Design Document

Table of contents:

Changes since milestone

- added “width” argument to Whiteboard.drawLine() (and to the protocol)
- the whiteboard class is no longer thread safe
- remove the copy() function from whiteboard
- add checkInBounds function to whiteboard
- fleshed out the testing strategy section
- overall design remains unchanged

Goals

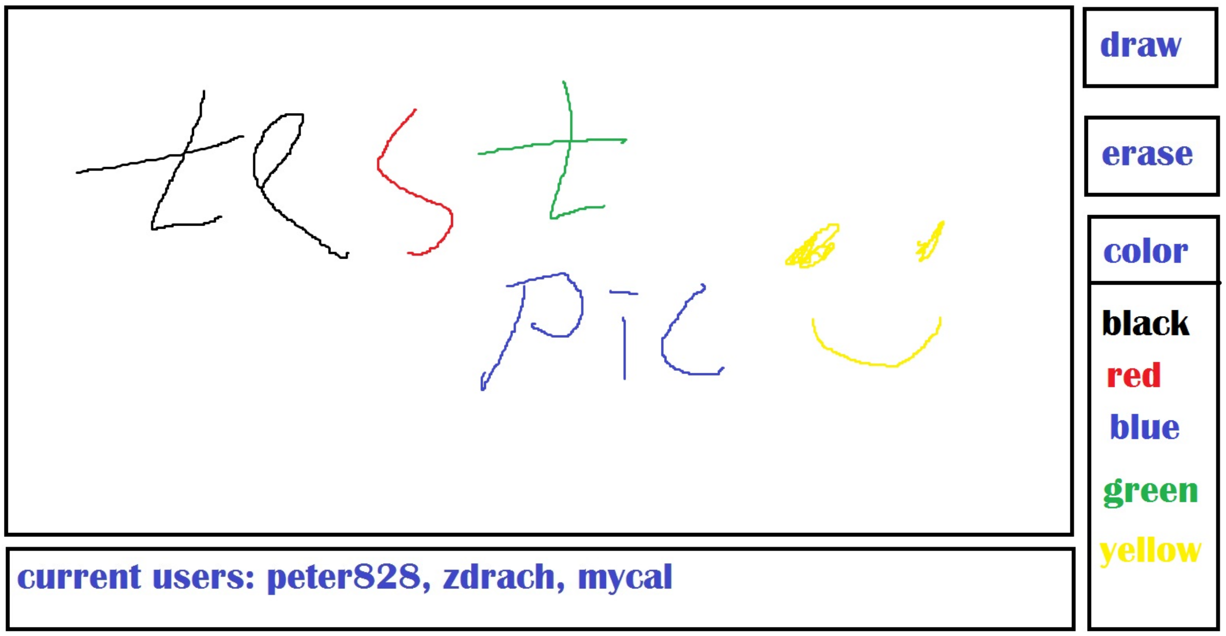
- Create raster-based collaborative editor
- Users can draw and erase an image at the same time
- The image will be hosted on a server, and users connect via internet
- A carefully-designed protocol will ensure that the image state remains consistent on all machines by avoiding race conditions and deadlocks

Design Overview

- Representation:
    - raster-based image editor
    - server and client maintain local pixel buffers
    - pixels are 24-bit RGB (8-bits for red, 8-bits for green, 8-bits for blue)
    - coordinates are 0-indexed from the bottom-left corner
- User actions:
    - client logs into server with a username (no password)

    - client can connect to a board
    - client can create a new board
    - client can disconnect from a board
- Real-time editing:
    - the server's pixel buffer is the authoritative representation of the image
    - client buffers are updated via the network to mirror the server's
    - client sends editing actions, such as drawing a line, to the server
    - whenever a pixel changes on the server, its new value gets sent to all clients
    - whenever a user connects or disconnects to the board, the server sends a list of all connected users to the client
    - to reduce perceived latency, client may draw the user's lines in its local buffer immediately when the user does an action
    - if the server disconnects for some reason, the GUI will close

GUI Design



The GUI for the program is expected to support following functions:drawing freely in 2D, choosing color for drawing, erasing freely in 2D and showing other users' names.

There are three main screens:
- Login screen
- Connection screen
- Drawing screen

The screens are shown and hidden depending on user actions.

The "drawing screen" consists of following components:

1. draw button & erase button & color buttons
2. Frame to hold the canvas
3. Frame to display the current users

Protocol

Each client communicates with a socket on the server computer via a message-passing protocol. When multiple clients are connected to a server computer, each client interacts via a separate socket on the server. For the rest of this section, we use the word “server” to refer to the socket on the server computer that talks to an individual client (as opposed to the entire, multithreaded server application).

All communication goes through a single socket. Each message is an ASCII string followed by a newline character. Numbers are represented in base 10. There are several different message types.

Messages are only allowed to be sent in a particular sequence. Each client/server pair can be modeled as a state machine. The state transition diagram in the next section indicates exactly which messages are allowed to be sent at any point in time.

