17CS352: Cloud Computing

Class Project: Rideshare

Backend for a Cloud Based Rideshare application

Date of Evaluation:19-05-2020
Evaluator(s): Spurthi N Anjan and Dilip Gurumurthy
Submission ID:1072
Automated submission score: 10

SNo	Name	USN	Class/Section
01	Anup B Sajjan	PES1201701730	6D
02	Suhas B N	PES1201701492	6D
03	K Suman	PES1201701767	6D
04	Shambu Nandish	PES1201701867	6D

Introduction

This is a Rideshare web server application. In this application, we built APIs using flask and used **Sqlite3** as a database where all the information related to rideshare applications will be stored. As a part of the assignments, we created API's and made them run on docker containers. As a part of the project, we have implemented a rideshare application using Database as a service model.

This has been implemented on the AWS platform where we created 3 VM,

1. For handing User related requests

2.For handling Rides related requests and

3. All these internally talk to our DBaaS VM which is scalable and highly available.

4. And all the requests are directly contacted through the load balancer which internally routes the request to a specific instance based on the target group path specified.

Related work

Accomplishing this project required understanding of concepts of message queues, stopping, and spawning containers dynamically and handling fault tolerance of containers.

This required us to understand the working of the following software and their relevant python bindings.

1.Rabbitmq (https://www.rabbitmq.com/getstarted.html)

2.Docker SDK (https://docker-py.readthedocs.io/en/stable/)

3.Zookeeper (https://kazoo.readthedocs.io/en/latest/)

ALGORITHM/DESIGN

The architecture of DBaaS:
To build this DataBase as a service for our rideshare application we started the following workflow-

1.Master-Slave Model.

To establish the master-slave model we required all workers to communicate and be updated with the changes in master. This was achieved by creating multiple queues using rabbitmq and connecting them to each container. Each worker would do its part of the reading or writing the database and update the same result into the relevant queues, from the other end the orchestrator would pick up the result from relevant queues and send the result as a response back to the user or rides VM which in turn would return the response to the request accordingly.

2. Real-Time syncing slaves:

As soon as the master was updated with a new write request, the same request write query would be sent to a fanout exchange, and in turn, the slaves would be asynchronously waiting and listening to new messages to sync and update themselves. Messages put in fanout exchange would reach all slaves as a broadcast and all of them execute the same SQL query to update the database.

3. Replication of database :

As soon as a new worker is spawned to scale out, the first thing it is supposed to do is to update itself to the current state of the database. To achieve this we again used message broker software rabbitmq. So every time a new write request comes to the master a durable queue with persistent messages would be created which would tolerate the server restart. To this queue, the SQL query executed successfully on the master database would be appended to this queue.

A new worker spawns and the first thing it is made to do is to read all the messages present in this queue, and execute the same in its database and update itself as other workers' current state. In this way, we are able to achieve consistency among all workers.

4.Scalability :

Just having 2 workers won't suffice. Hence when there is a high request period, the orchestrator keeps count of the number of requests and accordingly scales in an out. To achieve this docker SDK came into hand, where we were able to count the request count, and every 2 minutes based on the number of requests, we were able to spawn new containers or crash a worker using docker SDK.

5.High Availability (Fault tolerance):

To handle the fault tolerance of the worker containers, we needed someone to have a watch on all of the workers. Hence Zookeeper cluster coordination service was used in this case. We added a DataWatch in the orchestrator container which would be triggered whenever there is any change in data or state of each znodes of each container. This would allow us to know the change and handle the fault and spawn a new container if needed.

6. High Availability (Leader Election):

To accomplish leader election , In orchestrator , we added DataWatch on each of the worker containers . Initially using docker compose we run one slave and master container, which internally creates a znode with znode data as “slave” and “master” respectively . Now as soon a new container is spawned to scale out using docker sdk, it internally creates a znode and sets the znode data as “slave” and for the same znode path immediately a watch is set in the orchestrator using ChildrenWatch decorator.

Initially we know who the master is , and hence we store the master znode path as “/Worker O” ,Now if the master crashes , then the DataWatch is triggered in the orchestrator and based on the event path , if its same as current master znode path , we distinguish whether master died . If Master crashes then we get the worker list and the lowest pid containers znode data is set from “slave” to “master”, and we update the currentMasterZnode Path to this new master container znode path . This internally causes the watch to be fired inside the newMaster container , and hence we check if the znode data is equal to “master”, if yes , then we kill the current worker process , and re-run the process using command line flags , indicating that , this worker process should execute the master code and not slave code. Hence a new Master is elected and high availability is persisted. After this to compensate for the crash of the old master , a new container is spawned using docker sdk , making it run as a slave container.

This allows our DBaaS to be highly available even if there are a high number of requests.

TESTING

To test the DBaaS, we tried different test cases as :

- Sending a high number of write requests to the master container and checking if all slaves were getting updated and consistent by reading the messages from the queue updated by fanout exchange.
- Sending a large number of read requests to test scalability, and this resulted in spawning extra containers to handle requests.
- In automated submission, the requests from rides VM were not reaching DBaaS VM hence we were getting scale-up failed error. Hence we tested where this was failing in rides container and found out that the ports mapping was mismatched and the DBaaS VM IP address was given wrong.

CHALLENGES

- Replication of the database was a challenge, initially, we tried making volumes and transferring the sqlite.db file, but then we achieved the same with queues.

- ❏ Understanding Docker SDK and spawning containers dynamically took time. In terms of code, it's just 4 -5 lines but, from the documentation extracting only required syntax took time in understanding.
- ❏ Using RPC we tried to get a read response from the slave workers but the response was never reaching the orchestrator, then we figured out that the response correlation id and the actual correlation id are different hence the response is not getting returned.
- ❏ Letting slave know that it has to be master using znode set data and then making slave restart and run as master was a challenge.

Contributions

Suhas B N : Orchestrator API's
Suman K : Scalability using docker sdk , replication of database.
Shambu Nandish : Message Queues using RabbitMQ
Anup B Sajjan : High Availability : Leader Election and Fault tolerance using Zookeeper

CHECKLIST

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