17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 19-05-2020
Evaluator(s): Usha Devi BG and Sanjith
Submission ID: 786
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Arun Kumar Kashinath Agasar	PES1201701537	6B
2	Mahesh H A	PES1201701667	6B
3	Vikas Kalagi	PES1201701654	6B
4	Rakesh Devani	PES1201701602	6B

Introduction

- This project is about building a fault tolerant, highly available database as a service for RideShare application. We were provided with AWS cloud computing platform where we made use of ec2 instances, two microservices such as users and rides are created to communicate with the DBaaS (Database as a service) to perform the database operations such as creating a user/ride, deleting a user/ride and related operations. Where Load balancer was used to navigate the request to appropriate VM and containers were used to provide the isolation for particular microservice.

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.

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

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

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

ALGORITHM/DESIGN

Architecture of DBaaS:

Master-Slave Model.

It includes a master which is responsible for all the write operation to the database and slaves which are responsible for read operation from the database. To accomplish this model all slaves need to communicate and be updated with the changes in master. This was achieved with the help of RabbitMQ which acts as a message broker through which we created multiple queues and connected 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.

Real-Time syncing slaves:

As soon as the master performs a write operation it was published to a queue specifying exchange type as fanout which basically broadcasts the message. All the slaves will be subscribed to this exchange and would be asynchronously waiting and listening to new messages to sync and update themselves. In our case we will be sending SQL query as the message body through the fanout exchange which will be received by all the slaves and these SQL query will be executed to update the local database.

Scalability:

Just having fixed number of slaves won't suffice the requirement as the number of requests generally going to be vary. In our project scalability is done considering the number of read requests will be higher than write requests and is done only in case of slaves. To implement this we kept a count on the number of read requests in the orchestrator. Every two minutes a function was called to check the number of read requests, based upon this count we decide whether there is need of scale in or scale out. For this new containers has to be spawned or killed which was done through docker SDK. The exact number of slaves to be spawned is determined by calling the worker list to know the current number of slaves through which we will get to know the exact number of slaves to be spawned or killed.

Replication of database:

As soon as a new worker is spawned in case of 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. A queue was created of type durable and containing persistent messages which would tolerate a server restart. Whenever a successful write operation is done, the related SQL query statement is put into 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.

Fault tolerance and high availability:

- Fault tolerance is the ability of the system to operate properly in case of failure of some components of the system which in our case is the crashing of the slaves. So the requests will be sent to remaining workers if any particular worker stops.
- High availability means whenever a master fails a master should be elected among the remaining workers and a new worker should be spawned.
- If a slave fails then a new slave worker should be spawned.
- This is done with the help of zookeeper through kazoo which is a python binding for the zookeeper.
- A zookeeper watch is initiated on the children of "/producer/". The watch is triggered whenever a node is created under the path "/producer/". A node is deleted under the path "/producer/". Data on the node under the path "/producer/" is changed.
- Every time a new container is spawned a znode is created for that container. And a datawatch is kept on that znode.
- A children watch will be called whenever on of the above event occurs and whenever a slave fails or deleted. Upon calling this watch we check whether master is deleted or slave is deleted.
- If a master is deleted then we call for leader election in which slave with the lowest pid is elected as master. And its data is set to master which triggers a watch inside the container which stops the python process to running as slave and starts again to run as a master. And a new slave is spawned to compensate the loss of a slave.
- If a slave node is deleted then simply another slave container is spawned.

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 after crashing the slave it was failing to get the old slave workers pid. The problem was at starting the daemon process was taking some

time before actually waiting for two minutes after this was fixed the problem was solved.

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. Especially when we were trying to fetch only the slave worker from all the running containers.
- ❏ Implementing leader election using zookeeper was difficult and also making the elected leader behave as master was difficult to implement.

Contributions:

Mahesh H A: Management of queues using RabbitMQ

Vikas Kalagi: Apis and scalability related work

Rakesh Devani: Apis and scalability related work

Arun Kumar Kashinath Agasar: Leader election using 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