Nano Twitter

Ye Hong, Limian Guo, Chenfeng Fan
Brandeis University, Computer Science Department

Architecture

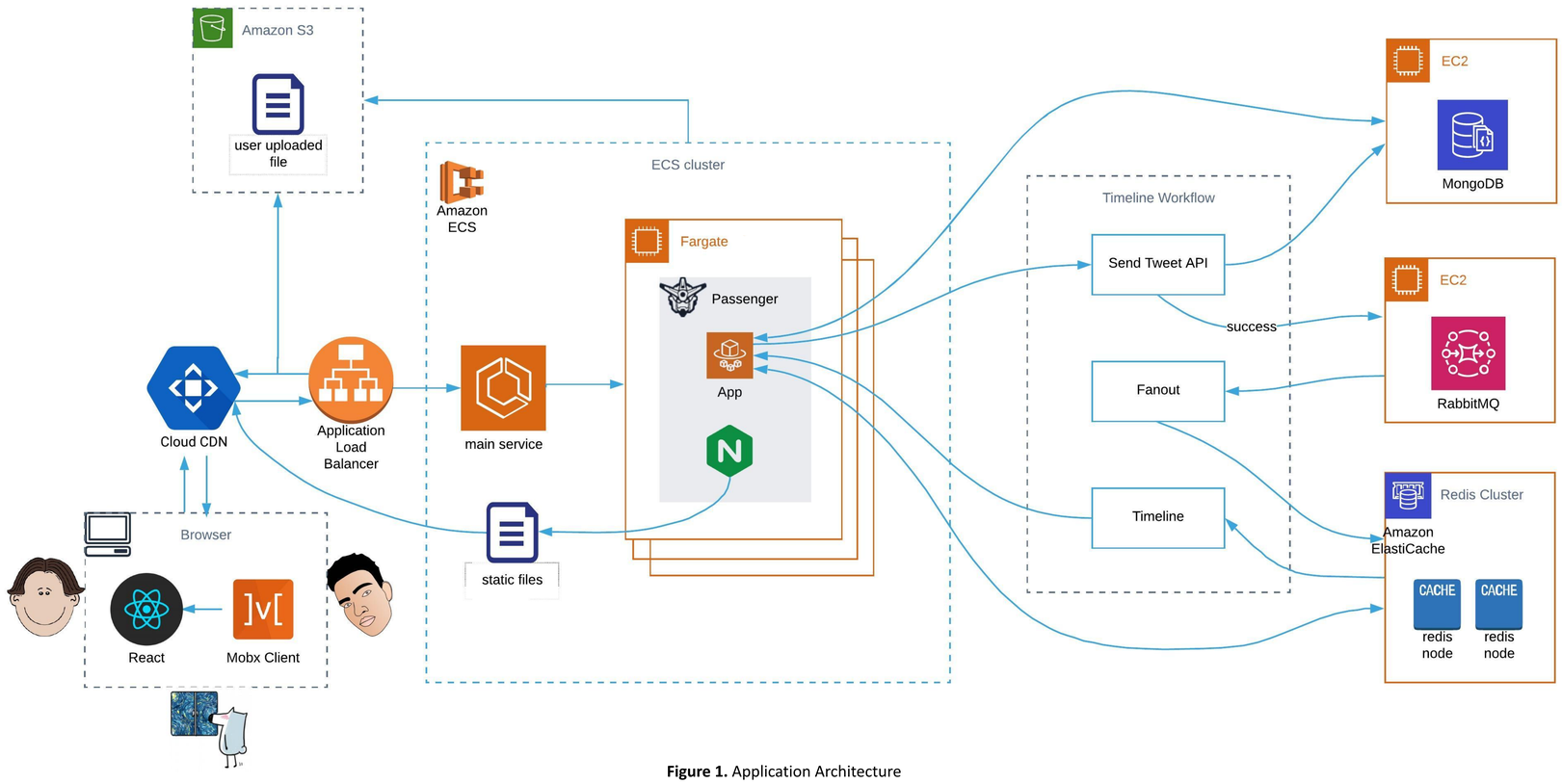

Figure 1. Application Architecture

Loader.io Test

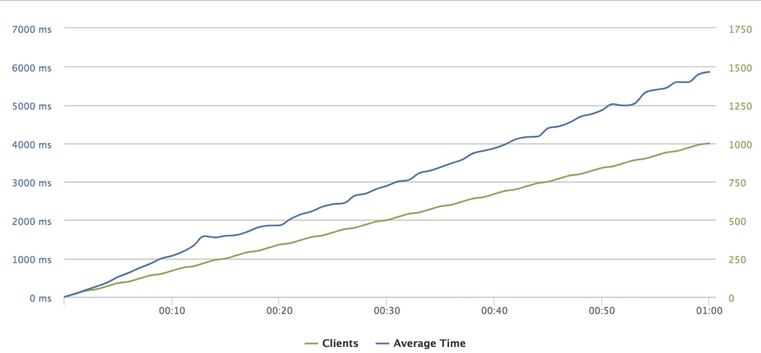

Chart 1. Loader.io Test 0 – 1000 Clients Over 1 Min

Clients Number	0 - 500	0 – 1000	0 – 1250
Average Response Times	1552ms	3051ms	3597ms
Success Counts	9255	8937	9265
Error Counts	0	3	0
Max CPU Utilization	>100%	>100%	>100%
Average Memory Utilization	22%	24.3%	26.6%

Chart 2. Loader.io Test Statistics Over 1 Min

Loader.io Test Result

Our back-end server clusters can support at least 150 requests per seconds. Under the pressure of 0-1000 clients in one minute, it can have more than 9000 successful responses and no error at all. The average response time is around 3000ms.

Since all of the tests' max CPU utilization exceeds 100%, it's highly possible that CPU is one of our biggest bottleneck.

Abstract

Our Nano Twitter supports the basic functions of twitter, such as tweeting, commenting, retweeting, following, listing the tweets of the people you follow when visiting the homepage (timeline), and accessing a person's profile. Our product front-end was built on React, it is very user-friendly and can quickly respond to users' operations. We also used MobX for state management.

Once a person posts a tweet, the people who are following him will have the tweet displayed on their home page. Comments, likes and the number of retweets will also be updated real time.

We extensively used Redis to cache data when necessary. However, we did this progressively to make sure the newest changes of the business can be reflected on user interface.

