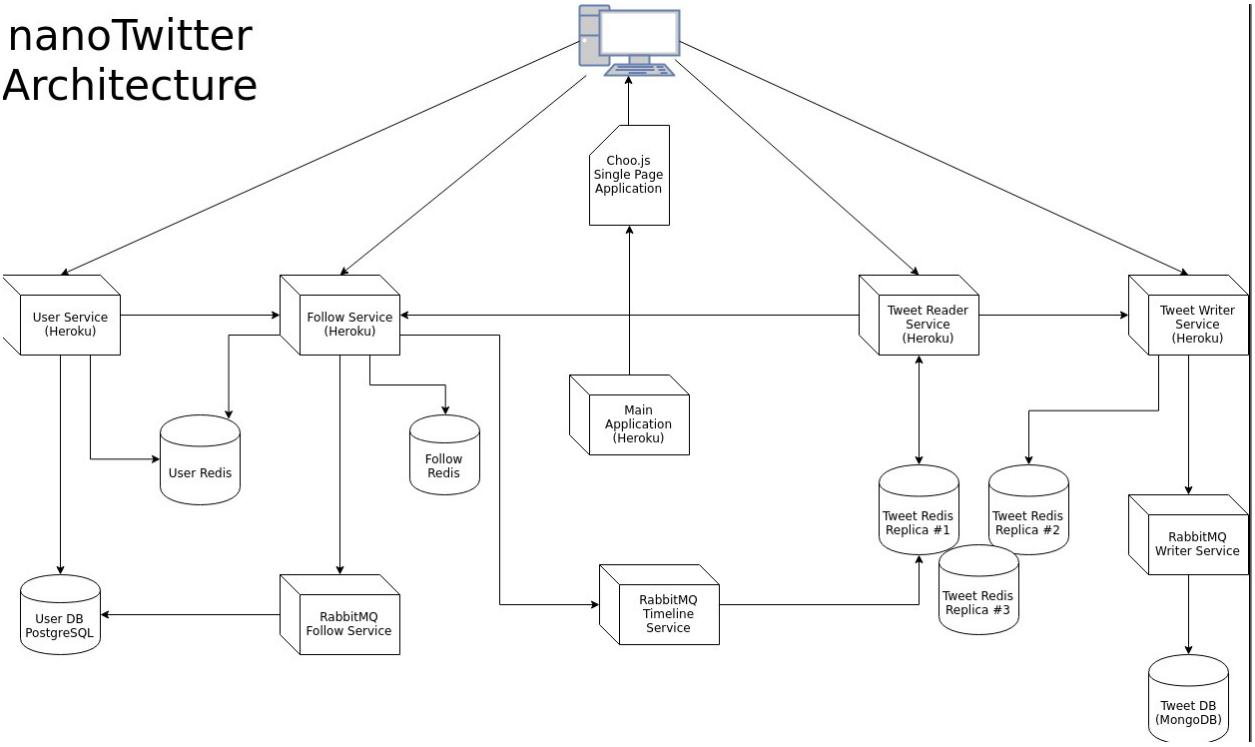
Fantastic Four Team: Thomas Willkens, Zhengyang Zhou, Cameron Cho, Alex Hu
Cosi105b

Assignment 24.1: nanoTwitter 1.0 Update

Overview

This is the nanoTwitter 1.0 update document for the Fantastic Four. As of this document's submission, our complete version of nanoTwitter should be finished (or just about finished). In this document, we will discuss our team's successes, challenges and remaining problems that confronted each of us individually, and as a team.



Cameron

Cameron was in charge of creating the services to read and write Tweets from and to the database or caches.

Successes

In particular, Cameron was able to establish the NoSQL database MongoDB for use via the data cloud mLab and interface with it using the Mongoid gem. He also started the jump into using microservices for our project.

Challenges

The problems he encountered included working around the intricacies of using MongoDB versus Activerecord, particularly the syntax of methods. Cameron also found it difficult to initially balance three services, ensuring that the data was correctly passed between all of them to ensure that the application was working as intended, especially when it came to debugging it. Keeping up with the constant introduction of new technologies (like Redis) and requirements (testing routes) was also difficult.

Zhengyang

Successes

Zhengyang spearheaded and completed the implementations of Redis, RabbitMQ and the timelines for users. In addition, he was responsible for creating the User creation/authentication service and all of the miniature microservices required for using RabbitMQ.

Challenges

In particular, he had to deal with the technological intricacies of using Redis and RabbitMQ, especially the BigWig server the latter employed that caused loader.io test problems. In addition, the complex nature of updating and retrieving timelines proved difficult to implement.

Remaining Tasks

Assuming we had more time, Zhengyang would like the chance to handle cases in which the write assignment on RabbitMQ failed to complete. That handling exists, but it is not used as of this moment.

Alex

Successes

Alex was largely responsible for creating the follow service and the entire test interface for all of the microservices. In addition, he managed the accounts we used for Heroku and mLab.

Challenges

The constantly changing nature of the application (considering that we pivoted at least once) forced him to constantly change the tests to match its structures. As more services cropped up, the more tests that he needed to write to address their functionality, and that also required him to ask the others constantly for new methods to implement that test functionality. Coordination between team members and comprehensive coverage of the application was especially difficult.

Tom

Successes

Tom handled the transition of the main application from Heroku to Amazon Beanstalk (and back again), and was responsible for writing the front-end with Choo. He also had a hand in everything during the final stretch, ensuring every service worked together using loader.io.

Challenges

Implementing these new technologies required a great amount of trial and error. In particular, communication between all of the services and having them pass data in the correct form between each other and Choo proved to be complex.

As a Team

Successes

The team managed to create an application that runs off of seven microservices using an array of different technologies: Heroku, Redis, Postgres, NewRelic, MongoDB and RabbitMQ, among the following.

Challenges

The effort it takes to coordinate just seven microservices is much greater than any of us imagined. Team communication was a challenge, especially concerning what was expected of the project.

Remaining Tasks/Problems

Tom would to like refactored the front-end a bit (one rendering bug in particular). Zhengyang would like to reimplment the microservices using Kafka.

Timeline push and pull model (By Zhengyang)

Timeline should be most difficult part of the backend service in the our nano-twitter, as it involves all of the services and contains a lot of scenarios. In order to speed up the loading of the timeline, we decide to use push model in the most of the cases and pull model when the Redis does not have the timeline of a target user.

The push model starts from posting a tweet. After a user finishes writing his tweets, this tweet will be send to the timeline queue all of his followers on Redis. Although it takes O(n) time, where n is the number of his follower, since this action is done by accessing cache and can be done with asynchronous saving with message broker,

the user who writes this tweet is able to do other things he wants without waiting the tweet saving is done. In other words, if the social graph does not change, i.e the following relation of a person doesn't change, his corresponding timeline can be retrieved by O(1) time through access his timeline list on Redis.

What if a user follows a new leader? Well, we just need to merge the new leader's feeds and the user's current timeline, partially. Here is how I approach it, observing the feed of the new leader and the timeline of the user is naturally ordered by time, this problem becomes sorting two sorted list and have the top 50 elements, which we can solve problem by 2 pointers.

What if a user unfollow a leader? In this case, it looks like we need to reassemble the timeline. However, similar with how we deal with following a new leader, this will become a sorting n sorted list and have the top 50 elements, which we can finish that by using n pointers.

Since this kind of push does not need to be done with users' waiting, we put all of these actions to a message queue and done it by another microservices.

When it comes to no timeline on redis, we are using the strategy we have for unfollowing a leader, but since we are now actively grasping the tweets when reading, we call this pull model.