17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 18/05/2020
Evaluator(s): Spurthi N Anjan , Dilip Gurumurthy

Submission ID : 409
Automated submission score: 10

SNo	Name	USN	Class/Section
1	REVANTH	PES1201700201	VI/H
2	PRINCE JHA	PES1201701608	VI/F
3	VIKAS MATAM	PES1201701665	VI/F
4	SACHIN	PES1201701725	VI/H

Introduction

This project is Focused on building a fault tolerant,scalability, highly available database as a service for the RideShare application. The db read/write APIs will now be exposed by the orchestrator. The users and rides microservices will be using 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. Zookeeper is used for High availability and for Scalability orchestrator is used. Basically orchestrator is a flask application.

Orchestrator:-

The orchestrator will be a flask application. It will be serving the endpoints 'api/v1/db/read' and 'api/v1/db/write'. Upon receiving a call to these APIs it will write relevant messages to the relevant queues. From the users/rides container, you will now be calling the db read/write APIs on the DBaaS Vm

ZooKeeper:-

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable.

High-Availability:-

ZooKeeper will be used to watch each worker. It used to make applications highly available. Here the process of who it works, here there is a concept called master and slave. master fails a new master is elected amongst the existing slaves. The slave whose container has the lowest , is elected as the master. A new slave worker is started and all the data is copied asynchronously to the new worker. slave worker fails. A new slave worker is started. All the data is copied to the new slave Asynchronously.

Scalability:-

The orchestrator has to keep a count of all the incoming HTTP requests for the db read APIs. For now, we assume that most requests are read requests and hence the master worker does not need scaling out. The auto scale timer has to begin after receiving the first request. After every two minutes, depending on how many requests were received, the orchestrator must increase/decrease the number of slave worker containers. The counter has to be reset every two minutes.In our implementation the orchestrator communicates to the zookeeper to know how many slaves are present and then scales according to the requirement. The zookeeper does all the book-keeping for running containers.

0 – 20 requests – 1 slave container must be running

21 – 40 requests – 2 slave containers must be running

And so on.

Related work

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

Python binding for zookeeper: https://kazoo.readthedocs.io/en/latest/

Docker sdk to start containers programmatically:
https://docker-py.readthedocs.io/en/stable/

and few more useful resources,

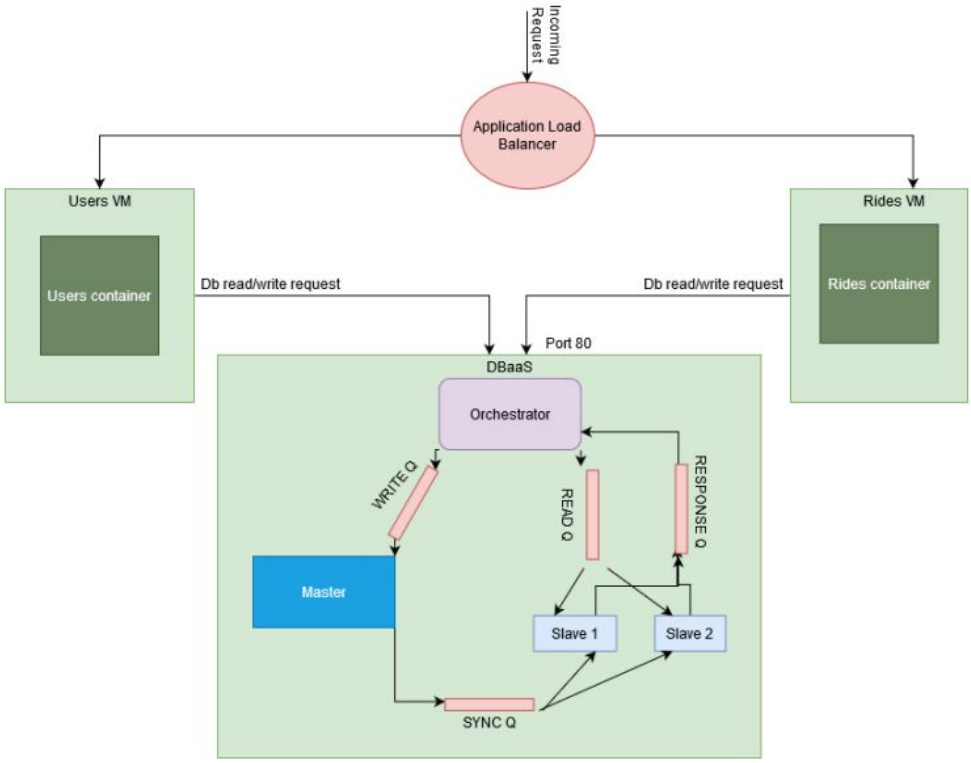
https://pika.readthedocs.io/en/stable/examples/blocking_consumer_generator.html

