COP5615 - DOSP Project 4 - Part 2

Team Members

Akash Pinnaka - UFID 1009-4703
Thilak Reddy Kanala - UFID 8800-3203

Video Demonstration
(https://www.youtube.com/watch?v=i2sDgZJtu7k)

Architecture

To integrate WebSockets into our existing Twitter engine, we utilise the ‘Simple Bridge’ web framework, which is built on top of the popular ‘Nitrogen’ web framework for Erlang. Simple Bridge allows easy implementation of web sockets with many examples and boilerplate code for rapid prototyping of websocket-based projects in erlang. We were able to easily integrate the Simple Bridge template code into our existing Twitter engine to add WebSocket functionality.

The architecture is split into three modules:

1. A bridge module that collects request information and allows us to construct a response
2. A supervisor that configures the base server with specific configuration
3. An anchor module that facilitates connection and communication over the underlying network, which then allows the handler module to be interfaced by the user code

The following are the standard erlang function calls triggered during Websocket communication:

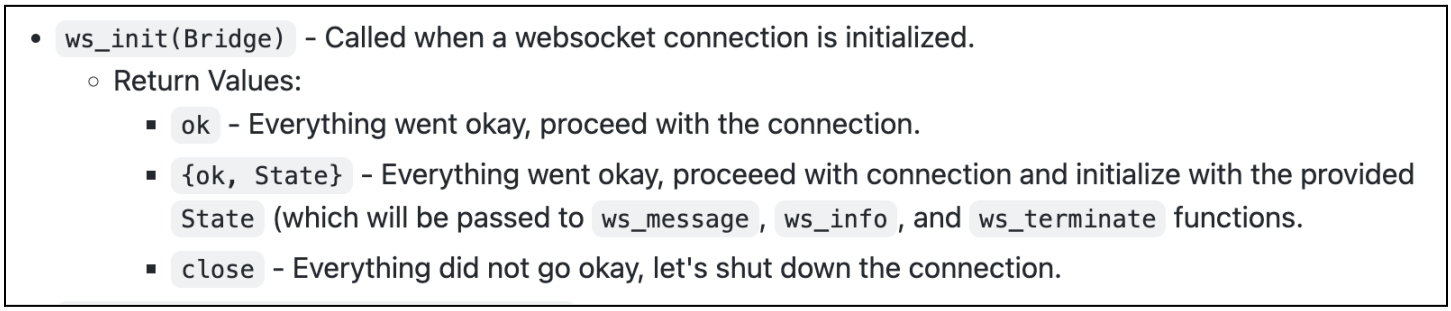

Fig 1. Websocket Initialization

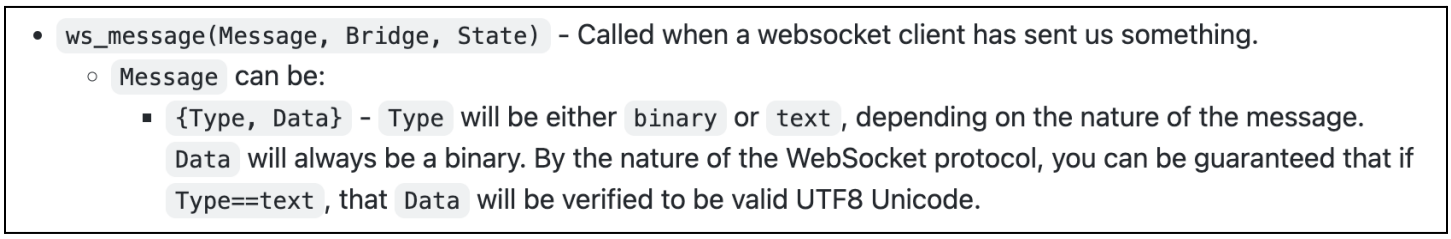

Fig 2. Websocket Communication

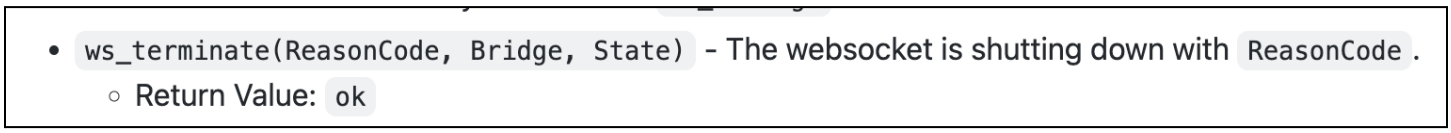

Fig 3. Websocket Termination

Demonstrating Functionality

1. Websocket demonstration for registering users

    a. For the purpose of demonstration, we register five randomly generated users
    b. The WebSocket callback initiates the relevant erlang process

```
ws_message({text, <<"{register_n_users_testing}">>}, _State, _Bridge) ->
  tester:register_all_users(5, 5, []),
  timer:sleep(500),
  {ok, _Data} = file:read_file("./scratch.txt"),
  Data = io_lib:format("~s",[_Data]),
  io:format("~p~n", [Data]),
  {reply, {text, Data}};
```

Fig 4. Code to demonstrate registering users

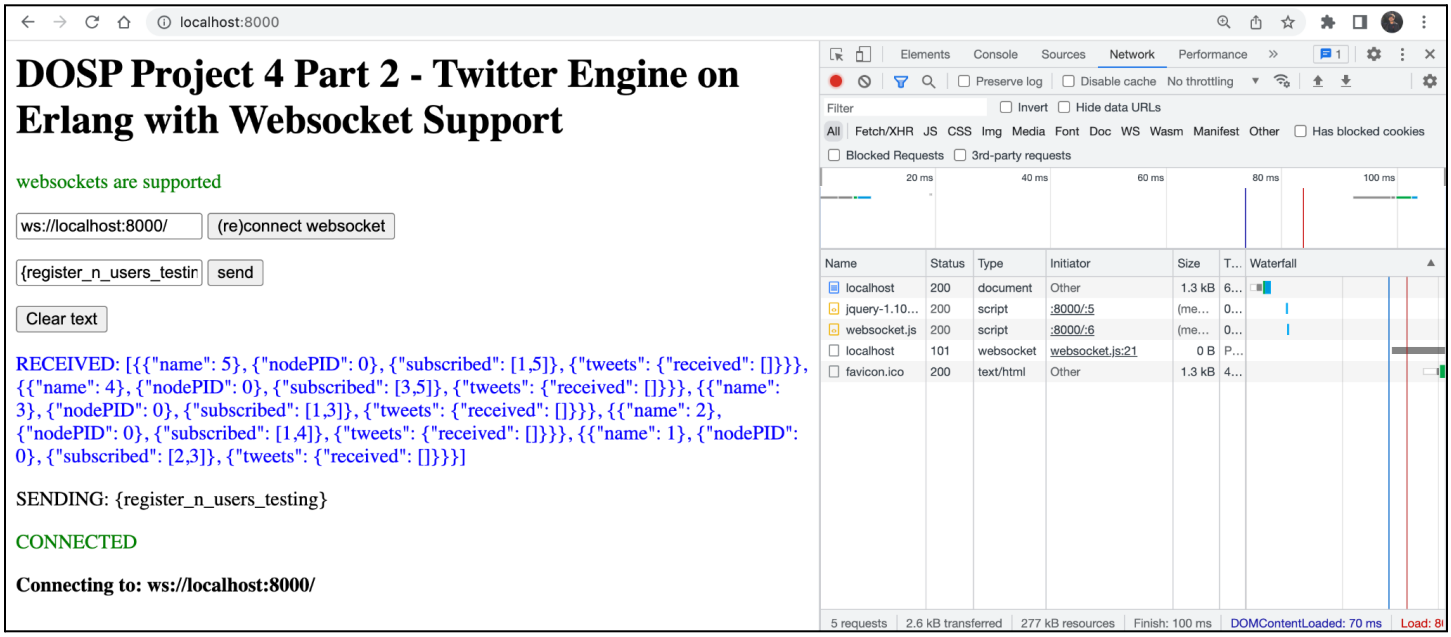

Fig 5. Result of demonstrating registering users

We observe the expected results, which is the JSON response of all the User's data stored on the Twitter engine gen_server.

We also observe in the chrome debugging engine that WebSocket messages are being utilized.

