17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 19/05/2020
Evaluator(s): Spurthi and Dilip
Submission ID: 1205
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Naveen D	PES1201701582	6/E
2	Nishchay Karle	PES1201701327	6/E
3	Mithun H M	PES1201700197	6/E
4	Anirudh Joshi	PES1201701558	6/E

Introduction

The main objective of this project was to build a fault tolerant, highly available database as a service for the RideShare application which was built progressively in the assignments. Using this RideShare application users can book rides between two locations at a time of their choice. There are various types of requests which are handled such as requests to Add a New User, Delete an Existing User, Creating and Deleting a New Ride, etc.

There were three Amazon EC2 instances which were used to deploy the project. The Load Balancer was also implemented to distribute the incoming HTTP requests to the Users and Rides instances. A message broker service (AMQP) was implemented using RabbitMQ. The Orchestrator is a Flask Application which is used to implement the various message queues such as 'readQ', 'writeQ', 'syncQ' and 'resQ' using RabbitMQ. It is also responsible for bringing up new worker containers as and when desired. Docker sdk was also implemented for Fault Tolerance for the Slave and Master. The Orchestrator will keep count of the incoming HTTP requests for the db read APIs and depending on the count the orchestrator will scale up or scale down the slave workers, that is increase or decrease the number of slave worker containers for every two minutes.

The users and rides microservices were no longer using their own database but a DBaaS was used. DBaaS stands for 'Database as a Service' which provides all functionalities of a database. It also has various services running such as RabbitMQ and Scaling Services.

The following tools were used for building this project:

- SqlAlchemy as our Database
- RabbitMQ as a Message Broker
- Flask to implement the API's and to route the requests
- Docker SDK for upscaling and down scaling

Related work

● https://www.rabbitmq.com/getstarted.html -We looked through Worker Queues and RPC tutorial from the above link.

● https://docker-py.readthedocs.io/en/stable/ - For the docker sdk part, We looked through the documentation from the above link.

● https://hackernoon.com/what-is-amazon-elastic-load-balancer-elb-16cdced bd485 - The tutorial above was used for implementation of Load-Balancer.

ALGORITHM/DESIGN

The orchestrator exposes the DB read and write APIs which are used by the API handlers in Users and Rides containers. The implementation of read and write API in orchestrator is as follows:

● Read API - The orchestrator publishes a request into a 'read_queue' which is consumed by slave workers in a round-robin fashion. The slaves publish a response back to the orchestrator in the 'res_queue'. RPC objects are used to handle the read API.

● Write API - The orchestrator publishes a request into the 'write_queue' which is consumed by the master worker only. The master updates its database and publishes the request further into the 'syncQ' exchange which is consumed by all the slave workers. Hence eventual consistency is attained. No response is sent back to the orchestrator. For implementation of auto-scaling of workers, we are running scale function in the orchestrator, which executes the function in intervals of two minutes. We created db to store number of read requests and increment row after each request giving us number of read requests. we then use read request count to scale up/down using docker sdk.

TESTING

Automated Testing:

We weren't able to pass one of the test cases where the request for the worker list wasn't returned properly. We updated our code to return the worker list in sorted order as it was mentioned in the specifications and were able to successfully pass all the test-cases. We faced difficulties on load balancing during automated testing so we had to create new load balancer and run it.

CHALLENGES

- We faced difficulties in updating db for the new slave created. We then stored all the write requests into a new table, then when the new slave is started, we extracted all the SQL statements that were previously stored in the table and executed all of them hence achieving persistency.
- We set an environment variable to differentiate between master and slave container which had the same code. Using the 'env' variable we check if it is a master or a slave

Contributions

Nishchay: Implemented the Read Queue and the Response Queue using RabbitMQ and the API for clearing database using Write Queue in the Orchestrator. Also wrote the code for performing Sync of the database present in the master whenever a new Slave starts.

Anirudh: wrote the code for crash slave and implemented the workerlist API which returns the sorted list of pid's of the containers of all the workers. Wrote code for consistency. wrote code for persistence when new slave starts.

Naveen: Implemented the Write API using Write Queues using RabbitMQ and also, the code used to run as master in the worker file. Also wrote code for clearing the database using Write Queue in Orchestrator.

Mithun: Implemented the auto scaling part which spawns new containers if the current number of running slaves is less than the required number of slaves and kills the slaves if the current number of running slaves is more than what is required.

CHECKLIST

SL No.	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