

WebSocket Relay Chat

Sponsor: Dr Ibrahim Imam

Mohaned Ahmed Sam Downs Madison Hamilton
Sam Smallwood

April 28, 2026

Contents

1	Executive Summary	4
2	System Description	5
2.1	System Requirements	5
2.2	Initial Systems Specification	5
2.3	Final Specifications	6
2.3.1	Protocol	6
2.3.2	Server	9
2.3.3	Client	10
2.3.4	Web Demo	11
2.4	System Diagrams	11
2.5	Hardware Overview Diagram	14
2.6	Software Overview Diagram	15
2.7	Economical, Technical and Time Constraints	16
3	Detailed Implementation	18
3.1	Server Implementation	18
3.2	Client Library Implementation	19
3.3	Demo Implementation	19
3.3.1	Webpages	19
3.3.2	Hardware	20
4	Testing	22
4.1	Test Environment	22
4.2	Methodology	22
4.3	Analysis of Results	23
4.3.1	Performance Evaluation	25
5	Impact	26
6	Societal Contribution	28
7	Engineering Standards	30
8	Conclusion	32
9	Recommendations for Future Works	33

A	Source Code	35
A.1	Java Server	35
A.2	Typescript Client	43

1 Executive Summary

The goal of this project was to create a messaging service using the WebSocket protocol. The benefit of using WebSockets is that they are lighter than HTTP and are full duplex, which eliminates the need for polling. The messaging service we created consisted of three parts: the protocol, the server, and the client. To allow client to client messaging over WebSockets, a client-server protocol, we needed to create a simple messaging protocol to run over the WebSocket connection. We created a system of JSON packets for each client to send commands to the server. We then created the server itself in Java. The server takes commands from clients and sends messages to other clients on their behalf. Java allows the server to be easily portable and run on a variety of systems. The last section of service, the client, was written in TypeScript. The client gives programmers a more convenient interface for interacting with the server, so that they don't have to manually generate and parse packets. In addition to the service itself, we also created a demo for the service, a set of websites for placing and managing orders in a restaurant. These pages use the client library to pass orders and order information back and forth, allowing the kitchen to see incoming orders and keep track of them. In addition to the code itself, we thoroughly documented the design of the protocol, the client interface, the demo code, and the repositories themselves, along with build instructions. To facilitate the showcase and demo, we created a continuous integration/continuous deployment pipeline to automatically pull changes to the server and web demos and host them on a private server.

2 System Description

2.1 System Requirements

In our initial meetings with our sponsor, Dr. Imam, the requirements for the project began to be outlined. Our sponsor wanted to learn more about WebSockets and their applications, so we were tasked with creating a messaging service that utilized them. It would have a central server and some kind of client that was capable of sending messages to other clients through the server. The system also needed to be lightweight, allowing it to run on low power devices. Outside of those specifications, we had broad flexibility with regards to design choices and programming languages. We also decided to try to make the system as portable as possible so that it could run on many systems. Later on, our sponsor also added a demo as a requirement. The demo was to be a pair of websites for ordering food and managing orders in a restaurant. The websites would use the protocol that we created to communicate with each other. The system requirements can be described using functional and performance constraints. The system should support multiple concurrent clients while maintaining reliable message delivery. It should provide low-latency communication under normal conditions. Each request must include a unique identifier to ensure correct matching of responses, and the system must handle invalid or malformed requests by returning appropriate error messages.

2.2 Initial Systems Specification

The first thing we designed was the protocol for the service. We decided to use JSON to define the packets and commands because it is human-readable, which is good for debugging, and is also highly supported, with parsing libraries available for almost every language; many even support it natively. The protocol began with 5 principle commands: log in, log out, check active users, send a message, and broadcast a message. Each packet would contain at least a `packetType` field, indicating its purpose. Some commands required fields for extra data such as `requestedUsername` for logging in, or `recipientList` for sending a message.

As we were designing the protocol, we decided to use Java for the server, since we were all familiar with it, and it has several libraries to choose from for working with WebSockets. Java also provided the portability we were looking for, allowing us to run the same code on almost any platform and architecture.

For our WebSocket library, we chose the library Java-WebSocket¹, written by GitHub user TooTallNate. It provided a painless system for setting up a WebSocket server and modifying it as desired by overriding the default event handlers.

For the client, we planned on writing it JavaScript. JavaScript is also highly portable, running in any browser on almost any system. JavaScript also has native support for both WebSockets and JSON, making it a natural choice for a client. A JavaScript client also made sense since our demos were going to be web based, and JavaScript is the primary language of the web.

2.3 Final Specifications

2.3.1 Protocol

The protocol is formatted in JSON. When a client wants something, it sends a packet containing a command to the server, to which the server will reply at minimum an acknowledgement or an error. The packet's purpose and format are defined by the numeric field **packetType**. The possible packet types are defined in Table 1. Each packet sent by the client also requires a numeric field **id**, which can be any number. When the server gets a request from a client, it stores the id number and copies it into the same field in its response packet, allowing the client to match the response to the request it belongs to.

Table 1: Packet type numbers and corresponding meaning

Type Number	Purpose
1	Login
2	Check Active Users
3	Send Message
4	Incoming Message
5	Broadcast Message
6	Logout
20	Error

For a message from the server to a client, there are 4 possible formats that packets may take. For requests which simply require acknowledgement (login, logout, send, broadcast), the server replies with a packet containing the **packetType** of the original request, as well as the **id**, as shown below.

1 {

¹<https://github.com/TooTallNate/Java-WebSocket>

```
2     "packetType": <type>,  
3     "id": <id>  
4 }
```

For responses to requests for active users, the server sends a packet including the basic acknowledgement, along with an **activeUsers** field containing string list of users which are online, including the requester. The format is shown below.

```
1 {  
2     "packetType": 2,  
3     "id": <id>,  
4     "activeUsers": [  
5         <usernameA>,  
6         <usernameB>,  
7         ...  
8     ]  
9 }
```

If the client has an incoming message, the server will send a packet of type 4, along with the string fields **sender** and **message** containing appropriate values. This packet type does not require an id since it is not sent in response to a request. The format is shown below.

```
1 {  
2     "packetType": 4,  
3     "sender": <sender username>,  
4     "message": <message content>  
5 }
```

The last type of response is an error, for which packets contain the **packetType** 20, a numeric **errorCode** field, a string **errorMessage** field, and an optional **errorData** field for extra data. The supported error codes are in Table 2, and a format of an error packet is shown below.

```
1 {  
2     "packetType": 20,  
3     "id": <id>,  
4     "errorCode": <error number>,  
5     "errorMessage": <text description of error>,  
6     "errorData": {  
7         ...  
8     }  
9 }
```

Covered next are the 5 requests a client may send to a server, along with the expected behavior of each. Every packet sent by the client must contain an **id** field to identify it. The server simply copies this field into the reply, so it is up to the client to ensure that each id is unique (our client simply increments the id used for each packet). Beginning with a login request, the client must send a packet with **packetType** 1, and a string **requestedUsername** field for the username they wish to log in under. The server will send a blank

Table 2: Error codes, corresponding meaning, and additional data

Error Code	Message	Data
1	Malformed packet	"inactiveUsers": []
2	Username is taken	
3	One or more users not found	
4	User is not logged in	
5	User is already logged in	
6	Server side error	

acknowledgement if the login was successful. If the client is already logged in, or the username they request is already in use, the server will respond with an appropriate error message. The format of a login packet is shown below.

```

1 {
2   "packetType": 1,
3   "id": <id>,
4   "requestedUsername": <username>
5 }
```

To check the active users, the client will send a packet with **packetType** 2. No additional data is required for this type of request. The server will respond with a packet containing **activeUsers** with a list of the active users. The format of the request is shown below.

```

1 {
2   "packetType": 2,
3   "id": <id>
4 }
```

To send a message to a set of users, the client will send a packet with **packetType** 3. This packet must contain 2 additional fields, **recipientList** with a list of recipient usernames, and **message** with the message to send those users. If the message is sent successfully to every recipient, the server will return a standard acknowledgement. If the client is not logged in, a corresponding error will be returned. If only some of the recipients are online, the server will respond with an error message containing the users which could not be reached. The message will still be sent to all recipients which were online. The format for a message request is shown below.

```

1 {
2   "packetType": 3,
3   "id": <id>,
4   "recipientList": [
5     <usernameA>,
6     <usernameB>,
7     ...
8   ],
```

```
9      "message": <message string>
10  }
```