2. Websocket demonstration of sending a Tweet
    a. For the purpose of testing, we use the same five randomly generated users above
    b. We generate a random tweet and post that through User 1

```
25
26    send_tweet_testing(NNodes, _State, _Bridge) ->
27        Tweet1 = get_random_tweet(1, NNodes),
28        engine:send_tweet_from_Nth_Node(1, Tweet1),
29        engine:update_all_nodes(NNodes),
30
31        engine:show_loopdata(_State, _Bridge).
```

Fig 6. Code triggered by Websocket request for posting a tweet

```
46  ws_message({text, <<"{send_tweet_testing}">>}, _State, _Bridge) ->
47    tester:send_tweet_testing(5, _State, _Bridge),
48    timer:sleep(500),
49    {ok, _Data} = file:read_file("./scratch.txt"),
50    Data = io_lib:format("~s",[_Data]),
51    io:format("~p~n", [Data]),
52    {reply, {text, Data}};
```

Fig 7. Code where Websocket request is processed for posting a tweet

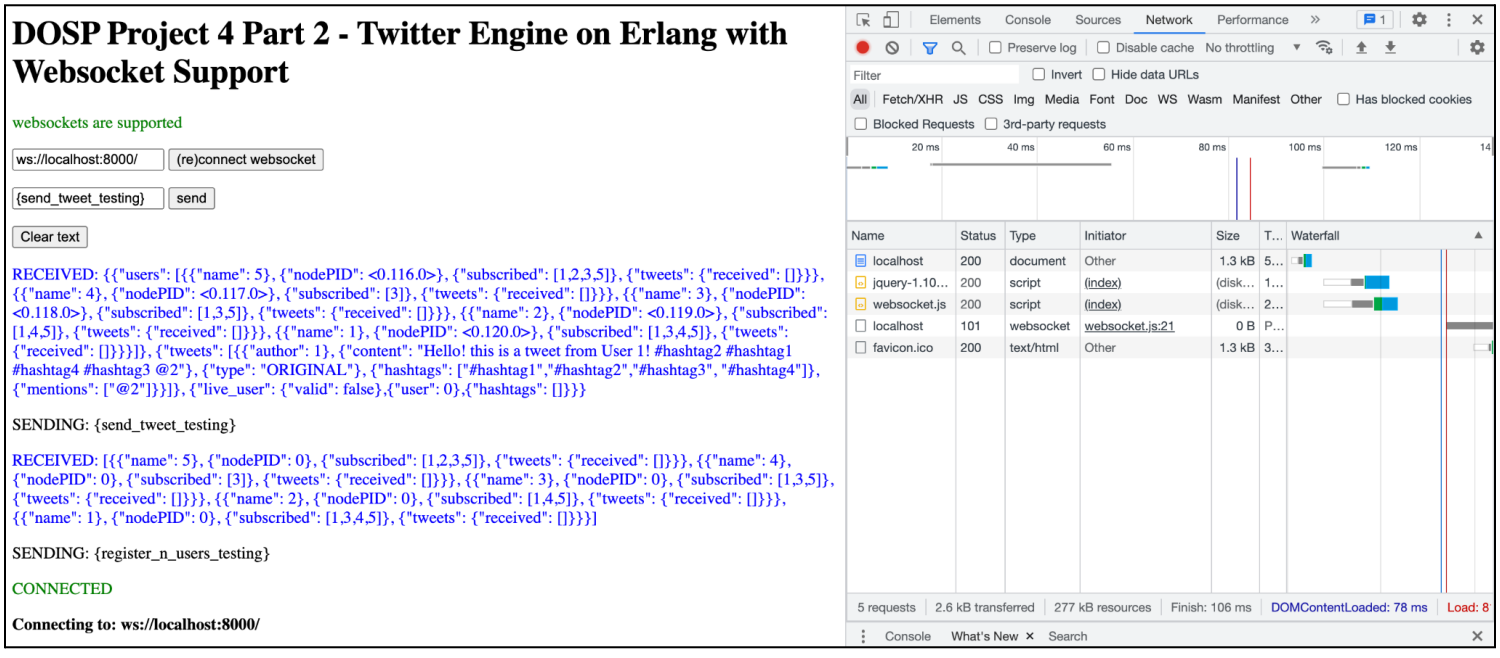

Fig 8. Result of demonstrating posting tweet

We observe the expected results, the JSON response is sent from the erlang Websocket.

