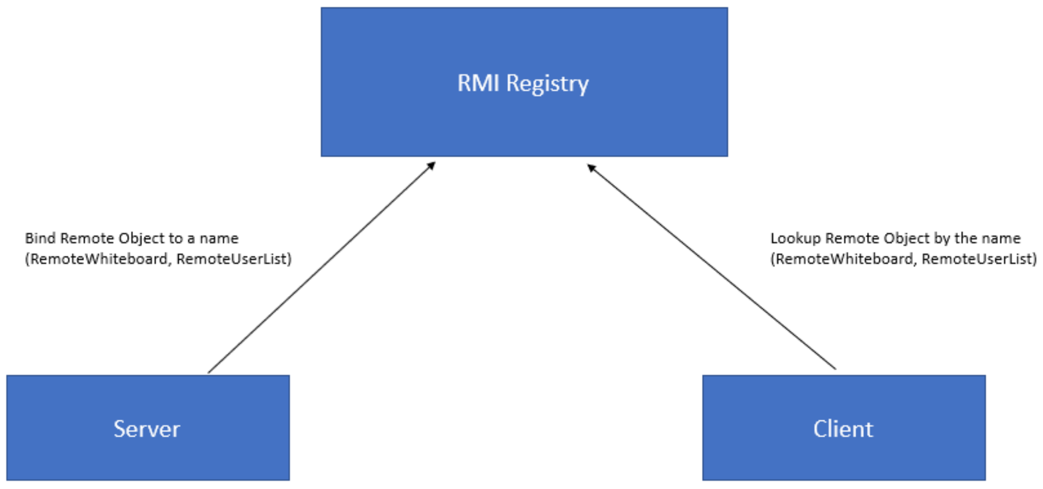

RMI Registry
Bind Remote Object to a name (RemoteWhiteboard, RemoteUserList)
Lookup Remote Object by the name (RemoteWhiteboard, RemoteUserList)
Server
Client

Fig4. Bind to and Lookup from RMI registry

Communication

As stated above, **remote method invocation (RMI)** is used in the project. The reason why RMI is chosen over socket is as followed:

- Easier implementation:
  RMI requires only bind and lookup to interact with remote objects, while socket needs to check ip address and port number as well as deal with connections. As a result, we can specify any whiteboardID we desire instead of being limited to use server ip and port as the identifier.

- Better scalability:
  In particular for multithreading, socket needs to keep accepting client connection and dispatch a separate thread for handling client requests, while using RMI simply needs the new client to lookup the remote service name binding to the object then methods of the remote object can be called directly. When there are hundreds of clients coming at the same time, using RMI is much efficient and easier.

Although RMI requires marshalling and unmarshalling objects, we are just sending buffered image which is not a complex and large object. Compared to the advantages above, marshalling and unmarshalling issues are trivial. Fig5 and Fig6 below present the interaction between client (manager and normal user) and remote objects (Remote user list and Remote whiteboard).

