17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 20-05-2020
Evaluator(s): Rachana B S and Deepthi
Submission ID: 364
Automated submission score: 10

SNo	Name	USN	Class/Section
1.	Aniket Hegde	PES1201700104	E
2.	Saransh Gupta	PES1201700267	E
3.	Ruthu G S	PES1201700856	E
4.	Shamitha K	PES1201700973	E

Introduction

In this project we have developed the backend for a cloud based RideShare application, that can be used to pool rides. The RideShare application allows the users to create a new ride if they are travelling from point A to point B.

The RideShare application has the following features:

- Add a new user
- Delete an existing user
- Create a new ride
- Search for an existing ride between a source and a destination
- Join an existing ride
- Delete a ride

We have built a fault tolerant, highly available database as a service for this application using rabbitmq and zookeeper.
The entire application is deployed on AWS EC2 instances.

Related work

For creating REST APIs and SQLite database, we referred to the Flask and FlaskSQLAlchemy documentation:

- https://flask.palletsprojects.com/en/1.1.x/
- https://flask-sqlalchemy.palletsprojects.com/en/2.x/

For understanding docker we referred its documentation:

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

For AMQP (rabbitmq) we followed these tutorials:

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

For zookeeper (kazoo) we referred to its documentation:

- https://kazoo.readthedocs.io/en/latest/

ALGORITHM/DESIGN

The project is set up on 3 AWS EC2 instances:

- Users instance
  It contains the users container which has all the users APIs for ex. add user, view users etc.
  It gets the request from the user and redirects it to the orchestrator APIs.
- Rides instance
  It contains the rides container which has all the rides APIs for ex. add user, view users etc.
  It gets the request from the user and redirects it to the orchestrator APIs.
- Orchestrator instance
  It contains the following containers:
  - orchestrator
    It has database read/write APIs, crash slave and container list APIs, auto scaling mechanism, zookeeper functionality and docker-sdk (spawning containers dynamically).
  - master
    It performs the database write operations.
  - slave
    It performs the database read operations.
  - shared_db
    We are using this to make the database persistent and sharable among all slaves.
  - rabbitmq
    It provides AMQP functionality.
  - zookeeper
    It provides high availability by keeping watch on the '/Workers/' path which has the z-nodes of the slave containers.

TESTING

- During testing one challenge we were facing was incorrect requests count. By going through the terminal requests, we figured out that we were not taking care of the **405** Method Not Allowed case. We solved this by adding another API for that.
- Another challenge we faced during testing was Auto Scaling failed. This was because an extra container was getting spawned each time. So, we solved this by limiting container spawning in only the orchestrator.

CHALLENGES

- One main challenge we were facing was to make the database persistent and also sharable among all the slaves. We implemented this by using docker volumes and creating another container shared_db which synchronizes the master db with the shared db.
- Another challenge we faced was to give unique name to each z-node in the slave. For this we used socker.hostname() which gives the container id of that particular slave.

Contributions

- Aniket
  Auto scaling functions and shared_db mechanism.
- Ruthu
  Crash slave and container list APIs.
- Saransh
  AMQP (rabbitmq) in orchestrator, master, slave, shared_db.
- Shamitha
  Zookeeper funtions and read/write APIs in orchestrator.

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