UE17CS352: Cloud Computing

Class Project: RideShare

Date of Evaluation: 18-05-2020
Evaluator(s): Srinivas K.S.
Submission ID: 1543
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Raghu G	PES1201700057	E
2	Bagyasree S	PES1021700164	E
3	Jayadev M	PES1201700166	E
4	Kaustubh Raghavan	PES1201700916	E

INTRODUCTION

The application, RideShare, aims to build a scalable, highly available system by which users can create and join rides. The application allows one to create users, retrieve users, and delete users as per requirements. Users who are registered in the database can create rides and join other rides. Rides can also be deleted.
The project involved building the backend for the above application, in a manner that allowed distributed access. It was built on AWS EC2 instances, which have their own IP addresses. This allows one to access the application remotely. It was built using Python, in conjunction with other tools, such as RabbitMQ, ZooKeeper, and Amazon's own Load Balancing Service.

RELATED READING

RabbitMQ : Documentation provided on website, plus tutorials on AMQP and the concepts involved.
ZooKeeper : Official Documentation
Docker SDK : Official Documentation

DESIGN

RideShare was built in stages, starting with the development of the APIs. It was then containerized, distributed across instances, and attached to a load balancer. Database functions were set up across containers, with RabbitMQ serving as the messenger. Finally, it was made highly available using ZooKeeper, and auto-scaling was implemented.

Development of APIs

Users

- Get Users
  A json object containing the necessary fields is created and sent as a post request to the orchestrator, to the /api/v1/db/read end point. Depending on the data received, a response is returned with status 204 if no data is present and 200 otherwise (along with the retrieved data).
- Add a User
  An initial check is made to see if the new user to be added has already been created. If yes, then a response is returned with the status as 400. Otherwise, the new user is added to the database via the /api/v1/db/write endpoint. A response with status code 201 is returned.
- Delete a User
  A check is made to see if the user to be deleted is present in the database. If not, then a response is returned with the status as 400. If the user is present, then they

are deleted. The rides created by the user are also deleted. Their username is removed from the users list in the rides they are a part of.

Rides

- Rides Counter
  It returns a response containing the number of rides created.
- Delete a Ride
  A check is made to see if the ride to be deleted is present in the database. If not, then a response is returned with the status as 400. If the ride is present then it is deleted.
- Get Ride by ride_id
  A json object containing the necessary fields is created and sent as a post request to the orchestrator to the /api/v1/db/read endpoint. This retrieves the ride details if it is present. If it is, a response containing the details is returned. Otherwise, a response with status code 204 is returned.
- Get Ride by Source and Destination
  A json object containing the necessary fields is created and sent as a post request to the orchestrator to the /api/v1/db/read endpoint. This aims to retrieve the rides with the appropriate source and destination codes. If none exist then a response with status code 204 is returned. Else the list of rides is returned.
- Add User to Ride
  A check is made to see if the user exists. If not, then response with status code 400 is returned. Else the user is appended to the users list of the ride and the values are updated.
- Add Ride
  A check is made to see if the user exists. If not, then response with status code 400 is returned. Else, a check is made to see if the source and destination codes are legitimate. Then the ride is created. A response with status code 201 is returned.
- Database Write API
  All write requests are funnelled to this API. Various rules exist to deal with the different requests from users and rides containers to add the data properly to the database. It also generates the ride ID for new rides to be added.
- Database Read API
  All read requests are funnelled to this API. It deals with the retrieval of data from the database in accordance with the details provided in the JSON files sent by the users and rides containers.

Containerization

Five containers are run at start time on the orchestrator instance, and one each on the users and rides instance.

Users

The users container is built from the users folder that contains the requirements text file, the Dockerfile, the app.py file and the api_count text file. Port 80 of the container is exposed to port 80 of the host. It uses the python:3 docker image.

Rides
The rides container is built from the rides folder that contains the requirements text file, the Dockerfile, the app.py file, the AreaNameEnum CSV file and the api_count text file. Port 80 of the container is exposed to port 80 of the host. It uses the python:3 docker image.

Orchestrator
The proj_mk_2 folder contains the database.db file, the orchestrator.py file, the worker.py file, the ride_id text file, the requirements text file, the Dockerfile and the docker-compose.yml file.
The zookeeper container is built using the official zookeeper image with port 2181 of the container exposed to the host's 2181. Container rabbitmq is built using the official rabbitmq:3.8.3-management image with ports 5672 and 15672 of the container exposed to the host's ports 5672 and 15672.
The master container is built from the proj_mk_2 folder, with Dockerfile as its dockerfile. An environment variable WORKER_TYPE is set with value as MASTER. It also has a label, which says its worker_type is 'master'. It has links to the zookeeper and rabbitmq containers and depends on them. Its worker.py file is run after executing a sleep command for 25 seconds.
The slave container is built from the proj_mk_2 folder, with Dockerfile as its dockerfile. An environment variable WORKER_TYPE is set with value as SLAVE, and the value for its label worker_type is also 'slave'. It has links to the zookeeper and rabbitmq containers and depends on them. Its worker.py file is run after executing a sleep command for 15 seconds.
The orchestrator container is built from the proj_mk_2 folder, with Dockerfile as its dockerfile. It has links to the zookeeper and rabbitmq containers and depends on them. Its orchestrator.py file is run after executing a sleep command for 15 seconds. The container's port 80 is exposed to the host's port 80.

All new containers that are spawned programmatically are built using the slave image, receive copies of the slave database, and are attached to the same network as the rest of the containers.

