17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation:
Evaluator(s):
Submission ID: 1213
Automated submission score: 9/10

SNo	Name	USN	Class/Section
1	Adithya Vardhan	PES1201700788	6 C
2	Malavikka Rajmohan	PES1201700794	6 C
3	Daksha Singhal	PES1201700847	6 C

Introduction

Ride Share is an application built using REST flask APIs and deployed on Amazon's AWS EC2 instance with the help of container. It has various functionalities such as creating a new user, creating a new ride, merging rides with users, deleting users. Database requests are handled by an orchestrator, which employs aa master node to write into the database and slave node(s)to read from the database. The orchestrator supports various features such as data consistency, scaling up, fault tolerance.

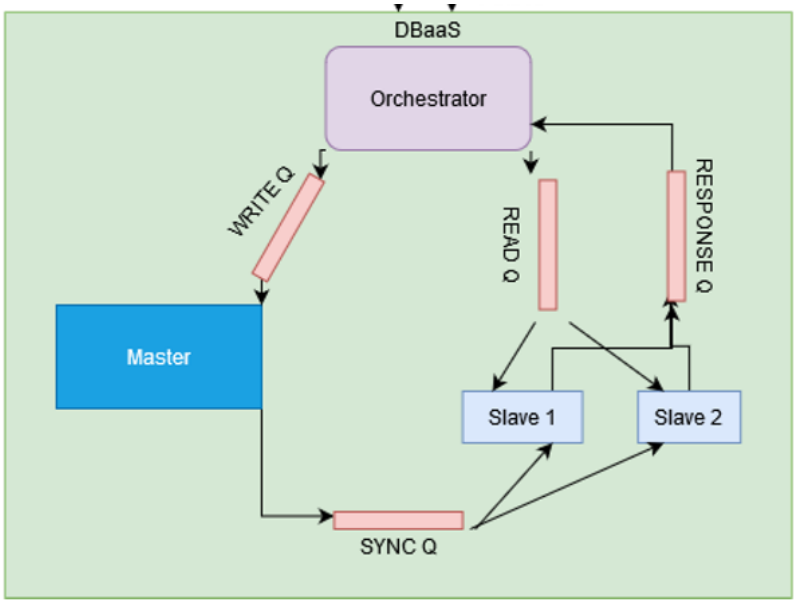
Related work

- https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- https://docs.docker.com/install/linux/docker-ce/ubuntu/#install-docker-ce
- https://hackernoon.com/what-is-amazon-elastic-load-balancer-elb-16cdcedbd485
- https://www.rabbitmq.com/getstarted.html
- https://www.rabbitmq.com/tutorials/amqp-concepts.html
- https://hub.docker.com/_/zookeeper/
- https://hub.docker.com/_/rabbitmq/
- https://kazoo.readthedocs.io/en/latest/
- https://docker-py.readthedocs.io/en/stable/

ALGORITHM/DESIGN

- 3 VMs were used for rides, users and orchestrator respectively
- Rides and user VMs were connected through a load balancer
- Any database-related requests(read/write) were redirected to the orchestrator VM
- The database used in this project was SQLite
- The queues were maintained by RabbitMQ containers
- The fault tolerance was handled by zookeeper
- Write requests were sent to a master container through a writeQ and read requests were sent to slave(s) containers(depending on the number of requests) through a readQ
- Data consistency was achieved by using a syncQ which updated changes in all the slaves as and when it was changed in the master
- RabbitMQ Setup
    - 4 queues were implemented namely: ReadQ, WriteQ, SyncQ and ResponseQ.
    - **ReadQ** : All read requests from orchestrator were published to one of the slaves using the exchange type called fanout which broadcasts all the messages it receives to all the queues it knows in a round-robin fashion.

- **ResponseQ** : The slave worker is responsible for responding to all the read requests coming to the DBaaS orchestrator and sending it back to the orchestrator from the ResponseQ.
- **WriteQ** : All write requests from orchestrator were published to the only master. The master listening on the WriteQ is responsible for writing on to the database.
- **SyncQ:** In order to implement eventual consistency the master will write the new DB writes on the "syncQ" after every write that master does, which will be picked up by the slaves to update their copy of the database to the lastest.



- Zookeeper setup –
  - The principal purpose of a zookeeper was to maintain a watch over the heartbeat of each container.
  - this was achieved by using the client.ChildrenWatch function of the kazoo library. Whenever a child node of the path "orchestrator" was created or deleted, the ChildrenWatch function was called and at that point, the number of children nodes currently running against the number of nodes that are supposed to be running were checked.
  - If there was a deficit, a new child node was created.
  - The child node was then assigned a PID and a slave.
  - The PID was later added to the list of child nodes.

TESTING

- Challenge: Getting no response on the terminal when sending the request and hence not knowing whether the error was from user or orchestrator instance.
  Solution: Putting flag variables in appropriate areas of the code to debug
- Challenge: zookeeper took time to update its children so had to wait before sending a list of workers requests
  Solution: Found difficulty in fixing this

CHALLENGES

Setting up the SyncQ was a challenging task as atomicity of the database had to be taken care of. Hence, individual threads were used in the slave for the read and sync queues.

Contributions

- Daksha Singhal – Handled the setting up of the RabbitMQ queues (read, write, sync, response)
- Malavikka Rajmohan – Handled scalability by keeping track of the count of incoming requests and increasing the number of slaves using dockerSDK
- Adithya Vardhan – Handled fault tolerance using zookeeper by creating new slaves every time the slave crashed

CHECKLIST

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