Following are the message types, their descriptions, and their representations.

draw-line(x1, y1, x2, y2, red, green, blue,width):

- Sender: client
- Description: Indicates that a line should be drawn with the given start and end points. The coordinates range from 0 to 65535 (16 bits long). The color is provided in RGB format. Each component ranges from 0 to 255 (8 bits long). Width is an integer.
- Format: “dr <x1> <y1> <x2> <y2> <red> <green> <blue> <width>\n”
- Example: “dr 0 0 50 50 128 128 128\n”

update-pixel(x, y, red, green, blue):

- Sender: server
- Description: Notifies the client that the pixel at the given coordinates has the given color. Coordinates are 16-bit, and color components are 8-bit (same as draw-line).
- Format: “p <x> <y> <red> <green> <blue>\n”
- Example: “p 50 50 128 128 128\n”

update-users(users):

- Sender: server
- Description: Notifies the client that the list of connected users has changed. Gives the client the new list of usernames. <users> is a comma-separated list of usernames.

- Format: “u <users>\n”
- Example: “u user1,user2\n”

login(username):

- Sender: client
- Description: Client request to login. This should be the first message sent to the server during a new connection. Username may only contain letters, numbers, and underscores, and must be unique among all currently connected users.
- Format: “l <username>\n”   // that’s a lowercase L
- Example: “l my_name\n”

login-success():

- Sender: server
- Description: Login with the current username was successful, and the client may issue further commands (such as connecting to a board).
- Format: “ls\n”
- Example: “ls\n”

error(code):

- Sender: server
- Description: Something failed. The reason for error is given by an integer code number. Possible code numbers include:
    - 100: username already taken
    - 200: invalid board id
- Format: “e <code>\n”
- Example: “e 100\n”

connect-to-board(id):

- Sender: client
- Description: Client request to connect to the board with the given ID number. The client must not currently be connected to a board when it sends this message. If the ID doesn’t exist, the server responds with an *error(200)* message. The server responds with an *connect-to-board-success* message to initialize the client buffer. After the response, the client is defined to be connected to the board. While the client is connected to a board, it can send draw-line messages or receive update-pixel messages.
- Format: “c <id>\n”
- Example: “c 13\n”

disconnect-from-board():

- Sender: client
- Description: Client request to disconnect from the board it is currently connected to. The client is not allowed to call disconnect-from-board() unless it is already connected to a board. The server responds with an *disconnected-from-board-success* message. After

the response, the client is not connected to any boards and must connect anew to start drawing or view boards.

- Format: “d\n”
- Example: “d\n”

disconnect-from-board-success():

- Sender: server
- Description: Client has successfully been disconnected from the board it used to be connected to.
- Format: “ds\n”
- Example: “ds\n”

new-board():

- Sender: client
- Description: Create a new board and automatically connect to it. The client must not currently be connected to a board when it sends this message. The client should expect a *acknowledge-connection* message in response.
- Format: “n\n”
- Example: “n\n”

connect-to-board-success(id, users, data):

- Sender: server
- Description: Client is connected to the board with the given id. The set of <users> currently connected to the board is given as a comma-separated list of usernames. The <data> string contains the RGB colors of the board pixels, row-by-row, in hex format (most-significant-nibble first) concatenated directly together. (more specifically, given a board with m rows and n cols, the first 3 elements of <data> contain the RGB values of the pixel with (x,y) = (0,0), and the first 3*n elements of <data> encode the RGB values of the first row: y = 0).
- Format: “cs <id> <users> <data>”
- Example: “cs 13 user1,user2 ffffffeeeeee...”

