

17CS352:Cloud Computing

Class Project: Rideshare

REPORT SUBTITLE

Date of Evaluation: 20-05-2020

Evaluator(s): Prof K Srinivas

Submission ID:1043

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Name | USN | Class/Section |
| 1. | Samhitha D | PES1201701274 | E |
| 2. | Meishty Pande | PES1201700224 | E |
| 3. | Sampreetha V | PES1201701542 | E |
| 4. | Rashmi | PES1201700196 | E |

# Introduction

## The project was based on building a fault tolerant, highly available database as a service for the RideShare application which we had developed initially through our assignments. We had to enhance the exiting DB APIs and implement a custom database orchestrator engine to listen to incoming HTTP requests from users and rides microservices and perform the database read and write according to the given specifications.

## 

## Related work

There were several online sources that we referred to including the hands on sessions that were conducted. Few of the links are mentioned below-

​http://zookeeper.apache.org/

​<https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php>

https://www.rabbitmq.com/getstarted.html​

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

​https://hub.docker.com/\_/zookeeper/ RabbitMQ

​https://kazoo.readthedocs.io/en/latest/

​https://docker-py.readthedocs.io/en/stable/

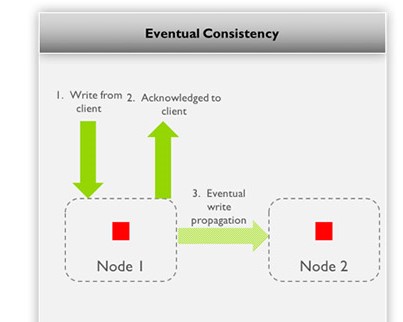
## ALGORITHM/DESIGN

We followed the same design specified in the pdf to construct the outline of our project

We had the users and rides in 2 different containers with the load balancer set up as per the previous assignment. Each dbwrite and dbread requests were directed to the DBaas orchestrator (port 80) which further sends writeQ to master and read/response to slave. Communication between them happens through syncQ.

Docker-compose was used to orchestrate the containers.

We used an eventual consistency model which is basically used to achieve high availability and if no new updates are made to a given data item, eventually all accesses to that item would return the last updated value. The design is as below-



Scalability is done for only the read apis i.e the slave and not the master. We setup a counter reset for every 2 minutes and directs requests accordingly.

## TESTING CHALLENGES

We encountered several challenges as it was a completely a new domain to work on. Figuring out how the EC2 instances worked, fixing the clearDB API, writing the code for AMPQ and orchestrator was quite challenging in the beginning.

For example, we encountered a few bugs like the db read API did not publish any response to the responseQ. (so we had to add connection.close() after publishing)

Unfortunately, we couldn’t beta test our project due to connectivity issues.

Also, communication and coordination between team mates was problematic due to network issues.

## Contributions

Samhitha D- Rabbitmq part

Meishty pande- master and slave workers

Sampreetha- also on orchestrator part

Rashmi- hosting on aws, setting up load balancer and uploading it for online evaluation

## CHECKLIST

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