Message Passing

Once the different parts of the application have been put into different containers, each with their own separate databases, a channel has to be established to allow communication, and to maintain consistency across the databases. RabbitMQ, with its AMQ Protocols, is used here. The parts of the application in the Users and Rides VM access the orchestrator via its IP address and port; however, the orchestrator, in order to communicate with the workers (the master and slaves), needs queues. These queues allow consistency to be maintained across the databases in a fairly straightforward manner. First, a channel is set up, and an exchange is created within the channel to which queues for reading, writing, and responding are attached. Whenever a read or write request reaches the orchestrator, along with its accompanying JSON object, it is routed to the corresponding function, which pushes it to the read or the write queue, with the object as the message body. The worker file is set up such that slaves read from the read queue, and

the master reads from the write queue. These queues are created as soon as the containers are run, and the workers start reading from the appropriate queue at once, based on their type.
A message in the read queue is fetched by a slave, and the required response is generated based on its database. This response is published to the response queue. Messages in the response queue are consumed by the orchestrator, and it sends the response back to VM that called it. To ensure that the responses for different requests don't get mixed up, the prefetch count is set to 1. This means that, until the slave acknowledges a message, it cannot consume the next message. This acknowledgment is sent only once the response has been published to the response queue, and thus, the responses will always arrive in order. Read requests are handled by the slaves in a round-robin fashion; worker queues are used for this purpose.
Similarly, messages in the write queue are fetched by the master. The database of the master is modified as indicated, and a response is sent back through a temporary response queue set up especially for the master. This prevents race conditions while responding even if read and write requests are received at the same time. Once the changes to the database have been made, these must be reflected in the databases of the slaves. To achieve this, a fanout exchange is created, which publishes the write request to multiple queues, each of which sends messages to one slave. The slaves then perform the same function the master performed to update their databases.

Availability

In order to make the RideShare application fault-tolerant, ZooKeeper is used.
Upon running the containers required for the application, a ZooKeeper hierarchy of nodes is created. There are two nodes at first – *master* and *slaves*. The slaves node further branches out into individual nodes for each slave worker. These are named 'slave*containerID*', which makes each node unique. The node stores the container ID and the PID of the slave in its data section. A dictionary of all the slaves, with the keys as container IDs and the values as the corresponding PIDs, is also created at start up. This dictionary is updated as slaves are created and destroyed.
When the slave crash API is called, the dictionary of slaves is used to find the slave with the highest PID. The container ID of that slave is then retrieved, and the corresponding slave node is deleted. Following this, the container itself is killed, the dictionary is modified, and the appropriate status code is returned.
ZooKeeper watches the children of the *slaves* node for deletions. While the watch function is triggered whenever changes are made to the children, in our application, the only changes possible are deletions, which makes watching the children sufficient. When the watch function is triggered, it checks a variable that indicates whether or not another slave must be brought up in place of the deleted slave (this is necessary, as scaling sometimes requires slaves to be killed without new slaves being spawned). If the variable is set to true, then the function proceeds to bring up a new container, mounts the volume of another slave onto it, and connects it to the same network. This ensures that it receives the database as it is at the present time, and it will be integrated into the read cycle. A

node for the new slave is created in the ZooKeeper hierarchy, and it is added to the slave dictionary.

Scalability

The application has an auto-scaling feature, and creates and kills slaves as needed. It checks the number of read requests received every two minutes; if it is between 0 and 20, one slave is allowed to run, if it is between 21 and 40, two are allowed to run, and so on. This polling is performed in another thread, so that it doesn't interfere with the normal functioning of the application. A variable indicating the number of requests is updated by the function whenever a call is made to the database read API. This tells the scaling function how many requests were received in the previous two-minute interval, and it computes the number of slaves required accordingly. It then brings up more slaves, or kills existing slaves (based on the same algorithm as the fault-tolerance mechanism) as per the difference between the existing number of slaves and the required number of slaves. This thread is killed as soon as the application stops running.

TESTING AND CHALLENGES FACED

We used Postman for unit and end point testing. We leveraged the use of collections and shared collections to ensure that there were automated test URL updates.

One of the primary challenges we faced was finding out how each of the tools we were meant to use worked, and how they fit in with our application. It required extensive research, and many calls to friends who had finished the part we were working on.

We also had problems with writing to the databases, such as when the /rides* APIs were called. It updated the databases three time. We fixed it by debugging our APIs, and modified them slightly to allow for the extra containers, which were causing issues due to the way the ride_id files were stored. Another write error occurred when we started both queues for both the master and the slave; this caused our application to treat both identically.

There were a number of other challenges we faced with queues and message passing, a lot of them to do with connections and pika exceptions. We solved these issues by going through the code together, with a fine-toothed comb, which allowed us to point out things the rest of our teammates had missed.

Despite multiple reviews of the code, and the AWS setup, we weren't able to successfully submit our code for days. We repeatedly received 'Load balancer not working' error message, despite our application running comfortably on Postman, and returning accurate status codes. The load balancer setup is the same as the one used for the previous assignment, and as we got full marks for that one, we failed to see what the problem is. Setting up a new load balancer with fresh target groups didn't work either. However, the data clear API worked, and we were able to view the requests on our terminal, which

showed that the orchestrator and the end points were working well. Later, we realized that we'd made a very basic mistake, by using the old database. Once we rebooted the database, the load balancer worked perfectly, and we were able to pass all the test cases.

CONTRIBUTIONS

The four of us worked on each stage together; one or two of us worked on the basic code for a particular functionality, and then the others added improvements. Errors were debugged as a team, via conference call.

CHECKLIST

	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