17CS352: Cloud Computing

Class Project: Rideshare

Implementing a Rideshare Service that works on the Cloud

Date of Evaluation: 23rd May 2020

Evaluator(s): Meghana, Venkat

Submission ID: 784

Automated submission score: 10

S. No.	Name	USN	Class/Section
1	Vithal P Nakod	PES1201701746	6E
2	Roshan B Poojary	PES1201701628	6H
3	Ganesha K S	PES1201701731	6B

Introduction

This project uses the features of Cloud computing and technologies like the Amazon Web Service and other offline features like docker and flask to implement a Rideshare service. The service is developed in such a way that the user gets the best experience without any hindrances while doing any of the numerous operations that are offered. Features like zookeeper and Docker SDK are used to make sure the service keeps on running even at times of failure. Once the user has signed up the user is a part of the user database and his information is stored in it until there is an override command issued by the administrator to clear the database. While designing and explaining this project we take the point of view of the administrator since he can see all the parts that make service functional.

ALGORITHM/DESIGN:

The main design elements of the service consist of three main instances along with an additional Load Balancer to handle certain requests. The instances are the orchestrator, the user and the rides. We will see the implementation of all these parts one by one followed by how we used the zookeeper and Docker SDK.

The Load Balancer we used is the Amazon's Application Load Balancer mainly because it is slightly superior to the classic Load Balancer in regards to handling requests. The ALB mainly differentiates the requests according to which instance they belong to. They go to the user instance if it has operations pertaining to user database or rides if ride database operations ae present. This is the main function of the ALB and all requests go to it.

Once the request has been classified with the help of the ALB, we arrive at the particular instance may it be user or rides. Here, we check whether the operation reads from the database or writes to the database. We do this by going through the APIs created. We send the APIs to the orchestrator based on whether they are read or write operations. According to this, the orchestrator selects whether to execute the read database API or the write database API. In these APIs we have also made sure that depending on the number of requests it scales up/down the slaves as specified. To ensure this the scaling function is used. The wait_seconds is set to 120s as specified in the scaling function to ensure that it sets to 0 after that interval.

In the scaling function we get the number of the current slaves and the number of required slaves (by dividing the count by 20 since for every 20 new requests there should be a slave working). Then we take the difference of these two. Thus, we get the number of slaves that we have to create or crash. Creating the slave is done simply by executing the python command using the run function in the container. In case there are extra slaves, we have to crash them. Since this is not part of the operation where a new slave is created every time an old one is deleted, we have written an exclusive function called crash_through_scale() to handle this. Then we change the values in the count.txt file accordingly.

We have written a class called RpcClient() to handle the queues of the read and write database requests/responses. Depending on which queue has to be called we send the queue name from either one of the data bases as a parameter to the call() function in this class. This class also has the on_response() function to identify which message the response has come from based on the correlation IDs.

After getting the response from the RpcClient class the orchestrator then sends this response to the user/ride, whoever made the request.

There are several more functions that we have implemented in the orchestrator.

1.The crash() is used by the administrator to crash whichever slaves that he/she wants. Its execution is direct. We fetch all the container with the tag project_slave and then get its PID and ID. Based on the PID we sort the slaves. We get the count of the fetched slaves. If it is zero then there are no containers else, we fetch the container to be crashed based on its ID and kill it using the stop() and remove() functions.

2.The all_worker() function is used to fetch the list of all the worker containers. This is done simply by excluding orchestrator container, the RabbitMQ container and the zookeeper containers and then appending all the remaining to a list and return the response.

3.The cleardb() is an extension of the write to database and is written separately since it is to be used for both users and rides. It executes the query in both the ride and user instances.

Additionally, the zookeeper was setup by first executing the necessary commands in the docker compose file which imports all the required files based on the commands in it. Then in the orchestrator we make the connection to the channel based on the port specified in the docker compose at the beginning of the code and also ensure the path. Now we keep an eye on this path. So based on the global variable that we have set which changes its value whenever a slave is crashed using crash() function - except in case of scaling which we have handled using crash_through_scale() function – we have to create a new slave by executing the python command using run() in that container.

The zookeeper is also used within the master and the slave. The master global variable is used to differentiate between the two. Both have the same functions with. The main difference is that all the read operations are handled in slaves and the master handles the write database operations. At the beginning of both the master and the slave we create the znode with sequence value to TRUE (to identify the different znodes of different containers), and the ephemeral to TRUE (to bind the znode to the zookeeper of that particular session. Once we do this, we make a pika connection (RabbitMQ) and all the responses of the queue operations are then broadcasted through this connection every time. The syn() function ensures that the new slave knows the queries executed by the old slave when it is newly created (just like a log of operations of the old slave) so that it can continue processing with this knowledge.

Lastly, the rides and the user instances have all the operations pertaining to the rides and user respectively with their own databases. These operations were specified at the beginning of the project. All the requests are sent to the orchestrator. The response is fetched from the orchestrator and this in turn becomes the response of the rides and the user instances.

TESTING

In Assignment 1,2 and 3 we had uploaded the data base separately along with our code. This was allowed at that time and caused no problems. But during the testing phase of the final project we encountered problems while trying to upload our database.

To overcome this, we wrote a separate code which creates the database with all the required fields. This eliminated the problem of uploading the database every time we had to submit.

There were no other major issues while testing.

CHALLENGES

The Docker SDK is used in many functions throughout the orchestrator. The main places we use Docker SDK are in crash() function, in scaling() function and in list all worker functions.

In crash() function we make use of the docker.from_env() so that we can access the containers through their tag names(project_slave).In all_worker() function we make use of the same above docker function so that we can get a list of all the workers.

The internal operations of the crash(), all_worker() and the scaling() functions once we make use of the Docker SDK is clearly explained in the design portion of this report.

In the Docker compose file we have created an image zookeeper under the service zoo. This re-starts every time there is a failure and we have written the necessary port numbers and created the environment. In the master, slave and the orchestrator containers we make use of this service. Using the correct port number, we establish the connection and start the zookeeper. We also ensure that the zookeeper has a clear path to the slave. The Zookeeper is implemented as an API called ChildrenWatch to ensure that a new slave is created whenever an old slave crashes.

Contributions

The Implementation of the crash API and the List all worker API in the orchestrator was done by Roshan Bhaskar Poojary. Also did the change to the code when the database could not be directly uploaded.

The implementation of the synchronization API in the master and the slave and Rpc Class in the orchestrator was done by Vithal Nakod. Also handled most of the debugging stages during offline testing using postman software.

The Cloud operations like creating and testing the instances and also creating the Load balancer was done by Ganesha K S. Also implemented the scaling function in the orchestrator by also taking care to separate it from the crash API.

Furthermore, Ganesh and Vithal handled wrote most of the user and ride APIs and also wrote the read and write database operations in the master and slaves.

Testing was done at various stages by whoever was making the changes.

Related work:

The following links were used so that we could better understand certain topics

1. Zookeeper: http://zookeeper.apache.org/

2. Zookeeper Container Image: https://hub.docker.com/_/zookeeper/

3. RabbitMQ Channels: https://www.rabbitmq.com/channels.html

4. Python binding for zookeeper: https://kazoo.readthedocs.io/en/latest/

5. RabbitMQ Container Image: https://hub.docker.com/_/rabbitmq/

CHECKLIST

S. No.	Item	Status
1.	Source code documented	Completed

2	Source code uploaded to private GitHub repository	Completed
3	Instructions for building and running the code. Your code must be usable out of the box.	Completed