Distributed Operating Systems

Dr. Alin Dobra

Meenakshi Shendye (0166-1233)
Advait Ambeskar (9615-9178)

Table of Contents

Introduction

A distributed system is a network of autonomous actors that are logically connected to each other to operate as a single entity using a distribution middleware. The connected actors share resources and capabilities to provide the user with a single network, while maintaining the separation of execution. An actor in distributed system can be attributed to a collective entity comprising of memory, processing capability and logical capacity that can independently be pursued to execute the required instructions. The inherent need for distributed systems comes from the increased complexity of the executed tasks, it is inherently important to reduce the overall execution time while utilizing the maximum CPU power and multiplicity of the CPU cores.

The distributed system designed as part of this project is based on Twitter. Twitter is a microblogging social networking service that allows users to interact with each other through messages sent through its internal services. A distributed system is an important design choice in such applications due to the large volume of messages that are sent concurrently. As part of this project, we implemented a small subset of functionalities to showcase the superiority of this design choice.

The project has been implemented in Elixir (1.9.1) with strict dependencies on **ecto** (ecto_sql: 3.0) and **postgrex** (postgrex: > 0.0.0). Ecto and postgrex allow creation of a persistent storage system that allows us to maintain state beyond different execution. It is important to understand that adding dependencies is not a favorable system design, however, the advantages of using a postgresql database are greater than the intricacies of solving the dependencies. The advantage of using elixir for building the system is that it allows decentralized execution and operations of actors.

Design

The system design comprises of modularized structure with a single entry-point to allow for better readability and execution flow. The following code structure is maintained –

-- Project41.Application

  -- Project41.Demo

    -- Project41.LiveUserServer

    -- Project41.ClientFunctions

    -- Project41.TweetFacility

    -- Project41.TweetEngine

    -- Project41.LoginEngine

The database schema is stored as part of User data **"Userdata"**, Topic **"Topic"**, Tweet **"Tweetdata"**, Follower **"Follower"**, User feed **"Feed"** hosted under the module name format **"Project41.XYZ"**.

The important distinction between a database dependent system and the currently designed system is that the system has been designed such that **it does not essentially depend on the database** for anything more than maintaining states between multiple executions and **allowing a sort of checkpoint** in case of failure.

Module and functions

- **Project41. LiveUserServer**
  Project41.LiveUserServer holds the functions that allow a central process to maintain the actors currently successfully logged into the system.

- **Project41.ClientFunctions**
  Project41.ClientFunctions holds the functions that allow a single end point for the server/ engine side functions.

- **Project41.TweetFacility**
  Project41.TweetFacility holds the functions that define the methods in which user can perform defined features (send tweet, retweet, query, read feed)

- **Project41.TweetEngine**
  Project41.TweetEngine holds the functions that define the methods which generate the various user processes for execution.

- **Project41.LoginEngine**
  Project41.LoginEngine holds the functions that define the methods that allow the users to perform login functionalities.

Persistent Storage

An important conscious choice made during development and implementation of this project was the choice to use persistent storage system in form of PostgreSQL. The idea behind this choice was simple, to provide the program with the ability to store the data transferred during execution beyond the risk of failure. Since PostgreSQL is a commonly use relational database management system (RDBMS) with linking in elixir through their **Ecto** module, the choice of using this RDBMS was advantageous. Use of a database allowed for easy checkpoints in case of process failure.

At the same time, it is important to understand that using a database results in a significant overhead. If a system design relies significantly on reading from a database, the overhead can become unbearable. Thus, a deliberate choice to minimize reading from the database was made. Most operations needed for proper function of the features implemented occur through **use of processes associated with each of the actors**. The major use of the database was to store the sent tweets, retweets, maintain the username and passwords of the registered users and store the feed associated with each of the users. This allowed us to revert back to the last-known state each time the user would log-off and log-in by retrieving the data from the storage.

Concurrent Design

Elixir provides the user with an opportunity to design applications which employ concurrent design in their roots through actor model. In the design of this project, we used GenServer processes to maintain the state of the actors. Actors hold the state in form of their user-id (a unique code which corresponds to their username), tweets (the tweets that each user has made), followers (the list of all the users who subscribe to the said actor) and feed (the collection of all tweets in their feed – a list of feeds by themselves, their followers and the tweets that mention the actor). Since each actor process is associated with a single user, this one-to-one association allows us to host multiple actors at the same time without causing scheduling errors.

Testing

The implementation of this system was testing using the ExUnit module provided with Elixir. A total of **8 tests** took place with 100% success. The tests concluded that the implemented features functioned the same manner as expected. The tests were run on both empty and pre-filled database.

System Implementation

The project implementation uses different modules each with their respective functions helping with the features implemented as needed. The mentioned module functions are only the end points for the features. There are multiple supporting/ secondary functions that are not written here due to the scope of this document. The user 'feed' is updated each time there is a new entry added to the current state of the actor. This allows for an easy way in which one can view the user feed.

