Pawel Cieslewski
Will Livesey
Distributed OS
11/30/15

Facebook Simulator Report Part 1

Introduction

In this project, students were assigned an open-ended assignment to implement a facebook simulator including a server and many clients. In order to accomplish this, the spray frame work was used in order to enable simple JSON message passing between the clients and server. In this report, we try to roughly model Facebook currently and try to implement some small portion of its backend.

User Study and Assumptions

According to Zephoria.com, these are a couple of rough statistics about Facebook which we use to draw a couple of conclusions.

- Daily Facebook Users : 968 Million
- Pieces of content shared daily : 4.75 billion
- Average time spent on Facebook per day : 20 Minutes

Using this information, we can draw some very simple and rough conclusions.

- Pieces of Content shared per second : ≈ 55,000
- Number of Users Concurrently Online : ≈ 13.5 Million

And from that we can estimate the following:

- Time Between Actions for Online User : ≈ 246 Seconds

However, we have decided that for the Facebook simulator this seems to add a large amount of memory overhead even though the clients would be taking an action every 4.1 minutes. Instead we will dramatically increase the time between actions which will test the server, without allocating as much memory.

We decide to make the following assumptions:

- Each of our clients will make 1 random action every 2 seconds because it is easy to monitor at that speed. There is enough activity to watch each client live.
- Since this is 123x as fast as the regular Facebook user, we will assume that each of our clients is approximately equivalent to 123 Facebook clients. Of course, this is not a perfect estimation because we are not allocating 123x memory for each client. Hopefully, other than memory constraints, this is roughly accurate.

Internals

To better explain our implementation it is important to explain our assumptions for our data structures. The following is a list of our structures and how the information is stored on the server.

- Page
  - Album
  - Profile
  - Friends List
  - Post List
- Album
  - A collection of pictures.
- Profile
  - Name
  - Relationship Status
  - Profile Picture
  - Birthday
- Friends List
  - Collection of Friends
- Post List
  - Collection of all posts written on that page
- Friend
  - Name and ID of another user
- Post
  - String message and name of the poster
- Picture
  - An array of bytes

To give a little insight to what is going on behind the scenes of the project, each client first registers with the server and allows the server to allocate some space for it in its backend.

After the registration, each user will randomly do an action, each actions weighted with the same likelihood.

- Add a Random Friend
  - Since each actor knows how many clients there are, they randomly generate a name and add them to their friends list.
- Post on Own Wall
  - Adds a post to the persons own Page.
- Post on Friend Wall
  - Queries the server for its own friend's list, and then posts on a random friends wall.
- Update Profile
  - Sends the server a profile object with an updated name, profile picture, birthday, and relationship status.
- Add Picture
  - A user creates a photo and adds it to their own album

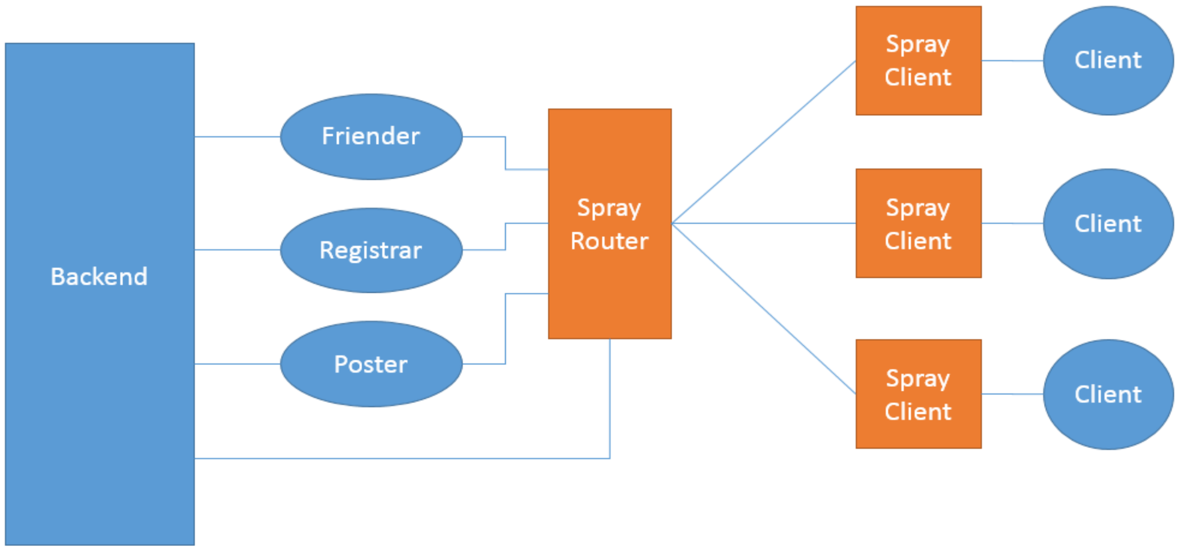
- Read Friend Page
    - Queries for the entire page of a random friend. The server response with the page including the posts, album with all pictures, and profile of the selected user.

We believed that this was enough variety for random tasks. Adding more actions would just be a matter of a little bit more implementation.

Difficulties

The most challenging part of the project was getting Spray to function properly. It was difficult learn how to get spray to properly marshall and unmarshall the objects properly over the wire. However, a more interesting problem we ran into was manipulating data structures in parallel. We realized that we could not have each connection to the server manipulating the backend at the same time, otherwise we risk having data conflicts.

In order to solve this, each resource that could have multiple actions simultaneously, is handled by an individual actor. This diagram will show our layout.



For example, each time a client wants to make a post, the post list of a page needs to be updated. However, if the post list is updated by two actors at the same time, an update will be lost. Therefore, to update a post list, each client sends a message to the "Poster" actor which then executes the posts in series. However, some of our actions did not require an additional actor since only one client can act on it at once. For example, when a profile is updated, only one client has access to do so, therefore no additional safeguards are required.

Results

On a single laptop we were able to simulate 2000 clients doing two requests per second. According to our previous assumptions this is equivalent to approximate 246,000 “Facebook” clients talking to the server. Of course, this number is inflated because we do not allocate 246,000 users worth of data. We just handle requests at that speed.

During a period of one minute, approximate 76 Mb of new data is written to the backend.

Each second, during the simulation the server was handling approximately 1000 requests per second. Each request could include a write, a read, or both.

Limitations and Future Work

Of course, in order to truly test how many requests the server can respond to, the client simulator would need to run on a separate computer. All of the results collected in our study there was a single computer that would run both the server and the clients.

Sources

https://zephoria.com/top-15-valuable-facebook-statistics/