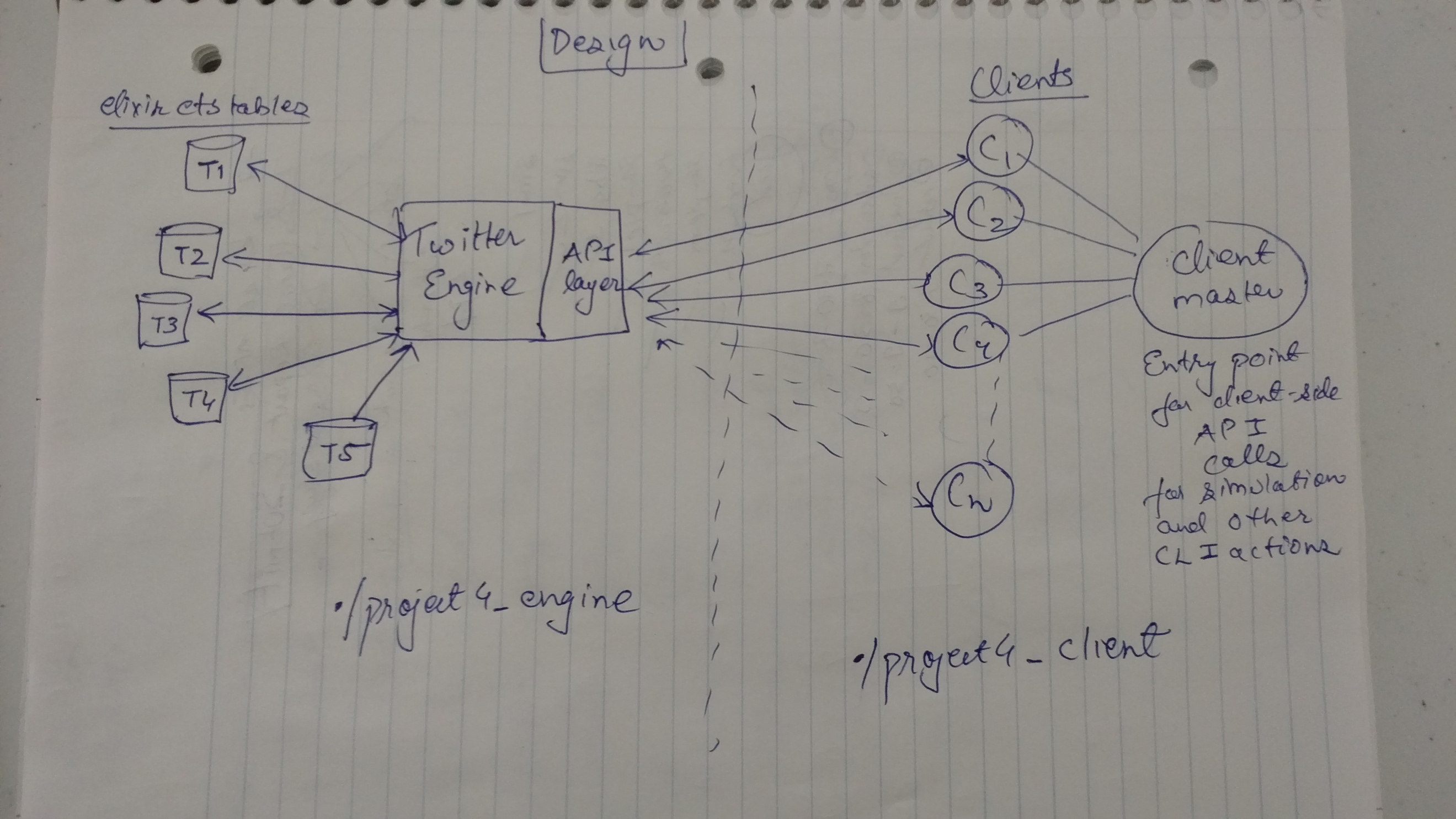
**README has directions as to how to run the project with the best practices for testing**

**In this Report, I discuss the design, parameters used to tune the simulation and performance of the solution**

# **Design**



## Engine

The Twitter Engine is started by running ./project4\_engine. It supports the following APIs:

1. Register
   1. Register a new user
   2. Register the client-master for reporting stats
2. Feed – Get Feed for a user based on the users it is subscribed to
3. Hashtag
   1. Tweets containing a particular hashtag
   2. Sample of hashtags from the database for CLI actions
4. Mention
   1. Tweets containing a particular mention
   2. Sample of mentions from the database for CLI actions
5. Subscribe – User A subscribes to User B
6. Tweet – A user tweets
7. Retweet – A user retweets something that was in it’s feed

## Client

The clients are spawned by running ./project4\_client *with the appropriate parameters*. More information of various Command line actions and the order in which they need to be performed is in the README file

## Tables

I have used 5 tables (ets tables) for various kinds of querying with the following schemas. The primary key is in **bold**:

1. **UserId (int)** | SubscribedToUserIds(array of ints)
2. **UserId (int)** | TweetIds (array of ints)
3. **Hashtag (string)** | TweetIds (array of ints)
4. **Mention (string)** | TweetIds (array of ints)
5. **TweetId (int)** | Tweet (string) | timestamp (System.monotonic\_time in microseconds)

The tables help me in efficient retrieval of all the basic queries and API calls

# **Parameters**

* *A pre-computed list* of 1K hashtags, 1K mentions and 8K tweets
  + The tweets are generated by a base-62 number system.
  + Every tweet has a hashtag, mention or both on a random basis
  + Before sending a tweet, it is prepended with ‘Tweet:USERID’
* *number of users* – This is inputted at the start of the simulation.
* *zipf-factor* –A simplistic zipf distribution has been modelled. The top 20% users tweet more than the bottom 80%. The zipf factor is used to quantify this. Default is 0.1 times the (20/80)% times 1 millisecond being interval between tweets for the user
* *print\_every\_factor* (p)– used to calculate and print the tweet-rate every ‘p’ times the number of users. A running average is calculated and printed . Default is 3
* *feed\_limit (f) –* When feed for a userid is requested, only the latest f tweets are returned. Default is 20

# **Performance**

## Number of Users supported

No theoretical limit imposed by the design. But on my machine I was able to simulate **100K users.** Any more would lead the VM to kill the process automatically

## Average number of tweets

* The average number of tweets processed is comparatively huge initially but later on degrades. Reason being the ets table gets filled up and subsequent querying simply takes more time
* By zipf law, approximately top 20% are tweeting every .02 millisecond and bottom 80% are tweeting per .08 millisecond. This can be tuned by the zipf\_factor parameter
* Below values are starting from a clean slate with empty tables which eventually fill up

|  |  |  |  |
| --- | --- | --- | --- |
| Number of users | Initial average rate | Eventual average rate | Average rate |
| 100 | ~10K tweets/s | ~2K tweets/s | ~6K tweets/s |
| 1K | ~11K tweets/s | ~5K tweets/s | ~7K tweets/s |
| 10K | ~9K tweets/s | ~7K tweets/s | ~7K tweets/s |
| 100K | ~5.5K tweets/s | ~5.5K tweets/s | ~5.5K tweets/s |

As we can see, with the given parameters, the average tweets processed by the engine is **~6.5K tweets/s**.