17CS352:Cloud Computing

Class Project: Rideshare

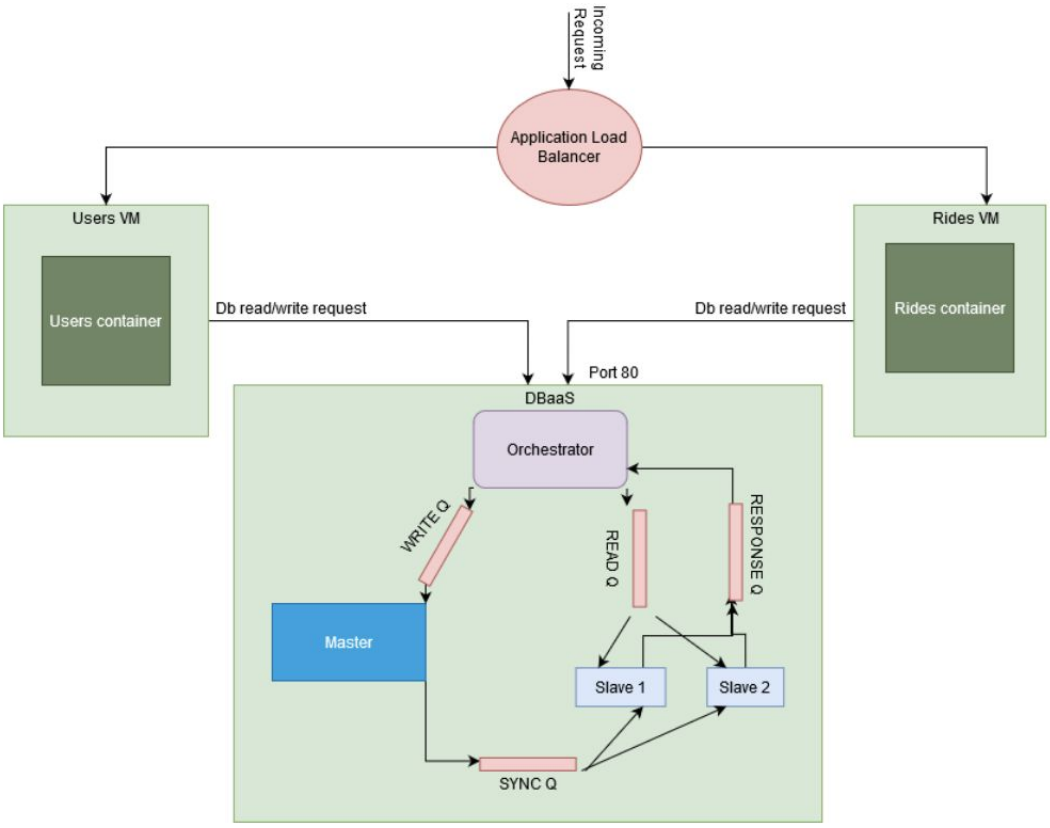
REPORT

Date of Evaluation: 15-05-2020
Evaluator(s): Venkatesh Prasad
Submission ID: 672
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Kritika Kapoor	PES1201701868	C
2	Shrutiya M	PES1201700160	C
3	Shubha M	PES1201701540	C

Introduction

A fault tolerant, highly available database as a service for the cloud based RideShare application was developed on AWS instances. The RideShare application allows the users to create a new ride if they are travelling from point A to point B and also pool rides. This is achieved with the help of REST APIs using Flask. We use two microservices, each for Users and Rides, started in separate docker containers running on one AWS instance. We also create an AWS Application Load Balancer which will distribute incoming HTTP requests to one of the two microservices based on the URL route of the request. Instead of using separate databases from each microservice, we implemented a 'DBaaS service' for the application. We created a custom database orchestrator engine that will listen to incoming HTTP requests from users and ride microservices and perform the database read and write operations.



Related work

We referred to the following resources in order to understand the concepts and implement the same:

1. AMQP(Advanced Message Queue Protocol) using RabbitMQ as a message broker was understood by going through the tutorials in the official site: https://www.rabbitmq.com/getstarted.html
2. Zookeeper concept was understood using the following sites.
    a. https://zookeeper.apache.org/
    b. https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php
3. Python implementation for zookeeper was understood from the official documentation : https://kazoo.readthedocs.io/en/latest/
4. Docker SDK as well as the low level API was understood from the official documentation : https://docker-py.readthedocs.io/en/stable/
5. Piazza posts for doubt clarification.
6. Cloud Computing slides shared and hands-on materials

ALGORITHM/DESIGN

1. **Defined the APIs**
    a. Add user
    b. Remove user
    c. Create a new ride.
    d. List all upcoming rides for a given source and destination
    e. List all the details of a given ride
    f. Join an existing ride
    g. Delete a ride
    h. List all users
    i. Clear db
    j. Get total number of rides
    k. Get total HTTP requests made to microservice
    l. Reset HTTP requests counter
    m. Crash master
    n. Crash slave
    o. List all the workers
2. **Deployed on Web**
    a. Deployed as a web application using gunicorn via supervisor. Supervisor will look after the Gunicorn process and make sure that they are restarted if anything goes wrong, or to ensure the processes are started at boot time.
    b. Set up NGINX, a HTTP and reverse proxy server.