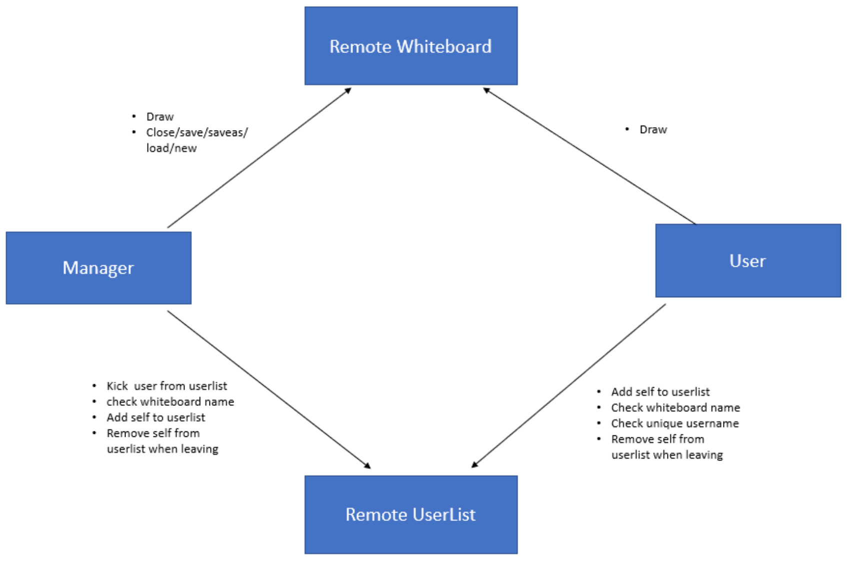


Fig5.interactions from client to remote objects

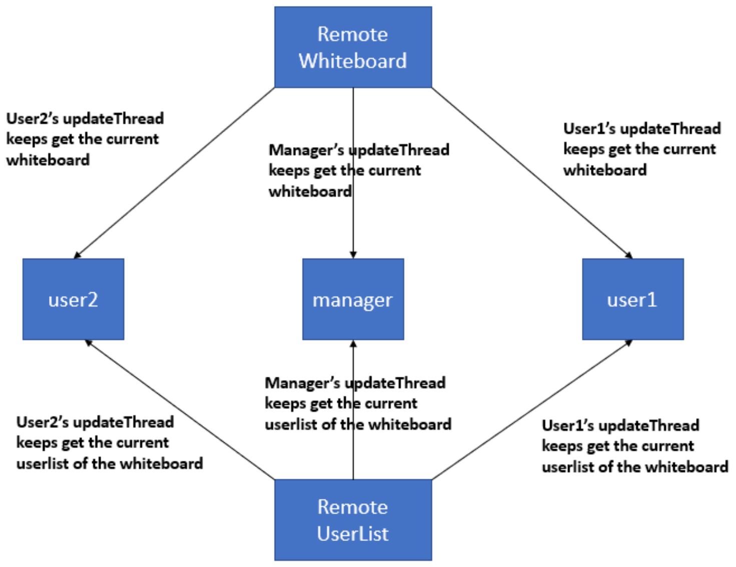

Remote Whiteboard
User2's updateThread keeps get the current whiteboard
Manager's updateThread keeps get the current whiteboard
User1's updateThread keeps get the current whiteboard
user2
manager
user1
User2's updateThread keeps get the current userlist of the whiteboard
Manager's updateThread keeps get the current userlist of the whiteboard
User1's updateThread keeps get the current userlist of the whiteboard
Remote UserList

Fig6. Interactions from remote objects to the client

For remote whiteboard, any client can draw to it. For remote userlist, any client can get all users, check unique username, check unique whiteboard name, add and remove himself/herself to and from current user list when joining and leaving the application.

For **manager**, there are some privileges a normal user doesn't possess – **save, load, new** and **close** a remote whiteboard and **kick any other user** other than the manager himself/herself from the current user list, resulting the kicked user leaving the application.

Synchronisation

- GUI Synchronisation

  An update thread is utilised by every client to keep fetching the latest image and userlist for the whiteboard, every 1.5 seconds the update thread calls the remote whiteboard's **getBoard()** to get the current drawing and remote userlist's **getUserList()** to get all active users of the whiteboard, then updates the client GUI.

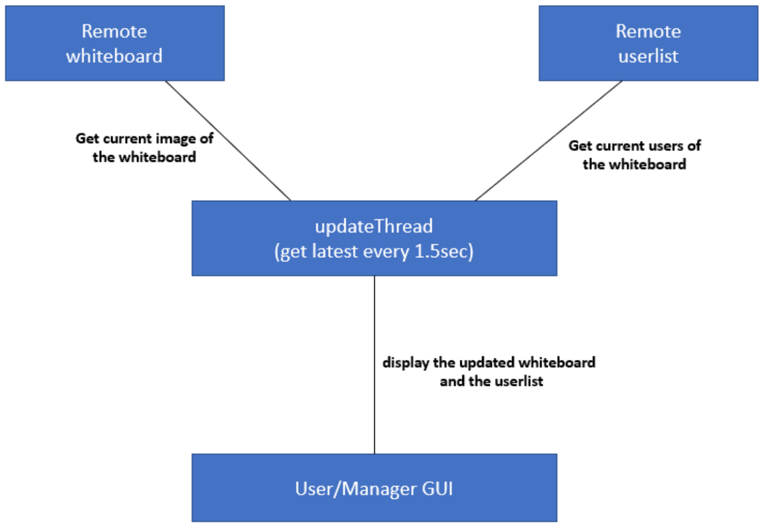

Remote whiteboard
Remote userlist
Get current image of the whiteboard
Get current users of the whiteboard
updateThread (get latest every 1.5sec)
display the updated whiteboard and the userlist
User/Manager GUI

Fig7. updateThread and GUI synchronisation

- Multiple WhiteBoard Synchronisation

  Each whiteboard has its own whiteboardID, which is specified by the manager initialising the whiteboard. In **RemoteWhiteBoard** class, a hashmap is used to map each whiteboardID to its drawing. When any client is drawing, it is drawn on the whiteboard which has key whiteboardID.