State Diagram

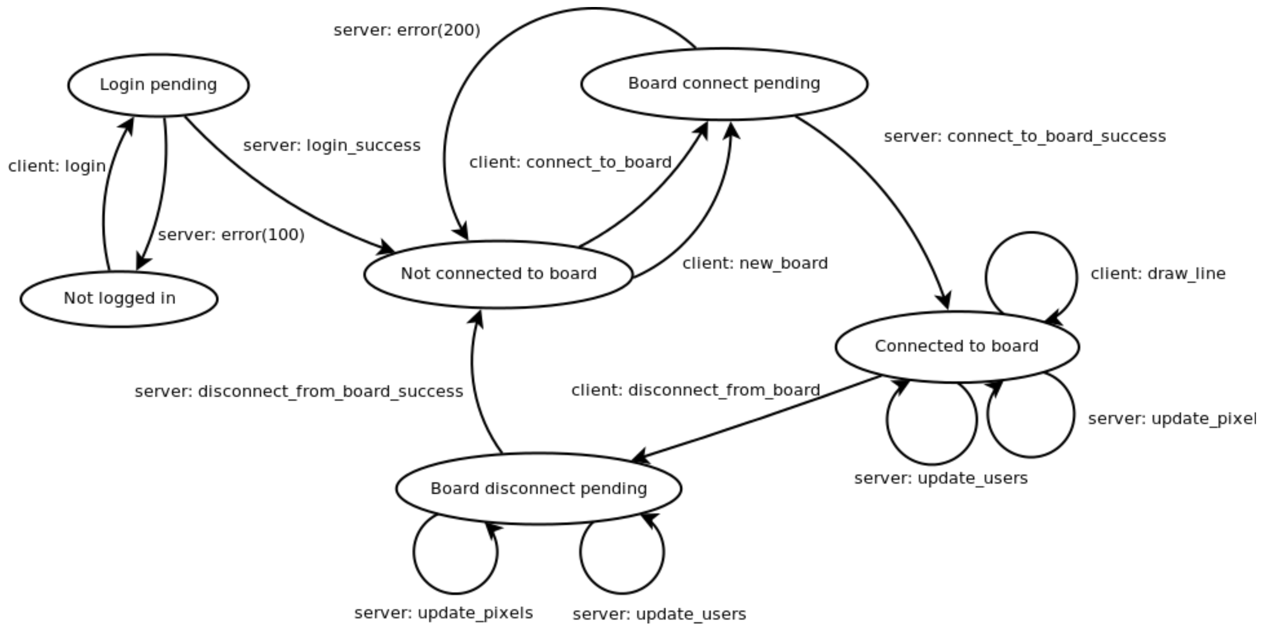
Each system, or client/server pair, can be modeled as a state machine. At any given point in time, the system is in one of the states listed in the state diagram below. State transitions (depicted by the arrows in the diagram) occur whenever the client or the server flushes a message over the network. When the system is in a particular state, the only messages that can be sent are the messages leading out of that particular state. For example, when the system is in the “Login pending” state, the client is not allowed to send any messages, whereas the server is allowed to send either a “login_success” message or an “error(100)” message.

When a new connection is initiated by the client, the system begins in the “Not logged in” state.

This state diagram ensures the correctness of our system at all points in time. Although the client and the server do not always have true knowledge of the system state, we can prove that they will never send a message that is inconsistent with the system state.

First consider the case where we ignore the self-state transitions. The remaining state transitions must alternate between a message sent by the client and a message sent by the server. This is similar to the request-response pattern implemented by HTTP. We know from experience that this pattern will produce consistent results.

Next, lets add back the self-state transitions. In the “Connected to board” state, both the client and the server are sending real-time updates to each other. When the client wishes to disconnect from the board, the client stops sending real-time updates to the server. Instead, it sends the “disconnect-from-board” message. Eventually, the server gets the message, stops pushing real-time updates to the client, and responds with the “disconnect-from-board-success” message. The server and client will continue interacting via the request-response pattern described above. By walking through this special case, we have shown that no messages are ever inconsistent with the state diagram.



Note: if an unrecoverable error occurs at any time, the socket simply gets closed

Data Types

common.Whiteboard:

- Description:
    - A two-dimensional bitmap of pixels representing an image
    - not thread safe
    - Internal representation is a fixed-size two-dimensional array of Color object.
    - The zeroth index corresponds to the bottom left hand corner of the board.
- Public Interface:
    - **public static final int WIDTH:** the width in pixels of all whiteboards
    - **public static final int HEIGHT:** the height in pixels of all whiteboards
    - **public Whiteboard()**; makes a board filled with the color white
    - **public List<Point> drawLine(Point point1, Point point2, Color color, int width)**; find all points on the line segment between and including point1 and point2 and draw a square with side-length width for each of them with the specified color. Returns an array of all the points that had their colors updated when while drawing the line. (Note that drawing a single point is the same as drawing a line from a point to itself.)
    - **public void setPixel(Point point, Color color)**; sets the color at that point to color. The point's x and y coordinates must be within the board's dimensions.
    - **public Color getPixel(Point point)**; return the color at the Point point. Point must be within the board's dimensions.
    - **public BufferedImage makeBuffer()**: construct a BufferedImage that can be used to copy the pixel data via the copyPixelData() function. Specifically, the BufferedImage should have the same width and height as the whiteboard, and use the predefined type "TYPE_INT_RGB".
    - **public void copyPixelData(BufferedImage buffer)**: copy the pixel data from the whiteboard into the given buffer. The buffer should be constructed using the makeBuffer() function, or should be constructed elsewhere with the same parameters.
    - **public static boolean checkInBounds(Point p)**: return true if the point falls within the boundary of a whiteboard.

common.Color:

- Description
    - Contains three final ints that range from 0 to 255 and stand for the RGB values.
    - Immutable and thread-safe
- Public Interface
    - **public Color()**; default constructor makes it white.
    - **public Color(int r, int g, int b)**; constructor sets the RGB values for this color. Precondition that r, g, and b are all between 0 and 255, inclusive.
    - **public int getRed()**; return the amount of red in this color
    - **public int getGreen()**; return the amount of green in this color
    - **public int getBlue()**; return the amount of blue in this color

common.Point:

