NT

Nanotwitter - Team Jia You

Aaron Gold, Julian Ho, Xiangran Zhao
COSI 105b: Software Engineering for Scalability, Pito Salas

@ Team Jia You

Overview

Our app server was programmed using Sinatra, which is a domain-specific language for building websites, web services and web applications in Ruby. Unlike other alternative Ruby web application frameworks such as Ruby on Rails, Sinatra is lightweight and focuses on a minimalistic approach to development. API calls to our nanoTwitter app server will initiate HTTP requests immediately.

Though designed with canonic Model-View-Controller (MVC) architecture for our app server, we transferred it to View-Controller (VC) because of the database we adopted, Dgraph. Dgraph is a distributed, low-latency, high throughput graph database, which outcompetes other databases in many aspects. It requires zero or little schema hacks, it's easier to write complicated queries in GraphQL+ than in SQL, and it connects entities via edges but not primary/foreign keys. As a social networking website, nanoTwitter requires for many complex joins among tables. We chose Dgraph as our database server because it avoids multiple joins among multiple tables. Instead, Dgraph collects all the data needed with recursive data retrieval, ranking and sorting, all in one query. Recursive data retrieval is essentially an edge traversal starting from a node, which is relatively cheap to Dgraph. Dgraph minimizes the number of network calls required to execute queries. The number of intermediate results won't increase the number of network calls.

Functionality & UI

Functionalities and Frontend Design
Functionalities of our nanoTwitter app:

- User sign up/sign in and sign out
- Follow/unfollow other users
- Post tweets
- Like and comment tweets
- Retweet
- Search
- View profile, timeline and trending tweets

All the functionalities listed above are achieved with Dgraph. New entities and/or edges are created via mutation on database, and one query on either profile or timeline is retrieved with one traversal of a node's edges in the graph. We also adopted Dgraph's own full-text search feature to achieve our search function.

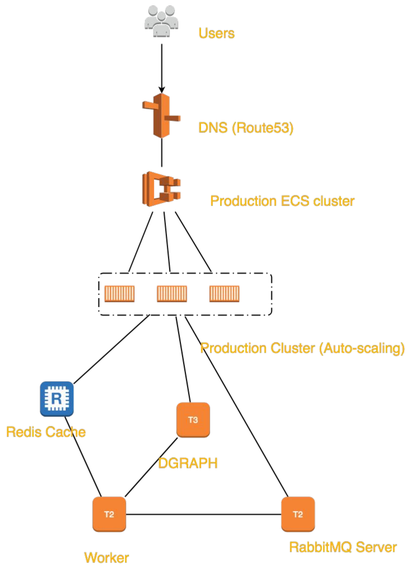
Bootstrap 4 is what we used for our app UI design. Bootstrap-Modal component is used for creating new tweets, retweeting and commenting. To avoid the network latency when tweeting, retweeting or commenting, these functionalities are achieved using AJAX.

Architecture

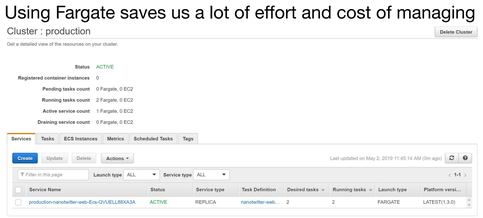
ECS/Fargate Cluster

In order to achieve high scalability while maintaining low running cost, we chose to build our app on AWS's Elastic Container Service (ECS). ECS leverages docker's container technology which allows us to quickly deploy, scale up, and scale down.

To host the containers, we chose to use AWS Fargate over a regular EC2 instance for even more flexible deployment. AWS Fargate is a service similar to EC2 which we can deploy and run our app on. The difference is that Fargate provide us the flexibility of deploying containers other than deploying individual machines that run containers. AWS will also manage the underlying systems that runs your container for you.



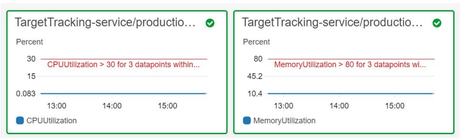
Using Fargate saves us a lot of effort and cost of managing



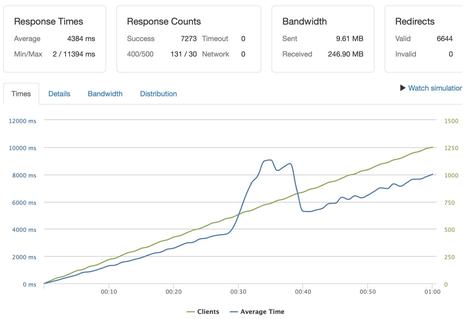
Auto-scaling cluster group

With the flexibility of ECS and Fargate, we are able to set up auto scaling that monitors our app performance and scale up/down accordingly. We have set up many scaling policies that will scale our app based on CPU/Memory usage, and traffic throughout.

Currently, our cluster is configured to have 2 Fargate containers running under normal load. It will scale up every 30 seconds if there is constant heavy load (e.g. Loader.io load testing).



Performance



Elastic Load Balancer

To balance our network traffic between multiple app services, we uses Elastic load balancer that does a round-robin network load distribution to our ECS cluster. It also automatically recognize newly created containers (from auto-scaling) and distribute traffic to it.

Combined with health checks that ELB provides, we were able to keep our app up time to close to perfect.

Worker: RabbitMQ

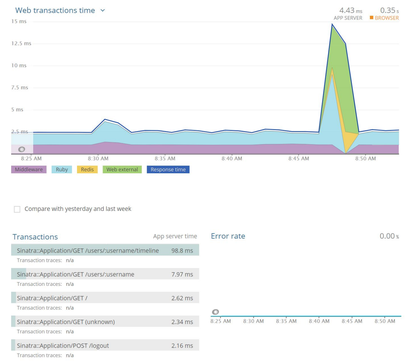
One of the most taxing and thus difficult features for the scaling of our Nanotwitter app is the action of a user tweeting a new tweet. When an individual tweets, we must not only update the database for the new tweet (new hashtags, retweet count, etc), but we must also update the timelines of every follower. Each individual's timeline is cached in a redis server. Therefore, we must update the caching server in order to limit database queries. Thus, we developed a worker to process and update all follower's timelines every time a user tweets. When a user tweets, the content of the tweet is sent as JSON to a RabbitMQ queue (the monolith is the producer). The RabbitMQ server, is a micro t2 instance on EC2 and automatically sends the new tweet information to any queue subscribers. Therefore, we have two queue subscribers (both on EC2 with a load balancer) whose main purpose is to complete the intensive process of updating timelines (aka fanning out tweets). The worker first adds the new tweet to the database. Next, the worker queries the database and finds all of the user's followers. It then expires all of the redis keys for both the user and his followers and then queries the database for their newly updated timeline information. From their, it adds these new timelines back to redis. This ensures that timelines are updated with new tweets spontaneously than with delay.

Redis Caching

In order to limit queries to the database, and increase scalability and speed for our app, we implemented Redis caching through Amazon ElastiCache. In general, the keys in the cache are as "username:profile" and "username:timeline". These keys map to JSON of all of the user's tweets/their followers tweets. Our main method, "Dgraph_or_Redis" first checks redis for the requested information, and if the key does not exifst, then it queries Dgraph and updates Redis. Redis is always up to date thanks to the background worker.

Monitoring

We use New Relic to monitor the performance of our app. It provides various useful metrics like latency, network load, and error rate. This is the first tool we use for debugging app bugs and unexpected crashes.



nanotwitter.com