17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 23/05/2020
Evaluator(s): Meghana and Venkat Datta
Submission ID: 1394
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Pavan Mitra	PES1201700239	G
2	Ravichandra Gidd K	PES1201701581	G
3	Venkatesh K	PES1201701626	G
4	Ambarish D Y	PES1201701635	G

Introduction

Rideshare is a cloud-based application that hooks users with upcoming rides in their area. Built on the DBaaS(database as a service) model, it is a highly available, fault tolerant and scalable service.

Related work

We have gone through tutorials on each topic for which the links were shared along with project specifications, some of which include

- Docker, Docker files, Docker-compose, Docker SDK, Containers
- Zookeeper
- RabbitMQ
- AMQP(Advanced Message Queue Protocol)

ALGORITHM/DESIGN

We have designed a Rideshare application as mentioned in specifications. Initially we had two microservices Users and Rides running on their own EC2 instances. We had set up a load balancer for the application so that user related requests go to Users microservice and rides related requests go to Rides microservice. Both microservices had their databases in the same container in which services are running.

In this project we built a DBaas service in a separate EC2 instance and all DB related API's are called on the IP address of DBaas service. The architecture of DBaas has an Orchestrator, two main workers Master and Slave, zookeeper, rabbitMQ running in containers. Docker SDK is used to programmatically create, run, and stop the containers. AMQP(Advanced Message Queue Protocol) using RabbitMQ is used as a message broker. Queues are created using rabbitMQ's Remote procedure call (RPC) through which messages are published across master and slaves workers.

The Orchestrator is the starting point of DBaas service. It is a flask application and serves the endpoints DB read, DB write, crash master, crash slave, worker pid list. We create readQ and WriteQ queues in orchestrator using the RPC technique of rabbitMQ. Docker SDK is used to kill and spawn new workers and the put_archive() method of containers is used to copy database files from old containers to newly spawned containers.

Master worker has its own database. It serves DB write requests and also sends write requests to Slave workers through the ‘SyncQ’ queue to maintain eventual consistency of database. Slave workers serve DB read requests and update its database according to write requests from the ‘syncQ’ queue.

We use Zookeeper for leader elections and bringing up new nodes in case of their failure. Master and slave are distinguished by a flag , which is true for master and false for slave. When the master crashes ,the slave with the highest pid is brought up as a master.

Initially we connected to the ZooKeeper container which exposes the port 2181 with the KazooClient python package.

Master and slave are nodes in zookeeper. A node in zookeeper maintains a stat structure in which we add a pid field. we create and delete nodes in zookeeper using create() and delete() methods respectively. We use a watcher on children of master which is executed asynchronously whenever there is a change in children of master i.e check for updates, deletions on children of masters.

When master crashes we delete the slave with the highest pid and update the pid of the master accordingly.

The watcher will trigger ‘watcher_function’ by default whenever the slaves are modified. By which we call to spawn another slave container which replaces the crashed one. To avoid creating redundant containers either during scaling up or down, we use another flag to check whether the callee is a crashed container or not. If not, we do not spawn another slave container.

TESTING

We had to spend quality time in testing, because building the orchestrator multiple times bombarded instance memory. This was solved by removing the “docker” dir and restarting the service to regain the space.With respect to Rabbitmq, the connection to the client used to get disconnected every sixty seconds, so we had to set the heartbeat to zero.

CHALLENGES

Pika connection errors were solved using a bigger instance.

Consequently, the Load Balancer crashed. We created another load balancer which solved the issue.

Contributions

Ravichandra G: Queues, Orchestrator, Master workers

Venkatesh K: Orchestrator, Zookeeper, and testing part

Pavan Mitra : implemented syncQ and Slave worker

Ambarish D Y: Zookeeper, fault tolerance, Docker SDK

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