17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 23-05-2020
Evaluator(s): Nishitha R and Vinay S Banakar
Submission ID: 1379
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1.	Anand Singhania	PES1201700130	6D
2.	Aniket Anand	PES1201700185	6H
3.	Aditya Manuraj	PES1201700252	6H
4.	Harsh Choudhary	PES1201700279	6H

Introduction

- A fault tolerant, highly available Database as a Service (DBaaS) designed for a RideShare application working on AWS EC2 instances. It includes three separate instances for Users, Rides and DBaaS Orchestrator.

- A load balancer is used to manage application traffic between Users and Rides container. In general, an incoming request first reaches the load balancer and is then forwarded to either Users or Rides container according to the keyword provides in the URL (api/v1/users or otherwise). Users and Rides container process this request and responds accordingly.

- There is a third server which is used for DBaaS. Users and Rides server communicate with this third server for Database queries. This provides a common database for Users and Rides. This DBaaS provides several functionalities such as reliability, availability and scalability. All the communication takes place using APIs.

Related work

Docker: https://docs.docker.com/engine/docker-overview/

Aws Load Balancer:
https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction. html

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

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

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

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

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

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

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

ALGORITHM/DESIGN

Design of the Zookeeper:

We are using the Kazoo Python library designed to make working with zookeeper a more hassle-free experience that is less prone to errors.

We created a container using the existing image of the zookeeper from the docker hub. Whenever the container is created dynamically a node is created as well. We are setting the name of that particular node as the PID of that container. We have implemented the fault tolerance for slaves i.e. when the slave is crashed/killed, a new slave is spawned through a watch event. The watch event is implemented by calling the @zk.ChildrenWatch decorator which gets triggered automatically and the new slave is spawned.

Design of the system:

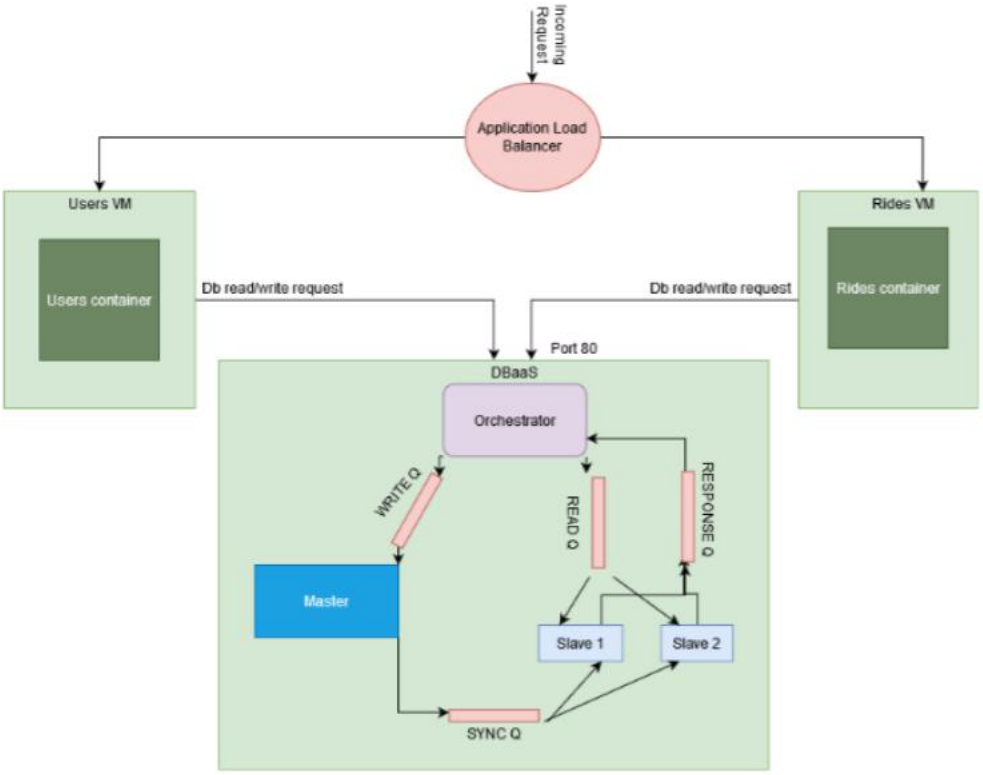
When the incoming requests are sent through the Load balancer, it further redirects it to the User or the Ride microservice based on the request sent by the User. The microservice then accepts these requests and forwards it to the orchestrator (via port 80) for db read and write operations. The Orchestrator then accepts these requests and sends it to the

1) Master through WriteQ if it's a db write/clear_db operation or

2) Slave through ReadQ if it's a db read operation.

The Master accepts the write requests and updates the master database. After the writing operation is successful, the same requests are then forwarded to slaves through SyncQ to update the slaves' databases to make sure they are eventually consistent. This happens through the fanout exchange.

The Slave accepts the read requests through ReadQ and performs the read operation in a round-robin fashion between the slaves. The details requested by the ReadQ are sent back to the orchestrator through the ResponseQ.



TESTING

1. Load balancer issues – The format of response expected by user and ride target group was not matching with the response of orchestrator which caused internal error.

CHALLENGES

1. Consumers i.e. Master and slaves were not able to consume from queues. It was solved by closing the connections to Rabbitmq.

2. Creating individual database for all the workers – This was solved by creating database with the names according to the PIDs of the containers.

3. Implementing Clear database API – It was solved by sending 'clear' as the body to the writeQ and checking if body='clear' and then calling clear function to clear the database.

Contributions

Anand Singhania: Implementation of RabbitMQ (read, write and clear database request)

Aniket Anand: Implementation of kill_slave, kill_master, number of slaves and listing PIDs of workers, implementing Load Balancer

Aditya Manuraj: Maintaining counter, calculating target slave and adjusting number of slaves accordingly wrt the specified time interval (120 s)

Harsh Choudhary: Changes in user and ride containers, implementation of zookeeper, deployment on AWS

CHECKLIST

SNo	Item	Status
1.	Source code documented	Documented
2	Source code uploaded to private GitHub repository	Uploaded
3	Instructions for building and running the code. Your code must be usable out of the box.	Readme.md uploaded with complete instructions.