Specifically, we notice the following JSON-parsed Tweet Data from User 1:

{{"author": 1}, {"content": "Hello! this is a tweet from User 1! #hashtag2 #hashtag1 #hashtag4 #hashtag3 @2"}, {"type": "ORIGINAL"}, {"hashtags": ["#hashtag1","#hashtag2","#hashtag3", "#hashtag4"]}, {"mentions": ["@2"]}}]}

3. Websocket demonstration of Retweets
    a. For the purpose of testing, we send a retweet from User 3
    b. We again initialise 5 random users and send a Tweet from User 1 mentioning User 3
    c. This Tweet from User 1, when received by User 3, is retweeted
    d. For the purpose of testing, we combine client requests for 'querying_retweets' and 'sending_retweets' into one message

```
53
54  ws_message({text, <<"{send_retweet_testing}">>}, _State, _Bridge) ->
55      tester:send_retweet_testing(5, _State, _Bridge),
56      timer:sleep(500),
57      {ok, _Data} = file:read_file("./scratch.txt"),
58      Data = io_lib:format("~s",[_Data]),
59      io:format("~p~n", [Data]),
60      {reply, {text, Data}};
61
```

Fig 9. Code where WebSocket request is handled

```
32
33  send_retweet_testing(NNodes, _State, _Bridge) ->
34      node:query_mention_tweets(3),
35      engine:send_random_retweet_from_Nth_Node(3),
36      engine:show_loopdata(_State, _Bridge).
37
```