To broadcast a message to every active user, the client may send a packet with `packetType` 5, along with the `message` field. This will send the message to every active user, including the requester. The server will respond with a standard acknowledgement. If the client is not logged in, the server will respond with an appropriate error. The format for a broadcast request is shown below.

```
1  {
2      "packetType": 5,
3      "id": <id>,
4      "message": <message string>
5  }
```

Finally, to log out, a client may send a logout request simply containing `packetType` 6. If the client is logged out, the server will send a standard acknowledgement. If the client is not logged in, the server will send an appropriate error. It is important to note that logging out does not terminate the underlying WebSocket connection. The primary purpose of logging out would be to log in again with a new username. If the client wishes to end the session entirely, they need not explicitly log out, as they will be automatically logged out when the WebSocket connection is terminated. The format for a logout request is shown below.

```
1  {
2      "packetType": 6,
3      "id": <id>,
4  }
```

2.3.2 Server

The server is built with Java 21. It implements the server side of the protocol we designed for the project. As mentioned previously, the WebSocket component is built on the MIT licensed Java-WebSocket library, written by GitHub user TooTallNate. This library provides a `WebSocketServer` class that can be extended to build a server using WebSockets. The extended class overrides the base class functions for common events like `onOpen` or `onMessage`. This kept us from having to start from scratch and prevented us from introducing bugs at the network level. We also used the Jackson² library for parsing, serializing, and deserializing JSON data. This library is widely used and well tested, giving us confidence that our JSON handling would be correct. It also

²<https://github.com/FasterXML/jackson>

provides an interface for converting JSON data directly into Java objects using records. This helped us ensure correctness in packet generation and parsing, as well as preventing the need for manually checking that incoming packets are formatted correctly. The project uses the Gradle build system, allowing simple, consistent, cross-platform builds.

2.3.3 Client

The client library was written in TypeScript. TypeScript was used instead of pure JavaScript because it provides type-safety, which is important for any protocol, while also compiling down to pure, web-compatible JavaScript. The client library is the client-side implementation of the protocol. If a developer wishes to use the protocol, the class provides the following functions:

- `WSClient(url)`: class constructor
- `getOpenPromise()`: return a promise that resolves true on successful connection
- `login(username)`: log in
- `checkActiveUsers()`: check active users
- `sendMessage(recipientList, message)`: send a message
- `addMessageListener(callback)`: add a callback for messages received
- `broadcast(message)`: broadcast
- `logout()`: log out

The callback function for the message listener takes two parameters, sender and message: `callback(sender, message)`. All functions except for `checkActiveUsers` and `sendMessage` return a `Promise<number>`. When a response is received from the server, the promise is resolved with a number corresponding to success (0) or an error code given by the server. Promises allow actions to be deferred, so that the main loop doesn't freeze while a response to a request is being awaited. The command `checkActiveUsers` returns a `Promise<string[]>`, which resolves to a string array containing the active users when the server responds. The command `sendMessage` returns a `Promise<SendMessagePromiseData>` which resolves to a custom type containing the success or error code, along with a list of recipients who were not online if applicable.

2.3.4 Web Demo

The demos were developed in HTML, CSS, and JavaScript. As the demos didn't need to be incredibly complex, no extra HTML/CSS frameworks or JavaScript libraries were used. The demo consisted of two pages: an ordering page and a kitchen page. The ordering page allows the users to choose from a list of items to order, add their name, and add special notes, and then place the order. The kitchen page shows incoming orders and allows the user to move orders between queues for placed, in progress, and completed. As a user in the kitchen moves the orders from section to section, the user on the order page can see the progress of their order update. To facilitate this process, we created a small standard for communication between the WebSocket clients. When an order is placed, a message is broadcast from the order client with the following data:

```
1 {  
2   "orderId": <random uuid>,  
3   "orderName": <name>,  
4   "orderItem": <item>,  
5   "orderNotes": <notes>  
6 }
```

The order UUID is a randomly generated universally unique identifier (UUID), which allows each order to be uniquely identified with an extremely low chance for collisions. Any kitchen client which hears the order puts a new entry in their placed queue. When a user on a kitchen client moves the order to the next queue, a message is broadcast to the other kitchen clients and the client which ordered the food, notifying them that the order has changed queues. Those messages are formatted as follows:

```
1 {  
2   "orderId": <uuid>,  
3   "orderStatus": <status>  
4 }
```

where `orderStatus` of 1 means being prepared, 2 means completed, and 3 mean removed from the system. This allows the kitchen clients and the ordering client to stay in sync with each other as the order progresses.

2.4 System Diagrams

Figure 1 gives an example of a simple communication using the protocol. Alice connects through the WebSocket and requests to log in. The server accepts her request. Bob then connects and requests to login, and the server accepts. Alice then requests to send a message to Bob. The server sends the message to Bob and then responds to Alice saying the request has been completed.

Alice then requests to broadcast a message. The server sends the broadcast to both Alice and Bob, then responds to Alice with an acknowledgement. Bob then sends a request to send a message to Alice and Eve. The server sends the message to Alice, but Eve is not online, so the server responds to Bob letting him know that Eve is not active currently. Alice then logs out, the server acknowledges, and Alice disconnects. Bob then requests a list of active users. Since Alice has logged out, the server sends Bob a list containing only his own username, as he is the only one online.

Figure 1: A communication example using the protocol

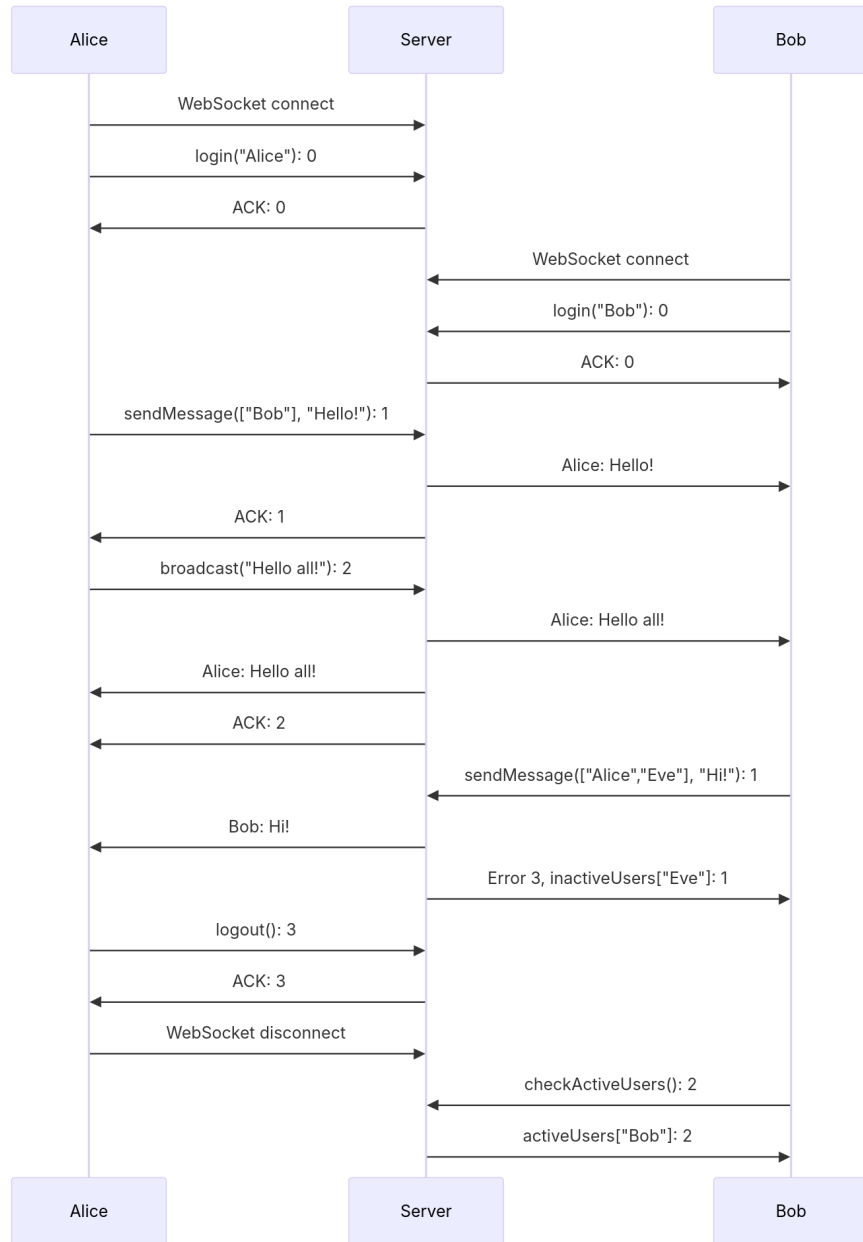
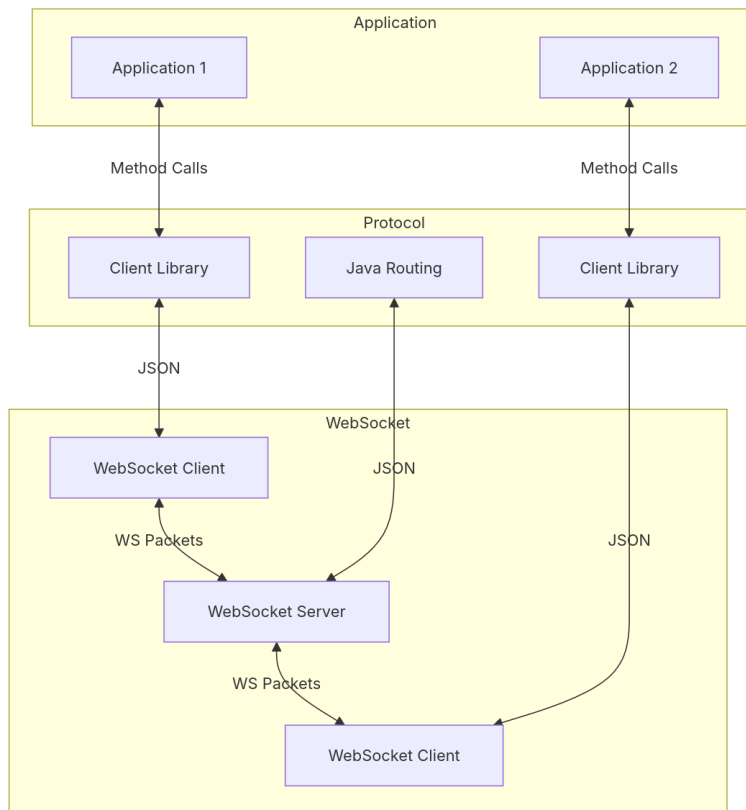


