17CS352:Cloud Computing

Class Project: Rideshare

Servicing Read Heavy Applications Using Flask, Docker, RabbitMQ, and ZooKeeper

Date of Evaluation:

Evaluator(s):

Submission ID: 265

Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Pratik Byathnal	PES1201700272	G
2	Sanjay Chari	PES1201700278	G
3	Aditya Shankaran	PES1201700710	G
4	Ziyan Zafar	PES1201701910	G

Introduction

The project deals with servicing a read heavy application using Flask, Docker, RabbitMQ, and ZooKeeper. There are two entities in the database of the project : users and rides. We programmed various RESTful API endpoints to access and modify the information related to these two entities.

Our infrastructure consists of an AWS application load balancer, that forwards incoming requests according to the path of the request to the respective target group. We have two target groups : the users instance and the rides instance. The API endpoints on these target groups send database read/write requests to the database orchestrator instance. RabbitMQ, Docker SDK are used for scale in/scale out in the Docker instance; and Zookeeper is used for high availability.

Related work

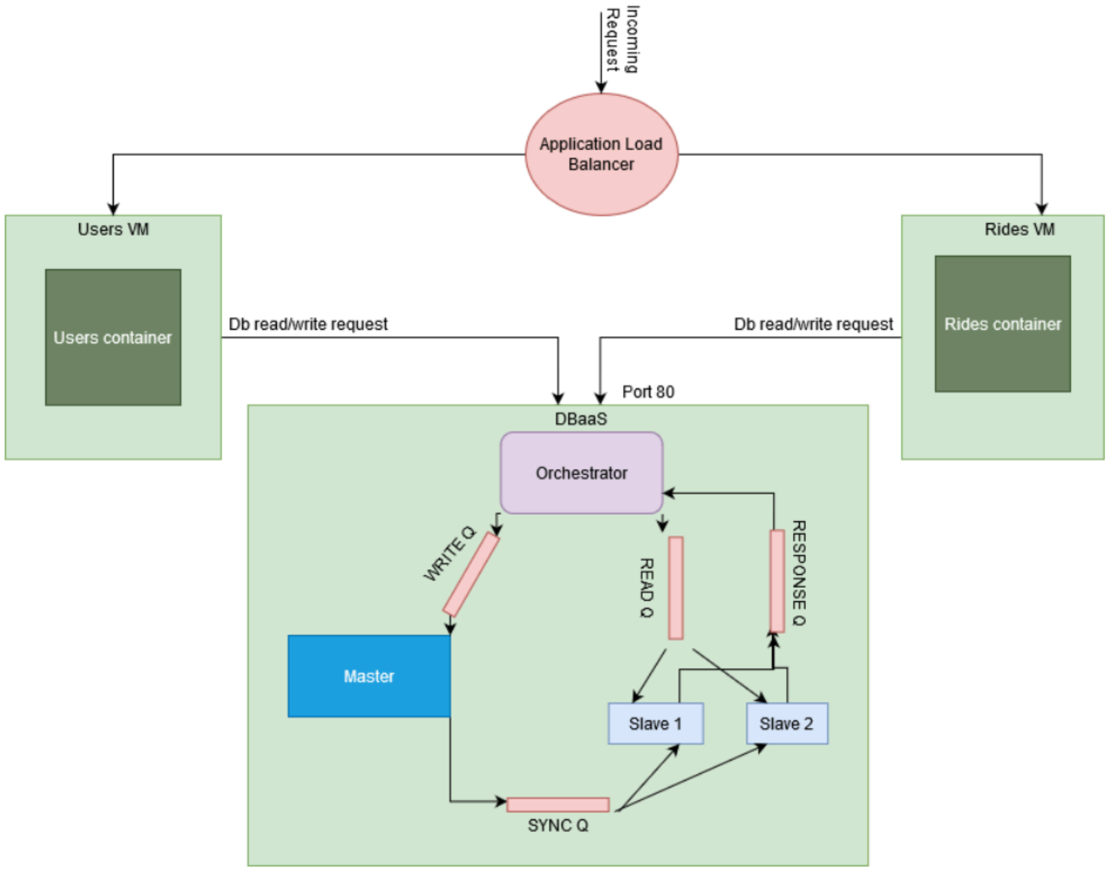
We referred to the official documentation of the software that we utilized in the project. Relevant links are listed below :

- https://www.rabbitmq.com/tutorials/tutorial-one-python.html
- https://www.rabbitmq.com/tutorials/tutorial-two-python.html
- https://www.rabbitmq.com/tutorials/tutorial-three-python.html
- https://www.rabbitmq.com/tutorials/tutorial-six-python.html
- https://docker-py.readthedocs.io/en/stable/
- https://docker-py.readthedocs.io/en/stable/containers.html
- https://kazoo.readthedocs.io/en/latest/

ALGORITHM/DESIGN

Our infrastructure consists of an AWS application load balancer, that forwards incoming requests according to the path of the request to the respective target group. All requests with path /api/v1/users are forwarded to the users target group; and all the other requests are forwarded to the rides instance. We have two target groups : the users instance and the rides instance. The API endpoints on these target groups send database read/write requests to the database orchestrator instance.

