DOS Project 4 Part II (Twitter Clone)

Aniketh Sukhtankar (UF ID 7819 9584)
Shikha Mehta (UF ID 4851 9256)

Brief Description:

The goal of this project is to use the Phoenix web framework to implement a WebSocket interface to the Twitter Clone and client tester/simulator built in part I. The problem statement is to implement an engine that is paired up with WebSockets to provide full functionality as per the requirements of part I. The client part (send/receive tweets) and the engine (distribute tweets) were simulated in separate OS processes. Multiple independent client processes that represented up to 5000 simulated clients were spawned during an iteration and handled by the Phoenix server.

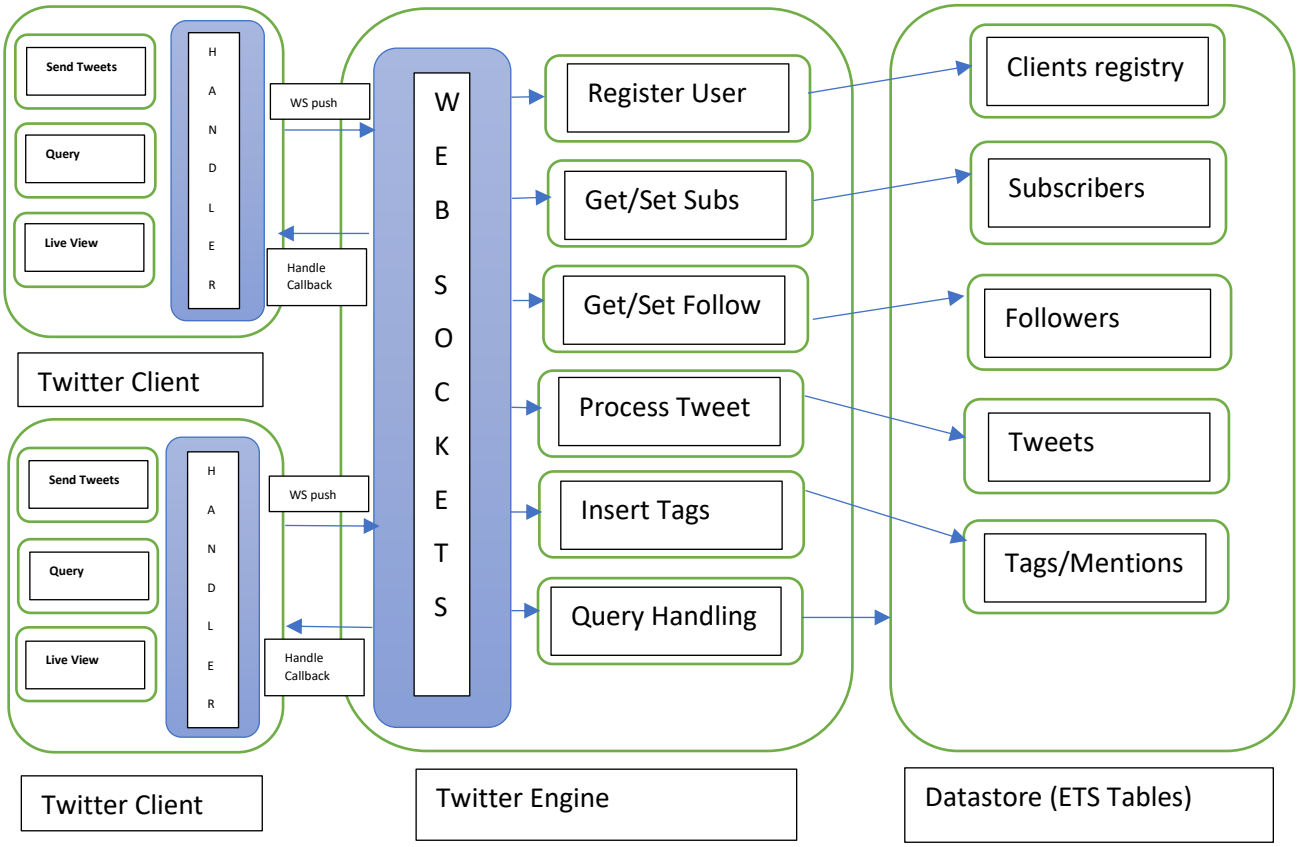
Specifically, we:

- Designed a JSON based API that represents all messages and their replies (including errors)
- Re-wrote our engine using Phoenix to implement the WebSocket interface
- Re-wrote our client to use WebSockets.

