17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 15-05-2020
Evaluator(s): Rachana B S, Deepthi B
Submission ID: CC_0099_0241_0633_0759
Automated submission score: 10

SNo	Name	USN	Class/Section
1.	Rishabh Jain	PES1201700099	D
2.	Saahil B Jain	PES1201700241	F
3.	Jeevana R Hegde	PES1201700633	D
4.	Phalguni Kamani	PES1201700759	F

INTRODUCTION

The project that has been developed is the backend for a cloud based RideShare application. The project was started off by creating a single monolithic service for the first assignment. It was then split into 2 micro services - one for user management and one for ride management - and dockerized in the second assignment. It was further developed by allowing the 2 micro-services to be accessible under the same IP by using a load balancer in the third assignment.

Finally, the fault-tolerant, highly available database as a service for the ride share application was built. Instead of having the users and rides micro service handle their own databases, they use the DBaaS service that was created by us. A DBaaS orchestrator is used to handle the database API endpoints along with a few other endpoints. A master-slave architecture has been implemented, with each slave having a copy of the master's database. Using zookeeper and rabbitmq, the fault tolerance, high availability and scalability has been ensured.

The final working project enables users to join the application, create a new ride if necessary, join rides as required and a few other options. A variety of quality checks have been performed to ensure that the information provided by the user is valid and applicable. This includes the correctness such as password validation, username uniqueness, and joining existing rides to name a few.

RELATED WORK

1. https://www.rabbitmq.com/getstarted.html - Understanding rabbitmq and learn the different ways of implementing queues and exchanges in ways they could be used to implement the project. Understanding how to use the publish/subscribe pattern for work queues using fanout exchange and learning RPC pattern for implementing readQ.
2. http://zookeeper.apache.org/ - To understand the theory behind zookeeper.
3. https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php - Understanding one possible way of implementing leader election with zookeeper to get a base idea to start with.
4. https://kazoo.readthedocs.io/en/latest/ - Understanding how kazoo (python library for zookeeper) works.
5. https://docker-py.readthedocs.io/en/stable/containers.html - Understanding docker sdk to create new containers or get list of all containers.
6. https://docker-py.readthedocs.io/en/stable/api.html - Understanding the low-level API, APIClient, to either get information about a single container, or even stop or kill it.

ALGORITHM/DESIGN

1. Updating the database

- Anytime a write request is made to the write API, the orchestrator writes the data into a queue called **writeQ.**
- The master continuously checks the queue for any messages sent. On receiving the message, it updates the database and an acknowledgment is sent.
- The existing slaves have to update their database as well so as to be in sync with the master.
- So, the orchestrator pushes the same data onto a fan-out exchange called **sync**, which broadcasts the message to all queues listening to that exchange. Each slave has a queue listening on that exchange.
- Finally, a new slave can be created at any point, and needs to update its database to be in sync with the master.
- Hence, the orchestrator also writes every write command to another queue called **commands**.
- Any time any slave is initialised, it reads all the messages in the queue and
    - acknowledges each one
    - updates it database and
    - writes all the messages back to the queue to be ready for the next newly created slave.

~ Clear database API is handled the same way as the write queues.

2. Querying the database

- The user and rides micro service make read requests in order to validate data entered by the user or to provide the user with data on requests.
- The orchestrator handles the read API endpoint.
- Slave workers are responsible for querying their database and providing the requested data.
- The orchestrator and slave communicate using an RPC pattern along with work queues.
- The RPC pattern is used so that the slave has a call-back queue to return the requested data to, which the orchestrator receives and sends to the client.
- Work queues are used to distribute the read requests among all slaves in a round robin manner.

3. Scalability, Fault tolerance and Zookeeper

- Zookeeper is used to keep a watch on each worker created, by associating a znode to each worker. Znode gets created in the worker container itself.
- Environment variables exist to provide a name to the worker and to indicate the type of the worker.
- The name is used to create a znode and to access that znode later.
- Any time a change occurs in the state of any worker, the zookeeper watch function is triggered and an appropriate action is taken.

~ Fault tolerance and scalability have been implemented.

- The orchestrator has API endpoints for crashing the master worker or crashing a slave worker. A textfile containing a json object is used by these 2 API's to indicate which type of worker has been killed. The moment a worker is killed, the zookeeper watch function is called. This reads the json object from the textfile and checks to see which worker died. If a master worker died, the znode associated with the worker having the lowest Pid is found and the value associated with the znode is changed from slave to master. Due to the change in the value of the znode, the worker updates its environment variable to indicate that it is a master now and starts to perform the functions of a master worker. A new slave is then created, and given an environment variable to indicate that it is a slave. If a slave is killed, then a new slave is created.

- A thread is created every 120 seconds to implement scalability. Based on the number of read requests being made, the thread being spawned checks if a new container needs to be created or if the number of containers needs to be reduced. A textfile is maintained to keep track of the number of slaves currently running. The number of containers required based on read requests is then calculated.

- Further, the action of creating new slaves or killing the running slaves is performed. However, this would trigger the zookeeper watch function. To ensure that the watch function doesn't create unnecessary slaves, the text file it reads from tells us if the API's for crashing a master or slave were called. The textfile also tells us if a slave or master was killed. If the API's weren't called, the zookeeper watch function won't do anything. The read requests counter is reset every 120 seconds by each thread that is created.

TESTING

Testing Challenges faced:

AWS EC2 instance kept running out of memory if tested multiple times.
The scalability function was initially in a separate docker container. Hence it wouldn't read from the same text file which the orchestrator continuously updated to keep track of the number of read requests.

How the issues were fixed:

A new instance of type t2.2x Large with 32 GiB memory was created.
Among the options of either mounting volumes or moving scalability function to the orchestrator file, the latter was implemented to avoid having an extra container.

CHALLENGES

The initial understanding of rabbitmq and zookeeper was slightly challenging. Figuring out how to let a worker consume from multiple queues at once was challenging but was overcome with our stable efforts. Trying to come up with a way to sync data between slave and master was a challenge we faced and finally developing an algorithm for the leader election was demanding.

CONTRIBUTIONS

Rishabh:
Helped implement worker files by taking care of slave functions. Helped set up zookeeper in worker file. Created Dockerfile for workers and orchestrator, and docker-compose file. Took care of read API in orchestrator and implemented RPC pattern. Helped in creating queue for syncing data between slave and master.

Phalguni:
Helped implement worker files by taking care of master functions. Helped set up zookeeper in worker file. Created Dockerfile for workers and orchestrator, and docker-compose file. Took care of write API and clear DB API in orchestrator. Helped in creating queue for syncing data between slave and master.

Saahil:
Helped in implementing crash slave and crash master API's in orchestrator. Helped to implement scalability function. Helped set up zookeeper watch function. Helped in creating list all workers API.

Jeevana:
Helped in implementing crash slave and crash master API's in orchestrator. Helped to implement scalability function. Helped set up zookeeper watch function. Helped in creating list all workers API.