- Description
  - contains two final ints for the x and y coordinate of the point
  - immutable and thread safe
- Public Interface
  - **public Point(int x, int y)**; constructor sets the point
  - **public int getX()**; returns the x coordinate of the point
  - **public int getY()**; returns the y coordinate of the point

common.SocketState

- Description
  - Enum for possible system states as specified by the protocol
- Public Interface
  - **NOT_LOGGED_IN**
  - **LOGIN_PENDING**
  - **NOT_CONNECTED**
  - **CONNECT_PENDING**
  - **CONNECTED**
  - **DISCONNECT_PENDING**

common.SocketWrapper

- Description
  - Contains generic functionality for reading lines from a socket on a background thread.
  - public interface is thread safe
  - users should attach a listener to implement specific functionality
  - listener.onReadLine, onReadError, onWriteError, and onReadFinish are guaranteed to be called from a single thread
- Public Interface
  - **public void SocketWrapper(Socket s)**: construct with the given open socket
  - **public void setSocketWrapperListener(SocketWrapperListener listener)**: set the listener
  - **protected final void writeLine(String line)**: write a string to the socket. a newline character is automatically appended to the string.
  - **public void start()**: start the background listener thread if not already started
  - **public void close()**: closes the underlying socket if it is still open.

common.SocketWrapperListener (interface)

- Public Interface
  - **public void onReadLine(String line)**: gets called for every line that arrives on the socket. subclasses should override
  - **public void onReadError(IOException e)**: gets called if there was an exception

reading the socket. subclasses should override
  - **public void onReadFinish()**: gets called if the socket is closed (null is read from the socket). subclasses should override
  - **public void onWriteError(IOException e)**: gets called if there was an exception writing to the socket. subclasses should override

common.ClientMessageListener (interface)

- Description
  - callback functions to be executed for events originating in the client
  - public interface is not thread-safe
  - none of the methods should throw any exceptions. instead, the class should call the clientClose() function when something bad happens
- Public Interface
  - **public void login(String username)**: send a login message with the given username. may only be called in the NOT_LOGGED_IN state
  - **public void connectToBoard(int id)**: request to connect to the board with the given ID. may only be called in the NOT_CONNECTED state
  - **public void newBoard()**: request to create a new board and connect to it. may only be called in the NOT_CONNECTED state
  - **public void disconnectFromBoard()**: request to disconnect from the current board. may only be called in the CONNECTED state
  - **public void drawLine(Point p1, Point p2, Color color, int width)**: request to draw a line. may only be called in the CONNECTED state
  - **public void clientClose()**: the client is requesting to close down. the first time this function is called, perform any necessary cleanup, such as forwarding the “close” message to any connected objects. if it is not the first time this function has been called, do nothing. it may be called in any state

common.ServerMessageListener (interface)

- Description
  - callback functions to be executed for events originating in the server
  - none of the methods should throw any exceptions under normal operation (including s ocket failures)
  - public interface is not thread-safe
- Public Interface
  - **public void loginSuccess()**: indicate that login was successful. may only be called in the LOGIN_PENDING state
  - **public void error(code)**: indicate that an error occurred. error(100) may only be called in the LOGIN_PENDING state, and error(200) may only be called in the CONNECT_PENDING state
  - **public void connectToBoardSuccess(int id, List<String> users, Whiteboard data)**: indicate that the board connection was completed, and provide some data about the board. may only be called in the

CONNECT_PENDING state
    - **public void updatePixel(point, color)**: notify that the pixel at the given location has changed color. may only be called in the CONNECTED or DISCONNECT_PENDING states
    - **public void updateUsers(users)**: notify that the list of connected users has changed. may only be called in the CONNECTED or DISCONNECT_PENDING states
    - **public void disconnectFromBoardSuccess()**: notify that the client is no longer connected to a board. may only be called from the DISCONNECT_PENDING state
    - **public void serverClose()**: the server is requesting to close down. the first time this function is called, perform any necessary cleanup, such as forwarding the "close" message to any connected objects. if it is not the first time this function has been called, do nothing. it may be called in any state

client.ClientSocketHandler implements common.ClientMessageListener

- Description:
    - Wraps around a socket, converts client events (function calls) into ASCII and sends them through a socket
    - provides method for registering callback functions for handling responses that arrive from the server
    - thread safe -- public interface is mostly synchronized. the non-synchronized method lock the object before accessing any data
    - arriving messages are parsed and callback functions are executed on a background thread
    - this class keeps track of the system state and asserts that messages are only being sent/received when they are allowed to be sent/received
    - the implementor should double check the code for thread safety
    - if the clientClose() function is called, the socket is closed if it isn't already closed.
    - If the socket throws an error while reading a message (eg, this may be caused by the previous bullet), the serverClose() function of the ServerMessageListener (if present) is called
    - none of the inherited interface functions should throw any exceptions. Instead, call the clientClose() function if anything bad happens
- Public Interface
    - **public ClientSocketHandler(SocketWrapper s)**: creates a handler that delegates to the given socketWrapper. automatically sets this object as the listener for the SocketWraper.
    - **public void setServerMessageListener(ServerMessageListener l):** use the given listener object to send server-messages to the client
- Inherited Interface
    - **public void login(String username)**
    - **public void connectToBoard(int id)**