Further, the now Phoenix-based Twitter engine (Server) has been implemented with the following functionality:

- Register account
- Send tweet. Tweets can have hashtags (e.g. #COP5615isgreat) and mentions (@bestuser)
- Subscribe to user's tweets
- Re-tweets (so that your subscribers get an interesting tweet you got by other means)
- Allow querying tweets subscribed to, tweets with specific hashtags, tweets in which the user is mentioned (my mentions)
- If the user is connected, deliver the above types of tweets live (without querying)

Architecture



Prerequisites

The following need to be installed to run the project:

- Elixir

- Erlang

- Phoenix Framework

Two terminals are required to execute Phoenix server and clients simulator separately.

Client Simulator Program Inputs

- numClients: the number of clients to simulate
- maxSubscribers: the maximum number of subscribers a Twitter account can have in the simulation (must be < numClients)
- disconnectClients: the percentage of clients to disconnect to simulate periods of live connection and disconnection

Running Project4 Part II

1. Go to the folder 'example' in one terminal using command line.

2. Start the Twitter engine in that by executing the following sequence of commands:

    a. mix deps.get

    b. mix deps.compile

    c. mix phx.server

3. Start the clients simulator in the other terminal using the following command: escript project4 <numClients> <maxSubscribers> <disconnectClients>

  e.g. escript project4 10 5 20

4. Note that both programs do not terminate by themselves.

  a. Twitter engine is designed to always stay on-line to handle incoming client connections.

  b. Clients simulator simulates recurring periods of live connection and disconnection.

5. To simulate the system with different parameter values, please restart both the programs and repeat the steps above.

Output

1. On the simulator's console we print query results for all the 3 types of queries, prefixed with the corresponding user's ID.

2. The simulator's console also prints live tweets for every user. User ID is prefixed to this output as well to identify which user's live view is getting updated.

3. If <disconnectClients> parameter is 0, the clients simulator console displays the performance statistics at the end. Otherwise, it prints the statistics and continues to simulate periods of live connection and disconnection.

Client Implementation Details:

Main.ex: This elixir file is the simulator's entry point and hosts the main method. Our implementation takes 3 parameters in the order - numClients, maxSubscribers, disconnectClients

- numClients: the number of clients to simulate (eg. 100)
- maxSubscribers: the maximum number of subscribers a Twitter account can have in the simulation (eg. 50)
- disconnectClients: the percentage of clients to disconnect to simulate periods of live connection and disconnection (eg. 10)

Client.ex: This file consists of the information pertaining to a single client participant that stands for a single twitter user. This includes the information of its userId which is stored as a numeric string passed to it upon initiation. The underlying logic of maintaining process state is done by the mainregistry ETS table that keeps track of various actors and is useful for disconnection and reconnection logic. **The client was rewritten to use WebSockets**. We used the phoenix_gen_socket_client library for websockets. The client was implemented as a behaviour, which allowed us a lot of flexibility. To send a message on the connected channel, we use GenSocketClient.push/4. If the socket has not joined the topic, an error is returned. On success, the message reference (aka ref) is returned. If the server-side channel replies directly (using the {:reply, ...}), the handle_reply/5 callback is invoked with the matching ref. If the server sends an asynchronous message (i.e. not a direct reply), the handle_message/5 callback is invoked.

Zipf distribution - As per the requirements we were asked to simulate a Zipf distribution on the number of subscribers. The Zipf distribution was handled by taking an additional parameter from the user, maxSubscribers which provides the maximum number of subscribers a client can have in the simulation. The client with the second highest number of subscribers had maxSubscribers/2 total subscribers while the client with the third highest number of subscribers had maxSubscribers/3 total subscribers and so on. This was achieved by calculating the number of clients a user should subscribe to in order to achieve zipf distribution using the formula noToSubscribe = round(Float.floor(totalSubscribers/(noOfClients-count+1))) – 1, where 'count' is the userId of a client. Every client then subscribed to noToSubscribe other clients thereby achieving the required Zipf distribution. Further, we also increased the number of tweets

For accounts with a lot of subscribers the number of tweets had to be increased so as to increase load on the engine for tweet distribution. This was done by ensuring that for any user account, its total tweets are

equal to the number of its subscribers, using the formula round(Float.floor(totalSubscribers/count)) for calculating noOfTweets.

Periods of live connection and disconnection for users – The third parameter taken by the program is disconnect clients which takes in the percentage of clients to disconnect to simulate periods of live connection and disconnection. If <disconnectClients> parameter is 0, the client's simulator console displays the performance statistics at the end. Otherwise, it prints the statistics and continues to simulate recurring periods of live connection and disconnection.

