Design Document

6.005 Project 2: Collaborative Whiteboard

Ankush Gupta, Justin Martinez, Kevin Wen

Datatype

GUI Description

Our GUI will consist of a single window composed of two parts: the InfoPanel and the Easel. The InfoPanel occupies the right side of the GUI, next to the Easel. Upon launching the GUI, the user is asked to input the server's IP address and port number. Upon connecting to the server, the GUI with its two major components will appear. The InfoPanel contains the list of active whiteboards and the users currently connected to the current whiteboard. From here, a client will be able to join a whiteboard or create a new whiteboard, as well as changing its username. The Easel contains the whiteboard and drawing tools. The client will only be able to have one whiteboard open at a time.

Client-Server Description

Each client will have one drawing space locally, which is able to be updated by the server, depneding on their currently selected whiteboard. The server will maintain the main copy of all actions that have been performd to each whiteboard. Whenever a client performs a drawing action, the action is shown on the the client's local whiteboard immediately. The action is also put onto an outgoing message queue of requests to be sent to the server. When the server responds with a drawing action, it will be put onto the client's incoming queue of image updates. The client's drawingBuffer will then receive these updates, and be appropriately layered underneath whatever action the client is currently performing.

When multiple clients modify at the same time (i.e. one draws where the other erases) the TCP protocol will guarantee ordering. Therefore, the edit which is sent to the server later will always take place after the first edit.

Implementation

Client

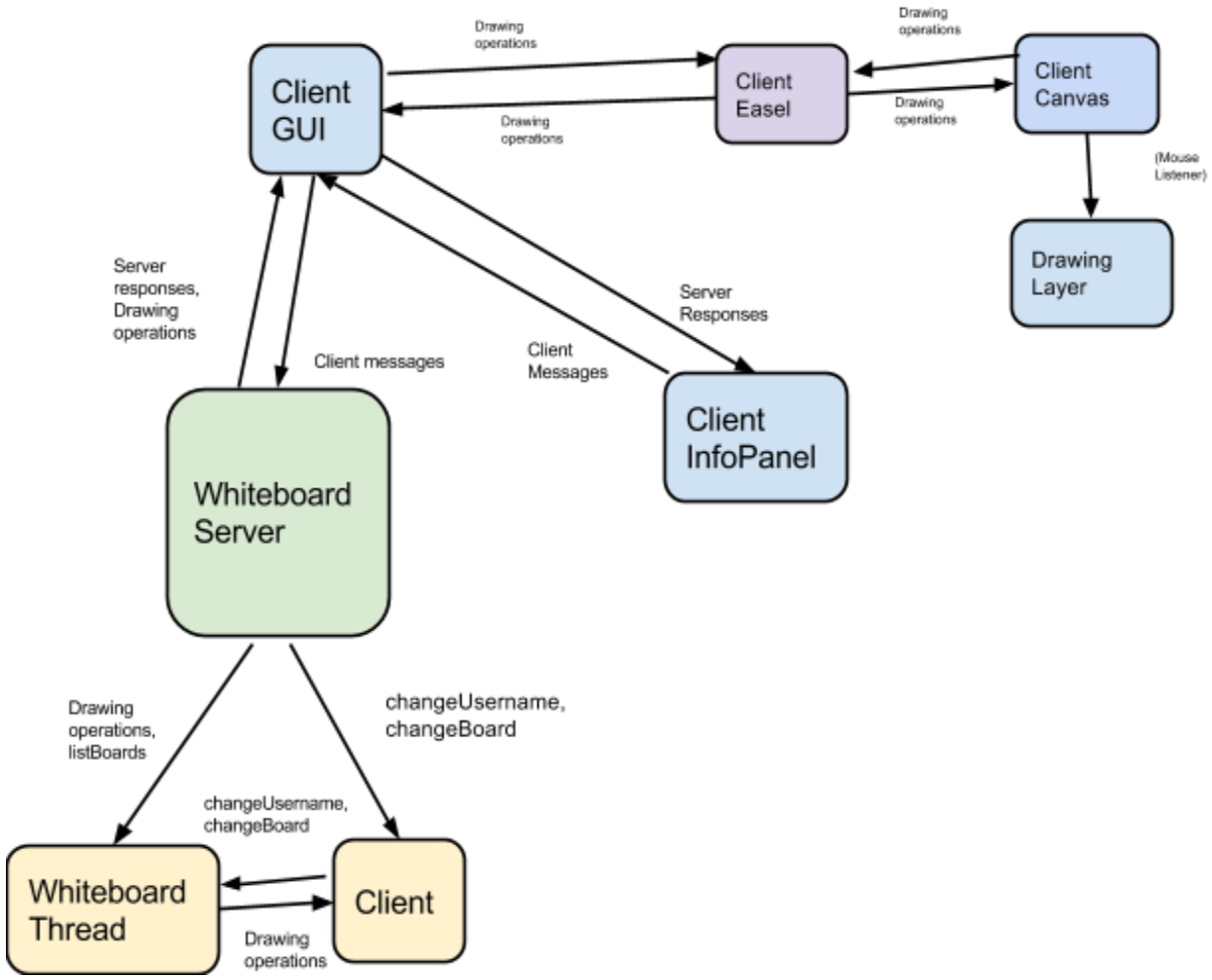
- **ClientGUI:** A GUI class that contains the information of the available whiteboards to edit, as well as the users connected to each whiteboard. This class will initialize and maintain the connection with a WhiteboardServer. The class contains two sub-GUI's to logically divide the information: a ClientEasel and a ClientInfoPanel.
- **ClientInfoPanel:** ClientInfoPanel is a GUI class that manages and displays information from the server to the client, as well as allowing the user to switch whiteboards, create new whiteboards, and change usernames.
    - Displays the current number of whiteboards on the server as a list and tells the

