# Project 4 part 2: Implementing Twitter Clone in Erlang using WebSocket interface

## 1> Project Description:

In this project, we aim to implement a WebSocket interface with the Twitter Clone and client simulator already developed in part 1 of this project. We design a JSON based API that is used to represent messages passed between client and server and the API is used to represent replies passed between them. In this study, we rewrite the server, client, and client simulator code developed in part 1 of this project. This Twitter Engine now paired with WebSockets offers complete functionality.

## 2> Methodology Description:

In this project, we make use of WebSockets along with client server architecture to develop different functionalities Twitter offers. We rewrote our server code, client code, and client simulator code so that all the functionalities developed in part 1 work along with WebSockets. The server now makes use of WebSockets interface, and the client uses the WebSockets. The server distributes tweets while the client sends and/or receives tweets. Initially, the main thread of the server process is spawned and all ETS databases are initialized. The server then waits for incoming requests from the clients. The server spawns a separate process for every client that is logged in and that process is killed when the client logs out. In this manner, the main server process was able to handle more than 1 million clients at the same time.

The Twitter Engine we implemented offers the following functionalities:

- Register an account
- Send tweets. The tweets can also have hashtags (#Happy) and mentions (@user1)
- Subscribe to user's tweets
- Re-tweets so that the subscribers of a client can get a popular tweet
- Allow querying tweets subscribed to, tweets with specific hashtags, tweets in which the user is mentioned
- Live tweeting without querying (If the user is connected, deliver the above types of tweets live)

In addition to the basic functionalities mentioned above, we also implemented the following functionalities:

- SHA-256 secure hashing of passwords as an encryption layer
- View all the registered users

- Ensuring usernames are unique at the time of registration

The architecture of our Twitter Engine implementation with WebSockets is depicted in Figure 1.

Our implementation has the following:

**server.erl: We rewrote our server code to implement the WebSocket interface.** This file contains the code for handling client requests by spawning a process for each client. The server has access to all the ETS tables essential for storing tweets, querying tweets, providing live feed, and handling subscribers. The server communicates with the clients using the sockets of the clients to distribute tweets to subscribers, to provide live feed and query results. We create user database, tweet database, subscribe database, and client Pid database ETS tables in server.erl and use them for different functionalities described above. **Our Pid database stores username and connection socket.**

**client.erl: We rewrote our client code to make sure the client uses WebSockets.** This file contains the code of functionalities like register, send tweet, subscribe, retweet, etc. It takes input from the user and sends the data to the server using Server socket. The information sent to the server will be stored in relevant ETS tables mentioned above. The server handles the client requests and relevant information is sent back from server to client.

**Zipf distribution:** We simulate a Zipf distribution on the number of subscribers. In order to create the Zipf distribution, we take max_subscribers as an argument from the user. max_subscribers provide the maximum number of subscribers a client can have in the simulation. The client with the second highest number of subscribers has max_subscribers/2 number of total subscribers. The client with the third highest number of subscribers has max_subscribers/3 number of total subscribers and so on. The Zipf distribution is obtained using the formula given below:

$$n\_subscribe = round(float.floor(max\_subscribers/(n\_clients - count + 1))) - 1$$

where $n\_subscribe$ is number of clients to subscribe
$count$ is user_id of a client
$max\_subscribers$ is maximum number of subscribers
$n\_clients$ is number of clients

Every client is subscribed to other clients using the formula described above to achieve Zipf distribution. In order to test the strength of our Twitter Engine, we also

generate a large number of tweets. It was necessary to increase the number of tweets for accounts with a large number of subscribers in order to put more strain on the tweet distribution engine.

$$num\_Tweets = round(float.floor(max\_subscribers/count))$$

where $num\_Tweets$ is number of tweets
$count$ is user_id of a client
$max\_subscribers$ is maximum number of subscribers

This was accomplished by utilizing the formula described above to calculate the number of tweets, which ensures that the number of tweets for each user account is equal to the number of subscribers.

**Live connection and disconnection for users:** disconnect_clients, the third parameter used by the program to simulate times of live connection and disconnection, collects the proportion of clients who disconnect. The client's simulator console displays the performance statistics at the conclusion if the disconnect_clients parameter is set to 0. If not, it outputs the statistics and keeps simulating recurrent live connection and disconnection times.

The tweet database consists of three fields: username, tweet content, and username of the user who tweeted it originally. We use the 3rd field in the tweet database to differentiate between tweets and re-tweets. For retweeting, we display all the tweets and then the user selects a tweet to retweet. We parse the tweet to get the username of the original tweeter and the tweet content. Tweet content is stored in the tweet filed and the username of the original tweeter is stored in the 3rd field.

By message forwarding to the server and showing the list of tweets received on the client side, all the queries with tweets subscribed to, tweets with hashtags, and tweets in which the user is mentioned were properly handled. A person who is online will receive the tweets via live view in real time.