Re-tweets are handled by making every client randomly pick a tweet from one of its subscribers and retweet it to its own subscribers. A "-RT" is appended to the end of a retweet to differentiate it from normal tweets as in original Twitter.

All the queries with tweets subscribed to, tweets with specific hashtags, tweets in which the user is mentioned (my mentions) were handled successfully by communicating over the web socket interface to the server and displaying the list of tweets received on the client side. The tweets will be delivered live to a user that is online by means of live view. User ID is prefixed to this output to identify which user's live view is getting updated.

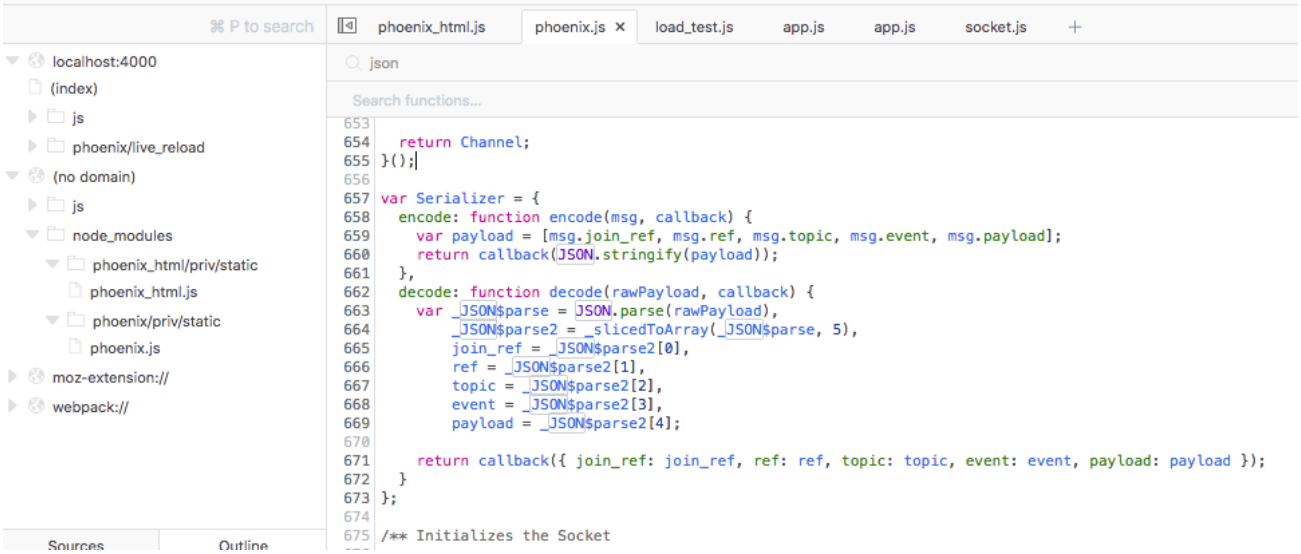
Server Implementation Details:

We also re-wrote our engine using Phoenix to implement the WebSocket interface. The 'example' folder contains the code for Twitter engine implementation that is responsible for processing and distributing tweets. The engine directly communicates with the database (implemented using ETS tables) to handle subscribers, tweets, queries, etc. It also communicates directly with clients via Phoenix channels / websockets to distribute the tweets to subscribers, send the query results etc. Inside the server maintaining process states is done by the clientsregistry ETS table that keeps track of various actors.

Channels allowed us to easily add soft-realtime features to our twitter application. Channels are based on web sockets. The server acts as a sender and broadcasts messages about topics. Clients act as receivers and subscribe to topics so that they can get those messages. Senders and receivers can switch roles on the same topic at any time.

Our Phoenix server holds a single connection and multiplexes channel sockets over that one connection. Socket handler, that is present in lib/example_web/channels/user_socket.ex, are modules that authenticate and identify a socket connection and allow us to set default socket assigns for use in all channels. The default transport mechanism is via WebSockets.

We also designed a JSON based API that represents all messages and their replies (including errors). Since Phoenix framework internally handles the data transfer using JSON the first criteria of designing a JSON based API was also handled. To confirm this, we looked at phoenix.js which is automatically included while served from Phoenix:



As we can see JSON is used for serialization. Hence, we learned that Phoenix handles the Elixir-to-JSON encoding on the server side and by using phoenix.js, we did not have to explicitly JSON.stringify or JSON.parse on the client side as the Phoenix framework took care of that.