https://stackoverflow.com/questions/24510310/consume-multiple-queues-in-python-pika

https://stackoverflow.com/questions/28550140/python-and-rabbitmq-best-way-to-listen-to-consume-events-from-multiple-channel

ALGORITHM/DESIGN

Schematic diagram for DBaaS machine



Rabbitmq

As shown in the schematic diagram above , the whole of the database has been migrated into a full fledged system. All the rest api requests come to the orchestrator container , which is basically a flask app.

If the incoming request is a write DB request , the orchestrator will dump the request to writeQ , where the master will fetch requests from this queue and do appropriate operations , commit to db if the request is valid . After this a commit status will be delivered back to the orchestrator , in an exclusive queue , which terminates after the queue is consumed. Master will also broadcast(fanout exchange) the same requests to SyncQ , for slaves to consume (eventual consistency).

If the incoming requests are read DB requests, then the orchestrator will dump these requests onto readQ , for slave(s) to consume in round robin fashion. After processing the requests, slave(s) will put the results into ResponseQ.

Zookeeper

We used kazooClient python interfacing library to connect to the zookeeper.

We first initialised paths for zookeeping. We created paths to store slave pids and master pid as nodes and the container object as the node's data.

We also create watch nodes for the corresponding znodes (running master/slave pid). When a slave or master is created a corresponding znode is also created along with a watch znode. The watch nodes are maintained for each of the znodes (running containers) so as to ensure immunity against crashes.

The watch value is set to 1 if fault tolerance is to be allowed(during crashes) and watch node value is 0(to allow scaling down). According to watch node values new slave containers might be up(for slave crash) or a new master might be elected(master crash).

If a master is crashed then delete that znode and select the lowest pid slave and change its data in watch path as it is master and create one new slave.

TESTING

We have written a testing script. which tests all the cases which are needed to implement. Initially we had a problem in maintaining consistency of database then we overcame that.

In automated submission we had a problem, there even though the slave crashed we got a review that the old slave is not stopped. That is due to time taking for killing the slave container.

CHALLENGES

1. Could Not expose a Pika Blocking connection inside Flask .

2. pika.BlockingConnection as the name suggests is a blocking connection , and cannot be used asynchronously in the code , and has to be a thread/sub_process on its own if there are multiple queues to listen from . Understanding this and making a thread safe code section tricky to tackle .

3.Master Election : after a master crash we had to communicate to the slave to indicate that it is now the master , this was done via zookeeping , the master and slaves needed to communicate to zookeeper periodically to know their identity.

Contributions

Revanth : Understood the basics of the rabbitmq , wrote code/created orchestrator . with the required queue mentioned as in the requirements.

Vikas Matam : 1. Migrated Database framework from FlaskSQLAlchemy to sqlite3

2. wrote thread safe slaves/master which consumes queues asynchronously.

Sachin : 1. Initialising KazooClient, Fault tolerance

2. Deployment on AWS.

Prince : 1. Master Election using Znodes.

2. Designed the zookeeper idea for tracking running containers allowed scaling up and down.

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 , but one has to remember to change ip address of load balancer and instance , if they are working on their system in the code