Real-Time Collaborative Whiteboard Application

1. Overview

This report explains the key components and their interactions in a real-time collaborative whiteboard application. The application is built using a backend WebSocket server (Node.js) and a frontend with HTML5 Canvas and vanilla JavaScript. The primary goal is to enable multiple users to draw and erase on a shared canvas in real time, assigning each user a unique random color. The system ensures that drawing events are synchronized across all connected clients, even when new clients join or network connections are disrupted.

2. Key Components and Their Interaction

2.1 Backend (Node.js WebSocket Server)

The backend is implemented using Node.js and the WebSocket library. It handles client connections, manages drawing events, and broadcasts updates to all connected clients.

2.1.1 Importing Dependencies

The WebSocket server is created using the `ws` library and is attached to an HTTP server.

```
let express = require("express");
let app = express();
let httpServer = require("http").createServer(app);
let WebSocket = require("ws");
let wss = new WebSocket.Server({ server: httpServer });
```

2.1.2 Color Generation and Assignment

These two functions generate a random hexadecimal color code. A loop runs six times to append a random character from the set '0123456789ABCDEF' to the color string and make sure the color for each user is unique.

```
function generateRandomColor() {
    const letters = '0123456789ABCDEF';
    let color = '#';
    for (let i = 0; i < 6; i++) {
        color += letters[Math.floor(Math.random() * 16)];
    }
    return color;
}
```

```
function getUniqueColor() {
    let color;
    do {
        color = generateRandomColor();
    } while (usedColors.has(color));
    usedColors.add(color);
    return color;
}
```

2.1.3 WebSocket Connection Handling

These codes handle new WebSocket connections. When a client connects, the server acts like this: Assigns a unique ID (ws.id) to the client. Assign a unique color (ws.color) to the client. Sends the client their assigned ID and color.Sends the entire drawing history to the client to synchronize the canvas state.Adds the client to the connections array. Logs a message indicating a user has connected.

```
wss.on("connection", (ws) => {
    ws.id = nextID++;
    ws.color = getUniqueColor();
    ws.send(JSON.stringify({ type: "assign_id_and_color", id: ws.id,
color: ws.color }));
    ws.send(JSON.stringify({ type: "history", history: drawingHistory
}));
    connections.push(ws);
    console.log("A user connected");
```

2.1.4 Message Handling

This part handles messages received from clients. When a client sends a message, the server first parses the message and then creates a broadcast object with the message type, client ID, and color. Depending on the message type, For "clear," Add a clear event to the drawing history. For "erase": Adds an erase event to the drawing history and includes coordinates. For drawing events ("ondraw", "ondown"): Adds the event to the drawing history with coordinates, then broadcasts the message to all other connected clients (excluding the sender).

```
ws.on("message", (data) => {
    let message = JSON.parse(data);
    if (["ondraw", "ondown", "clear", "erase"].includes(message.type)) {
        let broadcastMessage = {
            type: message.type,
            id: ws.id,
            color: ws.color
        };

        if (message.type === "clear") {
            drawingHistory.push({
                type: "clear",
                timestamp: Date.now()
            });
        } else if (message.type === "erase") {
            broadcastMessage.x = message.x;
            broadcastMessage.y = message.y;
            drawingHistory.push(broadcastMessage);
        } else {
            broadcastMessage.x = message.x;
            broadcastMessage.y = message.y;
            drawingHistory.push(broadcastMessage);
        }

        connections.forEach((con) => {
            if (con !== ws && con.readyState === WebSocket.OPEN) {
                con.send(JSON.stringify(broadcastMessage));
            }
        });
    }
});
```

2.1.5 Disconnection Handling

If disconnected, the server will log a message indicating the user has disconnected. Then, remove the client from the connections array.Moreover, Frees up the client's color by removing it from the usedColors set.

```
ws.on("close", () => {
    console.log("User disconnected");
    connections = connections.filter((con) => con !== ws);
    usedColors.delete(ws.color);
});
```

2.1.6 Static File Serving and Server Setup

Sets up the server to serve static files and listens on a specified port.

```
app.use(express.static("public"));

let PORT = process.env.PORT || 8080;
httpServer.listen(PORT, () => console.log(`Server started on port
${PORT}`));
```

2.2 Frontend (HTML5 Canvas and Vanilla JavaScript)

The front end is implemented using HTML5 Canvas and vanilla JavaScript. It handles user interactions, renders drawing events, and communicates with the WebSocket server.

2.2.1 Canvas Setting

To build the corresponding website, we use this function to dynamically adjust canvas size.

```
function resizeCanvas() {
    const displayWidth = Math.floor(canvas.clientWidth);
    const displayHeight = Math.floor(canvas.clientHeight);

    if (canvas.width !== displayWidth || canvas.height !== displayHeight) {
        canvas.width = displayWidth;
        canvas.height = displayHeight;
    }
}

resizeCanvas();
window.addEventListener('resize', resizeCanvas);
```

2.2.2 Coordinate Calculation

To convert mouse event coordinates to canvas coordinates we use getBoundingClientRect to get the canvas's position relative to the viewport which subtracts the canvas's left and top offsets from the mouse event coordinates to get the coordinates relative to the canvas.

```
function getCanvasCoordinates(event) {
    const rect = canvas.getBoundingClientRect();
    return {
        x: event.clientX - rect.left,
        y: event.clientY - rect.top
    };
}
```

2.2.3 WebSocket Connection

This part is responsible for handling WebSocket communication and rendering drawing events on the canvas in a collaborative whiteboard application. It establishes a WebSocket

connection to the server and listens for messages that dictate how the canvas should be updated based on actions taken by other connected clients.

