17CS352:Cloud Computing

Class Project: Rideshare

REPORT SUBTITLE

Date of Evaluation: 18/05/2020
Evaluator(s): Prof. Srinivas K S
Submission ID: 1438
Automated submission score: 9.0

SNo	Name	USN	Class/Section
1	Bhargavi Kumar	PES1201701802	6 'A'
2	Navneetha Rajan	PES1201700161	6 'A'
3	Sirisha Lanka	PES1201700294	6 'A'
4	Sukanya Harshvardhan	PES1201700214	6 'A'

Introduction

This project aims to develop a completely cloud-based back-end implementation which is used by our RideShare application via RESTful APIs. Through these APIs, the application allows addition of new users, rides and allows deletion of the same. It uses three microservices: rides-as-a-service, users-as-a-service and database-as-a-service. All RideShare requests are sent to either rides or users via the AWS Elastic Load Balancer. If either of these services need to perform database operations, they make the appropriate API call to the database service. The database service is highly scalable and has a clustered implementation. The cluster is managed with the help of Zookeeper and the nodes in the cluster communicate via AMQP-based Rabbitmq.

Related work

As Zookeeper and Rabbitmq were frameworks that were completely unknown to us, we had to perform an extensive research on the related python libraries; namely kazoo and pika.

Kazoo:

https://kazoo.readthedocs.io/en/latest/api/client.html

https://kazoo.readthedocs.io/en/latest/api/recipe/watchers.html

Pika:

https://pika.readthedocs.io/en/stable/modules/channel.html

https://pika.readthedocs.io/en/stable/modules/connection.html

https://pika.readthedocs.io/en/stable/modules/adapters/blocking.html

To check the TCP port connection to pika before running the orchestrator or slaves, we used an open source script.

https://github.com/vishnubob/wait-for-it

To execute docker related operations, we made us of Docker's python sdk.

https://docker-py.readthedocs.io/en/stable/containers.html

To grasp a better understanding of pika, we went through the rabbitmq python tutorials.

https://www.rabbitmq.com/tutorials/tutorial-one-python.html

To crash a container, we didn't use any docker sdk. Instead we referred to:

https://webkul.com/blog/docker-container-will-automatically-stop-run/

To isolate database operations from the rides and user services, we chose to switch from using flask-sqlalchemy to sqlite3.

https://docs.python.org/3/library/sqlite3.html

For the purpose of constantly checking the scaling condition, the advanced python scheduler library was used.

https://apscheduler.readthedocs.io/en/stable/modules/schedulers/background.html

https://apscheduler.readthedocs.io/en/v2.1.2/modules/scheduler.html

To debug our errors, we frequently referred to stack overflow, stack exchange, github community, docker forums and apache documentation (https://issues.apache.org/).

For guidance on certain implementation strategies and error resolutions, we also referred to Piazza.

ALGORITHM/DESIGN

Master Election: Each container creates a znode under '/workers/nodes' as soon as it is spawned and checks if the master exists under 'workers/master/'. If the master does not exist, it selects itself as the master by creating the master node and deleting the previous slave node. Each worker sets a watcher on their associated znodes so as to handle failure of the znode (in which case the entry-point process in the worker exits). On starting the entire DBaaS application, one worker container is built and run using docker-compose. This container elects itself as the first master. We went ahead with this implementation on the assumption that the master would not fail.

Scaling Out: The orchestrator runs a thread in the background that checks the number of database read requests that have been made within each minute. Depending on the number of slaves available and the number of slaves required (which increases by one for every 20 requests increase in number of database read requests), new slaves are appropriately spawned from an existing image in the orchestrator using docker python sdk.

Fault Tolerance: Each slave container has a watcher function on all the other slaves in the '/workers/nodes/' path. If at any a point an event occurs that leads to the number of children of the path being less than the number of children last recognized by the slave, new slaves are spawned from an existing image in the slave using docker python sdk.

Queues: The orchestrator publishes appropriate messages onto the queue on receiving database requests and waits for a reply. If the id of the worker matches the master's id, it starts consuming from the write queue. Otherwise, it starts consuming from the read and sync queues. If it is a write request, the worker forwards the message to the sync queue, which is bound to a fanout exchange. The worker returns the response from the respective database operations to the orchestrator with the help of a correlation id.

TESTING CHALLENGES

- Our initial implementation of database operations included the use of flask sqlalchemy libraries, which worked fine as the operations were performed by a process with an application context (Flask app). For our newer implementation, we decided to switch to using sqlite, which consequentially lead to changing the implementation of the necessary classes and their methods.
- Load Balancer connection issues: solved by changing the localhost address from 127.0.0.1 to 0.0.0.0
- Port Connection issues in pika and kazoo: issue solved by deleting all exited containers and dangling images. Ensuring there is sufficient delay between starting the zookeeper and rabbitmq servers and running the orchestrator and slaves.

CHALLENGES

- The design of the leader election process was quite challenging. Since in the end, the goal was to select one leader for the purpose of database write operations, any slave was selected as long as a master did not exist.
- Handling crash APIs, which signified crashing of workers, was a puzzle. Since our implementation doesn't make use of PIDs, we chose to exit the process that handles database operations in the worker containers. With no process running in the foreground, docker assumes the application is stopped and shuts down the container.

Contributions

Bhargavi – Zookeeper integration, fault tolerance, scaling out, debugging

Navneetha – Rabbitmq integration

Sirisha - API integration in orchestrator and worker responses, database replication

Sukanya – Database implementation and the necessary integration between slave RPC and database operations

CHECKLIST

SNo	Item	Status
1.	Source code documented	
2	Source code uploaded to private github repository	
3	Instructions for building and running the code. Your code must be usable out of the box.	