  In addition, a hashset is maintained in **RemoteUserList** to store all current whiteboardIDs. In other words, no two whiteboards with the same whiteboardID can exist at the same time. This mechanism allows multiple whiteboards to exist simultaneously without interfering with other whiteboards.

Command Line Invocation

(In this section, *Italics* indicates the string is a command)

1- Start the server

java Server

In the server, remote objects is bound to RMI registry using **createRegistry()**, thus we don't have to run *rmiregistry* command beforehand.

2- Create a new whiteboard

java create <whiteboardID> <username>

3- Join an existing whiteboard

java join <whiteboardID> <username>

There are some limitations for the commands, the following scenarios are not allowed and will throw exceptions.

- Username has already existed in the same whiteboard
  Each user in a whiteboard should have an unique username

```
Exception message: "Username has existed, please change to another"
```

- Username exceeds 25 characters long
  If an username is too lengthy, it will take up too much space on the screen.

```
Exception message: "Username should be less than 25 characters long"
```

- Create a whiteboard with an existing whiteboardID

```
Exception message: "The board is already in use. Try to join or
create a new one with different whiteboardID"
```

- Join a whiteboard with a non-existing whiteboardID

```
Exception message: "The board hasn't existed yet. Try to join another
existing board or create a new one"
```

GUI Display

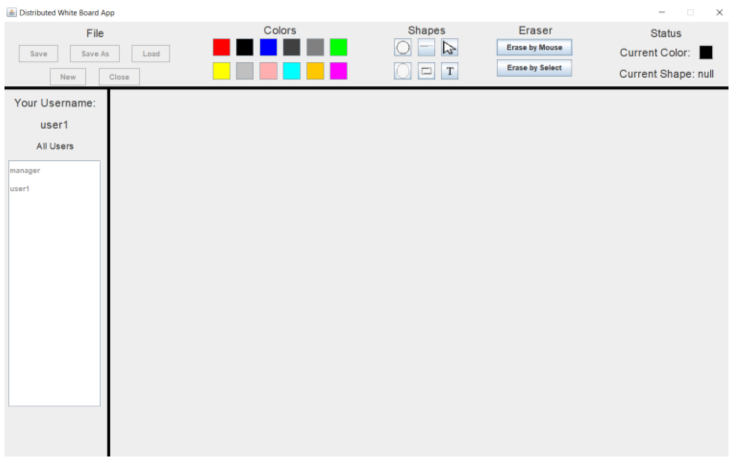
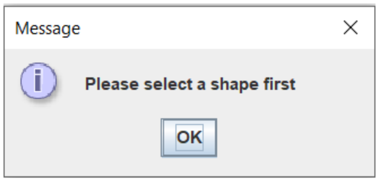


Fig8. Normal user GUI

Fig8 above shows the GUI of a normal user, compared to Fig3 which is manager's, the 5 File buttons on the top left corner are disabled, as well as the user in the all users window are also disabled from selection. On the status section on the top right corner, current color and shape are displayed. If the user doesn't pick a color before drawing, black is used as the default color. However, if the user doesn't pick a shape before drawing, a pop up window would appear to nofity the user as Fig9 below. When a whiteboard is closed and newed by the manager, the user is required to select the shape again.