When the WebSocket connection is successfully established, the onopen event is triggered, logging a message to the console to indicate that the connection has been made. The core functionality, however, lies within the onmessage event handler, which processes incoming messages from the server. These messages are expected to be in JSON format and contain a type field that specifies the kind of action to be performed.

If the message type is "assign_id_and_color", the client stores the assigned unique ID and color. This information is crucial for identifying the client's actions on the canvas and ensuring that each client's drawings are rendered in their assigned color.

For message types "ondown" and "ondraw", the code calculates the canvas coordinates from the normalized coordinates sent by the server. The "ondown" event initializes the drawing state for a client, storing the starting point of a new drawing action. The "ondraw" event, on the other hand, continues the drawing by creating a line from the last known position to the new position, effectively rendering the drawing action on the canvas. This is done using the CanvasRenderingContext2D methods moveTo and lineTo, followed by stroke to actually draw the line.

The "clear" message type triggers the clearRect method, which clears the entire canvas, effectively erasing all drawings. This is useful for resetting the canvas state, for instance, when a user decides to start a new drawing session.

The "erase" message type simulates an eraser tool. It draws a white circle at the specified coordinates, effectively removing any drawings within that area. This is achieved using the arc method to define the circle and the fill method to fill it with the background color (white in this case).

Lastly, the "history" message type is used to synchronize the canvas state when a new client joins. It clears the canvas and then iterates over the drawing history sent by the server, replaying each drawing event to reconstruct the current state of the canvas. This ensures that all clients see the same drawing, regardless of when they joined the session.

```
let socket = new WebSocket("ws://localhost:8080");
socket.onopen = () => console.log("WebSocket connected");
socket.onmessage = (event) => {
    let data = JSON.parse(event.data);
    if (data.type === "assign_id_and_color") {
        myID = data.id;
        myColor = data.color;
    } else if (data.type === "ondown") {
        const pos = {
            x: data.x * canvas.width,
            y: data.y * canvas.height
        };
        if (!drawingStates[data.id]) {
            drawingStates[data.id] = {};
        }
        drawingStates[data.id].lastX = pos.x;
        drawingStates[data.id].lastY = pos.y;
    } else if (data.type === "ondraw") {
        const pos = {
            x: data.x * canvas.width,
            y: data.y * canvas.height
        };
        if (drawingStates[data.id] && drawingStates[data.id].lastX !==
undefined) {
            ctx.strokeStyle = data.color;
            ctx.beginPath();
            ctx.moveTo(drawingStates[data.id].lastX,
drawingStates[data.id].lastY);
            ctx.lineTo(pos.x, pos.y);
            ctx.stroke();
            drawingStates[data.id].lastX = pos.x;
            drawingStates[data.id].lastY = pos.y;
        }
    } else if (data.type === "clear") {
        ctx.clearRect(0, 0, canvas.width, canvas.height);
    } else if (data.type === "erase") {
        const pos = {
            x: data.x * canvas.width,
            y: data.y * canvas.height
        };
        ctx.beginPath();
        ctx.arc(pos.x, pos.y, 20, 0, Math.PI * 2);
        ctx.fillStyle = "#ffffff";
        ctx.fill();
```

(part of the long code)

2.2.4 Mouse Event Handling

Several events like ‘mousedown’,’mouseup’,’mousemove’ are being listened to capture user drawing actions and send them to the server.

```
canvas.addEventListener('mousedown', (e) => {
    const pos = getCanvasCoordinates(e);
    ctx.strokeStyle = myColor;
    ctx.beginPath();
    ctx.moveTo(pos.x, pos.y);
    mouseDown = true;

    if (!isErasing) {
        socket.send(JSON.stringify({
            type: "ondown",
            x: pos.x / canvas.width,
            y: pos.y / canvas.height
        }));
    } else {
        socket.send(JSON.stringify({
            type: "erase",
            x: pos.x / canvas.width,
            y: pos.y / canvas.height
        }));
        ctx.beginPath();
        ctx.arc(pos.x, pos.y, 10, 0, Math.PI * 2);
        ctx.fillStyle = "#ffffff";
        ctx.fill();
    }
});
```

(part of long code)

2.2.5 Control Buttons

This is the button for the 'clear' and 'erase' functions. When clicking the 'clear' button, reset the drawingStates object. It will send the server a "clear" message to notify other clients.

When clicking the 'erase' button, toggle the isErasing flag. It changes the button text to reflect the current mode ("Erase" or "Draw").").

```
document.getElementById("clear").addEventListener("click", () => {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    drawingStates = {};
    socket.send(JSON.stringify({ type: "clear" }));
});

document.getElementById("erase").addEventListener("click", () => {
    isErasing = !isErasing;
    document.getElementById("erase").textContent = isErasing ? "Draw" :
"Erase";
});
```

3. Communication Workflow

3.1 Client Connection:

A client connects to the WebSocket server. The server assigns a unique ID and color to the client and sends them to the client. The server sends the current drawing history to the client to synchronize the canvas state.

3.2 Drawing Events:

The client captures user interactions (e.g., mouse events) and sends drawing events to the server. The server processes the events, updates the drawing history, and broadcasts the events to all connected clients. Each client receives the events and renders them on their canvas.

3.3 Client Disconnection:

When a client disconnects, the server removes the client from the active connections list and frees up the assigned color. The remaining clients continue to receive updates from the server.

4. Conclusion

This collaborative whiteboard application demonstrates the effective use of WebSockets for real-time communication and HTML5 Canvas for rendering. The backend server manages client connections, processes drawing events, and ensures state synchronization across all clients. The frontend captures user interactions, sends drawing events to the server, and renders the synchronized state of the canvas. The system is designed to handle network disruptions by implementing reconnection strategies and state recovery mechanisms. This solution provides a robust and interactive platform for real-time collaboration.