3. **Setup Docker containers and Load Balancer**
    a. Set up docker containers, using a python based image, for the two microservices - one catering to the user management, and another catering to the ride management.
    b. Created an AWS Application Load Balancer that forwards the incoming request to users microservice if it matches /api/v1/users otherwise forwards the request to the rides microservice.
4. **Orchestrator Setup**
    a. RabbitMQ was used as a queue service using pika Connection client in python, creating four queues, WriteQ, ReadQ, ResponseQ, SyncQ.
    b. A timer was triggered at the first read request, and at the end of every two minutes the number of read requests were monitored and the slaves were accordingly scaled up/down.
5. **Zookeeper**
    1. Zookeeper was used to create znodes for every slave, all the slaves stored on the same path and the data consisting of details such as type of worker(master/slave), PID of the container, and Container ID.
    2. For the Master Election, the list of workers were iterated and the slave with the highest PID was made master. Since start_consuming is a blocking call we were not able to switch from the slave code to the master code in the container despite changing the flag. Hence, a new queue was used with a fanout exchange. Whenever a master crashes, Zookeeper watch gets triggered. Here, the slave with the lowest PID is made the new master and the znode data is changed accordingly. When the new master is elected, a message is broadcasted that informs about this event and contains the container ID of the new master. With this information, the new master then stops listening to ReadQ, syncQ using basic_cancel and starts listening to the WriteQ. Hence a slave was successfully converted to a master. Also, a new slave was dynamically brought up for fault tolerance.
    3. For the Slave Crash, a new slave was brought up dynamically using docker sdk achieving fault tolerance.

TESTING

Since we had finished the project well before the deadline, we were able to test it appreciably well and resolve the challenges faced (described below) . Hence, we received 10 the first time we submitted in both Beta and the final submission.

CHALLENGES

1. Output of newly created containers : Newly spawned workers weren't printing on the screen. Hence we used Docker SDK's attach method on the container to get continuous logs of the container on screen.
2. Container's PID : To obtain the container's PID, we used Docker SDK's low level API inspect, using the State Pid attribute.
3. Spawning a new container with a consistent database : Using the existing worker image with mounted volumes wasn't working in the expected manner. Hence we built the image using Docker SDK and used that.
4. Master fault tolerance using zookeeper : Since start_consuming is a blocking call we were not able to switch from the slave code to the master code in the container despite changing the flag. Hence, we used a new queue with a fanout exchange. Whenever a master crashes, Zookeeper watch gets triggered. Here, the slave with the lowest PID is made the new master and the znode data is changed accordingly. When the new master is elected, a message is broadcasted that informs about this event and contains the container Id of the new master. With this information, the new master then stops listening to ReadQ, syncQ using basic_cancel and starts listening to the WriteQ. Hence we were able to efficiently switch the functionality of slave to that of master
5. Differentiation between master and slave : We store if it's a master or not in the znode data. A flag is present in the worker code too.
6. Zookeeper watch triggered multiple times : Flask runs main function twice. Giving app.run(debug=True, use_reloader=False) fixed it.
7. Contradiction between high availability and scalability : Deleting slaves as part of scaling down trigger's Zookeeper's watch. We exploited the property of watch being triggered only once per node. We check for the type of event to be deleted in watch and change the data of the slave to be deleted before deleting which cancels the watch for deleted.
8. Despite making the znodes ephemeral,znodes were not getting deleted in the desired manner as soon as the container crashed. Hence we set the host kazoo client timeout to 1 second, that achieved the heartbeat check of all the clients every second.

Contributions

1. PES1201701868 : rabbitmq, with all the queues, Scaling, new APIs, Fault tolerance (slave + master) using Zookeeper, Debugging, Report
2. PES1201700160 : rabbitmq, with all the queues, Scaling, new APIs, Fault tolerance (slave + master) using Zookeeper, Debugging, Report
3. PES1201701540 : rabbitmq, with all the queues, Scaling, new APIs, Fault tolerance (slave + master) using Zookeeper, Debugging, Report

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