Figure 2 shows the layers and protocols involved in the stack. In the application layer, applications which want to use the messaging protocol issue calls to the client methods to request services. The clients, in the protocol layer, do the necessary processing to convert those requests to JSON in the protocol's format. Those JSON packets are then passed to the WebSocket

clients in the WebSocket layer, which encapsulate them and send them to the server over the internet. The WebSocket server then decapsulates the JSON packets and hands them off to the Java server logic, in the protocol layer. Once it generates a response in JSON, it passes the response back to the WebSocket layer, which routes them to the appropriate clients. The data then moves back up through the layers until it reaches the application layer, completing a request.

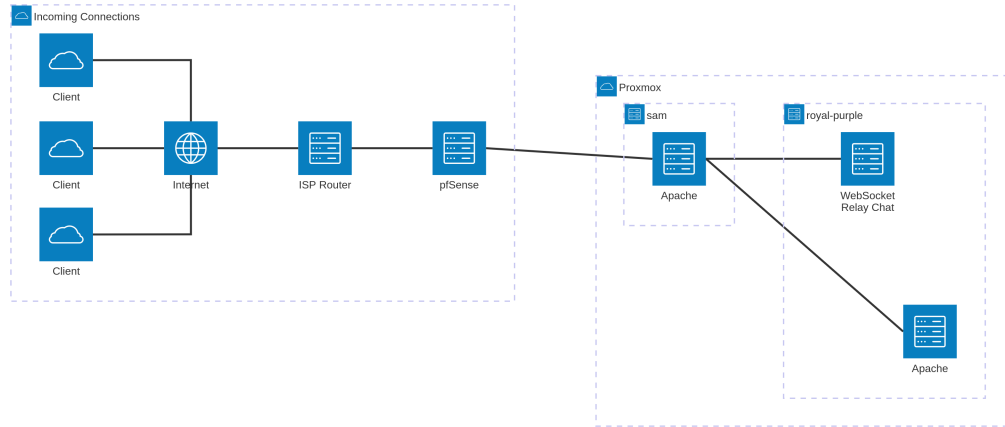
Figure 2: The protocols and layers involved in the stack



2.5 Hardware Overview Diagram

Figure 3 shows the path incoming connections take when connecting to the WebSocket server and accessing the web demo. The hardware lives in Sam Downs's house, so incoming connections route through the ISP's router, to the pfSense firewall, to an Apache reverse proxy, and finally to either the WebSocket server or the Apache server serving the demo's webpages.

Figure 3: Networking Layout of the Hardware



2.6 Software Overview Diagram

Figure 4: Several applications connected to the server through the client

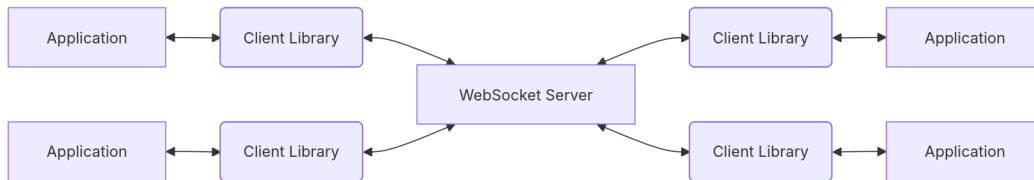
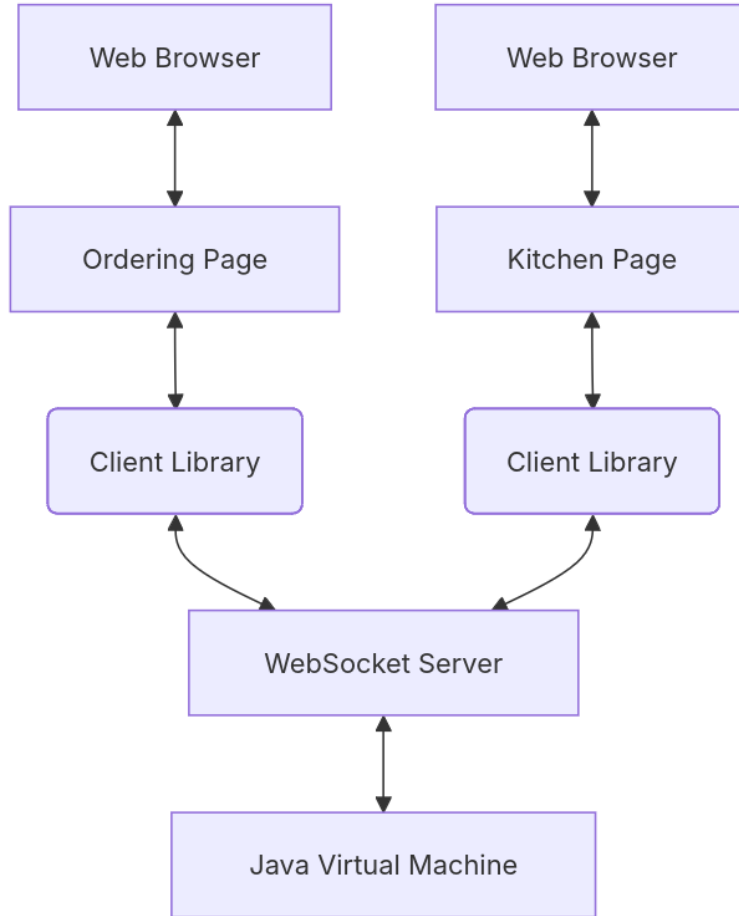


Figure 5: Software component layout for the demo



2.7 Economical, Technical and Time Constraints

The project was constrained economically by the requirement that the final system be lightweight and practical to run without paid infrastructure. This affected the design from the beginning. The server was written in Java and built with Gradle, the client was written in TypeScript and compiled to browser-compatible JavaScript, and the web demo used ordinary HTML, CSS, and JavaScript rather than a large frontend framework. The main third-party libraries, including Java-WebSocket, Jackson, and Google Guava, are available through public package repositories and do not require a commercial license for our intended use. As a result, the project can be built, deployed, and demonstrated without a paid software stack. The primary operating cost is the machine or virtual machine that hosts the server and static demo pages.

The main technical constraint was the need to support real-time communication while keeping the system simple enough to understand, debug, and extend. WebSockets solved the communication problem because they provide a persistent, full-duplex channel between the browser and the server. JSON solved the packet-format problem because it is readable during debugging and is supported directly by both JavaScript and Java libraries. These choices were not the most compact possible representation, but they made the protocol easier to verify with browser developer tools and easier for another programmer to implement in a different language. On the server side, Java 21 gave us a portable runtime, and the Java-WebSocket library handled the lower-level connection work so that our code could focus on packet routing and protocol behavior. On the client side, TypeScript gave us type checking around request and response objects while still producing JavaScript that runs in a standard browser.

