17CS352:Cloud Computing

Class Project: Rideshare

Date of Evaluation: 15/05/2020
Evaluator: Prof. Venkatesh Prasad
Submission ID: 263
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Amrutha S	PES1201700829	F
2	Krithika P	PES1201701369	F
3	Anagha M	PES1201701102	F
4	Ria Kalia	PES1201700769	F

Introduction

This project is a cloud based RideShare application that can be used to pool rides.

It is a DBSaaS service where the back-end has two parts: Users and Rides.
Users can: Add a new user, delete an existing user, view all users
Rides can: Create a ride from location A to B, join an existing ride, search for rides and delete rides

In order to perform any of the functionalities mentioned above, a call must be made to the relevant API using the correct request method. All calls are made to the load balancer, which redirects requests to either the users or rides microservice. The rides and users instance run a flask application that listens to requests directed to it, processes data and redirects all read / write requests to the orchestrator.

The orchestrator listens to incoming read / write requests and directs all read requests to the slaves and all write requests to the master. The result of the read request is sent back to the orchestrator, which is relayed back to the microservice that called it.

The orchestrator is also responsible for monitoring the workers, and if any slave crashes, it spawns a new slave to replace it. This monitoring of workers is done with the help of zookeeper.

Apart from this, the orchestrator also counts all incoming HTTP requests and if the number of requests is greater than 20, it increases the number of slaves by one. A similar pattern is followed if the number of requests is above or below 40, 60, 80 and so forth (scale up or scale down).

.

Related work

We referred to various tutorials to implement the project, including the official documentation of Docker SDK, Kazoo and RabbitMQ.

ALGORITHM/DESIGN

Our DBSaaS service has 5 docker containers when it is initially launched – for the RabbitMQ server, Zookeeper, orchestrator, and two workers. Apart from this, the rides and users instances along with the load balancer needs to be running before the system can start accepting requests.

When a request is sent to the load balancer, it's redirected to either the rides or the users microservice. These microservices process the request and redirect all read / write requests to the orchestrator.

The orchestrator is one of the most vital components of the system and is considered to be resilient to failure. It listens to incoming requests from the two microservices, and also monitors the workers.

When the system is started and the docker containers are launched, the two workers first create zookeeper nodes (znodes) with their PID as the path name, and the value stored in these nodes is initially -1, representing that they haven't been assigned as a master or slave yet. The workers also create their own copy of the database, where the details of users and the rides are stored.

Meanwhile, the orchestrator waits for the two workers to create znodes through a ChildWatch. Once the two workers are registered by the orchestrator, leader election is performed by selecting the znode with the lowest PID. The data attribute for this znode is set to 1, indicating that it's the master, and is set to 0 for the other znodes (the slaves).

Once the leader election is performed, the workers start consuming from the read / write queues.

When the orchestrator receives a read request, it extracts the JSON request and publishes it onto the ReadQ, which is listened to by the slaves. The requests are sent to these slaves in round robin format. After the request is executed by the slaves, the response is sent back to the orchestrator through the responseQ.

Write requests are executed in a similar fashion, where the requests are published onto the writeQ. To maintain consistency amongst the workers, write requests are sent to all the slaves so that they can update their database. Whenever a new slave is spawned, it makes a copy of the master's database and then starts executing requests.

The requests that are sent from the microservices to the orchestrator are in JSON format. However, only strings can be published through queues in RabbitMQ. To overcome this, we converted the JSON objects to strings and when the response was received, it was converted back into JSON and sent back to the microservice.

High availability of the slaves is maintained by keeping a ChildWatch in the orchestrator. Whenever there's a decrease in the number of znodes (not due to scaling down, but because the worker crashed due to unknown reasons), a new slave container is run using Docker SDK.

Based on the number of read requests sent to the orchestrator, the number of slaves are either scaled up or down. This is done by creating a thread that runs in the background with an auto timer that is reset every two minutes. When the number of requests crosses 20, a new slave is created and starts listening to the requests. If the number of requests is lesser than the threshold, then the slave with the highest PID is killed. When this happens, a flag is set to indicate to the orchestrator that the slave was killed on purpose, so that it doesn't spawn a new slave to replace the killed one (due to high availability).

TESTING

While testing our system, we primarily ran into problems with respect to the format of the output, as we had made changes to the rides and users API. When the orchestrator sent back the responses of the read requests, the format had to be rectified.

CHALLENGES

We faced quite a few challenges while implementing this project:

- We faced issues with implementing high availability of the master as we were unable to switch over the role of a slave to a master. We discovered that once a slave started consuming data, calls to DataWatch were being ignored and the worker failed to recognize when its data node was changed to 1, indicating that it was now the master.
  To make up for the lack of the DataWatch function, we created a thread which had a similar functionality, but the thread exhibited unknown behaviour, and we weren't sure about how to proceed further from that.

- We had tried to use Async methods to implement zookeeper, but weren't able to do so and faced issues due to lack of proper documentation. We then reverted back to the use of synchronous methods to perform the required operations.

- We also faced issues while sending the response of read requests back to the users,

due to the restriction of the data types that can be sent across the data structures / applications used.

Contributions

Amrutha – Worked on setting up RabbitMQ queues for the workers and the successful execution of read requests(from the slave to the user).

Krithika – Worked on the successful execution of write requests in the master, Scalability of the slaves, and the orchestrator.

Anagha – Worked on High Availability, the crash APIs and Zookeeper

Ria – Worked on setting up the RabbitMQ queues, the integration of the various APIs with the orchestrator, 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