

UE17CS352: Cloud Computing

Final Class Project: Rideshare

Date of Evaluation: 16-5-2020

Evaluator(s): MS. NISHITHA R & MR. VINAY BANAKAR

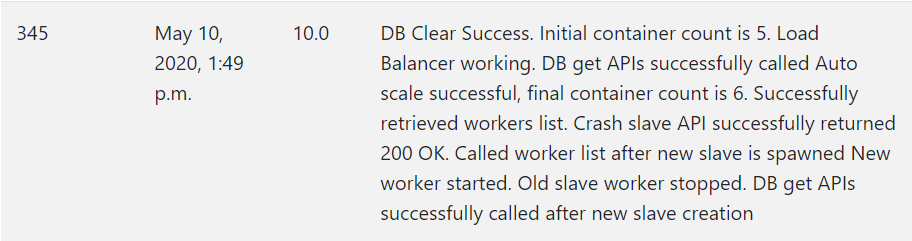
CC Team Name: 345

Submission ID: CC\_0227\_1123\_1139\_1526

Automated submission score: 10.0

GitHub source code: <https://github.com/madhavmk/Cloud-Computing-Course>

|  |  |  |  |
| --- | --- | --- | --- |
| **SNo** | **Name** | **USN** | **Class/Section** |
| **1** | Madhav Mahesh Kashyap | PES1201700227 | 6 H |
| **2** | Sangam Kedilaya | PES1201701139 | 6 H |
| **3** | K Sachin Nayak | PES1201701123 | 6 C |
| **4** | Anirudh Avadhani | PES1201701526 | 6 E |



## Introduction

In this report, we present our team’s approach to the final project of the Cloud Computing Course (UE17CS352). The final project built upon our learnings from the previous 3 assignments. This project reinforced our learnings on multiple concepts:

* REST paradigm
* Microservice architecture
* Python Flask to serve API endpoints
* PostgreSQL Database to store persistent data
* AWS Instances, Load Balancers, Elastic IPs, Virtual Machines, Target Groups etc.
* Docker Containerization
* RabbitMQ as an AMQP message broker (Python Pika library)
* Apache Zookeeper to maintain centralized information in distributed systems (Python Kazoo library)

In the assignments, we built a cloud server that could serve “User” requests and “Ride” request from different IP addresses using a Load balancer to distribute traffic. The “User” and “Ride” instances ran Flask and Postgresql DB using internal docker containers. Until Assignment 3, the “Rides” and “User” containers used their own separate databases.

In the project, we improve upon the microservice design further. The ”Users” and “Rides” microservices will no longer be using their own databases and will instead use the “DBaaS service”. The DBaas service to be built has to be scalable, fault tolerant and highly available.

## Related work and References

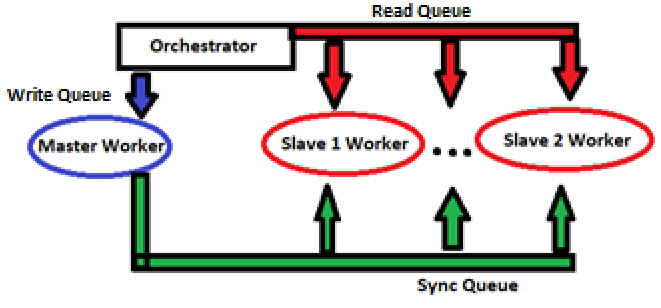
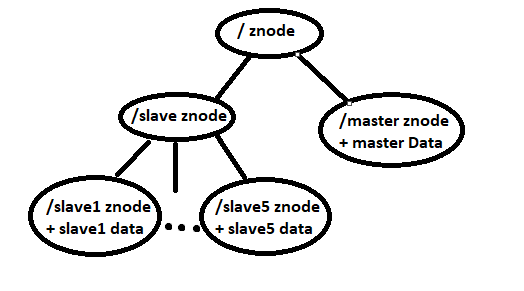
Kazoo and Zookeeper Resources:

* <https://kazoo.readthedocs.io/en/latest/>
* <https://readthedocs.org/projects/kazoo/downloads/pdf/2.5.0/>

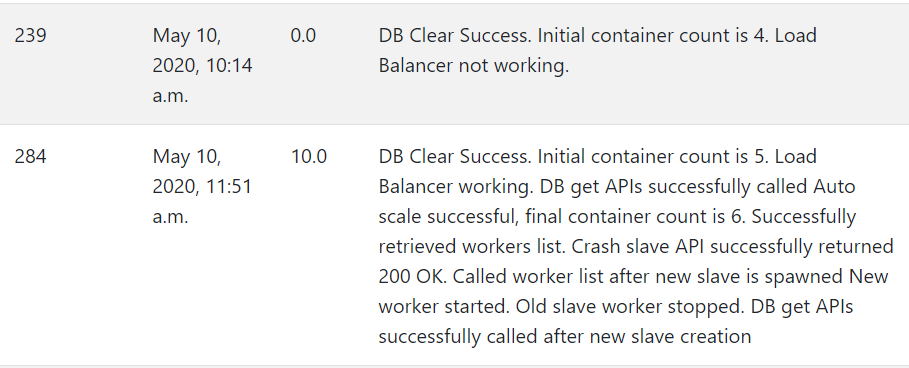
RabbitMQ Resources:

* <https://www.rabbitmq.com/getstarted.html>
* <https://levelup.gitconnected.com/rabbitmq-with-docker-on-windows-in-30-minutes-172e88bb0808>
* <https://github.com/cagrias/django-rabbitmq-celery-docker-example>
* <https://medium.com/@tonywangcn/how-to-build-docker-cluster-with-celery-and-rabbitmq-in-10-minutes-13fc74d21730>

## Algorithm and Design

* Pika Rabbit MQ Queues Setup:  
    
  There are 3 queues; Read. Write and Sync. All 3 queues use Remote Procedure Calls (RPCs) to send and receive information from the workers. Fanout exchange is used in Sync queue to send sync messages to all slaves. I used the Python library “Pika” to setup all the RabbitMQ message brokering and queue connections. We first load all the input parameters of the request to “byte” data type using the “Pickle” python library. Read requests are sent to Slave queue. Write requests are sent to Master queue. When a write message response is returned from the master worker, the same message is also sent to the Sync queue, so that there is data consistency between the slave and master workers.
* Worker configuration:  
  Each worker is a docker container containing all the Python dependencies. There are 2 types of workers: masters and slaves. The master worker handles write requests only and maintains a connection to only the Write queue. There are multiple slaves that are updated using the “eventual consistency” model. The slave worker maintains a connection to both the Read queue and the Sync queue. The Read queue services read requests, while the Sync queue helps to maintain data consistency between the master and slave workers. The worker code in “rpc\_server\_databse.py” contains code for both master and slave. If the file is run with command line argument flag as 0, worker runs master code. If flag is 1, worker runs as slave code.
* Auto scale of workers:  
  The number of slave workers servicing Read requests is directly proportional to the number of Read requests. Every additional 20 requests every 2 minutes require an additional slave worker. Similarly, the number of workers needed can reduce if the rate of requests reduces. Addition/Deletion of slave workers requires addition/deletion of children in the “/slave” znode and starting/stopping slave docker containers.
* Kazoo Zookeeper Znode Structure:  
    
  The Python library “Kazoo” is used to handle the zookeeper nodes running on the orchestrator instance. There is 1 znode “/master” to store data on the master worker. This contains information about the master worker PID, worker container ID and worker container name. There is also a “/slave” znode that is a parent node to all the running Slave workers. Each child of the “/slave” node contains information about that particular slave’s PID, container ID and name. The children of “/slave” znode are dynamically added/deleted when slave workers are started/stopped.
* Master Leader Election:  
  A Zookeeper Data Watch is placed on the “/master” znode. Whenever the “/crash/master” API purposely stops a master worker, the master data watch is triggered. This triggers a function that reads all the slave data from the children of “/slave”. It finds the slave with the lowest PID and converts that to master by restarting the Python process with 0 flag. “/master” data is updated with the new master’s info. Another new Slave container is spawned to take up the newly converted master’s place.
* Slave Leader Election:  
  A Zookeeper Children Watch is placed on the “/slave” znode. Whenever the “/crash/slave” API is called, all the children of the slave znode are scanned to check for the highest PID. Then, the highest PID slave container is stopped and the corresponding slave child znode is deleted. A new slave worker and znode is brought up to replace the crashed slave.

## Testing Issues

* Since Kazoo Zookeeper uses asynchronous calls, some of the lengthy function calls used to finish in unpredictable timings. This caused issues with the next code instructions running even though the function calls didn’t end. We fixed this by adding sleep timers in the Python code, so that the code did not go to the next step until we were sure that the asynchronous call finished.
* Initially the Automated testing gave us a score of 0, citing a “Load Balancer not working”. However, the load balancer worked upon sending POSTMAN requests. The issue was fixed by deleting the old Load Balancers, Target Groups and Rules, and then creating a fresh new set of Load Balancers, Target Groups and Rules. Upon doing this, we immediately received a **10 score** on the automated testing. 

## Challenges

* Poor documentation for the Python Kazoo library used for Zookeeper setup.
* Majority of Apache Zookeeper tutorials used JAVA, while we used Python.

## Contributions

|  |  |  |
| --- | --- | --- |
| **Name** | **USN** | **Contribution** |
| Madhav Mahesh Kashyap | PES1201700227 | Orchestrator, RabbitMQ queue, Zookeeper setup, Worker code, AWS Instance and Load balancers |
| Sangam Kedilaya | PES1201701139 | Worker code, AWS Instance and Load balancers |
| K Sachin Nayak | PES1201701123 | Worker code, AWS Instance and Load balancers |
| Anirudh Avadhani | PES1201701526 | Worker code, AWS Instance and Load balancers |

## Checklist

|  |  |  |
| --- | --- | --- |
| SNo | Item | Status |
| 1 | Source code documented | Done |
| 2 | Source code uploaded to private GitHub repository. (Also gave access to the evaluators) | Done (<https://github.com/madhavmk/Cloud-Computing-Course>) |
| 3 | Instructions for building and running the code. Your code must be usable out of the box. | Done  (Present in the README file of the repository) |