The system also had to operate within the limits of a semester-long capstone schedule. Because the project included a protocol, a server, a client library, documentation, and a working restaurant demo, we had to make several scope decisions. We prioritized the parts that proved the central idea of the project: clients can log in, discover active users, send direct messages, broadcast messages, and receive updates in real time. Features such as persistent order storage, password-based authentication, and administrative controls were treated as future work instead of core requirements. This kept the implementation focused on the WebSocket relay itself and prevented the demo from becoming larger than the communication system it was meant to showcase.

These constraints led to a design that is intentionally small but complete. The server listens on a single port and can be placed behind a reverse proxy. The protocol uses a small set of packet types and error codes. The client library hides the request identifiers and Promise handling from application code. The demo then builds on that foundation by broadcasting order and status messages between the customer page and the kitchen page. The result is a system that meets the sponsor's goals while leaving clear paths for a future team to add persistence, security, and larger-scale deployment features.

3 Detailed Implementation

The implementation was divided into three main parts: the Java WebSocket server, the TypeScript client library, and the web demo that uses the client library to show a real application of the protocol. Keeping these parts separate made the project easier to test and easier to explain. The server is responsible for maintaining connections and enforcing the protocol. The client library is responsible for hiding the raw JSON packets from application programmers. The demo is responsible for showing that the protocol can support a real-time workflow rather than only a simple chat example.

3.1 Server Implementation

The server was implemented in Java 21 using the Java-WebSocket library. The main server class extends `WebSocketServer` and overrides the event methods used by the library. When a client connects, `onOpen` records the connection in the server log. When a message arrives, `onMessage` passes the raw string to a private message handler. When a socket closes, `onClose` checks whether that socket belongs to a logged-in user and removes that user from the active login table.

Incoming packets are parsed with Jackson's `ObjectMapper`. The first validation step happens before a packet is routed: the server checks that the JSON contains both a numeric `packetType` and a numeric `id`. If those fields are missing or malformed, the server sends an error packet instead of attempting to process the request. If both fields are present, the server uses the packet type to dispatch the request to the correct handler function. This keeps the event-handling code small and moves the command-specific behavior into `PacketHandler`.

The server tracks logged-in users with a Guava `BiMap` from usernames to WebSocket connections. A bidirectional map is useful because the server needs to look up a socket from a username when sending a direct message, but it also needs to look up a username from a socket when generating an incoming message packet or cleaning up after a disconnect. The `BiMap` also prevents the same socket from being associated with two different usernames at the same time, which supports the protocol's duplicate-login error behavior.

Each protocol command has a corresponding handler. The login handler deserializes the request into a Java record, checks whether the requested username is already present, and either inserts the username/socket pair or returns an error. The active-user handler returns the current key set from the login table. The direct-message handler verifies that the sender is logged in, rejects

an empty recipient list, sends the message packet to every online recipient, and returns an error containing the inactive usernames when one or more recipients cannot be reached. The broadcast handler sends the message packet to every logged-in client, including the sender. The logout handler removes the socket from the login table but leaves the WebSocket connection open, allowing the same connection to log in again under a new username.

3.2 Client Library Implementation

The client library was written in TypeScript and exported as a `WSClient` class. Its purpose is to turn the packet protocol into a set of ordinary method calls. Application code can call methods such as `login`, `checkActiveUsers`, `sendMessage`, `broadcast`, and `logout` without manually building JSON packets or matching responses to requests.

The client maintains an incrementing request id. Before it sends a request, it increments the id, creates a Promise, and stores the Promise's resolve and reject functions in a waiting table indexed by that id. When the server later sends an acknowledgement or an error packet with the same id, the client uses the waiting table to resolve or reject the correct Promise. This design is important because several requests can be in flight at the same time over one WebSocket connection. The id field keeps the responses from being confused with each other.

Incoming message packets are treated differently from acknowledgement packets because they are not responses to a request. When the client receives packet type 4, it calls the registered message listener with the sender and message body. This gives the application a simple callback interface for real-time updates. The demo uses this listener to receive new orders and order-status updates, but the same interface could also be used for a chat room, a notification system, or another collaborative application.

3.3 Demo Implementation

3.3.1 Webpages

The web demo consists of an ordering page and a kitchen page. Both pages connect to the deployed WebSocket server through the client library and log in with generated usernames. The generated usernames avoid collisions between browser tabs and allow several customers or kitchen displays to be open at the same time during a demonstration.

On the ordering page, the user selects a menu item, enters a name, optionally enters notes, and submits the order. The page creates a UUID for the order and broadcasts a JSON message containing the order id, customer name, item, and notes. The ordering page then listens for status messages that contain the same order id. When a matching status update arrives, the page updates the visible progress indicator from placed to preparing or ready.

On the kitchen page, incoming order messages are rendered as cards in the queue column. Kitchen staff can move an order from queued to in progress, from in progress to ready, and from ready to complete. Each movement broadcasts a small JSON message containing the order id and the new status code. Other kitchen pages receive the same status broadcast and move their local copy of the card as well, which keeps multiple kitchen displays synchronized.

The demo intentionally does not store orders in a database. Instead, it demonstrates the behavior that the WebSocket relay was designed to provide: multiple clients can receive the same update immediately and adjust their local state in response. This makes the demo a direct test of the server and client library rather than a separate application with unrelated infrastructure.

3.3.2 Hardware

The demo had to be hosted on some hardware. Strictly speaking this isn't hardware since it was done via configuring software, but it's effectively the hardware the demos ran on. We had an Apache server serving the web pages, our WebSocket server running on that same hardware to service connections from clients, and webhooks from GitHub to keep everything up-to-date.

Figure 3 provides an overview of the hardware used. It consists of the Linux containers `sam` and `royal-purple` hosted on Sam Downs's Proxmox. `sam` was already made and being used for other things and has an Apache server running as a reverse proxy to route incoming HTTP requests and encrypt them via HTTPS. `royal-purple` was a brand new Arch container made for this project. It contains an Apache server hosting the HTML of the web demos and the WebSocket server we made. Tangentially used was a pfSense firewall and the ISP router, they're not too important, just included for completion's sake since traffic *does* route through them.

Figure 3 also shows the path network connections take to get to the server. Incoming connections from the Internet first encounter the ISP's router, that router forwards in-use ports to the pfSense firewall. The pfSense then forwards based on ports to the relevant machines. Importantly, the pfSense forwards all traffic on 80/443 (HTTP/HTTPS) to the `sam` container. `sam` is an in-use container that has an Apache server running in it.

The Apache server inspects the destinations subdomain and location and forwards it on as needed. Here it forwards any `sam` subdomain destination to `royal-purple`. There, an Apache server serves up the web demo pages, along with our report. There's an additional check for any destination of `sam.dwns.dev/api/ws` being forwarded to port 8080 instead of 80/443. This means any connection to `sam.dwns.dev/api/ws` goes to the websocket server instead of trying to load a webpage from the web demo Apache.

To keep the hosted demo up-to-date and not require us to manually update it each time we changed something, we set up webhooks from GitHub to keep it updated. Webhooks in GitHub are just HTTP requests GitHub will make to a user-defined URL whenever some event happens with information about that event attached.

For us, we had a webhook push to `sam.dwns.dev/api/push.php` every time a commit was pushed in the organization. The push contained a body with information on which repository had the new commit, and the PHP script handling the request would use the body to determine the relevant repository on the server to update. Once updated, it would use the Makefile each repository has to run whatever needs to happen to properly update the repository — e.g. the report's repository just needs to run `latexmk -pdf report.tex` to rebuild the document.

4 Testing

Testing the WebSocket service required more than checking that one function returned the expected value. The system depends on a persistent socket connection, JSON packet parsing, shared login state, and coordination between multiple browser clients. For that reason, our testing process combined protocol-level tests, client-library integration tests, and full demo tests using both the order page and the kitchen page.

4.1 Test Environment

We tested the system in two environments. During development, we ran the Java server locally and connected to it from browser tabs using the TypeScript client library and the browser developer console. This made it easy to send crafted JSON packets, inspect raw responses, and repeat failing cases quickly. For system testing, we used the deployed version of the project. The deployed environment placed the Java WebSocket server behind an Apache reverse proxy and served the static order and kitchen pages from the same host. This environment was important because it tested the same connection path that would be used during the project demonstration.

