17CS352:Cloud Computing

Class Project: Rideshare

Mini DBaaS for Rideshare

Date of Evaluation:

Evaluator(s):

Submission ID:438

Automated submission score: 10

SNo	Name	USN	Class/Section
1	Pratheek Kamath M	PES1201701595	H
2	Varsha C	PES1201701387	F
3	Namana	PES1201701898	H
4	Radhika Sadanand	PES1201701702	H

Introduction

The final project is focused on building a fault tolerant, highly available database as a service for the RideShare application.We have used our users and rides VM and the load balancer for the application. In addition we have enhance our existing DB APIs to provide this service.The db read/write APIs we had written are used as endpoints for this DBaaS orchestrator. The same db read/write APIs are exposed by the orchestrator.The users and rides microservices are using the “DBaaS service” that we have created for this project. Instead of calling the db read and write APIs on localhost, those APIs we have called on the IP address of the database orchestrator. We have implemented a custom database orchestrator engine that will listen to incoming HTTP requests from users and rides microservices and perform the database read and write accordingly.

Related work

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

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

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

ALGORITHM/DESIGN

There will be two types of workers running on the instance, “master” worker and “slave” worker.We have written the same code for master and slave for the sake of fault tolerance using zookeeper where there might be a chance of role change of either master or slave whenever there is a failure of any workers.

For RabbitMQ,we created a queue called readQ in slave which accepts read requests from orchestrator and send response through responseQ back to orchestrator.We have made the messages in readQ persistent such that if one slave gets busy with processing one message, another slave can take up the next message. In callback function of readQ,we call read(),which does db read and manually sends acknowledgment to orchestrator.Similarly,we created a queue called writeQ in master which accepts write requests from orchestrator and send response through responseQ back to orchestrator.We also call write(),which does db write as in readQ.Besides this in master,we have created syncQ which broadcasts the write requests to all the slaves which is implemented in the callback function of the writeQ to achieve eventual data consistency in all the workers.

We have implemented scalability where new slaves are spawned or old slaves are killed to maintain the number of slaves according to the number of read requests that have arrived to the orchestrator.For this we used docker sdk to spawn or kill slaves.When scaling down, we first obtain the container list after which we remove the following container – zookeeper,rabbitmq,master,orchestrator from the list in order to get only slave containers and then we randomly choose slave containers from the list to kill.And to implement auto scale timer where a function called "scale"(having logic for scalability) is periodically called every 2 minutes,a module called threading is used.Function called Timer() belongs to this module is used to periodically call scale().

To implement 2 crash apis and worker list api,we used docker sdk.To implement crash master api,we first get access to master container and then kill it. To implement crash slave api,we first obtain the container list after which we remove the following container – zookeeper,rabbitmq,master,orchestrator from the list in order to get only slave containers and the obtain slave with highest pid and kill it.For list worker api,we do the same as in crash slave api except removing master container and returning the list.

For zookerper,we have done slave leader election where if a slave fails then a new slave gets started and the data will be copied to it asynchronously.So what we have done is we create ephemeral znodes using zk.create() in the slaves as they get created with the main path as "/slave" and we perform a watch using zk.get_children("/slave",watch = watch_func) on these slaves in the orchestrator.If a slave goes down,ephemeral node gets deleted automatically and triggers the watch function in orchestrator to get called.In watch function,we spawn a new slave.

TESTING

Initially I got message "Load balancer not working".When I saw the logs of orchestrator,I came to know that db clear api was called to the orchestrator.Then I came to know that I have'nt included db clear api in orchestrator code.This fixed the error.Then I got "Worker list api failed".When I checked,we found out there was a syntax error in that api.Then I got 9 marks without showing the message

“Old slave stopped”.I fixed this by removing the old slave to be stopped from the container list and then killing that slave because I came to know that kill() on a container introduces a delay.Then this error finally got fixed.

CHALLENGES

Doing data replication felt little difficult for us.Implementing slave leader election in zookeeper posed a real challenge to us.

Contributions

Pratheek Kamath M has done RabbitMQ

Radhika Sadanand has done 2 crash apis and list worker api

Varsha C has done Scalability

Namana has done Slave leader election in Zookeeper

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