https://github.com/Gaupeng/RideShare

17CS352:Cloud Computing
Class Project: Rideshare

CC_0088_0144_0667_1103

Date of Evaluation: 11/05/20
Evaluator(s): Nishita & Vinay
Submission ID: 498
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Anantharam R U	PES1201700088	H
2	Gaurang Rao	PES1201701103	H
3	Shashank Prasad	PES1201700667	H
4	Tanay Gangey	PES1201700144	F

Introduction

- Rideshare is a Database as a Service (DBaaS) that helps users create and share rides. This project was done as a part of our University's course project under the subject of Cloud Computing. The application is fault tolerant,highly available and scalable based on the number of concurrent users in real time.

- Some of the functionalities we have implemented are adding/modifying users, creating new rides, joining existing rides and listing upcoming and current ride details. We used multiple AWS Ec2 instances to host the app and used Postman as a frontend to make requests and receive feedback on the same.

- Technology Used:

    - Flask - Our application ran on Flask. This included the rides microservice, the users microservice and the orchestrator.
    - PostgreSQL - Our primary database used was PostgreSQL. Sometimes, we used files to keep track of counts persistently.
    - RabbitMQ - RabbitMQ acted as our message-broker through the different components of the application.
    - Apache ZooKeeper - ZooKeeper was used to create and maintain working entities in the form of nodes.

Related work

RabbitMQ: https://www.rabbitmq.com/getstarted.html

Zookeeper: http://zookeeper.apache.org/

RabbitMQ Container Image: https://hub.docker.com/_/rabbitmq/

Python binding for zookeeper: https://kazoo.readthedocs.io/en/latest/

Docker sdk to start containers programmatically:
https://docker-py.readthedocs.io/en/stable/

Zookeeper:
https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php

AMQP: https://www.rabbitmq.com/tutorials/amqp-concepts.html

RabbitMQ Channels: https://www.rabbitmq.com/channels.html

Zookeeper Container Image: https://hub.docker.com/_/zookeeper/

ALGORITHM/DESIGN

- We had three instances running:

    a. The Users micro-service.

    b. The Rides micro-service.

    c. The Orchestrator.

- Any incoming request would reach the Application Load Balancer, where based on target group rules, would be redirected to either the Users or Rides micro-service.
- Upon requiring the usage of a database on one of these micro-services, the request would be forwarded to the Orchestrator from these micro-services.
- The orchestrator maintained workers, in the form of a single master worker, and multiple slave workers. Each worker maintains its own copy of the database.
- We used Object Oriented Programming in python with SQLAlchemy to set up and use these aforementioned databases
- We also used classes to define and publish to the RabbitMQ queues.
- On receiving a write request, the orchestrator would write to a RabbitMQ queue, called the 'writeQ'.
- The master worker is responsible for all writes to the database, and hence has to ensure that the slaves keep their database in sync with itself. After writing to its database, the master would write to a RabbitMQ exchange, to notify all slaves to update their database.
- The slaves on receiving this message to sync their database with the master, would update their copy of the database with the new values.
- Any read requests to the orchestrator would be processed by the slave workers in a round-robin fashion, after the orchestrator publishes the message to a RabbitMQ queue 'readQ'.
- Thus, any response, whether write or read, would at the end, be acknowledged at the orchestrator, when a slave confirms its actions.
- If there were more than a threshold number of reads in the past two minutes, we add new slaves to assuage this influx of requests and use an internal API in the orchestrator to make sure their database is up to date so as to provide consistency.
- Zookeeper was used to keep track of the status of each worker, including master and the slaves using znodes. We used a single root node with all workers as children nodes. The slaves were named in incrementing order based on the process id (and indirectly, the time of creation), with the master having “master” as its name and data value. A “Watch” was kept on each slave worker to keep track of failures that may occur. In the occurrence of a failure, a new slave worker is created and database synced with the master to perform reads in place of the failed slave container.

TESTING

One of the first challenges we faced was that it took us a while to realize that the slaves were connecting to the same DB container which led to Integrity Error when we were adding data to our database while testing. We fixed this by creating a new postgres container for every slave.

One major challenge was looking at the docker-logs for every little bug. One bug that took us around 3-4 days to debug was because of this issue. When we did 'docker-compose up' our services/containers in the docker-compose file had a status of 'running' but when we checked the status of one of our dynamically created containers, it was stuck at 'created'. Seeing this as a problem, we would do 'ctrl+C' and see the log of the new container and see the error - "pika.exceptions.StreamLostError". We were absolutely bamboozled for four days trying to change and fix our rabbitMQ queues and whatnot when it suddenly struck us that we should see docker logs while the container is running so as to see at what point the error gets thrown - to our surprise - it never did. Turns out the error was being thrown when we did ctrl+C because the queue lost connection due to the process being killed.

Another challenge we faced was during the master election. Earlier our new containers had random names, but for the purpose of master election, we named them 'worker_worker_'+<slave count>. However when we were scaling up, the container was being created much slower than the part of the code that incremented <slave_count> which was used as the node name so we kept getting zookeeperNodeExists error because the count had incremented much before the container had been created.

We did not face any issues at the time of automated submission

CHALLENGES

- Zookeeper Master Election
- Dockers

Contributions

Anantharam R U - Dockers, Zookeeper, Reverse Proxy

Gaurang Rao - API's, RabbitMQ, Setting up Microservices

Shashank Prasad - Zookeeper, Dockers, Documentation and Code cleaning

Tanay Gangey - RabbitMQ, APIs, Setting up Load Balancer and instances

CHECKLIST

SNo	Item	Status
1.	Source code documented	Completed
2.	Source code uploaded to private github repository	Completed
3.	Instructions for building and running the code. Your code must be usable out of the box.	Completed