UE17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 16/05/2020
Evaluator(s): Prof. Srinivas KS
Submission ID: 1136
Automated submission score: 0.0

SNo	Name	USN	Class/Section
1.	Abhijeet Murthy	PES1201700139	C
2.	Sakshi Goel	PES1201700148	B
3.	Sanjana Shekar	PES1201700905	B
4.	Prajwala D	PES1201701736	D

Introduction

The objective of the project is to build a Rideshare application with a mini DBaaS system.

Database-as-a-service is a database service that typically runs on a cloud architecture. Key characteristics are:

1. Self-service
2. On-demand
3. Dynamic
4. Security
5. Automation

Following were the expected functionalities that need to be implemented:

1. Queues for message passing
2. Eventual consistency handled
3. High availability and fault-tolerance
4. Scaling
5. Orchestrator, worker set up
6. Rabbitmq, zookeeper usage

Related work

- About DBaaS: http://www.vlss-llc.com/blog/what-is-database-as-a-service-dbaas
- Basics of kazoo: https://kazoo.readthedocs.io/en/latest/basic_usage.html
- Zookeeper Container Image: https://hub.docker.com/_/zookeeper/
- Python binding for zookeeper: https://kazoo.readthedocs.io/en/latest/

ALGORITHM/DESIGN

Step 1 : Orchestrator-message passing

With respect to the previously done assignments, now the database will be created in the orchestrator container instead of the rides and users container, and all the read and write requests will be forwarded to the orchestrator which will perform the necessary action. A queue protocol is used to forward the messages received by the orchestrator. The queues are built through Rabbitmq.

Step 2 : Queues

There are four types of queues - readQ, writeQ, responseQ and syncQ. These queues transfer the message to the workers which are of 2 kinds, master and slave. The master listens to messages from writeQ and slave listens to readQ. Slave then writes to the responseQ which is then picked up by orchestrator. A round-robin method is used by the slaves when there are multiple instances.

Step 3 : Eventual consistency handling

A problem arises when eventual consistency needs to be handled since each worker runs on its own and the replica of changes the master makes needs to be reflected on the slave databases. Hence, another queue called syncQ is set up. Here, every time the master makes a write to the database, the slaves update their copy.

step 4 : Availability

2 scenarios :

1. In case a slave fails, a new slave needs to be elected to start its process.
2. In case a master fails, an existing slave with the lowest pid container will take over and a new slave will be created.

This is done using zookeeper's leader election. The data to the new slave is copied asynchronously.

step 5 : Scalability

Here, the number of workers to be generated is based on the number of http requests that are incoming. The count of the number of slave containers is first initialized to one. It will increase or decrease based on the requests and is set to zero after 2 minutes.

step 6 : Execution

Docker containers on all 3 AWS instances are run using the following commands :

```
sudo docker-compose build
sudo docker-compose up
```

Testing is done using postman.

TESTING

- Load balancer testing on postman was difficult as it wasn't getting set up properly.
- On the submission portal, the IP address of the load balancer should not contain 'http://'. It was changed after one submission.

- The load balancer wasn't working due to issues related to security groups and availability zones. We added additional availability zones and corrected the security groups.

CHALLENGES

1. The implementation of queues in rabbitmq was challenging.
2. We also faced problems regarding ports being already in use while running the command 'sudo docker-compose up'. Had to use netstat and kill to remove any existing processes.
3. Since our load balancer still wasn't working even after setting it up multiple times, we lost 10 marks during evaluation. We faced difficulty in setting it up.

Contributions

Assignment 1:

Sakshi - Creating read and write APIs, Nginx and gunicorn set up

Sanjana - Add, delete and display user and rides api, Nginx and gunicorn set up

Prajwala - List existing ride based on rideid, creating db, Nginx and gunicorn set up

Assignment 2:

Sakshi - Dockerfile and docker-compose.yml creation and docker set up on the instance

Sanjana - Adding clear db api and list user api.

Prajwala - Routing of db read and write to user db and ride db accordingly

Assignment 3:

Sakshi - new instance set up, testing and running, creation of load-balancer

Sanjana - count request APIs, creation of load-balancer

Prajwala - docker set up on 2 instances, creation of load-balancer

Abhijeet(joined new) - helping with load-balancer creation

Project:

Master/slave worker creation + orchestrator set up - Sanjana, Sakshi

Queues and containers set up - Prajwala, Abhijeet

Setting up rabbitmq - All 4

Load-balancer set up - All 4

Zookeeper - All 4 [didn't work as expected]

Scaling - All 4 [apis created didn't work as expected]

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 (added as readme file on github repository)