Performance Results:

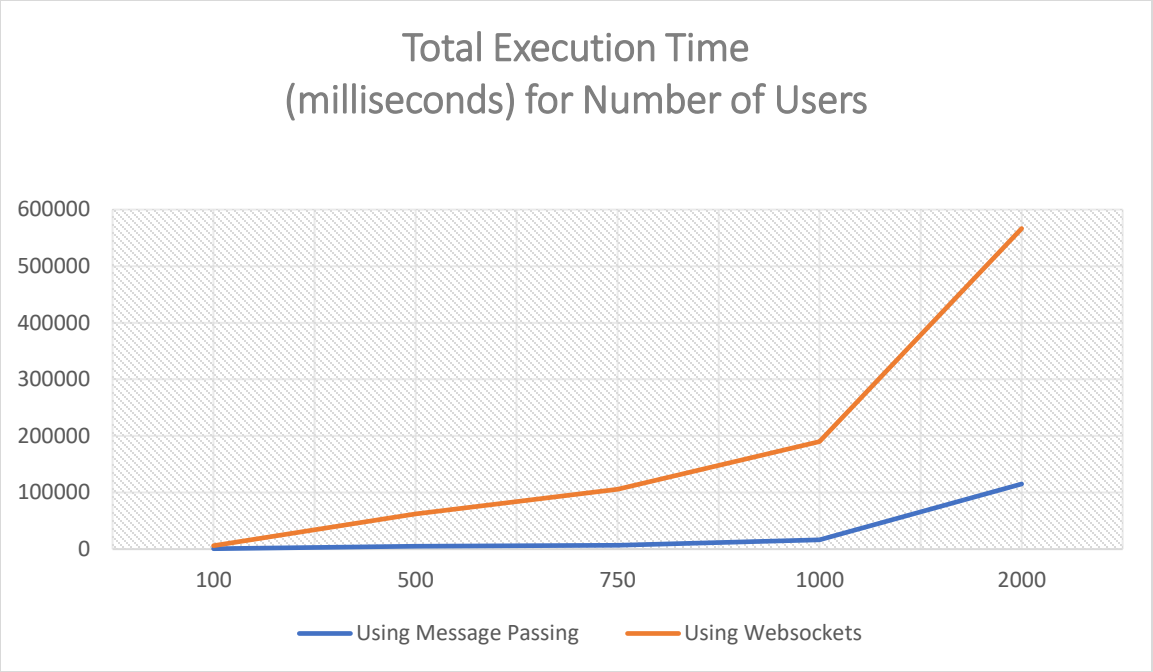
The time taken for the simulation to run vs the number of clients is plotted as a graph for both Message passing and Web socket implementations. A maximum of **5000 clients** have been spawned on our system. The values and consolidated graphs for results we obtained have been disclosed below:

Performance using Erlang Message Passing when Max Subscribers(and max tweets/account) = Number of Clients (0% disconnectClients)

Number of Clients	Max Subscribers	Time to Tweet, Retweet (milliseconds)	Query Tweets Subscribed To (milliseconds)	Query Tweets by Hashtag (milliseconds)	Query Tweets by Mention (milliseconds)	Query All Relevant Tweets (milliseconds)	Complete Execution (milliseconds)
100	99	43.96	122.83	36.52	15.83	6.42	734
500	499	253.39	2309.92	320.418	333.19	146.386	5109
750	749	301.61	3523.74	633.11	418.57	206.92	7047
1000	999	1256.88	6616.81	1294.74	703.46	197.14	16531
2000	1999	5855.21	54364.68	7393.04	17024.85	2494.94	115141

Performance using Websocket interface when Max Subscribers(and max tweets/account) = Number of Clients (0% disconnectClients)

Number of Clients	Max Subscribers	Time to Tweet, Retweet (milliseconds)	Query Tweets Subscribed To (milliseconds)	Query Tweets by Hashtag (milliseconds)	Query Tweets by Mention (milliseconds)	Query All Relevant Tweets (milliseconds)	Complete Execution (milliseconds)
100	99	265.47	1323.48	395	86.87	58.76	6125
500	499	2549.17	17329.73	2145.30	1109.77	303.21	62297
750	749	3158.22	21654.68	9416.32	9150.92	5303.03	105515
1000	999	4712.62	46087.93	19548.13	5672.02	4665.21	189984
2000	1999	5878.17	75068.48	13878.66	11994.47	1742.33	566516