    - **public void newBoard()**
    - **public void disconnectFromBoard()**
    - **public void drawLine(Point p1, Point p2, Color color)**
    - **public void clientClose()**

client.ClientGUI implements common.ServerMessageListener

- Description
    - public interface is not thread-safe
    - store the list of currently connected usernames internally
    - stores the state of the board in an internal Whiteboard object
    - blits the contents of the whiteboard onto the screen at regular intervals using the graphics2d.drawImage(BufferedImage, null, 0, 0) function
    - when the serverClose() function is called, close the GUI if it is not already closed
    - when the GUI is closed, call the clientClose() function of the ClientMessageListener
    - none of the inherited interface functions should throw any exceptions. Instead, call the serverClose() function if anything bad happens
- Public Interface
    - **public void setClientMessageListener(ClientMessageListener l)**: use the given listener object to send messages to the server
    - **public void start()**: display the GUI on the screen, and start listening for events from the user
- Inherited Interface
    - **public void loginSuccess()**
    - **public void error(code)**
    - **public void connectToBoardSuccess(int id, List<String> users, Whiteboard data)**
    - **public void updatePixel(point, color)**
    - **public void updateUsers(users)**
    - **public void disconnectFromBoardSuccess()**
    - **public void serverClose()**

client.ClientController

- Description:
    - constructs a ClientGUI and a ClientSocketHandler, and connects them together (by adding them to each other as listeners)
    - public interface is not thread-safe
- Public Interface
    - **public ClientController(Socket s):** construct a ClientController that communicates to a server through the given socket
    - **public void run():** execute the main loop

server.ServerSocketHandler implements common.ServerMessageListener

- Description:
    - see ClientSocketHandler, switching the words “client” and “server”
- Public Interface
    - **public ServerSocketHandler(SocketWrapper s)**: creates a handler that delegates to the given socketWrapper. automatically sets this object as the listener for the SocketWraper.
    - **public void setClientMessageListener(ClientMessageListener l):** use the given listener object to handle incoming messages from the client
- Inherited Interface
    - **public void loginSuccess()**
    - **public void error(code)**
    - **public void connectToBoardSuccess(int id, List<String> users, Whiteboard data)**
    - **public void updatePixel(point, color)**
    - **public void updateUsers(users)**
    - **public void disconnectFromBoardSuccess()**
    - **public void serverClose()**

server.SessionHandler implements ClientMessageListener

- Description
    - listens for client events, and performs the appropriate server-side actions when they occur. this includes
        - logging in
        - joining/creating boards
        - drawing lines
        - cleaning up when the session “closes”
    - there is one session controller for each client connection
    - public interface is not thread-safe
    - when the clientClose() function is called for the first time (and only the first time), unregister the listener in proper WhiteboardStruct (if one is registered) and remove the current username from the AuthenticationBackend (if present), and call the serverClose() function of the ServerMessageListener (if present)
    - none of the inherited interface functions should throw any exceptions. Instead, call the clientClose() function if anything bad happens
- Public Interface
    - **public SessionHandler(AuthenticationBackend auth, WhiteboardMap boards)**: construct with the given parameters
    - **public void setServerMessageListener(ServerMessageListener l):** use the given listener object to send messages to the client
- Inherited Interface
    - **public void login(String username)**
    - **public void connectToBoard(int id)**
    - **public void newBoard()**

    - **public void disconnectFromBoard()**
    - **public void drawLine(Point p1, Point p2, Color color, int width)**
    - **public void clientClose()**

server.ServerController

- Description
    - Listens for incoming client connections. For each new client connection, it constructs a new ServerSocketHandler, constructs a new SessionController, and connects them together (by adding them to each other as listeners)
    - public interface is not thread-safe
- Public Interface
    - **public ServerController(ServerSocket s):** construct a ServerController that listens for new connections on the given server socket
    - **public void run():** execute the main loop

server.AuthenticationBackend

- Description
    - Authenticates usernames (ensures that they are unique)
    - thread-safe -- the methods are synchronized
- Public Interface
    - **public boolean login(username)**: return true and log in if username is available, else return false
    - **public void logout(username)**: make the username available if it is taken

server.WhiteboardStruct

- Description
    - An object containing a final Whiteboard object, a List<ServerMessageListeners> of objects listening on the whiteboard, a list of usernames of conencted users, and the ID number of the whiteboard
    - public interface is not thread-safe
- Public Interface
    - **public WhiteboardStruct(Whiteboard board, List<String> users, List<ServerMessageListener> listeners, int id)**: construct with the given variables
    - **public getWhiteboard()**: return a reference to a mutable whiteboard
    - **public getUsers()**: return a reference to a mutable list of usernames
    - **public getListeners()**: return a reference to a mutable listeners list
    - **public getID()**: return the integer ID

