17CS352:Cloud Computing

Class Project: Rideshare

Building a fault tolerant, highly available database as a service for the RideShare application

Date of Evaluation:
Evaluator(s):
Submission ID: 161
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Mithali Shashidhar	PES1201700190	F
2	Sharanya Venkat	PES1201700218	F
3	Aishwarya M A Ramanath	PES1201700872	F

Introduction

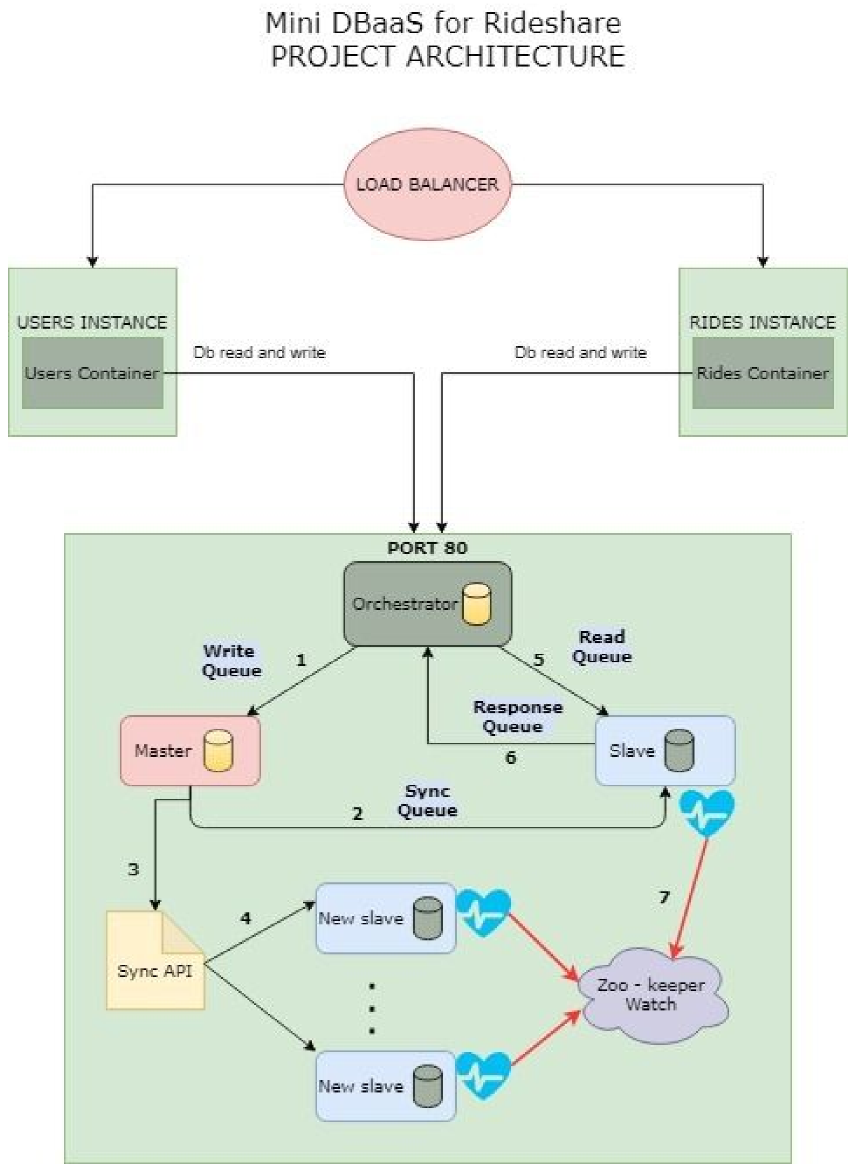
This project is a cloud-based application deployed on AWS that simulates the backend of a ridesharing application that can be used to pool rides. We used technologies such as docker, rabbitmq, zookeeper and amazon web services to implement the same. The various services provided by this application include: adding a user, creating a ride, joining a ride and others. These services were implemented with REST apis and the information regarding the users and rides was stored in a sqlite database.

The specific focus of this project was to design a fault tolerant and highly available database service for this application. A database orchestrator listens to requests from the rides and users microservices and with the help of master and slave containers, performs database operations.

Related work

- 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://docker-py.readthedocs.io/en/stable/api.html
- https://stackoverflow.com/questions/39125064/how-to-use-kazoo-client-for-leader-election
- https://corejavaguru.com/bigdata/zookeeper/leader-election
- https://kazoo.readthedocs.io/en/latest/api/client.html#kazoo.client.KazooClient.__init__

ALGORITHM/DESIGN



The above diagram is the design of how we were able to implement a fault tolerant and highly available database service with the creation of a database orchestrator. All requests to access the database from either of the two microservices are now routed through an orchestrator engine, which further ensures that these operations are successfully executed.

The various arrows in the aforementioned design schema can be elaborated as follows:

1. All write requests from either of the microservices are published to the master worker through the write queue.
2. In order to implement the "eventual consistency" model, the sync queue is implemented to ensure data consistency among the workers present at a given time.
3. When the master is written to, the universal table for syncing new slaves is also being written to, so that when the sync api is called, it can access the latest, updated data.
4. On bringing up new slaves, they call the sync API, which holds all the latest data that is part of the existing slaves, and updates the newly created slaves, so as to ensure data consistency.
5. All the read requests on the other hand, are published to the slave workers in a round robin fashion through the read queue.
6. The slave workers query the database based on the read request and output the result into the response queue, which is then picked up by the orchestrator.
7. DBaaS has to be highly available, hence all the workers will be "watched" for failure by zookeeper. Zookeeper on being triggered, will bring up a new slave.

TESTING CHALLENGES

1. On calling crash-slave api, our slave would crash but then be restarted before the list workers api was called, i.e , our timing was off for the test case, we fixed this by using an app scheduler rather than the threading function we had implemented originally.
2. Each time a new slave was brought up, the syncing had to take place so it could be up-to-date with the correct data to ensure data consistency before a read request could be made; if this was happening parallely we had to ensure sync occurred before read did, we made this work by creating two threads, so that reading and syncing could occur parallely without inconsistency in results.

Other challenges:

- Documentation for docker-sdk was hard to interpret in some cases, as there weren't enough examples, and there were no corresponding resources on the internet for the same.
- Prevention of Zookeeper being called on scale down, proved to be a little challenging to figure out.

- Postgres proved to be problematic as dynamic creation of containers caused multiple port errors, causing us to shift from postgres to sqlite midway through the project.
- Mechanism for syncing between slaves.

Contributions

All team members collaborated to work on the project, with each working on smaller sub-tasks as and when required and switching between tasks when the other got stuck at any point. All project work was done over video calls with screen-sharing for easier debugging of problems, and to balance workload appropriately.

CHECKLIST

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