17CS352:Cloud Computing

Class Project: Rideshare

CC_0107_0175_1133_1501

Date of Evaluation: 18/05/2020
Evaluator(s): Prof. Venkatesh Prasad
Submission ID: 578
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	P R Aravind Perichiappan	PES1201700107	6F
2	Aditya Prasanna	PES1201700175	6F
3	Nilay Gupta	PES1201701133	6F
4	Shubham Hosalikar	PES1201701501	6F

Introduction

- We have designed a fault-tolerant, highly available database as a service for a ridesharing application.
- The application consists of two services hosted on two AWS EC2 instances – a Users service and a Rides service.
- The Users service does work related to registering, removing and managing the users of the RideShare application.
- The Rides service does work related to creating, deleting and managing the details of rides for the application.
- The goal of this application is to enable users to create rides and have other users join these rides.
- The Users and Rides service use the database as a service as the backend to store and retrieve data in a reliable and efficient manner.
- The database as a service is hosted on a separate AWS EC2 instance.
- The database architecture used is a master-slave architecture with a single master database and multiple slave databases. There is a worker associated with each database.
- The master worker services write requests and slave worker(s) service read requests.
- These workers are managed by an orchestrator container, in which a flask app is run. The orchestrator is responsible for communication between Users and Rides services and the database backend.
- Initially, only one slave worker runs. Based on the number of read requests received in a two-minute period, we scale out or scale in the number of slave workers.
- The workers and orchestrator communicate through an AMQP message broker, RabbitMQ.
- The slave workers' database is synced to the master's database on every write.
- The workers are watched for fault tolerance with the help of zookeeper.
- When a worker crashes, a new worker is started and all data is replicated asynchronously in it.

- When master crashes, a slave is elected as new master. Thus, the role of a worker can change at any point and this is also facilitated using zookeeper.
- Each of the workers run in a separate container. They contain a mongoDB database which is managed by a worker python script.
- The orchestrator manages the worker containers using the docker SDK for python.
- The orchestrator and workers also use pika for communication with RabbitMQ server and kazoo for communication with Zookeeper server.

Related work

- https://www.rabbitmq.com/tutorials/amqp-concepts.html - AMQP concepts
- https://www.rabbitmq.com/getstarted.html - RabbitMQ tutorials
- https://pika.readthedocs.io/en/stable/ - pika documentation
- https://docker-py.readthedocs.io/en/stable/ - Docker SDK documentation
- https://kazoo.readthedocs.io/en/latest/ - kazoo documentation
- https://pymongo.readthedocs.io/en/stable/ - pymongo documentation
- https://medium.com/greedygame-engineering/an-elegant-way-to-run-periodic-tasks-in-python-61b7c477b679 - for autoscaling

ALGORITHM/DESIGN

The orchestrator receives the read/write requests from the Users and Rides services and sends message to the appropriate worker. Write messages are sent through a direct exchange to the master's writeq. Read messages are sent through a direct exchange to the slaves' readqs in round-robin fashion. A fanout exchange is used to sync database of slaves with master. DB Clear API is also implemented using writeq.

Master election is done using zookeeper server which runs in its own docker container. Whenever there's a change in the state of running workers, the orchestrator goes through the pids of all running workers, chooses the lowest pid and sets it as the data of a znode. All of the workers watch this znode, and on any change in the znode, they compare their own pid with the pid in the znode. Based on this comparison, the workers assign themselves to be a master or slave.

For autoscaling, threading.Timer function is used to run a function every two minutes after the first read request. This function checks for number of running slaves, and based on number of read requests received in the two-minute period, it also calculates the number

of slaves that need to be running. Based on this, scaling out or in is performed by starting new slaves or crashing existing slaves respectively.

Fault tolerance is implemented using the zookeeper server. Every worker, during initialization, creates an ephemeral znode as a child of /workers znode. Since the znodes are ephemeral, when a worker crashes, its corresponding znode is also deleted. The orchestrator watches the /workers znode's children and on any change that was not due to scaling, it checks how many containers are running. If this number is less than the expected number (determined by scale factor of slaves), then new workers are started.

Data replication is performed by using a common database which stores the writes done by the master. When a new slave starts up, it reads this database and replicates the data as required.

TESTING

It was difficult to debug issues with the orchestrator instance as the logs were spread out over the various containers. To overcome this, we used lazydocker, which helped in viewing the logs and the configuration of all containers in an easy-to-use terminal interface. We tested the APIs using Postman.

We did not face any issues on the automated evaluation of our submission.

CHALLENGES

- For master election, we had to obtain the pids of docker containers from within the containers. To solve this, we mounted the docker binary and docker.sock socket file of the instance into the containers and used these to find the pids.

- We had to design the worker.py file such that the worker could be converted between slave and master roles at any point instantly. The code for both slaves and master was kept together in a single file to achieve this.

- We were not able to cancel/close queues/channels while converting a worker from slave to master or vice-versa due to some errors that we couldn't debug. Hence, all workers have a readq, a writeq and a syncq irrespective of their role of master or slave. We used routing keys that were set to the pids of each worker to route the messages from the orchestrator to the appropriate queues.

- Due to the read queues being in all workers, we had to manually implement a round-robin way of sending the messages to the slaves.

- We wanted the database to be in the same container as its corresponding worker in order to make container management simpler. But in mongodb image, only mongod can run as the top process. So, we used a shell script that ran the python command to start the worker after mongo started running. This shell script was stored in /docker-entrypoint-initdb.d/ directory of the container and it would automatically run on database initialization for the first time.

- While scaling down, we had to ensure that the fault tolerance is disabled temporarily. For this, we used a global flag that was set while scaling.

Contributions

Aravind – PES1201700107

Master election, fault tolerance (slave + master) using zookeeper, and docker containers configuration.

Aditya – PES1201700175

Worker management in orchestrator, and reading & writing using RabbitMQ server to communicate between orchestrator and workers.

Nilay – PES1201701133

Data replication in new workers, sync using RabbitMQ server, and integration of DBaaS with Users and Rides services.

Shubham – PES1201701501

Autoscaling using threading, and orchestrator APIs to list workers & crash master or slave workers.

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.	✔