user which whiteboard he or she is currently connected to.
    - The user is able to switch to different whiteboards by either using the arrow keys to maneuver up and down the list, singleclicking a designated board name, or typing the name of the server on the textbox provided.
    - Entering a name unique to all the whiteboard names on the server will create a new whiteboard.
    - Displays the users connected to the current whiteboard as a list.
- **ClientEasel:** ClientEasel represents the sub-GUI surrounding a ClientCanvas, which has a drawing surface that allows the user to draw on it using tools such as freehand, shapes, etc. and allows the user to erase as well.
- **ClientCanvas:** ClientCanvas represents the actual drawing surface where the user can draw using tools such as freehand, shapes, etc., and allows the user to erase as well. This keeps a list of local DrawingLayers, which represent actions that were performed by the local client, and a drawingBuffer, which is updated by the server with the actions of other clients. Once the DrawingLayers have been processed by the server, they are removed from the list of DrawingLayers, since these drawings have been performed on the drawingBuffer.
- **DrawingLayer:** A class to represent a layer in a ClientCanvas. This layer is given an ID number so that it can be removed once the server responds that this message has been drawn to the buffer. This class also generates the message for drawing itself to the server.

Server

- **WhiteboardServer:** A server class that handles all of the incoming client to server messages, as well as keeping track of the messages used to construct all boards.
    - Creates a new Client instance and a WhiteboardThread for each client who connects.
    - Adds WhiteboardThread to a synchronized List<WhiteboardThread> and keeps track of boards in a Map<String, List<String>> where the String is the name of the board and the List<String> is the list of global messages sent to construct the given board.
    - The WhiteboardServer has helper methods to send global messages to all connected clients and to send messages to all clients connected to a particular whiteboard (while simultaneously adding the message to the board's list) and these methods are synchronized to ensure that each client receives the messages in the same order.
- **WhiteboardThread:** An internal class in WhiteboardServer that is created for each client connected to the server instance.
    - Contains a reference to a Client object which is modified on certain requests (such as changeBoard and changeUsername requests) and serves to identify which WhiteboardThread is connected to which client and thus uniquely identify them.
- **Client:** A model used to represent a client.
    - Contains an id which is unique to each client. The id is also used to create an initial username and serves to distinguish clients which are connected to the same board.

Snapshot Diagram of Client-Server Interaction



Protocol

Using telnet, clients connect to a centralized server and send commands. Each request will have its own flags and values. From a server architecture standpoint, the protocol functions similarly to a chat client that supports multiple chatrooms.

There are a few commands that manipulate the Client's state from the Server's perspective (setting the username, changing the Board that the client is connected to, etc), and the rest of the commands are messages that are sent to every Client connected to the same Board, similarly to how chat servers handle chat messages. The server will also keep a "transcript" of all previous messages that are not client-specific to allow new clients to get caught up to the current state of the board.

It's up to the client to act on operations received from the server, but since messages are received

in the same order and each client handles them the same, each client will have the same end result.

Client Messages

CLIENT_OPERATION messages

setUsername [string]
- server sends you a USER_CHANGED message
  - if the username is unique:
    - server sends everybody connected to the same board (including yourself) a USER_QUIT message followed by a USER_JOINED message for your old and new username
  - if username was not unique:
    - server sends nobody a USER_QUIT and USER_JOINED message

getUsername
- server returns a USERNAME message

listBoards
- server replies with CURRENT_BOARDS message

changeBoard [string]
- server creates a new board with this ID should it not exist
  - server sends ALL connected threads a NEW_BOARD message
- server sends you a BOARD_CHANGED message
- server also sends BOARD_CONTENTS message whose elements are to be processed just like normal CLIENTSIDE_OPERATION responses

exit
- sends a USER_QUIT message to coworkers and server closes the current socket

DRAWING_OPERATION messages

```
DRAWING_OPERATION: OPERATION SPACE BRUSH FILLED_FLAG? SPACE POINTS SPACE
    DRAWING_ID | ERASE_ALL
ERASE_ALL: "eraseAll" SPACE DRAWING_ID
BRUSH: "-c" SPACE "[" INT "]" SPACE "-w" SPACE "[" INT "]" SPACE
OPERATION: "drawLineSegment" | "eraseLineSegment" | "drawRectangle" | "drawOval"
POINTS: "-p" SPACE "[" POINT+ "]"
POINT: INT SPACE INT SPACE?
DRAWING_ID: "-i" SPACE "[" STRING "]"
FILLED_FLAG: SPACE "-f"
STRING: ([a-z] | [A-Z] | INT )+
INT: [0-9]+
SPACE: " "
```

The server responds to every DRAWING_OPERATION by sending ALL clients connected to the same board (including the client that sent the message) the exact same message it received and letting

the Clients handle processing in the order that the messages were received.