The deployed environment also allowed multi-client testing across machines instead of only within one local browser. Several team members could connect to the same server at the same time, log in under different generated usernames, and verify that direct messages, broadcasts, and demo order updates reached the correct clients. This caught issues that would be easy to miss in a single-client test, such as whether a status update sent from one kitchen display was reflected on another kitchen display.

4.2 Methodology

The first level of testing focused on the protocol handlers in the Java server. For each packet type, we tested a successful request and the main error paths documented in the protocol. The login handler was tested with an available username, a duplicate username, and a socket that was already logged in. The direct-message handler was tested with online recipients, offline recipients, an empty recipient list, and an unauthenticated sender. The broadcast and logout handlers were tested from both authenticated and unauthenticated sockets. Malformed-packet tests were also run by sending JSON that was missing required fields or contained the wrong field types.

The second level of testing used the TypeScript client library rather than raw packets. These tests verified that the library generated the correct packet types, incremented request ids, resolved Promises on successful acknowledgements, rejected Promises on error packets, and delivered incoming packet type 4 messages to the registered listener callback. This level was necessary because the client library is the interface that application programmers would actually use; a correct server would still be inconvenient if the client failed to match responses to requests correctly.

The final level of testing exercised the restaurant demo as a complete system. We opened multiple order pages and multiple kitchen pages at the same time. A customer page placed an order, a kitchen page received it, and the kitchen page moved it through the placed, preparing, ready, and complete states. While this happened, the original order page updated its status display, and other kitchen pages moved their copy of the same order card. These tests verified not only that broadcast messages were delivered, but also that the application-level JSON messages contained enough information for each browser to update its local state.

4.3 Analysis of Results

The final build passed the representative cases in Table 3. The protocol tests confirmed that the server follows the packet contract described in Section 2. Successful requests produced acknowledgement packets with the original request id, and error cases produced the expected error code. The login-table tests were especially important because much of the server’s behavior depends on the relationship between usernames and WebSocket connections. Closing a browser tab without sending a logout packet still removed the user from the active table, which prevents stale usernames from remaining online after a disconnect.

The client-library tests confirmed that the library hid the packet details correctly. Requests were sent with unique ids, acknowledgements resolved the Promise associated with the matching id, and server error packets rejected the matching Promise. The special direct-message error case also returned the inactive-user list, which allows application code to tell the user which recipients could not be reached. Packet type 4 messages were delivered through the listener callback instead of being treated as request responses, which is the behavior needed for real-time application updates.

The demo tests showed that the broadcast pattern was sufficient for the restaurant workflow. An order message contained the order id and order details, so every kitchen page could create a matching card. A status message

Table 3: Representative test coverage and final outcomes

Area	Scenario	Result
Authentication	Login with an available username	Pass
Authentication	Login with a duplicate username on another socket	Error 2 returned
Authentication	Login request from an already authenticated socket	Error 5 returned
Routing	Direct message to online recipients only	Delivered, then acknowledged
Routing	Direct message including offline recipients	Error 3 with <code>inactiveUsers</code>
Routing	Direct message from an unauthenticated client	Error 4 returned
Validation	Request with missing fields or wrong field types	Error 1 returned
Broadcast	Broadcast sent from an authenticated client	Delivered to all active clients
Session state	Browser tab closed without explicit logout	User removed by <code>onClose</code>
Client library	Response matched back to request by id	Promise resolved/rejected correctly
Client library	Incoming packet type 4 handled by listener callback	Callback fired with sender/message
System demo	Order created on customer page appears on kitchen page	Pass
System demo	Kitchen status update is reflected on customer page	Pass
System demo	Two kitchen pages stay synchronized for same order id	Pass

contained the same order id and a status code, so order pages and kitchen pages could update only the order that changed. This pattern worked with multiple browser windows open at once, which was the most important system-level requirement for the demo.

Testing also clarified the limits of the current design. Because orders are stored only in browser memory, a newly opened kitchen page will not automatically receive old orders until another relevant status update is broadcast. That behavior is acceptable for the capstone demo because persistence was outside the required scope, but it would need to change in a production restaurant system. A future version should add server-side order storage, stronger authentication, and automated regression tests so that the manual test cases used in this project can be repeated consistently after each change.

4.3.1 Performance Evaluation

To further evaluate the system, we conducted basic performance observations under typical usage conditions. During testing, the system demonstrated low latency in message delivery, with responses generally occurring within milliseconds under local and deployed environments. The WebSocket-based architecture allowed for persistent connections, reducing communication overhead compared to traditional request-response models.

The system also supported multiple concurrent client connections without failure. During multi-client testing, several users were able to send and receive messages simultaneously, and all messages were delivered correctly and in order. No packet loss or duplication was observed during these tests.

While the system was not stress-tested under high-load conditions, the current design using asynchronous event handling and lightweight JSON messaging suggests that it can be extended to support a larger number of users with appropriate server resources.

These observations confirm that the system meets its performance expectations for real-time communication and provides a solid foundation for future scalability improvements.

5 Impact

An ethical and legal consideration for our project is the lack of security in it. Our protocol makes no attempt to prevent people from logging in as any name aside from not allowing multiple logins for the same name; meaning an attacker could spoof being another client as long as they logged in before the real client. For example, with the restaurant demo, an attacker could log in as the kitchen and log all the orders people make for some nefarious purpose.

Another security issue is the lack of authorization. Any client could log in to the server since there's no checks done by the server. This is an issue because it allows anyone to log in and send messages instead of only authorized clients. In some cases, it might be beneficial to allow anyone to log in, but for our demo, it wouldn't have been good if the restaurant only wanted to a few kiosks to handle orders. Without authorization, any one could connect and send messages; potentially even overwhelm the kitchen since it would be easy to set up a script to send orders as fast as possible.

Both of these issues could be solved or at least mitigated without changing the protocol at all. Clients could first be required to authenticate themselves to other clients via asymmetric cryptography or similar before any of their messages get processed. This isn't perfect since it won't prevent a denial of service attack, but it would at least reduce the impact. A complication of this solution would be each client needs to go through the authentication process before communicating. As more clients connect and want to communicate, more authentications need to happen if every client is communicating to every other client.

A better solution would be changing the protocol itself to require authentication and authorization as part of the login process, and maybe allow clients to choose if they allow communicating with unauthenticated clients. This way, each new client only needs to authenticate once with the server and every other client can trust the server to have properly authenticated the client and thus trust the new client. The server could still be susceptible to denial of service attacks, but as the server it could also have more compute power and be much harder to deny.

A real-world example of these concerns can be seen in messaging platforms where lack of proper authentication can lead to impersonation or spam attacks. For instance, an attacker could connect to the system and send misleading messages to multiple users, disrupting normal operations. In a restaurant scenario similar to the demo, this could result in incorrect orders being displayed or processed, affecting service quality. Addressing these risks through authentication and access control mechanisms is essential before deploying such a

system in production environments.

6 Societal Contribution

The overall direct contribution of this project on society will likely be small. If the protocol and code is published publicly, it is unlikely to receive much adoption. This is because there are already many existing protocols designed for messaging and communication which are more mature and are widely adopted. For communication between people, protocols like IRC, Matrix, and Signal are much more well known and widely adopted. As for communication between systems, many software packages come with their own proprietary protocols or use well known protocols like Zigbee, REST, or Matter.

That said, this project can serve as an example to learn from. This project shines in that it is complete and well-designed, while not being so large that it's unintelligible to an outsider. It provides a basis for what a messaging protocol might look like, as well as well-written Java and TypeScript code showing technologies like JSON and WebSockets in use. Code examples like this can be a valuable tool for those who are learning, who are looking for real, practical examples of technologies they're trying to get familiar with. The project as a whole can also serve as a positive example of complete, professional-quality, thoughtful software design. The protocol is very well documented, and serves as a strong example of using well-defined standards to facilitate communication between various programming languages and distributed systems. The final outcome shows the result of combining many technologies such as WebSockets, JSON, Java, TypeScript, HTML, CSS, Make, and Gradle all into a professional-quality codebase. This skill is vital for software engineering, and this can serve as a reference for future software engineers on the complete software design process, beginning to end. Real-time communication systems similar to this project are widely used in critical real-world applications. For example, hospitals rely on real-time messaging systems to coordinate patient care between doctors and nurses, while industrial systems use similar architectures to synchronize machines and monitor operations. These applications demonstrate how reliable and low-latency communication can improve efficiency, safety, and decision-making in society.