Optimization

We only cached timelines for users who often log in. Building a timeline from scratch is very costly, so we used a Redis as a global lock to make sure only one server is building the timeline for a certain user.

We modified our business logic to make use of bulk operation and atomic operation supported by MongoDB and Redis, this will lead to quicker respond and lower bandwidth usage.

Redis and MongoDB can be sharded quite easily. By choosing them, our architecture has better horizontal scalability. We store the follower IDs inside the user documents in our MongoDB. As we can store them together, we could avoid the cost of remote join which usually becomes a problem when you build a distributed system with RDBMS.

We serve our static file through CDN and use Nginx to maintain unhandled connections. When the load suddenly gets huge, CDN, load balancer and Nginx can help our system to hold the connection and let our app server handle them. This is our key to avoiding error during load testing.

Contact Information

Ye Hong: yehong@brandeis.edu
Limian Guo: limianguo@brandeis.edu
Chenfeng Fan: fanc@brandeis.edu

References

1. React: https://reactjs.org/docs/getting-started.html
2. MobX: https://mobx.js.org/getting-started.html
3. Redis: https://redis.io/commands
4. RabbitMQ: https://www.rabbitmq.com/getstarted.html
5. Mongoid: https://docs.mongodb.com/mongoid/current/
6. Passenger: https://www.phusionpassenger.com/docs/tutorials/what_is_passenger/
7. AWS ECS: https://aws.amazon.com/ecs/getting-started/