Fig 10. Code triggered for retweet testing

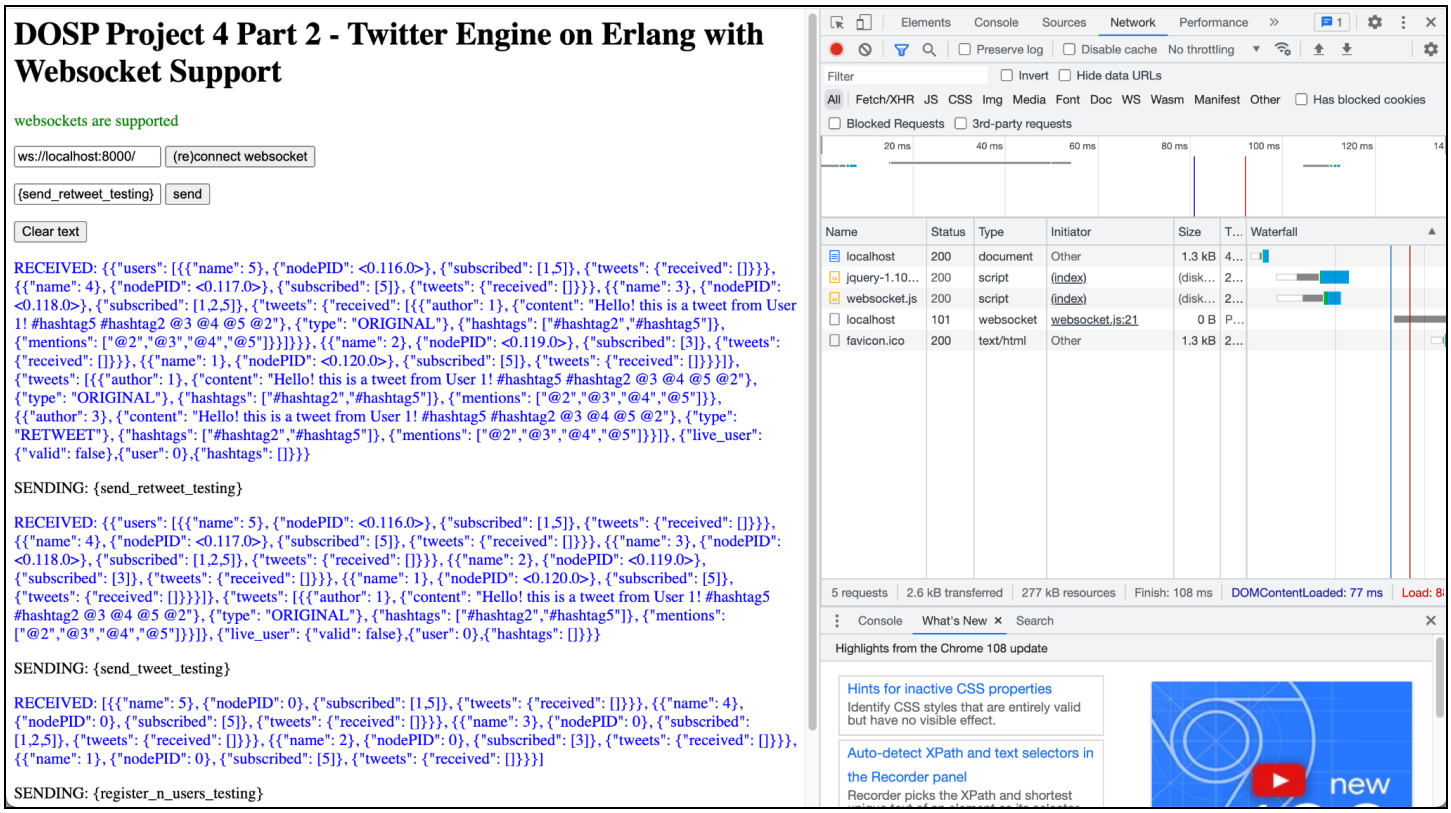

Fig 11. Result of demonstrating retweet testing

The observed result is as expected; there is a tweet posted by User 1 which mentions User 3. This tweet is then Retweeted by User 3, which can be observed in the final message from Twitter engine gen_server, where one of the tweets is:

`{"author": 3}, {"content": "Hello! this is a tweet from User 1! #hashtag5 #hashtag2 @3 @4 @5 @2"}, {"type": "RETWEET"}, {"hashtags": ["#hashtag2","#hashtag5"]}, {"mentions": ["@2","@3","@4","@5"]}}`

Which is a retweet from User 3 (as we can see in the “author”: 3 field).

4. Websocket demonstration of live feed, along with querying for subscribed, specific hashtags and mentioned tweets
    a. For the purpose of testing, we combine several functions of the Twitter engine to trigger on receiving the WebSocket message `{demo_live_user_testing}` from the client
    b. The functions triggered are
        i. registering five random users
        ii. registering User 2 as the live user with interest in #hashtag5
        iii. posting three random tweets, from User 1, User 3 and User 4

```
43  demo_live_user_testing(_NNodes, _State, _Bridge) ->
44      engine:start(),
45      timer:sleep(500),
46      _AllUsers = register_all_users(5, 5, []),
47      LiveUser = {'\"live_user\":', {'\"valid\":', true}, {'\"user\":',2}, {'\"hashtags\":', ["#hashtag5"]}},
48      engine:register_live_user(LiveUser),
49
50      Tweet1 = get_random_tweet(1, 5),
51      engine:send_tweet_from_Nth_Node(1, Tweet1),
52
53      Tweet2 = get_random_tweet(3, 5),
54      engine:send_tweet_from_Nth_Node(3, Tweet2),
55
56      Tweet3 = get_random_tweet(4, 5),
57      engine:send_tweet_from_Nth_Node(4, Tweet3),
```

Fig 12. Code triggered for live user testing

```
63  ws_message({text, <<"{demo_live_user_testing}">>}, _State, _Bridge) ->
64      tester:demo_live_user_testing(5, _State, _Bridge),
65      timer:sleep(500),
66      {ok, _Data} = file:read_file("./scratch.txt"),
67      Data = io_lib:format("~s",[_Data]),
68      io:format("~p~n", [Data]),
69      {reply, {text, Data}};
```