In addition to these benefits, this project raises important ethical and security considerations related to real-time communication systems. The current implementation lacks authentication and authorization mechanisms, which makes it vulnerable to misuse. For example, any client can log in using any available username, allowing an attacker to impersonate another user or send misleading messages. Additionally, the absence of access control enables unauthorized users to send spam messages or overwhelm the system, which could disrupt normal operations, such as in the restaurant demo. These issues high-

light risks related to data integrity, user trust, and system reliability.

These concerns align with the ACM Code of Ethics, particularly Principle 1.2 (avoiding harm) and Principle 2.9 (designing secure systems). To address these risks, future improvements should include user authentication, message encryption, and proper validation of user permissions. Implementing these measures would enhance system security, protect users, and ensure that the system operates in a responsible and ethical manner.

7 Engineering Standards

The design and implementation of the WebSocket Relay Chat system were guided by established engineering standards, particularly those defined by the IEEE for software lifecycle, documentation, verification, and testing. These standards informed both the structure of the system and the development process used throughout the project.

The overall development process aligns with IEEE Std 12207 (Software Life Cycle Processes). The project followed a structured workflow including requirements definition, system design, implementation, testing, and deployment. Each of these phases is reflected in the organization of this report, as well as in the separation of the protocol, server, client library, and demo application.

System requirements and functional behavior were defined in a manner consistent with IEEE Std 1233 (Guide for Developing System Requirements Specifications). The protocol design, including packet types, required fields, and expected responses, serves as a formal specification of system behavior. This ensures that both the server and client implementations follow a consistent and well-defined contract.

The system architecture and component relationships are documented in accordance with IEEE Std 1016 (Software Design Description). The use of layered diagrams and system models illustrates how the application layer, protocol layer, and WebSocket communication layer interact. These design representations support clarity, maintainability, and extensibility of the system.

Verification and validation practices were implemented following IEEE Std 1012 (System and Software Verification and Validation). Testing was performed at multiple levels, including protocol validation, client-library behavior, and full system integration using the web demo. These tests ensured that the system meets its functional requirements and behaves correctly under both normal and error conditions.

Testing procedures and results are documented in a structured format aligned with IEEE Std 829 (Software Test Documentation). Test cases included authentication scenarios, message routing, error handling, and multi-client synchronization. The results demonstrate that the system performs as expected and meets the intended use cases defined during the design phase.

Software quality considerations are consistent with IEEE Std 730 (Software Quality Assurance Plans). Code was developed with a focus on readability, modularity, and maintainability. The use of Java records, structured packet handling, and TypeScript type safety contributed to reducing errors and en-

sureing consistent behavior across components.

Coding standards were applied to ensure clarity, consistency, and usability. The system emphasizes modular design, clear naming conventions, structured error handling, and user-focused abstractions. The client library, in particular, provides a simplified interface that allows developers to interact with the system without needing to manage low-level protocol details, improving usability and reducing complexity.

In addition to technical standards, the system was developed with awareness of engineering constraints and broader impacts. The use of open-source tools and lightweight technologies supports economic feasibility. The system's real-time communication design using WebSockets improves efficiency and performance. Time constraints influenced the prioritization of core functionality, ensuring a complete and demonstrable system within the capstone timeline.

Security and ethical considerations were also recognized. While the current implementation does not include authentication or authorization mechanisms, these limitations were identified and analyzed. Addressing these concerns in future iterations would improve alignment with standards related to user safety, data integrity, and secure communication.

Overall, the project demonstrates the application of established engineering standards to guide the design, implementation, testing, and documentation of a real-time distributed system.

8 Conclusion

This project successfully demonstrated the design and implementation of a lightweight, real-time messaging system built on the WebSocket protocol. By developing a custom JSON-based communication protocol, a Java WebSocket server, a TypeScript client library, and a browser-based demo application, we were able to create a complete system that supports real-time interaction between multiple clients. The final implementation met the original goal of providing a simple and portable messaging service while also showcasing a practical use case through the restaurant ordering demo.

One of the most important outcomes of this project was gaining a deeper understanding of how real-time communication systems are structured. The separation between the protocol, server, and client layers allowed each part of the system to be developed, tested, and reasoned about independently. The use of JSON for packet formatting and WebSockets for communication simplified debugging and made the system easier to extend. Additionally, the client library abstracted much of the complexity of asynchronous communication, allowing applications to interact with the system through simple method calls rather than raw packet handling.

The testing process confirmed that the system behaves as expected across a variety of scenarios, including multiple concurrent clients, error handling, and real-time updates in the demo application. The restaurant demo specifically demonstrated that the system can support synchronized state changes across multiple clients, which is a key requirement for many modern applications beyond messaging, such as collaborative tools and live data systems.

While the system is intentionally lightweight, it provides a strong foundation for future development. The current implementation focuses on core messaging functionality, but the architecture allows for additional features such as authentication, persistence, and scalability improvements to be added without requiring a complete redesign.

Overall, this project achieved its intended objectives and provided practical experience in protocol design, client-server architecture, and real-time system development. It also highlights how relatively simple technologies, when combined effectively, can be used to build flexible and functional distributed systems.

9 Recommendations for Future Works

While the current implementation of the WebSocket Relay Chat system meets the core requirements of real-time messaging and demonstrates a functional client-server architecture, there are several areas where the system could be extended and improved in future work.

One of the most important improvements would be the addition of authentication and authorization mechanisms. The current system allows any client to log in with any username, which introduces potential security and identity spoofing issues. Implementing a secure login system, such as token-based authentication or credential verification, would significantly improve system reliability and better protect user identity.

Another major enhancement would be the introduction of persistent data storage through a database system. Currently, all user sessions, messages, and order data exist only in memory, meaning that information is lost when the server restarts or when clients disconnect. Integrating a database, such as a relational database (e.g., MySQL or PostgreSQL) or a NoSQL solution, would allow the system to store previous messages, maintain order history, and persist user state. This would make the system more practical for real-world applications, such as tracking past orders in the restaurant demo or enabling message history in a chat application.

Scalability is also an important area for future development. The current implementation is designed for a small number of concurrent clients. Future versions of the system could explore load balancing, distributed server architectures, or message queuing systems to support a larger number of users and improve performance under higher traffic conditions.

From a usability perspective, the client and demo application could be expanded to include a more advanced user interface. Features such as improved error feedback, better visualization of active users, and more intuitive interaction flows would enhance the overall user experience. Additionally, expanding the demo beyond a restaurant ordering system to other real-time applications, such as collaborative tools or chat platforms, would further demonstrate the flexibility of the protocol.

Additional improvements could also be made to error handling and protocol robustness. While the current system handles common error cases, more detailed validation, logging, and recovery mechanisms would improve maintainability and debugging. Formalizing the protocol further, potentially with versioning or schema validation, would make it easier for external developers to implement compatible clients.

Finally, automated testing and continuous integration could be expanded

to include regression tests and stress testing. This would ensure that future changes do not introduce unintended behavior and would improve confidence in the system as it evolves.

Overall, the current system provides a strong foundation for real-time communication, and these recommended improvements would extend its functionality, security, and scalability, bringing it closer to a production-ready system.

A Source Code

A.1 Java Server

Listing 1: MessageServer.java

```
1 package dev.dwns.sam;
2
3 import com.fasterxml.jackson.databind.JsonNode;
4 import com.fasterxml.jackson.databind.ObjectMapper;
5 import com.google.common.collect.BiMap;
6 import com.google.common.collect.HashBiMap;
7
8 import dev.dwns.sam.packet.ErrorPackets;
9 import org.java_websocket.WebSocket;
10 import org.java_websocket.handshake.ClientHandshake;
11 import org.java_websocket.server.WebSocketServer;
12
13 import java.io.IOException;
14 import java.net.InetSocketAddress;
15
16 public class MessageServer extends WebSocketServer {
17
18     private BiMap<String, WebSocket> loginTable;
19
20     private void handleMessage(WebSocket socket,
21                               String request) {
22         ObjectMapper mapper = new ObjectMapper();
23         try {
24             JsonNode root = mapper.readTree(request);
25             if (root.hasNonNull("packetType") && root
26                 .get("packetType").isInt() && root
27                 .hasNonNull("id") && root.get("id").
28                 isInt()) { // ensure packet type and
29                             id are present
30                 int id = root.get("id").asInt();
31                 switch (root.get("packetType").asInt
32                     ()) {
33                     case 1: // login request
```

```

28         PacketHandler.loginRequest(
                request, loginTable,
                socket, id);
29         break;
30     case 2: // check active users
31         PacketHandler.
                checkActiveUsers(request,
                loginTable, socket, id);
32         break;
33     case 3: // send message
34         PacketHandler.
                sendMessageRequest(request
                , loginTable, socket, id);
35         break;
36     case 5: // broadcast request
37         PacketHandler.
                broadcastRequest(request,
                loginTable, socket, id);
38         break;
39     case 6: // logout
40         PacketHandler.logoutRequest(
                request, loginTable,
                socket, id);
41         break;
42     default:
43         socket.send(ErrorPackets.
                MalformedPacket(id));
44     }
45     } else {
46         socket.send(ErrorPackets.
                ERR_MALFORMED_PACKET_NOID); //
                return error malformed packet
47     }
48 } catch(IOException e) {
49     socket.send(ErrorPackets.
                ERR_MALFORMED_PACKET_NOID); // return
                error malformed packet
50 }
51
52 }

