17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation:
Evaluator(s):
Submission ID: 178
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Aiswarya Y B	PES1201701271	H
2	Varun Venkatesh	PES1201701403	C
3	Suhail Rahman	PES1201701420	H
4	Riya Vijay	PES1201701814	H

Introduction

This project is focused on building a fault tolerant, highly available database as a service for the Rideshare application. We used Amazon EC2 instances and existing users and rides VM, their containers, and the load balancer for the application. The existing db read/write APIs are used as endpoints for this DBaaS orchestrator. We also implemented an orchestrator that works with read/write API. The users and rides microservices are using the "DBaaS service". Instead of calling the db read and write APIs on localhost, those APIs will be called on the IP address of the database orchestrator.

Orchestrators: It is an intermediate between the rides, users microservices and the workers. Upon receiving a call to the APIs, it will write relevant messages to the relevant queues.

Rides Microservice: This microservice deals with creation of rides, deletion and listing all the rides between a source and destination, the count and the details of a given ride.

Users Microservice: This microservice deals with creation of users, deletion and listing all the users.

Queues using RabbitMQ: There are 4 Different Queues i.e. Write Queue, Sync Queue, Read and Response Queue.

Related work

https://www.tutorialspoint.com/python/python_multithreading.htm

https://stackoverflow.com/questions/10525185/python-threading-how-do-i-lock-a-thread

https://github.com/marshmallow-code/flask-marshmallow/issues/56

https://marshmallow.readthedocs.io/en/stable/quickstart.html

https://ihong5.wordpress.com/2014/06/24/znode-types-and-how-to-create-read-delete-and-write-in-zookeeper-via-zkclient/

https://www.tutorialspoint.com/zookeeper/zookeeper_fundamentals.htm

https://bowenli86.github.io/2016/07/11/distributed%20system/zookeeper/ZooKeeper-Watches/

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

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

http://zookeeper.apache.org

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

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

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

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

https://hub.docker.com/_/zookeeper/

https://hub.docker.com/_/rabbitmq/

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

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

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

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. The data of the node is set as "0" for the slave and "1" for the master to differentiate between the two. 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.

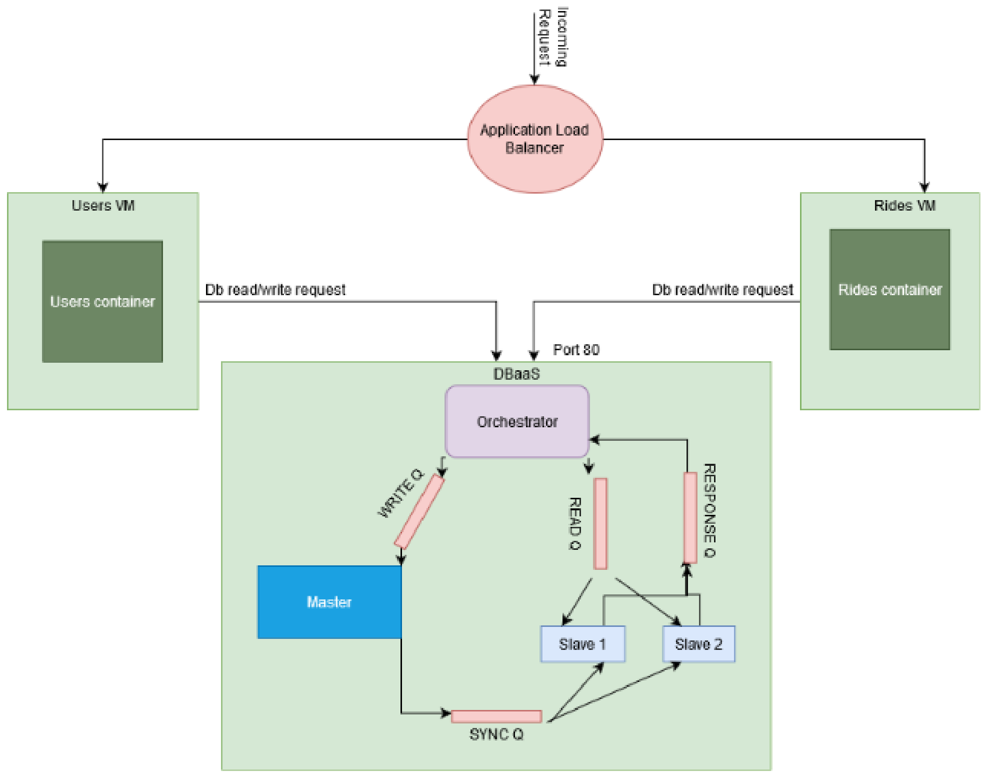
Design of the system:

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 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

Testing challenges:

1) Crashing the slave and scaling up happening at the same time - The problem arises due to the mismatch of the slave count. We fixed this by incrementing the slave count before creating new slaves.
2) ma.ModelSchema - we used this syntax to format the output according to the required specifications during the first 3 assignments. However, upon implementing the same syntax in this project, we faced a syntax error. We could not figure out this issue as the same exact syntax was used previously, and we had to use an alternative method to format the output.

Issues faced during automated submission and their solutions:

Initially, while calling the listing of rides API, we were getting a “400 bad request” and we weren’t counting this request as a Read request for scaling up. Hence, the auto scale was failing. To fix this, we considered this bad request as a Read request for scaling up/down, and while doing so, we were able to overcome the issue.

CHALLENGES

- First challenge was when we tried to implement both the Read and Sync operations to the slaves simultaneously. Initially each operation was occurring atomically. To solve this, we decided to incorporate a multi-threaded implementation.

- Second big challenge was to do with data replication and sync. Whenever we created a new slave, it needed to be consistent with all the existing slaves. Initially, the new slave was pointing to the existing database as opposed to creating a new one. We realized that this was the issue while using the "--scale" command in the terminal. When we added a new user, the data was being inserted twice into the same database. A "Unique Constraint Error" error message was being displayed. To solve this, we decided to use a different name for each database that is linked to a newly created container. For replication, we copied the contents of the master database to the newly created slave database. While this operation is happening, we made sure that all other operations are halted until this replication is completed.

- Third challenge was dealing with the mounting of volumes during the dynamic creation of containers. Docker sdk command was used for this. This initially posed to be a challenge due to the lack of proper documentation. However, we were able to solve this after referring to piazza.

- Fourth challenge was fetching the parent pids (ppid) of the containers. We found a solution to fetch these ppids using the "pgrep" command, but unfortunately, we did not get the desired result. The output was an empty list instead of a list of ppids. We fixed this issue by fetching the pids of the containers using the "docker inspect" command.

- Another small challenge we faced was trying to integrate the master and slave program. To solve this, we used sys argument to identify whether it should run a master or slave program.

Contributions

Aiswarya Y B: Sync Queues, scale in and out.

Varun Venkatesh: Read and Response Queues. Getting pids of the container.

Suhail Rahman: Write Queues, Zookeeper-fault tolerance for slaves.

Riya Vijay: Data replication and sync, 3 API's.

CHECKLIST

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