In the database orchestrator instance, worker containers take care of read/write to the database. Since we are servicing a read heavy application, it is assumed that a single master container for write to the database is sufficient, and slave containers are slave containers are scaled in/out depending on the number of incoming read requests to the database. To achieve this infrastructure, we utilize RabbitMQ with the following layout.



The database orchestrator consists of 5 containers running initially : RabbitMQ, Zookeeper, master container, slave container, and the orchestrator container.

We used pika to connect to our RabbitMQ container from the orchestrator container.We followed the official examples on the RabbitMQ website to achieve the above design. For scale in/out, we have a manager thread that runs for the entire duration of the orchestrator life cycle. In this manager thread, we utilise Docker SDK to create/kill containers according to the number of incoming read requests in the past 2 minutes.

For high availability using Zookeeper, we made use of a manager thread; along with the watch APIs that are built in to Kazoo. We connect to the Zookeeper container from the orchestrator container using the Kazoo client. We ensure two paths "/workers/slaves" and "/workers/master"; to which children are added and deleted according to the list of containers at that point of time. In the manager thread, creation and deletion of znodes is taken care of; according to the container list. If a container is killed at any point of time apart from auto scaling, the Zookeeper manager thread detects it as a crash.

- If the master crashes, the slave with the lowest pid is elected as master; and a new slave is created. To implement the transition from slave to master, we utilised two methods : change of environment variable, and container rename. When a slave is elected to master, we changed the value of the IS_MASTER environment variable to True and renamed it to <slavename>-MASTER. In the worker code, we check for the name of the worker container; and run the master code if MASTER is present in its name.
- If a slave crashes, a new slave is created.

TESTING

We tested the API endpoints using Postman. To test for scale in/scale out, we sent multiple curl requests using the subprocess module in Python. To test for high availability, we called the crash endpoints from Postman.
On the beta submission portal, we faced the following challenges.
1. When we first submitted, we got 2.0. We realized the bug was in handling 400 status code(We were not sending these requests to the orchestrator instance), and corrected it accordingly.
2. Second submission : We got 9.0. We realized the bug was in gandling 204 status code on the rides instance, and fixed it accordingly.
3. We got 10.0 on the third submission.

CHALLENGES

We faced various challenges at different stages of the project :

- RabbitMQ :
    - When programming the READQ/RESPONSEQ RPC, the orchestrator instance would become unresponsive. We realised this was because start_consuming is a blocking call, and were able to fix it by closing the connection after consuming the response from RESPONSEQ.
    - When a new slave is spawned, it wouldn't receive messages from the SYNC fanout exchange, as the messages had already been consumed. To solve this, we attached another queue to the SYNC fanout exchange, apart from the multiple slave queues. The messages from this other queue are consumed in the orchestrator, and stored in a list. We implemented an additional API in the orchestrator to return the content of this list; and called this API in the slave code; to get the content of the SYNCQ.
- Docker SDK :
    - Without the network parameter to the client.containers.run() command, we would get pika connection errors. We fixed this by configuring the network correctly.

    - Without the detach = True parameter to the client.containers.run() command, the orchestrator would become unresponsive. We fixed this bug by including the parameter.
- Zookeeper :
    - We were calling zk.create() without first calling zk.ensure_path(). We realized our error and fixed it.
    - During scaling down, killing of a slave should not be counted as a crash. To achieve this functionality, we utilised a mutex variable that would prevent the crash registration code from being executed during autoscaling.

Contributions

- Pratik Byathnal : ReadQ, ResponseQ, SYNC fanout exchange.
- Sanjay Chari : High Availability using Zookeeper(Master and slave leader election), AWS load balancer and target groups setup.
- Aditya Shankaran : WRITEQ, Scale in using Docker SDK.
- Ziyan Zafar : Scale out using Docker SDK, Ensuring that new slave receives SYNCQ content, Testing.

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