```

```

53
54     public MessageServer(int port) {
55         super(new InetSocketAddress(port));
56         loginTable = HashBiMap.create();
57     }
58
59     @Override
60     public void onOpen(WebSocket webSocket,
61         ClientHandshake clientHandshake) {
62         System.out.println("Connected: " + webSocket.
63             getRemoteSocketAddress());
64     }
65
66     @Override
67     public void onClose(WebSocket webSocket, int i,
68         String s, boolean b) {
69         System.out.println("Connection closed: " +
70             webSocket.getRemoteSocketAddress().
71             getHostString());
72         String username = loginTable.inverse().get(
73             webSocket);
74         if(username != null) {
75             System.out.println("Logging out user: " +
76                 username);
77             loginTable.remove(username);
78         }
79     }
80
81     @Override
82     public void onMessage(WebSocket webSocket, String
83         s) {
84         System.out.println("Received: " + s);
85         handleMessage(webSocket, s);
86     }
87
88     @Override
89     public void onError(WebSocket webSocket,
90         Exception e) {
91     }
92 }

```

```

84
85     @Override
86     public void onStart() {
87
88     }
89
90
91     public static void main(String[] args) throws
        InterruptedException {
92         MessageServer server;
93         if(args.length == 1) {
94             server = new MessageServer(Integer.
                parseInt(args[0]));
95         } else {
96             server = new MessageServer(8080);
97         }
98         server.start();
99         while(true) {
100             Thread.sleep(1000);
101         }
102     }
103 }

```

Listing 2: PacketHandler.java

```

1 package dev.dwns.sam;
2
3 import com.fasterxml.jackson.core.
    JsonProcessingException;
4 import com.fasterxml.jackson.databind.
    DeserializationFeature;
5 import com.fasterxml.jackson.databind.ObjectMapper;
6 import com.google.common.collect.BiMap;
7
8 import dev.dwns.sam.packet.*;
9
10 import org.java_websocket.WebSocket;
11
12 import java.util.HashSet;
13 import java.util.Set;

```

```

14
15 import static dev.dwns.sam.packet.ErrorPackets.
    NotLoggedIn;
16
17 // This file uses Jackson's ObjectMapper along with
    the records defined in ErrorPackets and Packets to
    automatically convert JSON strings to objects and
    vice-versa
18
19 public class PacketHandler {
20     private static ObjectMapper mapper = new
        ObjectMapper().configure(
            DeserializationFeature.
            FAIL_ON_MISSING_CREATOR_PROPERTIES, true); //
            throw an error if any fields are missing
            during deserialization
21
22     public static void loginRequest(String message,
        BiMap<String, WebSocket> loginTable, WebSocket
        socket, int id) {
23         try {
24             Packets.LoginRequest request = mapper.
                readValue(message, Packets.
                LoginRequest.class);
25             String username = request.
                requestedUsername();
26             if (!loginTable.containsKey(username)) {
27                 try {
28                     loginTable.put(username, socket);
29                     System.out.println("User logged
                        in: " + username);
30                     socket.send(mapper.
                        writeValueAsString(new Packets
                            .LoginResponse(1, request.id()
                                )); // return success
31                 } catch (IllegalArgumentException e)
                    {
32                     socket.send(ErrorPackets.
                        AlreadyLoggedIn(request.id()))
                        ; // error if this socket is

```

```

                                already logged in with a
                                different username
33         }
34     } else {
35         socket.send(ErrorPackets.
                        UsernameTaken(request.id()));
36     }
37 } catch (JsonProcessingException e) {
38     socket.send(ErrorPackets.MalformedPacket(
        id));
39 }
40 }
41
42 public static void checkActiveUsers(String
    message, BiMap<String, WebSocket> loginTable,
    WebSocket socket, int id) {
43     try {
44         Packets.ActiveUsersRequest request =
            mapper.readValue(message, Packets.
                ActiveUsersRequest.class);
45         socket.send(mapper.writeValueAsString(new
            Packets.ActiveUsersResponse(2,
                request.id(), loginTable.keySet())));
46     } catch (JsonProcessingException e) {
47         socket.send(ErrorPackets.MalformedPacket(
            id));
48     }
49 }
50
51 public static void sendMessageRequest(String
    message, BiMap<String, WebSocket> loginTable,
    WebSocket requestClient, int id) {
52     if (!loginTable.containsValue(requestClient))
    {
53         requestClient.send(ErrorPackets.
            NotLoggedIn(id)); // error if user is
                                not logged in and tries to send a
                                message
54         return;
55     }

```

```

56     try {
57         Packets.SendMessageRequest request =
            mapper.readValue(message, Packets.
                SendMessageRequest.class);
58         if (request.recipientList().isEmpty()) {
59             requestClient.send(ErrorPackets.
                MalformedPacket(id)); // error if
                recipient list is empty
60             return;
61         }
62         Set<String> missingUsers = new HashSet
            <>();
63         String messagePacket = mapper.
            writeValueAsString(new Packets.
                MessagePacket(4, loginTable.inverse().
                    get(requestClient), request.message())
            );
64         request.recipientList().forEach(recipient
            -> {
65             if (loginTable.containsKey(recipient)
66                 ) {
67                 loginTable.get(recipient).send(
68                     messagePacket); // send
69                     message to each user in the
70                     list
71             } else {
72                 missingUsers.add(recipient); //
73                     add users which are not found
74                     to list
75             }
76         });
77         if (missingUsers.isEmpty()) {
78             requestClient.send(mapper.
79                 writeValueAsString(new Packets.
80                     SendMessageResponse(3, request.id
81                         ())))); // all recipients are
82                     online, success
83         } else { // otherwise send an error with
84             the users which were not found
85             ErrorPackets.SendMessageError.

```

```

        SendMessageErrorData errorData =
            new ErrorPackets.SendMessageError.
                SendMessageErrorData(missingUsers)
            ;
75        ErrorPackets.SendMessageError
            errorPacket = new ErrorPackets.
                SendMessageError(20, request.id(),
                    3, "One or more users not found",
                    errorData);
76        requestClient.send(mapper.
            writeValueAsString(errorPacket));
77    }
78    } catch (JsonProcessingException e) {
79        requestClient.send(ErrorPackets.
            MalformedPacket(id));
80    }
81 }
82
83 public static void broadcastRequest(String
    message, BiMap<String, WebSocket> loginTable,
    WebSocket requestClient, int id) {
84     if (!loginTable.containsValue(requestClient))
85     {
86         requestClient.send(NotLoggedIn(id)); //
            error if user is not logged in
87         return;
88     }
89     try {
90         Packets.BroadcastRequest request = mapper
            .readValue(message, Packets.
                BroadcastRequest.class);
91         String messagePacket = mapper.
            writeValueAsString(new Packets.
                MessagePacket(4, loginTable.inverse().
                    get(requestClient), request.message())
            );
92         loginTable.forEach((name, client) -> { //
            send the message to every client
            currently logged in
            client.send(messagePacket);

```

```

93         });
94         requestClient.send(mapper.
            writeValueAsString(new Packets.
                BroadcastResponse(5, request.id())));
95     } catch (JsonProcessingException e) {
96         requestClient.send(ErrorPackets.
            MalformedPacket(id));
97     }
98 }
99
100 public static void logoutRequest(String message,
    BiMap<String, WebSocket> loginTable, WebSocket
    socket, int id) {
101     try {
102         Packets.LogoutRequest request = mapper.
            readValue(message, Packets.
                LogoutRequest.class);
103         if(loginTable.containsValue(socket)) { //
            if users is logged in, log them out
104             loginTable.inverse().remove(socket);
105             socket.send(mapper.writeValueAsString
                (new Packets.LogoutResponse(6,
                    request.id())));
106         } else {
107             socket.send(mapper.writeValueAsString
                (NotLoggedIn(request.id()))); //
                error is they are not logged in
                and try to log out
108         }
109     } catch (JsonProcessingException e) {
110         socket.send(ErrorPackets.MalformedPacket(
            id));
111     }
112 }
113 }