Message

Please select a shape first

Fig9. Message when trying to draw without selecting a shape

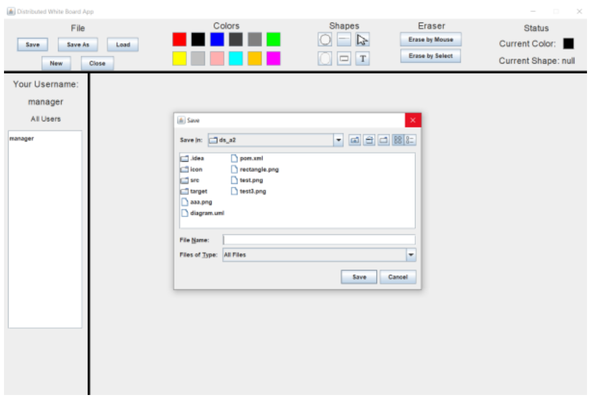


Fig10. save, saveas and load

Fig10 above presents a file chooser will pop up once the manager click save, saveas and load button. Note that for the save scenario, the file chooser only pops up if the manager hasn't saved the same image before.

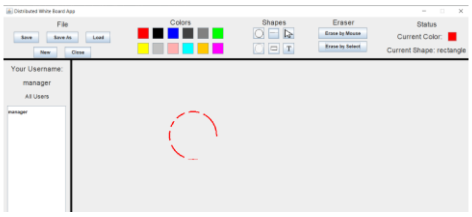
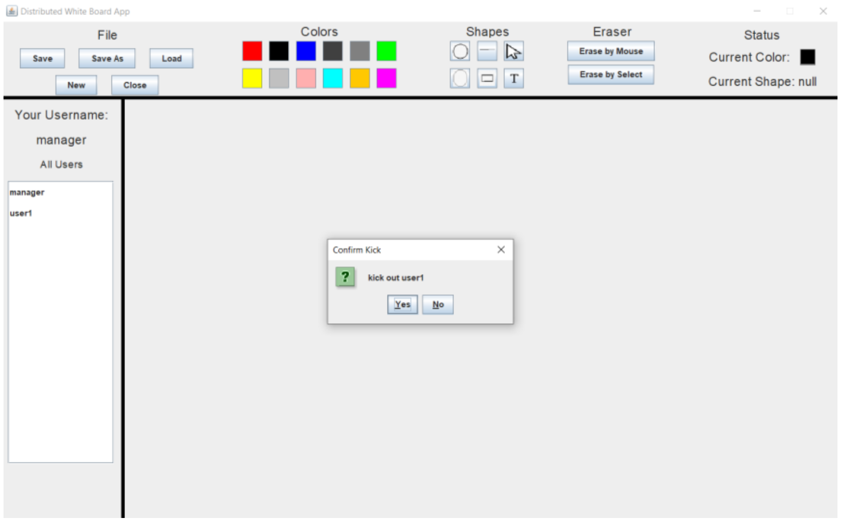


Fig11. Eraser effect

Fig11 above shows the eraser effect. There are two kinds of eraser, one is by mouse and the other is by select. The former kind is shown on the top left corner of the red circle with gaps in between and the latter kind is shown on the bottom right missing corner of the red circle.

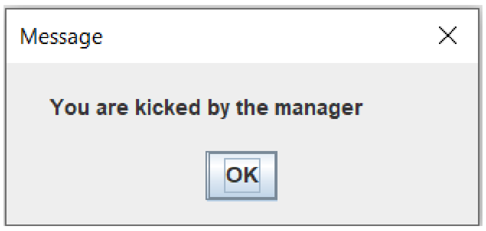


Confirm Kick

kick out user1

Fig12. Manager kick out user

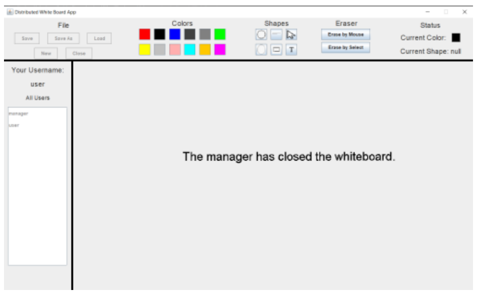
Fig12 above shows the screen when manager wants to kick out a user. The manager just click the username in all users list, and a confirm dialog would pop up to confirm kickout. If yes button is pressed, the selected user is kicked out from the whiteboard. On the screen of the kicked user, kickout message as shown in Fig13 below will appear and close user's application.



Message

You are kicked by the manager

Fig13. Kickout message



The manager has closed the whiteboard.

Fig14. No whiteboard screen

Suppose the manager has closed the whiteboard, all user including the manager would have a default null image shown as fig14 above. Although colors and shapes can be selected, they will not be drawn on the board until the manager creates a new whiteboard.