Project41.Client

register(username, password) – registers the user (creates a new entry in the database associated with the username and password that the user can later use to login into the system. If the registration is successful, the user is also logged into the system.

login(username, password) – logs in the user (checks if the user is previously registered) such that the username, password combination matches the records with which the registration has occurred. It also detects if the user is already logged into the system

logout(username) – logs the user out of the system. Log out can occur only if the user is currently logged in.

delete(username, password) – deletes the registration entry of the username, password. This can occur only if the user is currently logged into the system. Deleting the user permanently deletes the entries associated with the user.

subscribeToUser(subscriber, username) – allows the *subscriber* to follow the *username*. The subscriber will receive all the updates created by the username (retrieved from the feed in case the *subscriber* was **offline** when the *username* updated their feed.

tweet(user, tweet) – allows the user to send a tweet. The tweet can contain @otherusers, or #hashtags (mentions and hashtags). Tweet is essentially a string, which is sent only if the user is currently online and the mentioned users get updated only if they exist.

retweet(user, tweetid) – allows the user to retweet a previously existing tweet. The tweetid is the unique binary-id of the tweet that is being retweeted and is created when the user originally sent the said tweet. The retweet occurs with the prefix ***"retweet by @user ->"***

Project41.TweetFacility

hashtagSearchFacility(hashtag) – allows the user to query all the tweets that are associated with the hashtag.

userSearchFacility(user) – allows the user to query all the tweets that have been generated by a given user.

Project41.DatabaseFunctions

mentions(username) – allows the user to query all the tweets that have mentioned the specified username.

Constraints

The application was run on **10000 users, each sending 1000 messages**. Due to the sheer volume of this numbers, and the inability of the testing systems to handle them (due to lack of better configurations), we have not tested the values beyond those described. However, due to scalability of the suggested solution. It is easy to determine that the implementation will run as expected.

Observations

A few observations made were as part of the testing of the implementation. Due to a unique (random) nature of this implementation, it is difficult to ascertain the outputs. However, through testing of each individual functions, it is possible to

Tests

We have tested functions in different modules of the Project. There are total of eight tests:

test_registered_user: This is a test to test the working of the function registerUser in the LoginEngine Module which is supposed to register a new user and log him in. If the user is already registered, it replies with :olduser and in the other case the expected reply is :newUser. We have tested this by the assert functionality.

test_delete_user: This is to test the ***deleteUser*** function of LoginEnngine. If a user is an old user, they must be deleted and if they are a new user then the function returns indicating that we need to login first before deleting. We also check if the return of the function is never empty or null. We write assert test for this function.

test_username_exists: This is to test the ***username_exists*** function in the LoginEngine module. The expected reply for this function is a boolean value which states if a username exists. In this test, we create a new user and log him in and then check if this method returns true.

test_login: This function is to test the ***login*** function in LoginEngine. We are trying to login a user two times and then we check if the expected output is :loginUnsuccessful. We are using assert testing in this function.

is_logged_in: This tests the ***is_login?*** Function in LoginEngine. The return type is *boolean* and it returns true of a user is logged in. We are initially logging a user and then checking if this method returns true.

log_out: This function tests the ***delete_user*** function in the LoginEngine. The user is deleted if he is already logged in. Else the functions assert that you need to log in first. We are creating and logging in a new user and when we try to delete the user, he is successfully deleted.

test_subscribe_to_user: This is a function to test the ***subscribe_to_user*** function in TweetEngine. We created and logged in two users. Then we ran this function and matched the output where user1 must be following user2. We have used assert function for testing.

test_tweet_format: This function tests the ***tweetFormat*** in TweetFacility. We have created a new tweet and checked if this function correctly separates the hashtags, tweet and mentions in the tweet. We have used assert function for testing.

```
C:\Documents\github\DOS\project-4\final\project41>mix test
warning: test/TweetEngine_test.ex does not match "*_test.exs" and won't be loaded
warning: unused import ExUnit.CaptureIO
  test/TweetFacility_test.exs:2

."User hello is an old user. Attempting login instead."
."User goodbye is an old user. Attempting login instead."
."User hello is an old user. Attempting login instead."
....

Finished in 0.6 seconds
7 tests, 0 failures

Randomized with seed 794000
```

```
C:\Documents\github\DOS\project-4\final\project41>mix run proj4 2 2
starting demo...
"User User_1 is an old user. Attempting login instead."
"User User_2 is an old user. Attempting login instead."
@User_1 already follows @User_2
You cannot follow your own self.
@User_1 already follows @User_2
You cannot follow your own self.
You cannot follow your own self.
You cannot follow your own self.
@User_2 already follows @User_1
New tweet added to User_1's feed : This is Tweet 1 for @User_1
New tweet added to User_1's feed : This is Tweet 1 for @User_1
New tweet added to User_2's feed : This is Tweet 1 for @User_1
New tweet added to User_1's feed : This is Tweet 2 for @User_2
New tweet added to User_2's feed : This is Tweet 2 for @User_2
New tweet added to User_2's feed : This is Tweet 2 for @User_2
New tweet added to User_2's feed : This is Tweet 1 for @User_2
New tweet added to User_2's feed : This is Tweet 1 for @User_2
New tweet added to User_1's feed : This is Tweet 1 for @User_2
New tweet added to User_2's feed : This is Tweet 2 for @User_1
New tweet added to User_1's feed : This is Tweet 2 for @User_1
New tweet added to User_1's feed : This is Tweet 2 for @User_1
Demo finished...
```

Discussion & Conclusion

This implementation of a twitter-like system is designed to work like the backend server that logically handles all the implemented features as described by the problem statement. It is imperative to understand that this implementation is designed to be always-up, to form a server that will always serve the users who might want to connect and perform meaningful functions. By use of persistent data, we have allowed the system to become more meaningful and improved resilience to data loss due to process failure. This implementation will be further extended to work on the Phoenix framework with a better expression of data and to create a more reliable application.