UE17CS352: Cloud Computing

Class Project: Rideshare

Rideshare App on AWS

Date of Evaluation: 19/05/2020
Evaluator(s): Pushpa and Nithin
Submission ID: 570
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Chandan M	PES1201701549	D
2	Samarth S	PES1201701359	D
3	Zenkar S	PES1201701532	C
4	Dheeraj Gharde	PES1201700075	D

Introduction

The project is based around building a fault-tolerant, highly available database as a service for the RideShare application which we built as a part of previous three assignments. The RideShare Application helps users carpool with other users. They can create a redi from one location to another. Other users can join the ride. The application is built in containers and deployed in AWS.

The term “Database-as-a-Service” refers to software that enables users to setup, operate and scale databases using a common set of abstractions without having to either know or care about the exact specifications/implementations of those abstractions for the specific database.

DBaaS is often delivered as a component of a more comprehensive platform, which may provide additional services such as Infrastructure-as-a-Service. The DBaaS solution would request resources from the underlying IaaS which would automatically manage the provisioning compute, storage and networking as needed, essentially removing the need for IT to be involved.

A DBaaS solution provides an organization a number of benefits, the main ones being:

- Developer agility
- IT productivity
- Application reliability and performance
- Application security

Related work

The major technologies/software/modules that were used in order to build this project include:

Docker-https://docker-curriculum.com/

Docker SDK- https://docker-py.readthedocs.io/en/stable/

RabbitMQ- https://www.rabbitmq.com/getstarted.html

AMQP- https://www.rabbitmq.com/tutorials/amqp-concepts.html

Zookeeper-
https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php

https://www.tutorialspoint.com/zookeeper/zookeeper_overview.htm

Kazoo(Zookeeper client)- https://kazoo.readthedocs.io/en/latest/index.html

A lot of time was spent on learning and understanding these technologies as they were fairly new to all of us.

ALGORITHM/DESIGN

The specification requires us to continue using the Load Balancer and two VM Instances which houses a container for Rides and Users app. A third VM Instance is used as our DBaaS, which initially has five containers: **Orchestrator, RabbitMQ, Zookeeper, Master and Slave**.

Any requests sent to the DBaaS VM are received by a container which is named as the orchestrator. The job of the orchestrator is to publish the db requests into the respective queues, manage workers, auto scaling, implementing fault tolerance, and leader election.

The **Orchestrator** itself doesn't have a database. Each of the Workers (Master and Slave) have their own copy of the database.Orchestrator uses AMQP with RabbitMQ to distribute incoming db requests to master and several slave containers. Four Queues are used for this purpose, namely **ReadQ, WriteQ, SyncQ, ResponseQ**. Orchestrator publishes all db write requests to WriteQ. All db read requests are published to ReadQ using the RPC pattern, which waits for a corresponding reply from ResponseQ.

Master container consumes messages from WriteQ, and writes them to the database. Then the Master publishes those messages to SyncQ which is a Fanout exchange. SyncQ is used to synchronise the slaves' database with master. All incoming read requests in the orchestrator are saved in a .txt file. When a new slave is spawned, its database is synchronised using *get_sql_stmts* api in the Orchestrator, which retrieves the saved sql statements, all these statements are written into the database.

Slave container consumes messages from ReadQ and SyncQ. All messages from SyncQ fanout exchange are written into the Database.The requests from the ReadQ are consumed by slaves in Round-Robin fashion. All read db requests from ReadQ are executed on the database, and result is sent back on the ResponseQ.

Both master and slaves have the image called **Worker**. The Worker container has three files **worker.py, master.py, slave.py**. worker.py runs either master.py or slave.py using Subprocess. This enables us to change a Worker's function between Slave and Master. We have created a flask app which runs in the host VM. Here we have API's to kill a worker

container, spawn a new worker container, and get the process id of a container using Docker SDK. All workers spawned, are initially Slaves, until they are elected as Master. All Workers create ephemeral z-nodes, under the path `"/t",` containing the following data about the worker:

```
{"cont_id":Container ID, "proc_id":Container's Process ID, "master":True/False}
```

master field is initially False. Ephemeral z-node is a z-node which exists as long as the session which created it exists. So when a Worker dies, the z-node automatically gets deleted. Then it runs the slave.py using Subprocess.

