UE17CS352 : Cloud Computing

Class Project: Rideshare

Date of Evaluation:
Evaluator(s):
Submission ID: 508
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Hardik Gourisaria	PES1201700129	6E
2	Shreyas Mavanoor	PES1201700837	6E
3	K Shrinidhi Bhagavath	PES1201701525	6E

Introduction

This project demonstrated the implementation of a Fault Tolerant, Highly Available Database as a Service for the backend of a RideShare Application. Fault Tolerance is required to ensure that the application does not have any down time leading to business loss. High Availability is required to ensure quick servicing of requests made to the application from across the web.

Implementing fault tolerance and high availability lead to some other issues that have been resolved using various other design strategies such as Data Redundancy and Eventual Consistency. Data Redundancy is required to ensure high availability of the application across various domains and quick servicing of the incoming requests. Eventual Consistency is required to ensure that the data across the redundant databases remains accurate and consistent so as to provide accurate responses to the incoming requests.

The database is initially deployed on a single Docker container and based on the number of requests incoming to the database per minute, the number of containers used for servicing the database read and write requests can be scaled in or out using Docker SDK and RabbitMQ. The DBaaS instance receives its requests from the User and Ride instances which have been implemented on another instances. These two instances are deployed over a load balancer and the whole setup has been deployed on AWS.

Related work

The following few web links provided us with great information and assistance to us in successfully completing the project as per the specifications:

1. Load Balancer: https://hackernoon.com/what-is-amazon-elastic-load-balancer-elb-16cdcedbd485
2. Docker and Docker Compose: https://docs.docker.com/
3. Rabbit MQ: https://www.rabbitmq.com/tutorials/tutorial-two-python.html
4. Zookeeper: https://www.tutorialspoint.com/zookeeper/zookeeper_leader_election.htm
5. Eventual Consistency: https://www.youtube.com/watch?v=fIfH-kUaX4c

ALGORITHM/DESIGN

Master Election:

We wait for 3 sec if as slave is created within 3 sec if no master is present then delete the slave_node that we want to elect from slave path and add it into master path. This is done by sending a sigterm to slave to convert it into a master.

Slave Patrol:

If any changes are made on db then we perform commit on master. We start a slave using Docker SDK and run db_manager on slave to remove duplicates entries in the database if any.

Scale Down:

Once every two minutes we check if the number of slaves is as expected. If more slaves are present, we crash the extra slaves without triggering the Slave Patrol action.

Scale Up:

We check the count every time we get a read_db request. If (count - 1)/20 + 1 number of slaves are not running, we immediately start the additional required number of slaves without waiting for the two minutes to pass to ensure optimal performance.

TESTING

While developing the solution, the difficulty of the testing was that we could not easily validate the working of our orchestrator and master and slave election. We attempted on adding redundant print statements when certain conditions were met. However, this didn't help and hence to check if our solution was working, we had to introduce a lot of dump files to which data was dumped so that we could read it later to know what exactly took place.

During the automated testing phase, the major issue that we encountered was that the Load balancer kept crashing and we had to reconfigure the load balancer from scratch.

CHALLENGES

As kazoo creates separate thread for most of the function with no return value, there was no way to know when jobs were finished. This created a few issues where the workers kept crashing. We later figure out the reason and hence implemented lock along with kazoo. This was done to ensure that the currently running process is completed before the new task starts on a new thread.

We feel that the documentation of kazoo is not a great resource for reference as it has not implementation examples. A lot of time of the project went into figuring out and fixing the issues cause by kazoo.

Contributions

Hardik Gourisaria contributed to the development of API involved with the application including making changes to existing APIs. Deployment of the Application over AWS was also handled by him.

Shreyas Mavanoor contributed to upscaling and downscaling workers aspect of the project.

K Shrinidhi Bhagavath handled the docker, leader election and orchestration aspect of the application.

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