17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 16/05/2020

Evaluator(s): Prof. Nishitha and Vinay

Submission ID: 986

Automated submission score: 10

SNo	Name	USN	Class/Section
1	Sreedhar	PES1201700019	6 B
2	Rohit	PES1201700167	6 B
3	Vinayaka M Hegde	PES1201701600	6 B
4	Ramakrishnan	PES1201701906	6 B

Introduction

Through this project we aim at developing the backend for a cloud-based application called RideShare. The RideShare application allows the users to create rides from one place to another. The application allows users to join existing rides and view the details of the rides as well. This application is deployed on the cloud making it easily available, accessible, scalable and fault tolerant. The backend is developed using REpresentational State Transfer Application Program Interface (RESTful APIs).

The APIs are developed using python flask. The flask application is run on an AWS instance using NginX as the web server. The APIs generate the corresponding status code depending on the input and provide different services based on the request method. The application is built using 3 microservices namely the users, the rides and the database microservice.

The users microservice provides APIs to add new users, list all users and delete users. This microservice handles all the functionality related to the users. Hence it is run on a separate container in the AWS instance. The APIs are accessible through /api/v1/users. These APIs create and use the “users” table in the database as and when required through the APIs provided by the database microservice.

The rides microservice provides APIs to create a new ride, list all rides between two locations, list details of a given ride, join a ride and delete a ride. This microservice handles all the functionality related to the rides and is run on a different container in the AWS instance. The APIs are accessible through /api/v1/rides. These APIs create and use the “rides” table as when required through the APIs provided by the database microservice.

The database access made by the microservices are done using the database related API calls. The database APIs are implemented as a Database as a Service. They support APIs to read from, write to and clear the database. To support high availability and fault tolerance, the database microservice is implemented using a master slave architecture using an Orchestrator to manage them. Zookeeper ensures high availability by creating a new worker if a worker crashes or when scaling out to handle more requests. Zookeeper also handles scaling in by crashing few of the workers based on the number of requests received. Each of the worker runs in their own containers including the orchestrator and zookeeper. The containers use queues to transfer messages between them. Here RabbitMQ and AMQP are used to ensure that the communication using queues is successful even in the case of a crash. The DBaaS is compiled into an image which is run using docker compose. The related images such as RabbitMQ and Zookeeper are downloaded and used by docker when bringing up the containers. Docker SDK is used to handle the containers in the application instead of using the command line. APIs related to the workers such as get worker list, crash master, crash slave are also implemented.

Related work

For learning the basics of RabbitMQ and AMQP protocol, we referred tutorials from official rabbitmq website. From the six tutorials, we understood how to publish and consume message from a queue, types of exchange and when to use a particular exchange.

For using methods in kazoo library, we referred the official kazoo documentation and got an idea of the various parameters to be passed and when to use them.

For starting/stopping containers dynamically using docker sdk, we referred the official docker sdk documentation.

ALGORITHM/DESIGN

The design of our project is as follows:

1. **Users/Rides microservices**

   The users and rides microservices are containers running in separate Amazon EC2 instances. In users microservice, we have a function to check if received password is in valid SHA format – this is used when creating a new user. When required, api calls are made to orchestrator microservice for reading/writing the database and corresponding status codes/response data is used to carry out the operations.

   In the rides microservice, we have a method for checking if received source/destination are valid – these source/destination are present in a csv file. We also have a method for checking if a username is valid. In this method, we make a call to the users microservice and use the returned status code. Like the users microservice, the rides microservice also makes api calls to orchestrator to carry out database read/write operations.

2. **Orchestrator microservice**

   Orchestartor is a flask application that has api endpoints for reading and writing database. In the orchestrator file, we used docker SDK to bring up master and slave containers and we also set an environment variable to distinguish between master and slaves. A variable called slave_count is incremented when a new slave container starts. We have created two classes called writeMsg and readMsg inside which we declare read and write queues. We have used RPC for implementing the aforementioned queues. So, there is a callback function for response that is triggered as soon as it gets a response. The read and write apis use methods present in the above mentioned class to publish data to corresponding queues.

In the worker file, we declare the same queues as that of orchestrator file and we have callback functions that gets triggered on receiving a read/write request and performs the necessary read/write operations on database. Finally, the status codes and valid responses are published back to corresponding queues.

The next part is scaling the number of slaves based on the number of read requests received. For this, we have a method called scale() that gets the expected number of slaves (based on number of requests) and we use slave_count to get current number of slaves and we take the difference between them to scale up/down slave containers. Since the no requests has to be reset after every two minutes, we use time.sleep and since this is a blocking call, we create a separate thread for scale().

Since we are using sqlite3 module for database, the database file of master is copied to the new slave container's path (since master has the updated database) as soon as the new slave starts. This is done using docker cp command and it ensures that the new slave container has updated database. And for syncing the messages from master to all slaves, we used a publish/subscribe method using fanout exchange. So, the master publishes the write messages to an exchange and all slaves have a queue of their own through which they consume these messages that ensures consistency of database between master and all slaves.

When a new slave is brought up, a new znode is created which will have the pid of slave container as data inside it. We have used ephemeral and sequence parameters while creating znodes. Setting ephemeral to true ensures that znode gets deleted when the corresponding slave crashes. Setting sequence to true ensures that every znode is given a concurrent sequence number.
In the orchestrator flask application, we have a variable slave_crash that indicates the number of slaves that have crashed as a result of calling the api /crash/slave. We have used ChildrenWatch decorator on callback function. So, when a new znode is created/existing znode is destroyed, this function gets triggered. Inside this function, we examine the value of slave_crashed to get the number of slave container that are to be brought up.

3. Setting up Application load balancer

For setting up a load balancer, we created two target groups – for users and rides. Then, we defined rules that indicate the target group the request had to be redirected to based on the REST api path of the incoming request. We have also given proper paths for health checks to be performed by load balancer.

TESTING CHALLENGES

1. In the initial phase, we were getting AMQP Connection Refused error from rabbit mq frequently. Then, we figured out that rabbitmq closes the connection if it doesnot get a request in 60 seconds. So, we used the heartbeat parameter while establishing a rabbitmq connection.

2. Although our code was correct (we had copied database file from master to slave when a new slave starts), the database of new slave was not consistent with that of master. Then we understood that before orchestrator copied the updated database to new slave, worker file established a connection to database. As a result, it was using the old database file. So, we added a time.sleep for 5 seconds before connecting to database in worker file.

Contributions

Sl No.	**NAME**	**CONTRIBUTIONS**
1.	Ramakrishnan K	Data Replication and Syncing
2.	Sreedhar Karnam	Orchestator part
3.	Rohit	Scaling up/ down slave containers
4.	Vinayaka M Hegde	Zookeeper part