17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 18-05-2020
Evaluator(s): Usha Devi BG and Sanjith
Submission ID: 374
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Suryanarayan N	PES1201700094	6 'C'
2	Harikrishnan V	PES1201700155	6 'A'
3	Rahul Pandia	PES1201700260	6 'B'
4	Kruthik JT	PES1201701509	6 'B'

Introduction

A cloud based RideShare application , that can be used to pool rides.

The features provided by the application are:

- Adding a new user
- Delete an existing user
- Creating a new Ride
- Search for an existing ride between a source and a destination
- Join an existing ride
- Delete a ride

Related work

Docker:

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

RabbitMQ:

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

Docker-SDK:

https://docker-py.readthedocs.io/en/stable/

ALGORITHM/DESIGN

The design is as follows:

- The orchestrator will listen to incoming requests on port 80 and will publish the "Read" message into the "ReadQ" and the "Write" message into the "WriteQ". It spawns new (or) deletes worker containers based on the number of requests received in the 2-minute time gap and the API that was called (like /api/v1/crash/slave (or) /api/v1/crash/master).

● The master worker will listen to the “writeQ” and picks up all the write requests in the “writeQ” and write them to a persistent database, while it also writes these messages to the “syncQ” and an exchange called “fan” which forwards it to every existing slave to write into them for consistency.

● The slave worker will be listening to the readQ and picks up all the read requests in the “readQ” queue. Upon querying the database based on the request, the slave will write the response to the “responseQ”.

● Multiple instances of the worker are created such that the messages from the readQ are picked up by the slave workers in a round robin fashion.

● Whenever a slave is created, it initially connects to the “syncQ” to execute all the requests that were received to the “writeQ” (master) without sending back an acknowledge which makes the state of every request in the “syncQ” as “Not Acknowledged”. We then close the connection between the “syncQ” and the new slave which then changes the state of all the requests back to “ready” state for consumption again.

● DBaaS must be highly available, hence all the workers will be “watched” for failure by Zookeeper, a cluster coordination service. Every worker initially created a “znode” and is given a “znode-data” called “slave”. The slave is made to watch this “znode-data”. If the slave is of the lowest PID, and if it must be made the master, the “znode-data” is changed to “master”. The “watcher” then identifies this change and make the slave run as the master. The orchestrator watches over every “znode”, and for any event, it checks if a “znode” died, in this case, a new slave is spawned and appropriate changes are made based on whether the “znode” that died it a master or not.

TESTING

The testing challenges were:

- Maintaining and checking correlation between test requests and auto-scaled containers.
- Zookeeper leader election triggers nodes for all kinds of events, but we needed to differentiate between creation and deletion events.

How we fixed the issues on automated submission:

- Had issues with using count manager and files, hence we used global variables instead.
- We used many “print” statements to debug the errors.

CHALLENGES

- To bring up new containers from another container using Docker-SDK.
- To implement Master Crash

Contributions

Everyone sat together on a video call the entire time to finish the project. Hence there was an equal contribution from every member of the team.

CHECKLIST

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