Fig 13. Code where WebSocket request is processed

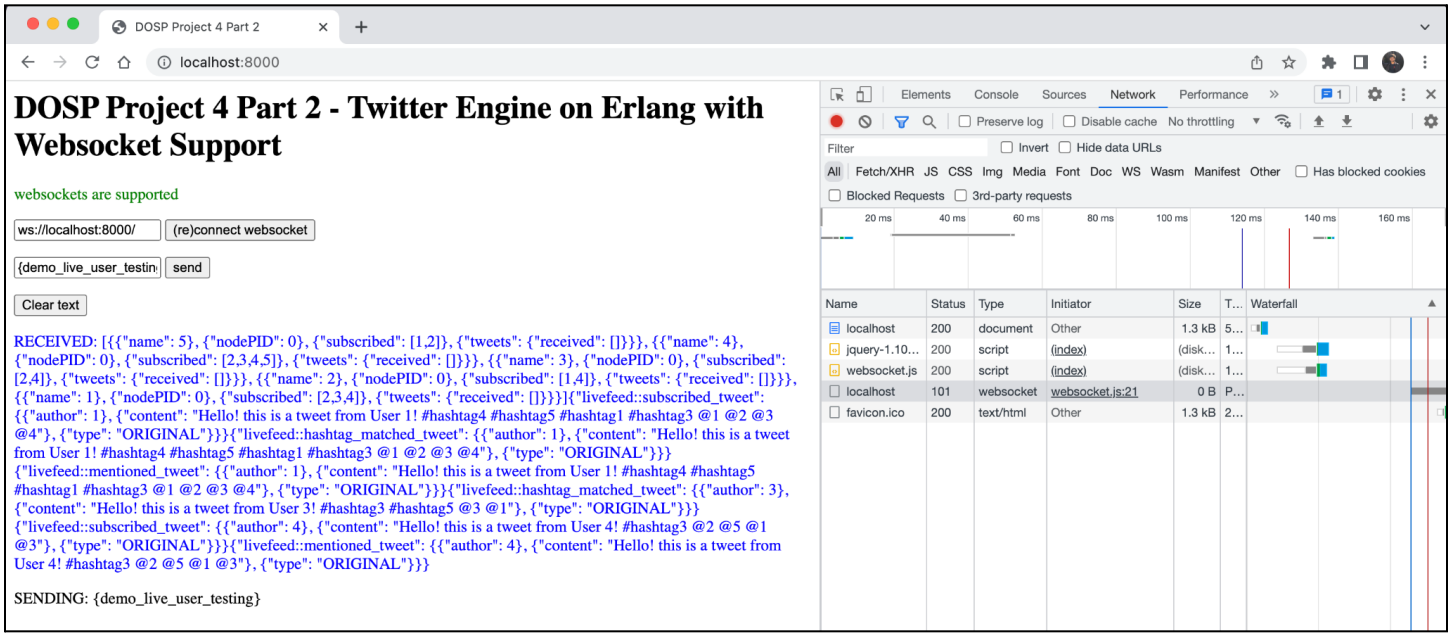

Fig 14. Result of demonstrating live user testing

The observed result is as expected, we notice that since User 2 is registered as a live user, on their live feed, there are tweets from User 1 to which User 2 is subscribed, and also mentions #hashtag5 in which User 2 is interested, and tweet from User 4 who mentions User 2 in their tweet.

5. Demonstrating Zipf Distribution

a. For the purpose of testing, similar to the previous demonstration, we combine several Twitter engine functions into one trigger, {demo_zipf_distribution_testing}

b. We register ten users, with a total of 1000 Tweets in this session, and simulate a Zipf Distribution with these parameters along with a Zipf Constant of 0.1

c. The method for calculating the Zipf Distribution is mentioned in our report for Part 1

d. $$\text{Tweets Posted} = \frac{(Total\ Tweets\ in\ Session) * (Zipf\ Constant)}{Rank}$$

```
71  ws_message({text, <<"{demo_zipf_distribution_testing}">>}, _State, _Bridge) ->
72    tester:simulate_zipf_distribution(10),
73    timer:sleep(500),
74    {ok, _Data} = file:read_file("./scratch.txt"),
75    Data = io_lib:format("~s",[_Data]),
76    io:format("~p~n", [Data]),
77    {reply, {text, Data}};
78
```