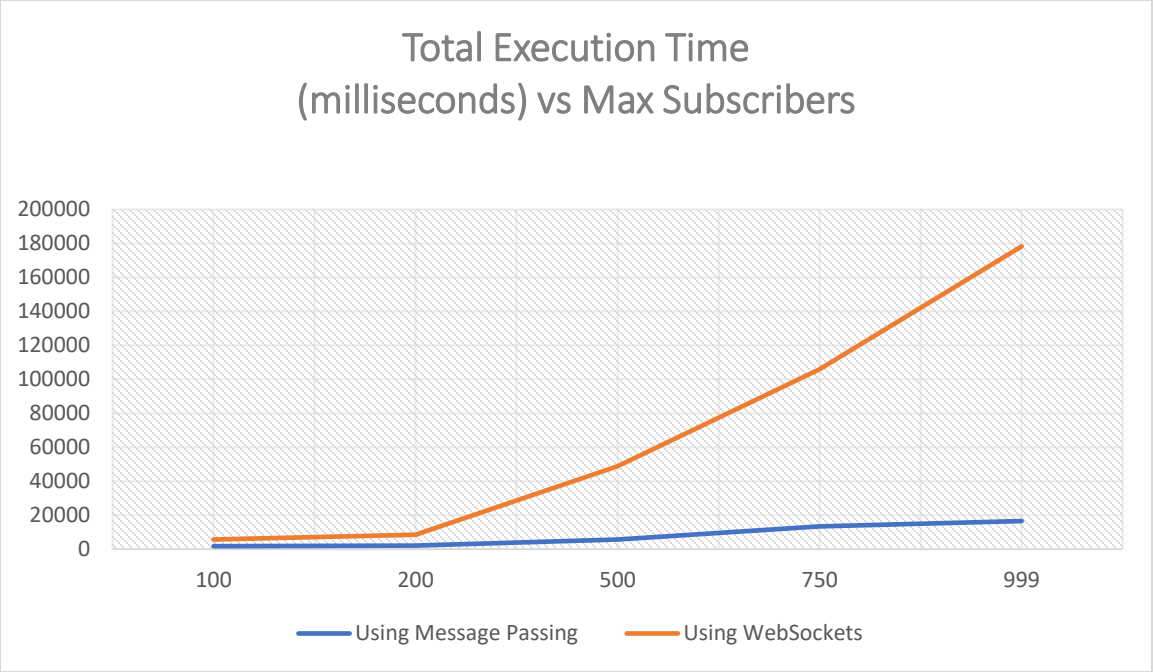

Total Execution Time
(milliseconds) for Number of Users
600000
500000
400000
300000
200000
100000
0
100
500
750
1000
2000
Using Message Passing
Using Websockets

Performance using Erlang Message Passing for different number of Max Subscribers / Max Tweets per Account (0% disconnectClients)

Number of Clients	Max Subscribers / Max Tweets per Account	Time to Tweet, Retweet (milliseconds)	Query Tweets Subscribed To (milliseconds)	Query Tweets by Hashtag (milliseconds)	Query Tweets by Mention (milliseconds)	Query All Relevant Tweets (milliseconds)	Complete Execution (milliseconds)
1000	100	36.16	172.65	364.73	311.63	167.23	1734
1000	200	100.00	380.17	377.56	335.57	111.49	2094
1000	500	412.78	1468.85	1311.19	404.98	160.67	5688
1000	750	139.92	7047.60	2999.58	1700.60	271.67	13484
1000	999	1256.88	6616.81	1294.74	703.46	197.14	16531

Performance using Websocket Interface for different number of Max Subscribers / Max Tweets per Account (0% disconnectClients)

Number of Clients	Max Subscribers / Max Tweets per Account	Time to Tweet, Retweet (milliseconds)	Query Tweets Subscribed To (milliseconds)	Query Tweets by Hashtag (milliseconds)	Query Tweets by Mention (milliseconds)	Query All Relevant Tweets (milliseconds)	Complete Execution (milliseconds)
1000	100	247.11	224.68	1017.83	594.71	218.51	5672
1000	200	252.40	380.54	1362.83	549.22	197.97	8500
1000	500	244.45	2791.65	1774.67	876.11	589.93	48875
1000	750	2443.35	18098.63	5158.14	4887.51	2263.81	105859
1000	999	4104	37803.92	19021.37	12507.01	7031.84	178329


Total Execution Time
(milliseconds) vs Max Subscribers
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
100
200
500
750
999
Using Message Passing
Using WebSockets

References:

- http://highscalability.com/blog/2013/7/8/the-architecture-twitter-uses-to-deal-with-150m-active-users.html
- https://elixirschool.com/en/lessons/specifics/ets/
- https://medium.com/@Stephanbv/elixir-phoenix-build-a-simple-chat-room-7f20ee8e8f9c
- https://elixir-lang.org/
- https://medium.com/@benhansen/lets-build-a-slack-clone-with-elixir-phoenix-and-react-part-1-project-setup-3252ae780a1
- https://blog.diacode.com/trello-clone-with-phoenix-and-react-pt-1