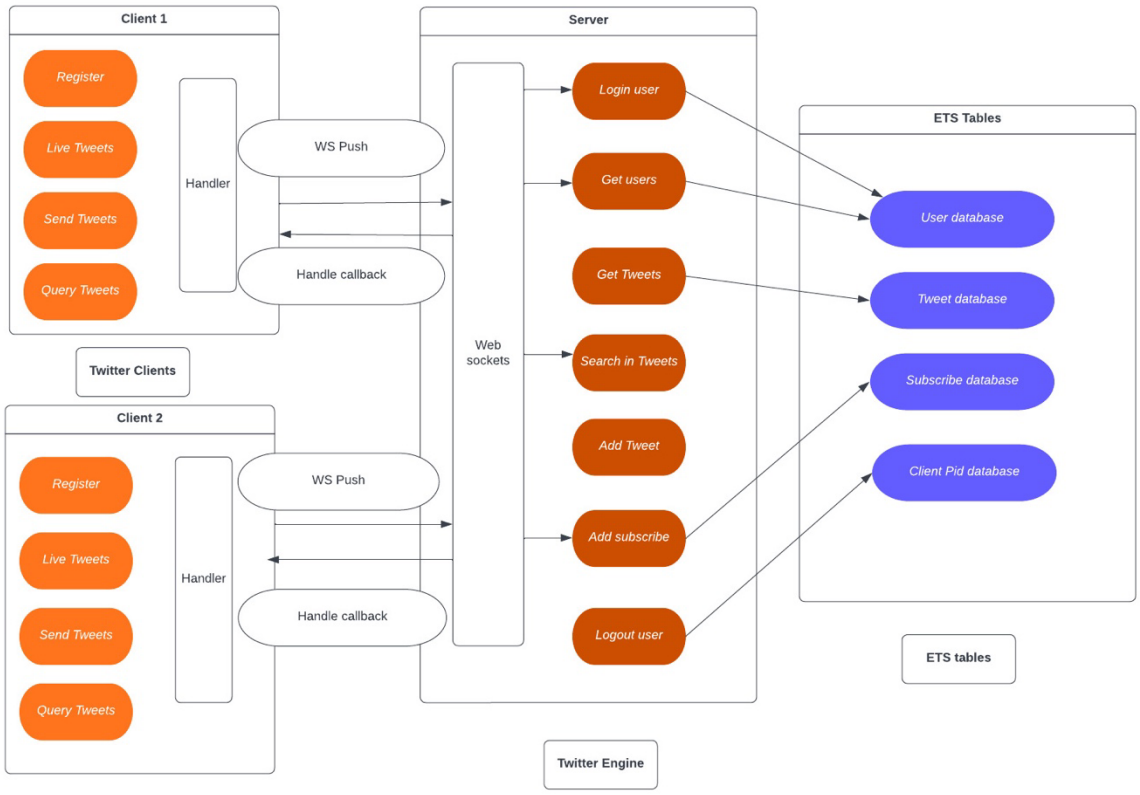


Figure 1: Twitter Engine Architecture using WebSockets

## 3> Running the code:

- Start server using the following commands:

  c(server).
  server:start().

- Start client using the following commands:

  c(client).
  client:start().

## 3> Results:

| Number of Clients | Max Subscribers | Total Execution time |
|---|---|---|
| 100 | 99 | 116 |
| 500 | 499 | 1625 |
| 750 | 749 | 2567 |
| 1000 | 999 | 6168 |
| 2000 | 1999 | 46241 |

Table: Total execution time estimation for different number of clients
(Performance based on Erlang message-based communication)

| Number of Clients | Max Subscribers | Total Execution time |
|---|---|---|
| 100 | 99 | 929 |
| 500 | 499 | 19468 |
| 750 | 749 | 36742 |
| 1000 | 999 | 70939 |
| 2000 | 1999 | 231278 |

Table: Total execution time estimation for different number of clients
(Performance using WebSocket Interface)