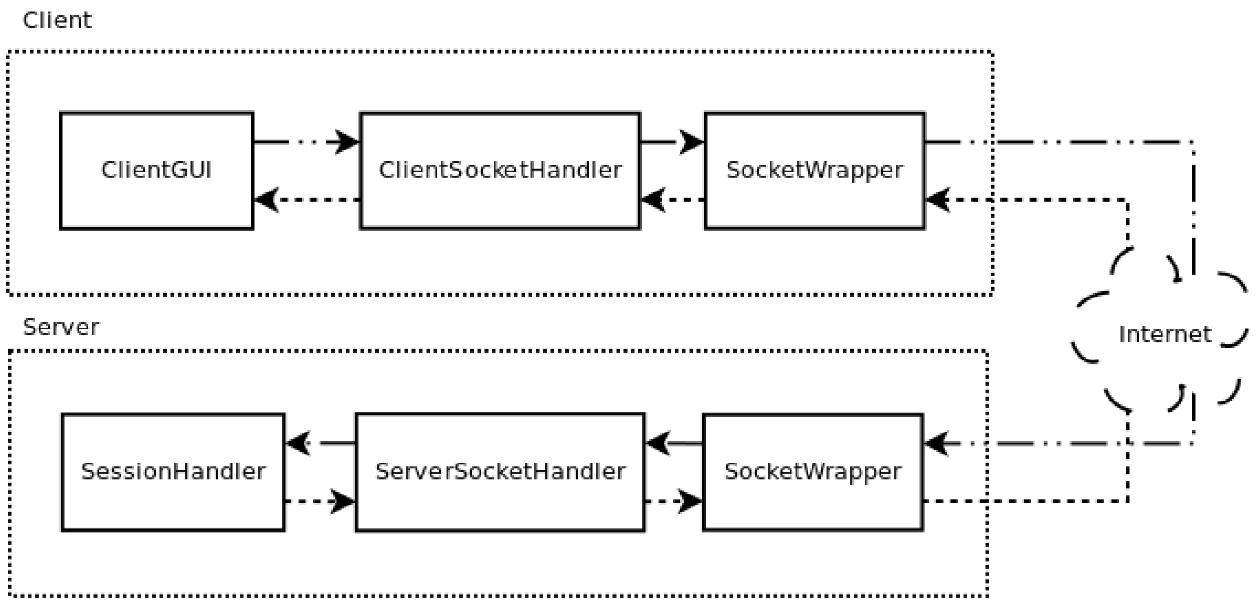
server.WhiteboardMap

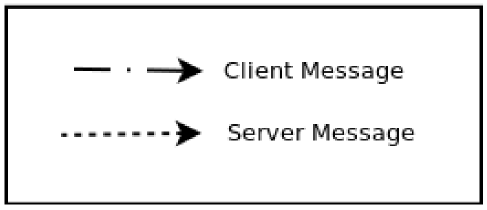
- Description
    - keeps maps from id's to boards with listeners
    - thread-safe -- the methods are synchronized

- Public Interface
    - **public WhiteboardMap()**: create empty collection
    - **public WhiteboardStruct newBoard()**: create a new Whiteboard Struct, add it to the map, and return it
    - **public WhiteboardStruct getBoard(int id)**: return the board struct with the given id, or null if none exist

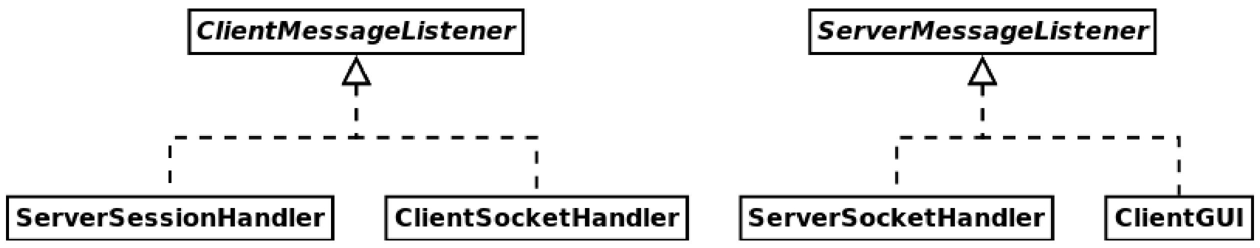
System Diagrams

Client messages originate from the ClientGUI, get passed through the socket, and are received by a SessionHandler. The SessionHandler performs the desired action, then generates one or more server messages that get returned to the ClientGUI through the socket.