```

A.2 Typescript Client

Listing 3: Typescript Client

```
1 class LoginRequest {
2   packetType: number = 1;
3   id: number;
4   requestedUsername: string;
5
6   constructor(id: number, requestedUsername: string)
7     {
8     this.id = id;
9     this.requestedUsername = requestedUsername;
10  }
11
12 class ActiveUsersCheck {
13   packetType: number = 2;
14   id: number;
15
16   constructor(id: number) {
17     this.id = id;
18   }
19 }
20
21 class SendMessageRequest {
22   packetType: number = 3;
23   id: number;
24   recipientList: string[];
25   message: string;
26
27   constructor(id: number, recipientList: string[],
28     message: string) {
29     this.id = id;
30     this.recipientList = recipientList;
31     this.message = message;
32   }
33
34 class SendMessagePromiseData {
35   errorCode: number;
36   inactiveUsers: string[];
```

```

37
38     constructor(errorCode: number, inactiveUsers:
        string[]) {
39         this.errorCode = errorCode;
40         this.inactiveUsers = inactiveUsers;
41     }
42 }
43
44 class BroadcastRequest {
45     packetType: number = 5;
46     id: number;
47     message: string;
48
49     constructor(id: number, message: string) {
50         this.id = id;
51         this.message = message;
52     }
53 }
54
55 class LogoutRequest {
56     packetType: number = 6;
57     id: number;
58
59     constructor(id: number) {
60         this.id = id;
61     }
62 }
63
64 class ResolveRejectData {
65     resolve: Function;
66     reject: Function;
67     data: any;
68
69     constructor(resolve: Function, reject: Function,
        data: any) {
70         this.resolve = resolve;
71         this.reject = reject;
72         this.data = data;
73     }
74 }

```

```

75
76 export default class WSClient {
77
78     static SUCCESS: number = 0; // Named constants for
        error codes
79     static ERR_MALFORMED_PACKET: number = 1;
80     static ERR_USERNAME_TAKEN: number = 2;
81     static ERR_INACTIVE_USERS: number = 3;
82     static ERR_REQUESTER_NOT_LOGGED_IN: number = 4;
83     static ERR_REQUESTER_ALREADY_LOGGED_IN: number = 5;
84     static ERR_SERVER_SIDE: number = 6;
85     static ERR_BAD_RESPONSE: number = 7;
86
87     socket: WebSocket;
88     id: number = 0;
89     currentUsername: string = "";
90     waitingTable: Array<ResolveRejectData> = [];
91     messageListener: ((sender: string, message: string)
        => void) | null = null;
92
93     constructor(url: string) {
94         this.socket = new WebSocket(url); // Create
            socket and register message listener
95         this.socket.addEventListener("message", (event)
            => {
96             this.handleIncomingPacket(event.data);
97         });
98     }
99
100     addMessageListener(callback: (sender: string,
        message: string) => void) {
101         this.messageListener = callback;
102     }
103
104     private handleIncomingMessage(object: any) {
105         if (this.messageListener) {
106             this.messageListener(object.sender, object.
                message);
107         }
108     }

```

```

109
110     getOpenPromise(): Promise<void> {
111         const { promise, resolve, reject } = Promise.
            withResolvers<void>(); // Create promise
112         if(this.socket.readyState == this.socket.OPEN) {
            // Resolve immediately if socket is already
            open
113             resolve();
114             return promise;
115         }
116         this.socket.addEventListener("open", () => { //
            Wait for socket to connect to open
117             resolve();
118         });
119         return promise;
120     }
121
122     login(username: string): Promise<number> {
123         this.id++;
124         this.socket.send(JSON.stringify(new LoginRequest(
            this.id, username))); // Send login request
125         const { promise, resolve, reject } = Promise.
            withResolvers<number>(); // Create promise
126         this.waitingTable[this.id] = new
            ResolveRejectData(resolve, reject, username);
            // Add promise resolvers to waiting table
127         return promise;
128     }
129
130     private loginSuccess(id: number) {
131         if(this.waitingTable[id]) { // If request id is
            in the waiting table, resolve success
132             this.currentUsername = this.waitingTable[id].
                data;
133             this.waitingTable[id].resolve(WSCClient.SUCCESS)
                ;
134             delete this.waitingTable[id]; // Free
                ResolveRejectData from array
135         }
136     }

```

```

137
138     checkActiveUsers(): Promise<string[]> {
139         this.id++;
140         const {promise, resolve, reject} = Promise.
            withResolvers<string[]>();
141         this.waitingTable[this.id] = new
            ResolveRejectData(resolve, reject, null);
142         this.socket.send(JSON.stringify(new
            ActiveUsersCheck(this.id)));
143         return promise;
144     }
145
146     private activeUsersSucess(object: any) {
147         const incomingId: number = object.id;
148         if (this.waitingTable[incomingId]) {
149             this.waitingTable[incomingId].resolve(object.
                activeUsers);
150             delete this.waitingTable[incomingId];
151         }
152     }
153
154     sendMessage(recipientList: string[], message:
        string): Promise<SendMessagePromiseData> {
155         this.id++;
156         this.socket.send(JSON.stringify(new
            SendMessageRequest(this.id, recipientList,
            message))); // Send message request
157         const {promise, resolve, reject} = Promise.
            withResolvers<SendMessagePromiseData>();
158         this.waitingTable[this.id] = new
            ResolveRejectData(resolve, reject, null);
159         return promise;
160     }
161
162     private sendMessageSuccess(id: number) {
163         if(this.waitingTable[id]) {
164             this.waitingTable[id].resolve(new
                SendMessagePromiseData(WSCClient.SUCCESS, []))
                ); // Success with no users not found
165             delete this.waitingTable[id];

```

```

166     }
167 }
168
169 broadcast(message: string): Promise<number> {
170     this.id++;
171     this.socket.send(JSON.stringify(new
        BroadcastRequest(this.id, message)));
172
173     const { promise, resolve, reject } = Promise.
        withResolvers<number>();
174     this.waitingTable[this.id] = new
        ResolveRejectData(resolve, reject, null);
175
176     return promise;
177 }
178
179 private broadcastSuccess(id: number) {
180     if (this.waitingTable[id]) {
181         this.waitingTable[id].resolve(WSCClient.SUCCESS)
            ;
182         delete this.waitingTable[id];
183     }
184 }
185
186 logout(): Promise<number> {
187     this.id++;
188     this.socket.send(JSON.stringify(new LogoutRequest
        (this.id)));
189
190     const {promise, resolve, reject} = Promise.
        withResolvers<number>();
191     this.waitingTable[this.id] = new
        ResolveRejectData(resolve, reject, null);
192
193     return promise;
194 }
195
196 private logoutSuccess(id: number) {
197     if(this.waitingTable[id]) {

```

```

198         this.waitingTable[id].resolve(WSCClient.SUCCESS)
199         ;
200     delete this.waitingTable[id];
201 }
202
203 private error(object: any) {
204     if(this.waitingTable[object.id]) {
205         // will need a special check for inactive users
206         error after send message, all others are
207         simple numerical errors
208         if(typeof object.errorCode === "number") { //
209             If error code exists, reject with error code
210             if(object.errorCode == 3) { // Special error
211                 type for missing users
212                 this.waitingTable[object.id].reject(new
213                     SendMessagePromiseData(object.errorCode,
214                     object.errorData.inactiveUsers));
215             } else {
216                 this.waitingTable[object.id].reject(object.
217                     errorCode);
218             }
219         }
220         delete this.waitingTable[object.id];
221     }
222 }
223
224 handleIncomingPacket(packet: string) {
225     const object = JSON.parse(packet);
226
227     if(object.packetType === 4 && object.sender &&
228         object.message) { // special case for incoming
229         message since it doesn't contain an id
230         this.handleIncomingMessage(object);
231     } else if(object.packetType && object.id) {
232         switch(object.packetType) {
233             case 1: // Login success
234                 this.loginSuccess(object.id);
235             break;
236             case 2: // check active user success

```

```
228         this.activeUsersSucess(object);
229         break;
230     case 3: // Send message success
231         this.sendMessageSuccess(object.id);
232         break;
233     case 5: // Broadcast success
234         this.broadcastSuccess(object.id);
235         break;
236     case 6: // Logout successss
237         this.logoutSuccess(object.id);
238         break;
239     case 20: // Error
240         this.error(object);
241         break;
242     }
243 }
244 console.log("Packet Received: " + packet);
245 }
246 }
```