Auto-Scaling is implemented using BackgroundScheduler, which calls a function (auto_scale) every 2 minutes. The scheduler is started after the first Read Request. Scaling Up/Down is performed based on the number of read requests received in the previous interval. The number of read requests is stored in counter.txt file. The auto_scale function performs the following tasks:

- It first gets the request count**.** Then resets the count to zero. It calculates the number of slaves required based on count, and stores it in the global variable **required_slaves.** Then it gets the list running workers and calculates the number of **current_slaves.**
- If required_slaves is greater than current_slaves, then it spawns needed number of new slaves.
- If required_slaves is lesser than current_slaves, then it kills the extra number of slaves in decreasing order of their pids.

Zookeeper is used to implement Fault Tolerance in Master and Slave. The ChildrenWatch and DataWatch functions are used for this. A **ChildrenWatch** is set on the root node "/t". Whenever there is any change in the number of workers(worker dies or new worker is added), the watch function is triggered which performs the following tasks.

- **Master Fault Tolerance - Leader Election -** It loops through the list of children and finds if a master znode exists. If there is no master, then it elects the slave with lowest pid as master, by setting its znode data to master=True. This causes DataWatch to be triggered in the newly elected worker.
- **Slave Fault Tolerance -** It checks if the current number of slaves is less than required_slaves. This condition arises when a worker fails, or **crash/master** or **crash/slave** is called. If so it spawns a new worker.

In every Worker, **DataWatch** is set on it's znode. Whenever there is change in data of its znode, the watch function is called and it first checks if data znode data is master. Then it kills the running **slave.py** subprocess and starts the **master.py** subprocess. Hence the Slave changes to **Master**.

TESTING

Most of the testing was done during the development stage, and any errors found were corrected there itself. Further Postman was used to requests in different scenarios to test the working of various features like load balancing, auto scaling, master and slave fault tolerance.

Beta Testing Portal helped us in testing the correctness of code before final submission. As There were no errors in logic and implementation at this stage.The Beta Test considered Invalid Read Requests to be counted as in the Orchestrator. But these Invalid requests were handled in the API's itself and appropriate responses were returned,and hence no further db read requests were made to the Orchestrator. This issue was resolved in the Final Test Portal, hence we received full marks in the second submission.

CHALLENGES

- Slave Replication was difficult as the suggested methods in the project specification (make message 'durable' or not sending 'ack' for consumed message) didn't help at all. This was solved by using a fanout exchange for SyncQ and saving write requests to a text file, which is read by a new slave to synchronize its database.
- Running the same code for both the master and the slave containers but changing the roles. This was overcome by creating a third program "worker.py" which uses subprocess to run slave.py and master.py.
- Using Zookeeper for High Availability, was a challenge due to lack of proper resources and tutorials to understand Zookeeper and Kazoo. Read the Kazoo Client docs and created simple programs using kazoo to understand usage.
- Performing Leader Election based on the PIDs was a challenge as it was difficult to get the PIDs of the containers.
- Faced problems while using Docker SDK inside the orchestrator container. Hence created a flask app that runs in the host, to use Docker SDK to spawn/kill workers, and get pid of containers.

Contributions

Chandan:

- Implemented Orchestrator APIs crash_master, crash_slave, list_workers.
- APIs to spawn a new worker container, kill a worker, get pid of worker using Docker SDK.
- Slave Fault Tolerance using zookeeper ChildrenWatch.
- Auto Scaling Worker Containers using Background Scheduler.

Dheeraj:

- Implemented Orchestrator APIs read_db, write_db, clear_db using RMQ Queues.
- Master Fault Tolerance using zookeeper ChildrenWatch (Leader Election).
- Changing Slave to Master using zookeeper DataWatch after Leader Election.

Samarth:

- Implemented the functionality of master worker.
- RMQ queues WriteQ, SyncQ fanout exchange.
- Integrating with Rides and Users instances, AWS setup, Load Balancing.

Zenkar:

- Implemented the functionality of slave workers.
- RMQ queues ReadQ, ResponseQ (RPC), SyncQ fanout exchange.
- Slave Replication - Synchronizing database of new slave.

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