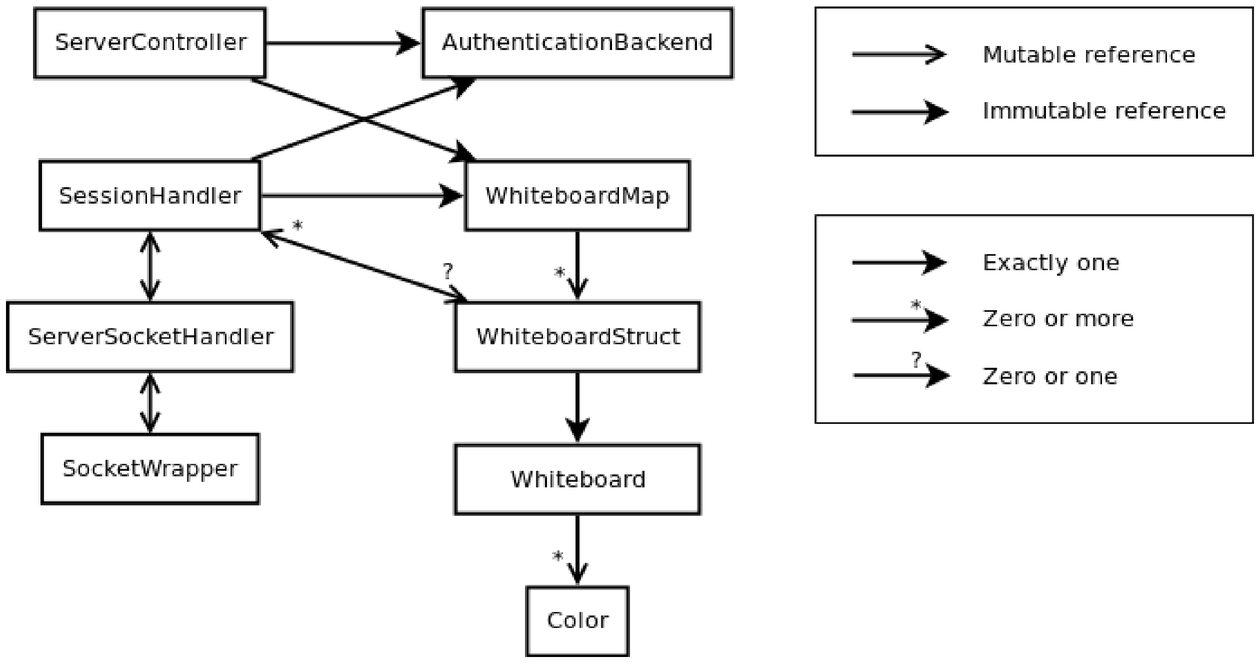


Here is the type hierarchy diagram. The listener interfaces exist for the purposes of linking together the modules in the above diagram. If a class is a “ServerMessageListener”, it means that its public interface contains java function calls that correspond to messages sent by the server.

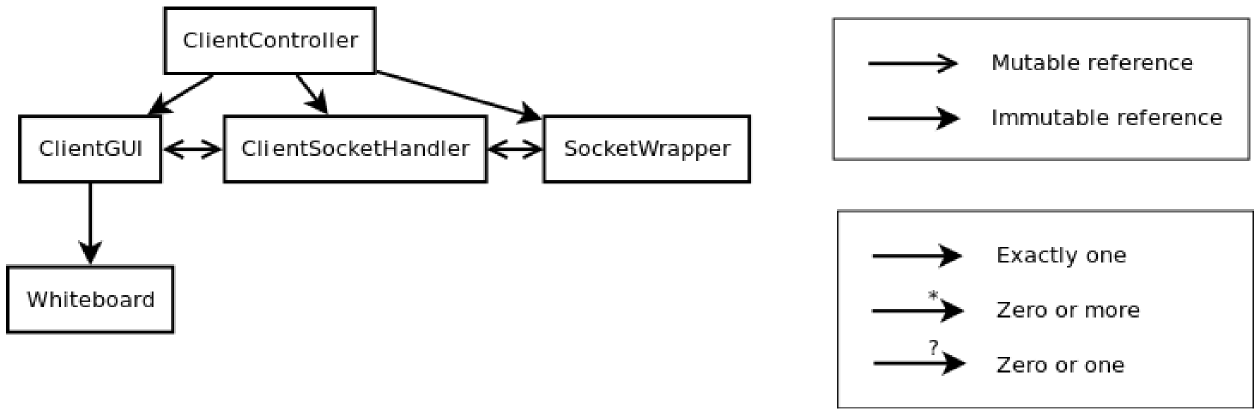


Here is a snapshot diagram of a server that is connected to multiple clients. The ServerController creates multiple SessionHandlers, one for each connected client. Each SessionHandler is connected to a ServerSocketHandler for sending and receiving messages over the network. Each SessionHandler communicates with the AuthenticationBackend, WhiteboardMap, and WhiteboardStruct classes.

Note that some objects (arrays, primitives, etc.) have been left out of the snapshot diagram for simplicity.



The snapshot diagram for the client is much simpler. The ClientController creates a ClientGUI, which contains the logic for the user interface, and a ClientSocketController, which sends and receives messages over the network. The ClientGUI holds a reference to a Whiteboard to keep track of what should be displayed on screen.



Concurrency Strategy

The correctness of the server/client protocol under concurrency is described in the State Diagram section. Here, we show why the server and client are free of race conditions and deadlocks.

Some of the classes in our program have thread-safe public interfaces. The rest do not. Here we categorize the classes based on this criteria.