The Strings of operations will be placed in an ordered list to ensure proper action order (to deal with overlapping). When a client connects to a board, all of the previous DRAWING_OPERATIONS (along with a few other operations as defined in the CLIENT_OPERATION section) will be sent to the client to process allowing it to reconstruct the current state of the board from a transcript of previous actions.

A separate method on the client-side will analyze the Strings and rebuffer the whiteboard accordingly.

Server Responses

MESSAGE_NAME:	**message;**
CURRENT_BOARDS:	“currentBoards " (string)+;
NEW_BOARD:	"newBoard " string;
CLIENTSIDE_OPERATION:	DRAWING_OPERATION | PUBLIC_USER_OPERATION;
PUBLIC_USER_OPERATION:	USER_QUIT | USER_JOINED;
BOARD_CONTENTS:	(CLIENTSIDE_OPERATION \n)+;
BOARD_CHANGED:	"boardChanged " string string;
	// old_board and new_board in that order
USER_QUIT:	"userQuit " string;
USER_JOINED:	"userJoined " string;
USERNAME_CHANGED:	"usernameChanged " string string;
	// old_username and new_username in that order
USERNAME:	"username " string;

Concurrency

Each of the clients will be connected to the server on their own thread. All information about the brush type/drawing type will be included in the drawing operation messages. The server will have one event handling thread for each of it’s own whiteboards, forcing all drawing operations to each whiteboard to be single-threaded. Additionally, the ClientGUI will be listening to all incoming messages and sorting them accordingly, and the ClientEasel/ClientInfoPanel will make sure that any incoming messages are handled in a thread safe manner, since Swing is not thread safe.

Testing Strategy

The ClientCanvas and GUI have user interface components that are tested manually. These classes also have methods that send/receive messages to the server, so we separated these message-sending methods so that we can write automated tests for these. For example, DrawingOperationProtocol is a separate class that handles the message parsing between the WhiteboardServer and DrawingLayer classes.

Many more detailed and precise tests are detailed in the respective test classes in the project, but here is an overview of our testing strategy:

Areas tested:

- Can join and leave boards multiple times
- All drawing operations function properly
- Usernames update when other clients join/leave/change username
- Each outgoing message is properly formatted
- Receives and parses messages properly

We manually tested WhiteboardServer and ServerCanvas for concurrency issues by simultaneously connecting with multiple users. These classes do not have user interface components that will need to be tested. We also wrote automated tests for any message-generating method, and made sure that the server can parse messages properly as well

Areas tested:

- Multiple clients can connect to same whiteboard
- Multiple clients can connect to different whiteboards
- Each outgoing message is properly formatted
- Receives and parses messages properly
- Organizes incoming commands into a single thread
- Organizes outgoing commands properly based on the commands that have come in

(Defining whiteboards. One of your first tasks is to determine what whiteboard is in your system. Is a whiteboard more like a 2D array of pixels, or more like an arrangement of shapes? How are they named? What drawing operations are available, and what does it mean to remove or erase a drawing? What happens when two users draw on or modify the whiteboard simultaneously? Will both of their edits be reflected or might one override the other? Are some changes independent and some interfering, and if so, how is interference resolved? The most important piece of advice here is to **keep it simple**. You can always extend your program, but if you've created a complex mess, it's hard to go back.)

(Designing architecture and protocol. You must also devise a network architecture and a protocol for this project. A client/server architecture is potentially the easiest choice here, but it isn't required. You should strongly consider using a text-based protocol, which is generally easier for testing and debugging. We used text-based protocols in the problem sets. Services that use plain text protocols — e.g. HTTP or SMTP — can talk to a human just as well as another machine by using a client program that sends and receives characters. You've already used telnet as a tool for interacting with text-based protocols in the problem set. Make use of it here too.)

(Design for safe concurrency. In general, making an argument that an implementation is free of concurrency bugs (like race conditions and deadlocks) is veryt difficult and error-prone. The best strategy therefore is to design your program to allow a very simple argument, by limiting your use of concurrency and especially avoiding shared state wherever possible. For example, one approach is to use concurrency only for reading sockets, and to make the rest of the design single-threaded. And recall that even though user interfaces are concurrent by nature, Swing is not thread-safe. Recommended reading: Threads and Swing.)

Handling multiple users. Since realtime collaborative whiteboard is not realtime without at least two people, your system must be able to handle multiple users connected at the same time. One reasonable design approach could be using one thread for reading input from each client and having a central state machine representing the state of the server (using one more thread, to which each of the client threads pass messages through a shared queue).