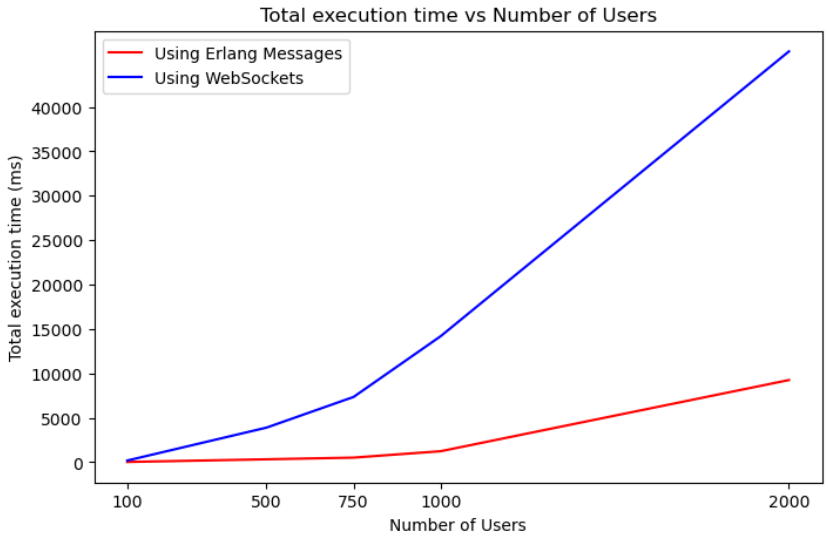


Figure: Total execution time in milliseconds vs number of clients

| Number of Clients | Max Subscribers | Total Execution time |
|---|---|---|
| 1000 | 100 | 1161 |
| 1000 | 200 | 1849 |
| 1000 | 500 | 4157 |
| 1000 | 750 | 9781 |
| 1000 | 999 | 12646 |

Table: Total execution time estimation for different number of max_subscribers (Performance based on Erlang message-based communication)

| Number of Clients | Max Subscribers | Total Execution time |
|---|---|---|
| 1000 | 100 | 5046 |
| 1000 | 200 | 9087 |
| 1000 | 500 | 38319 |
| 1000 | 750 | 74312 |
| 1000 | 999 | 103751 |

Table: Total execution time estimation for different number of max_subscribers (Performance using WebSocket Interface)

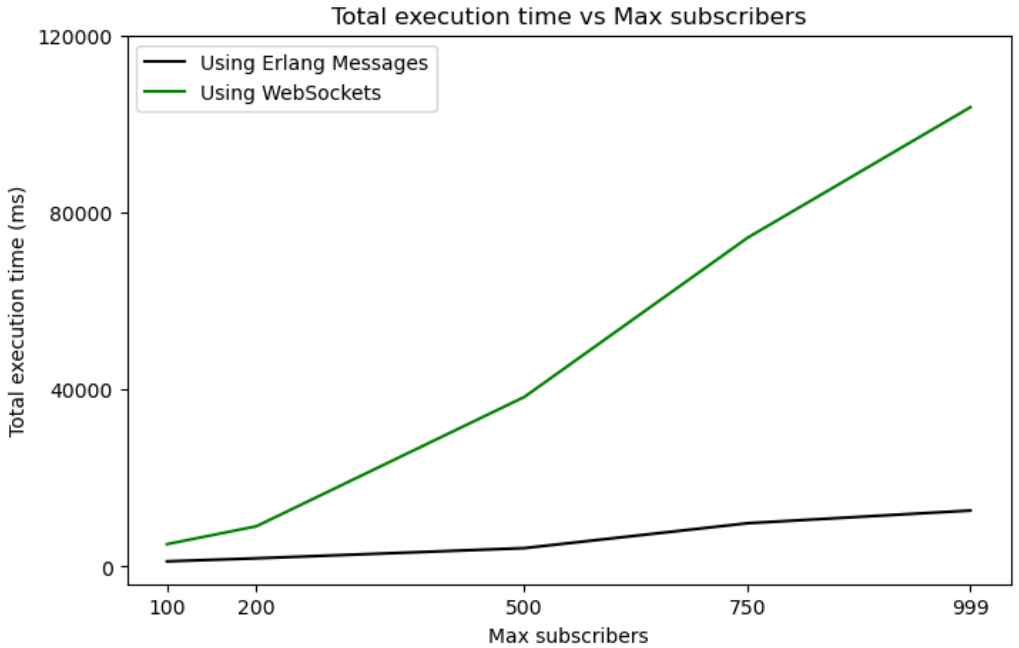


Figure: Total execution time in milliseconds vs Maximum number of subscribers

**Analysis:**

- We have successfully spawned more than 20000 clients. Since our client-server architecture is completely distributed, we will be able to spawn many more clients with a machine that has more computation power.
- When the maximum subscribers increase, the number of tweets increase by a large extent adding load on the server. Because of this increased load, the number of clients that can be spawned will decrease.

## 4> Additional functionalities:

For two parties to communicate securely, we need to create an exchange secret key that is exceptionally hard to compute if not known. We can use a single secret key and then bootstrap it for all the future secrets. Suppose we have a hashing scheme. For example, SHA-256. To compute hash, we need two parameters: seed and message. An example of seed is nonce for every user. We combine seed with SHA-256. In this project, we added SHA-256 for securely hashing the passwords as an encryption layer.

## 5> YouTube video link:

https://youtu.be/lQzvTMUaiXQ

## Team members:

The team members involved in this project are:

1. Sri Sai Sruti Kuppa, UFID: 45303714, srisaisrutikuppa@ufl.edu
2. Sai Nikhil Dondapati, UFID: 22286439, sa.dondapati@ufl.edu