Thread-safe and immutable

- Color
- Point

Thread-safe (strategies documented in the code)

- SocketWrapper
- ClientSocketHandler
- ServerSocketHandler
- AuthenticationBackend
- WhiteboardMap

Not thread-safe

- ClientGUI
- ClientController
- SessionHandler
- ServerController
- WhiteboardStruct
- Whiteboard

The immutable classes don't need any further discussion. The other thread-safe classes will encounter no race-conditions within their methods, but we still need to show that their usage doesn't have any race-conditions among several method calls. Finally, we need to show that the not thread-safe classes are used in a thread-safe manner.

First, lets address the thread-safety of the client. In the client, there is a main thread which briefly runs to construct the ClientController, there is a Swing event loop thread, and there is a thread inside the SocketWrapper that listens for incoming messages on the socket.

The classes we need to consider are: ClientSocketHandler, SocketWrapper, Whiteboard, ClientGUI, and ClientController.

ClientController

- only accessed from the main thread

ClientGUI

- public interface is only accessed from the ClientSocketHandler thread
- the Swing GUI is only modified by scheduling functions to be executed on the swing event loop
- some private variables are accessed by both the public interface functions and the functions in the Swing event loop: the Whiteboard, the board id, and the List<String> of connected users. The ClientGUI object should be manually locked wherever these variables are accessed.

Whiteboard

- this variable is protected by the ClientGUI's lock

ClientSocketHandler
SocketWrapper

- thread-safe public interface
- preconditions are easily met in a multithreaded environment

In the server, there is a main thread which constructs the ServerController and listens to the ServerSocket. Whenever a new connection arrives on the ServerSocket, the main thread constructs a SocketWrapper, which starts a new thread that listens for incoming messages on the socket. Thus, there is one new thread for each open connection to the server.

The classes we need to consider are: ServerController, SessionHandler, ServerSocketHandler, SocketWrapper, AuthenticationBackend, WhiteboardMap, WhiteboardStruct, and Whiteboard

ServerController

- only accessed from the main thread

SessionHandler

- after being constructed by the main thread, it is only accessed from the single SocketWrapper's thread

ServerSocketHandler
SocketWrapper
AuthenticationBackend
WhiteboardMap

- thread-safe public interface
- preconditions are easily met in a multithreaded environment

WhiteboardStruct

- all users of this class should manually lock the object before using any of its contents

Whiteboard

- contained within WhiteboardStruct, so it is protected by its lock

We have shown that the design is free of race conditions. We need to show that there are no deadlocks.

Argument for deadlock safety:

- when the server locks the WhiteboardStruct and the ServerSocketHandler at the same time, the WhiteboardStruct always gets locked first
- otherwise, server threads lock objects one at a time

Testing Strategy

Overview

Testing plan has been broken up into three main categories, depending on their scope. Unit tests are for testing individual modules. The mockito library will be used to mock inter-module connections. Unit tests will test the functionality of individual classes. Automatic integration tests will test the behavior of several modules in combination, particularly related to socket communication. Finally, manual human tests will be used to make sure that the system and user interface work as a whole.

Exact test cases are not provided here. See the codebase for details.

Unit Tests

- **Whiteboard**
- **Point**
- **Color**
- **AuthenticationBackend**

- **WhiteboardMap**
- **WhiteboardStruct**

Integration Tests

- **Socket Communication**
    - (ClientSocketHandler, ServerSocketHandler, SocketWrapper)
    - Construct a ClientSocketHandler and a ServerSocketHandler connected together
    - For each public interface function, calling the function on one handler should trigger the callback on the other with the same arguments
- **SessionHandler**
    - Test the SessionHandler in conjunction with a real WhiteboardStruct/Whiteboard, and a mock AuthenticationBackend/WhiteboardMap to ensure that the proper messages are sent to the client under the correct conditions

Manual Tests

- **ClientGUI, ClientController, ServerController**
    - Tested manually to ensure that they run as expected
- **System**
    - **Single client single board:** Single client logs in with a valid username and creates a new board.
    - **Single client failed board connect:** Single client logs in and tries to join a board that doesn't exist.
    - **Single client several boards:** Single client logs in and creates a new board with id 1. The client draws a bit. The client then disconnects from board1. The client creates a new board with id 2. The client draws a bit. The client disconnects from board2. The client connects to board1 again and should see his original drawings from board1.
    - **Several clients same username:** Single client logs in with username "user1". Another client then tries to login with username "user1". The second client should be told that it must choose a different username.
    - **Several clients one and several boards:** A client logs in with username "user1". Another client logs in with username "user2". User1 creates a new board with id 5. User1 starts drawing. User2 connects to board5. User2 should see the drawings that user1 has already made. Both user1 and user2 should see that they are both connected to board5. User1 disconnects from board5 and creates a new board with id 6. User2 should now see that it is the only one editing board5, and user1 should see that it is the only one editing board6. Both users should be able to independently draw on their separate boards.