17CS352: Cloud Computing

Class Project: Rideshare

RIDESHARE API ON AWS

Date of Evaluation:
Evaluator(s):
Submission ID: 253
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Eshwar H S	PES1201700193	E
2	Vaibhav K T	PES1201700924	F
3	Kethan M V	PES1201701085	E

Introduction

This project is based on developing the backend for a cloud based RideShare application that can be used to pool rides. The RideShare application allows users to create new rides if they are travelling from point A to point B. The application has an endpoint API for each of the functionalities. Some of them are: add a new user, delete an existing user, create a ride, join an existing ride, delete a ride, etc. The application is containerized and deployed on AWS.

Related work

- https://docs.docker.com/engine/api/v1.40/ - To dynamically spawn and kill containers.
- https://www.rabbitmq.com/tutorials/amqp-concepts.html - To understand how RabbitMQ works and various features of a queue.
- https://kazoo.readthedocs.io/en/latest/basic_usage.html - For Zookeeper's DataWatch and ChildrenWatch, ephemeral nodes, etc.

ALGORITHM/DESIGN

The rides' and users' related API endpoints are written as separate flask applications and hosted on Rides and Users instances using a docker container each. The database for the application is hosted on DBaaS instance which has 5 containers initially – Orchestrator, RabbitMQ, Zookeeper, Master and Slave.

The Orchestrator controls the entire database functionality, high availability, fault tolerence and scalability. RabbitMQ queues are used to distribute write requests to Master container and read requests to Slave container. The different queues are: **WriteQ, ReadQ, SyncQ and ResponseQ**. To achieve synchronization of slaves' databases with that of the master's, SyncQ is used as a fanout exchange. For any incoming write request, **the SQL statement is stored on a file on the Orchestrator**. This file is read by a new slave as soon as it's spawned and all the SQL statements are executed on its database to be in sync with the master. Further write requests are sent to the slave using the fanout exchange.

`

The Master and Slave containers are built using the "Worker" image. Containers are spawned using **Docker Engine APIs** which is running on the host system on port 5555. When a worker container is spawned, by default it is considered to be a slave. The orchestrator application maintains a dictionary, **container_ids** which has the key as **worker container ID** and the corresponding value as the **PID of container** on the host system. This dictionary is used when scaling up/down. Also, master dictionary keeps track of which container is the master.

To scale up/down based on the read requests to the database, the count is maintained in a text file **count.txt**. This file is updated after every read request. A **Background Scheduler** is used to schedule a task that scales up/down the slave containers at every 2-minute interval. The scheduler calls a function at each interval. The function checks the number of read requests that the database got in the previous interval and spawns a **new slave container for every 20 requests** it got. Or the function can reduce the number of slave container by scaling down using the **container_ids** dictionary and the **Docker Engine API**.

Zookeeper is used to maintain High Availability. When a slave container fails, the number of required slave container and number of slave container deployed currently is checked. If there is a difference, the difference number of container is spawned/killed. **ChildrenWatch and DataWatch** of Zookeeper is used to maintain High Availability. When a new worker container is spawned, Zookeeper's **DataWatch** is triggered. A node is created with the name node<seq_number>. The node is made ephemeral so that it is destroyed when the container is killed. Each node has data as a string stored in binary format. The string is a semicolon separated string of the format **container_id;process_id;role** where the role is either slave or master. By default, the role is slave. All the nodes are created in the path **/worker.** When there's a change in the path, Zookeeper's **ChildrenWatch** is triggered. This causes the master leader election to happen. List of all the children nodes is received, the data string of each node is checked if any container's role is master. In no container is found, the children nodes are checked again to find the node with **lowest PID** to be elected as master. The data string of that node is changed from **container_id;process_id;slave** to **container_id;process_id;master.** This change in data of the node causes Zookeeper's **DataWatch** to be triggered in that particular container.

When **DataWatch** is triggered, subprocess is used to change the container's role from slave to master. By default, the container will be running **slave.py** script. When the role is changed to master, the current subprocess is stopped and a new

`

subprocess is created to run **master.py** in that container. By doing this, fault tolerance of master container is achieved.

TESTING

The beta testing helped test our code before the final submission. Since there were no errors in our code, we received full marks in the first attempt. There were no testing challenges of sort that we faced.

CHALLENGES

- Making write messages to be persistent on SyncQ was a challenge. This was solved by using fanout exchange for SyncQ and making the master as producer and all the slaves as consumers of that exchange. The SQL statements for write requests were stored on **commands.txt** file on the orchestrator to make write messages persistent.
- Mounting Docker socket file and using Docker SDK so spawn new containers from Orchestrator was a challenge. Instead, Docker Engine API was used to spawn containers. The API is running on host machine on port 5555.
- Implementing High Availability using Zookeeper was a challenge because of less documentation and hard to understand. Kazoo Client documentation helped overcome this problem and provided information about ChildrenWatch and DataWatch.
- Running the same code for both master and slave containers but changing the role from master to slave. This was achieved by using subprocesses.

Contributions

Implementation of Orchestrator, spawning/killing containers using Docker Engine, implementation of WriteQ, ReadQ and ResponseQ was done by Eshwar H S.

`

Implementation of Fault tolerance and Master election, synchronization of slave containers with that of master's was done by Vaibhav K T.

Implementation of Scaling every 2 minutes using BackgroundScheduler was done by Kethan M V.

CHECKLIST

SNo	Item	Status
1.	Source code documented	Yes
2.	Source code uploaded to private GitHub repository	Yes
3.	Instructions for building and running the code. Your code must be usable out of the box.	Yes

`