Fig 15. Code where WebSocket request is processed

```
79  simulate_zipf_distribution(NNodes) ->
80      engine:start(),
81      timer:sleep(500),
82
83      AllUsers = register_all_users(NNodes, NNodes, []),
84      timer:sleep(500),
85
86      io:format("~p~n~n", [{"Simulating Zipf Distribution with: ", {'Total Users: ', NNodes}, {'Total Tweets: ', ?
        NTOTALTWEETS}, {'Zipf Constant: ', ?ZIPHCONSTANT}}]),
87
88      UserSubscriberCount0 = get_empty_user_subscriber_count_structure(NNodes, []),
89      UserSubscriberCount1 = evaluate_user_subscriber_count(AllUsers, UserSubscriberCount0),
90      UserSubscriberCount2 = lists:keysort(2, UserSubscriberCount1),
91      UserSubscriberCount3 = update_user_subscriber_count_rank(lists:reverse(UserSubscriberCount2), 1, []),
92
93      {_UserSubscriberCount, _NRanks, _NSubs, _NTweets} = simulate_zipf_distribution_each_user(UserSubscriberCount3,
        NNodes, [], [], [], []),
94
95      % io:format("~p~n~n", [{_UserSubscriberCount}]).
96      _StringUserSubscriberCount = io_lib:format("~p",[_UserSubscriberCount]),
97      StringUserSubscriberCount = lists:flatten(_StringUserSubscriberCount),
98
99      file:write_file("./scratch.txt", io_lib:fwrite("~s", [StringUserSubscriberCount])).
100
```

Fig 8. Code triggered for Zipf Distribution Simulation

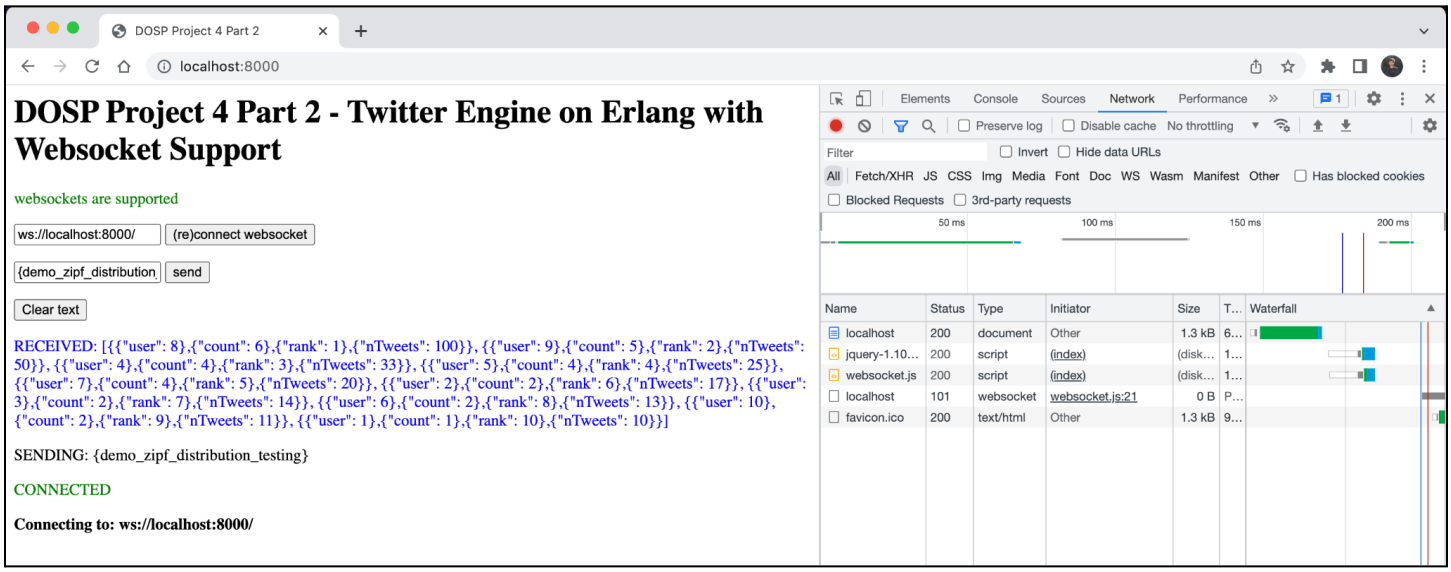

Fig 8. Result of demonstrating Zipf Distribution Simulation

The observed result is as expected, every user is ranked according to how many other users have subscribed to them. Based on this ranking, the number of tweets to be posted by that user is determined to conform to a Zipf Distribution. User 8 is ranked first, as they have six users subscribed to them and will be posting 100 tweets, and likewise, User 1 is ranked last, which is ranked #10, as they have only one user subscribed to them and will be posting only ten tweets.