Documentation: WebSocket Server

This document explains how I implemented features like connection throttling, rate limiting, session management, state sharing, dynamic heartbeat intervals, and message prioritization in my WebSocket server from scratch without using any packages, along with steps to test each feature using your frontend.

---

1. Connection Throttling

Goal: Limit the number of WebSocket connections from the same IP to prevent abuse (e.g., limiting a user to 3 connections per IP).

Implementation:

- Use a `Map` to store the number of connections per IP (`connectionCounts`).
- Check the count when a client tries to connect. If the IP exceeds the limit, reject the connection and close it.

Code Snippet:

javascript

```
wss.on("connection", (ws, req) => {
    const clientIp = req.socket.remoteAddress;
    const currentConnectionCount = connectionCounts.get(clientIp) ||
0;
    if (currentConnectionCount >= MAX_CONNECTIONS_PER_IP) {
        console.log("Too many IP connections");
        ws.send("Exceeded the limit of users for WebSocket");
        ws.close();
        return;
    }
    connectionCounts.set(clientIp, currentConnectionCount + 1);
});
```

Testing:

- Open multiple browser tabs with your `index.html` file and try to connect more than the allowed number of clients from the same IP. The WebSocket server should reject the excess connections and display the error message.

---

2. Rate Limiting

Goal: Control the number of messages a client can send to the server within a given time period to avoid spam or overload.

Implementation:

- Use Redis to store session data for each client, including how many messages they've sent and the last time they sent one.
- Allow a maximum of `MAX_MESSAGES_PER_SECOND` messages per client within a defined timeframe (e.g., per second).

Code Snippet:

javascript
Copy code

```
ws.on("message", async (message) => {
    const sessionData = await
redisClient.hGetAll(`session:${clientId}`);
    let messageCount = parseInt(sessionData.messageCount) || 0;
    let lastMessageTimestamp =
parseInt(sessionData.lastMessageTimestamp) || Date.now();
    const currentTime = Date.now();

    if (currentTime - lastMessageTimestamp < 1000) {  // 1 second
        messageCount++;
    } else {
        messageCount = 1;
        lastMessageTimestamp = currentTime;
    }

    if (messageCount > MAX_MESSAGES_PER_SECOND) {
        ws.send("Rate limit exceeded. Please slow down.");
        return;
    }

    await redisClient.hSet(`session:${clientId}`, {
        messageCount: messageCount,
        lastMessageTimestamp: lastMessageTimestamp,
    });
});
```

Testing:

- Open `index.html` in multiple browser tabs and try sending messages rapidly (e.g., type and hit send quickly). Once the rate limit is exceeded, the server should reject messages with a warning.

---

3. Session Management

Goal: Manage client sessions by storing and retrieving session information (like message count, last activity, etc.) across multiple WebSocket server instances.

Implementation:

- Store session data (e.g., message count, timestamps) in Redis, using the client ID (`clientId`) as the key.

Code Snippet:
javascript
Copy code

```
const clientId = uuidv4();
const exists = await redisClient.hExists(`session:${clientId}`,
"messageCount");
if (!exists) {
    await redisClient.hSet(`session:${clientId}`, "messageCount",
0);
    await redisClient.hSet(`session:${clientId}`,
"lastMessageTimestamp", Date.now());
}
```

Testing:

- Open multiple tabs in your browser with `index.html` and observe if each session stores unique data. Check the Redis store to see if session data is being created and updated correctly.

---

4. State Sharing (Across WebSocket Server Instances)

Goal: Ensure that session state is shared across multiple WebSocket server instances by using Redis Pub/Sub to manage messaging between servers.

Implementation:

- Publish messages to Redis on one server and subscribe to the same channel on other servers to receive and process those messages.

Code Snippet:

javascript
Copy code

```
// Publish message to Redis
pubClient.publish("broadcast", JSON.stringify(clientMessage));

// Subscribe and handle messages from Redis
subClient.on("message", (channel, message) => {
    if (channel === "broadcast") {
        const parsedMessage = JSON.parse(message);
        enqueueMessage(parsedMessage.message, parsedMessage.priority
|| 1);
        processMessages();
    }
});
```

Testing:

- Set up two instances of your WebSocket server on different ports (e.g., 8200 and 8300) and observe message synchronization across both instances. Sending a message from one server should be reflected in all clients connected to either server.

---

5. Dynamic Heartbeat Intervals

Goal: Adjust the frequency of heartbeat messages based on the number of connected clients to manage load dynamically.

Implementation:

- Adjust the `heartbeatInterval` depending on the load (e.g., fewer clients = shorter interval; more clients = longer interval).

Code Snippet:

javascript
Copy code

```
const adjustHeartbeatInterval = () => {
    if (connectedClients <= 1) {
        heartbeatInterval = 5000;
    } else if (connectedClients <= 2) {
        heartbeatInterval = 15000;
    } else {
        heartbeatInterval = 30000;
```

```
    }
    console.log(`Adjusting heartbeat interval to ${heartbeatInterval
/ 1000} seconds.`);
};

setInterval(() => {
    wss.clients.forEach((client) => {
        if (client.readyState === WebSocket.OPEN) {
            client.send(JSON.stringify({ type: "heartbeat",
timestamp: Date.now() }));
        }
    });
}, heartbeatInterval);
```

Testing:

- Open multiple tabs with `index.html`. Observe the heartbeat frequency and ensure that it adjusts dynamically based on the number of connected clients. Check the console to verify the adjustment.

---

6. Message Prioritization

Goal: Prioritize the handling of certain types of messages (e.g., system alerts or critical messages) over regular chat messages.

Implementation:

- Use a priority queue to store messages and process them in order of priority (e.g., high-priority messages are processed first).

Code Snippet:
javascript
Copy code

```
const enqueueMessage = (message, priority) => {
    priorityQueue.push({ message, priority });
    priorityQueue.sort((a, b) => b.priority - a.priority);
};

const processMessages = () => {
    while (priorityQueue.length > 0) {
        const { message } = priorityQueue.shift();
        wss.clients.forEach((client) => {
```

```
            if (client.readyState === WebSocket.OPEN) {
                client.send(message);
            }
        });
    }
};
```

Testing:

- Open multiple tabs and send different types of messages from the client (`critical`, `normal`, `low priority`). Ensure that high-priority messages are processed first. You can test this by sending low-priority messages first, followed by a high-priority one, and observing the order of display.

Testing Using `index.html` and `app.js`

Your frontend (`index.html` and `app.js`) is used to join the WebSocket server and send messages. To test each feature:

1. Open multiple tabs in your browser.
2. Observe how the WebSocket server handles connections, message rate limiting, session persistence, message prioritization, and heartbeat intervals.
3. Check your browser's console and UI (`<ul>`) for feedback on message delivery, connection limits, and rate limits.

By modifying `app.js`, you can send different types of messages to test prioritization:

javascript
Copy code

```
document.querySelector('button').onclick = () => {
    const messageContent = document.querySelector('input').value;
    const priority = document.querySelector('select').value;  // 
Assuming you have a select for priority
    const message = {
        type: priority,  // e.g., "critical", "normal", or "status"
        content: messageContent
    };
    socket.send(JSON.stringify(message));
};
```

Conclusion

This document provides an end-to-end guide on how to implement and test key features for your WebSocket server, including connection throttling, rate limiting, session management, dynamic heartbeat intervals, and message prioritization. You can easily test these features using multiple tabs of your `index.html` file loaded in the browser.