

17CS352:Cloud Computing

Class Project: Rideshare

A HIGHLY SCALABLE MASSIVE PARALLEL RIDESHARING PLATFORM

Date of Evaluation: 15th May, 2020 11:00 AM

Evaluator(s): Prof. Venkatesh Prasad

Submission ID: XXXX (TBD)

Automated submission score: 5

|  |  |  |  |
| --- | --- | --- | --- |
| SNo | Name | USN | Class/Section |
| 1 | Hrishikesh V | PES1201700276 | VI-C |
| 2 | Varun R Gupta | PES1201700652 | VI-C |
| 3 | Ravendra Singh | PES1201700706 | VI-C |
| 4 | Aekansh Dixit | PES1201701808 | VI-C |

# Introduction

* Rideshare is a highly scalable massively parallel ridesharing platform that can be used to schedule rides that are shared between multiple users going along the same route at the same time.
* Users can access the application with public IP addresses and further more access each functionality of the application (Creating a ride, user etc.) by specifying the URI of the APIs
* To simplify the interaction of the users with the APIs, we used REST to create simple user interfaces.
* The primary goals of the project were to build an application that is –
* easily accessible
* independently scalable by using microservices as opposed to monolithic
* handles automatic distribution of incoming traffic with load balancers
* highly available and fault tolerant.
* The application, being a cloud based service, is deployed on AWS
* The basic functionalities include creating/deleting users, creating/deleting rides, fetching rides, users and integrating these operations with the database, which is implemented with MongoDB
* The features that we’ve used to build this project are
* Node JS (Express) as the basic backbone of the project. Every API is implemented with Express
* Mongo DB is used to store the data related to the application
* Docker is used to split the monolithic application into micro services that run on separate containers
* AWS Application load balancer is needed to route incoming data to relevant containers when accessed by the same IP Address
* RabbitMQ was used to build an orchestrator that set up a master slave architecture to handle database operations
* Zookeeper was implemented for fault tolerance (Bonus)

## Related work

We have referred to the following sites to find the required documentation or instructions to get us started with the technologies used in this project:

* <https://zookeeper.apache.org/>
* <https://www.npmjs.com/package/node-zookeeper-client>
* <https://www.rabbitmq.com/documentation.html>
* <https://www.rabbitmq.com/tutorials/tutorial-one-java.html>
* <https://www.npmjs.com/package/amqp>
* <https://nodejs.org/en/docs/>
* <https://docs.docker.com/>
* <https://www.npmjs.com/package/dockerode>

## ALGORITHM/DESIGN

### Backend Framework

Node JS (Express) was used to build the backend of the application. The reason for doing this was the plethora of specific networking/cloud related modules such as dockerode and amqplib that were available to us to use. These modules reduced the burden on internal implementation of the services and made it easier to focus on the design.

Node JS helped in improving the readability of the code with the use of a router. The router helped split the functionalities into different files for independent testing and implementation.

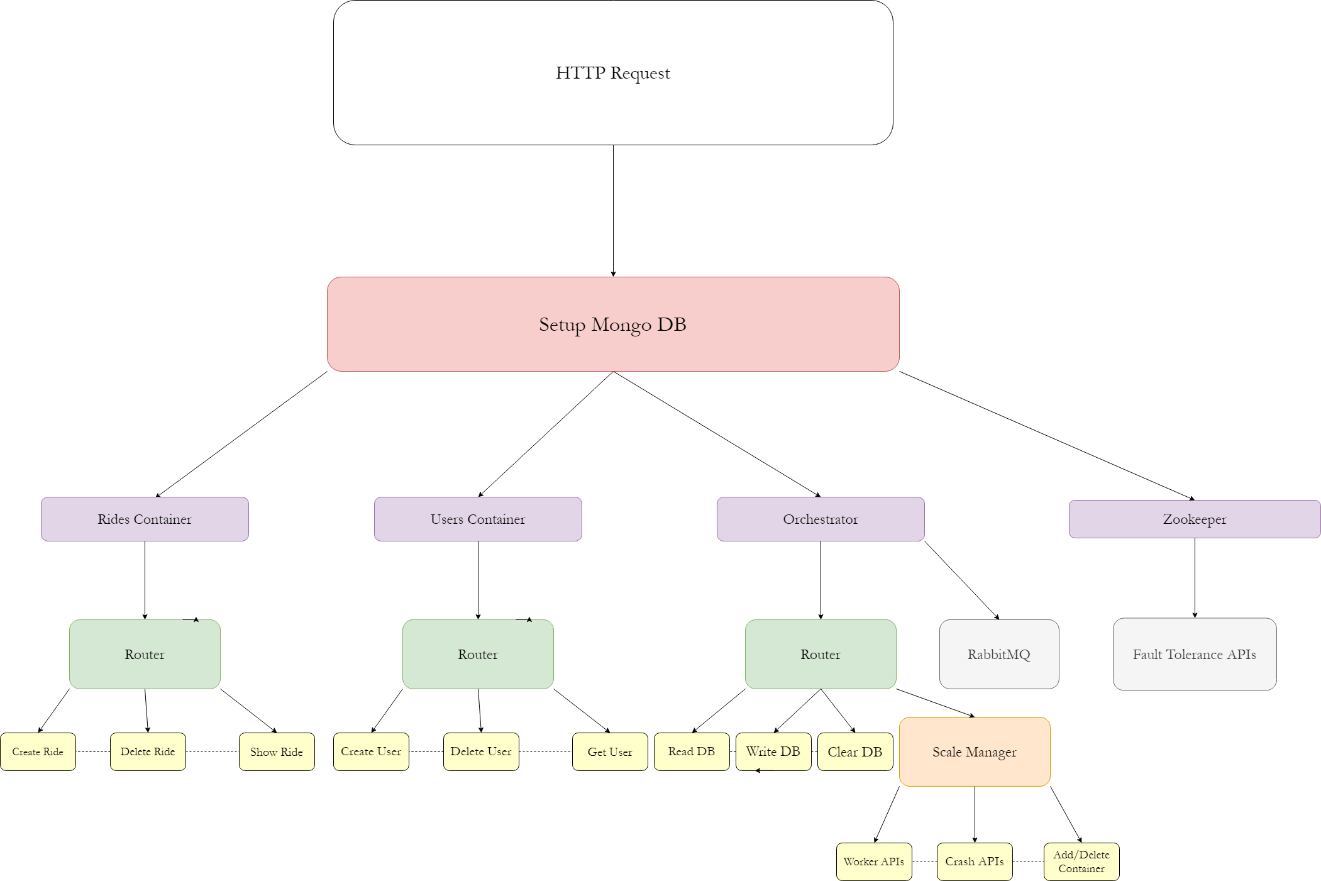
In the below diagram, upon receiving a request, the first task of the application is to set up a Mongo DB database, if not present already.

The router then routes the request to the appropriate API, which may reside anywhere in the container. Care must be taken to ensure that appropriate routes are first declared in the router. Once done, each API is completely sealed from the others in terms of run time requirements.

**A very strict requirement was to handle all possible situations that the API might face.**

Some common situations are:

* Internal Errors (Database Failures more often than not)
* Forbidden Requests
* Improper Input
* Accessing data that isn’t present
* Calling create APIs with wrong input

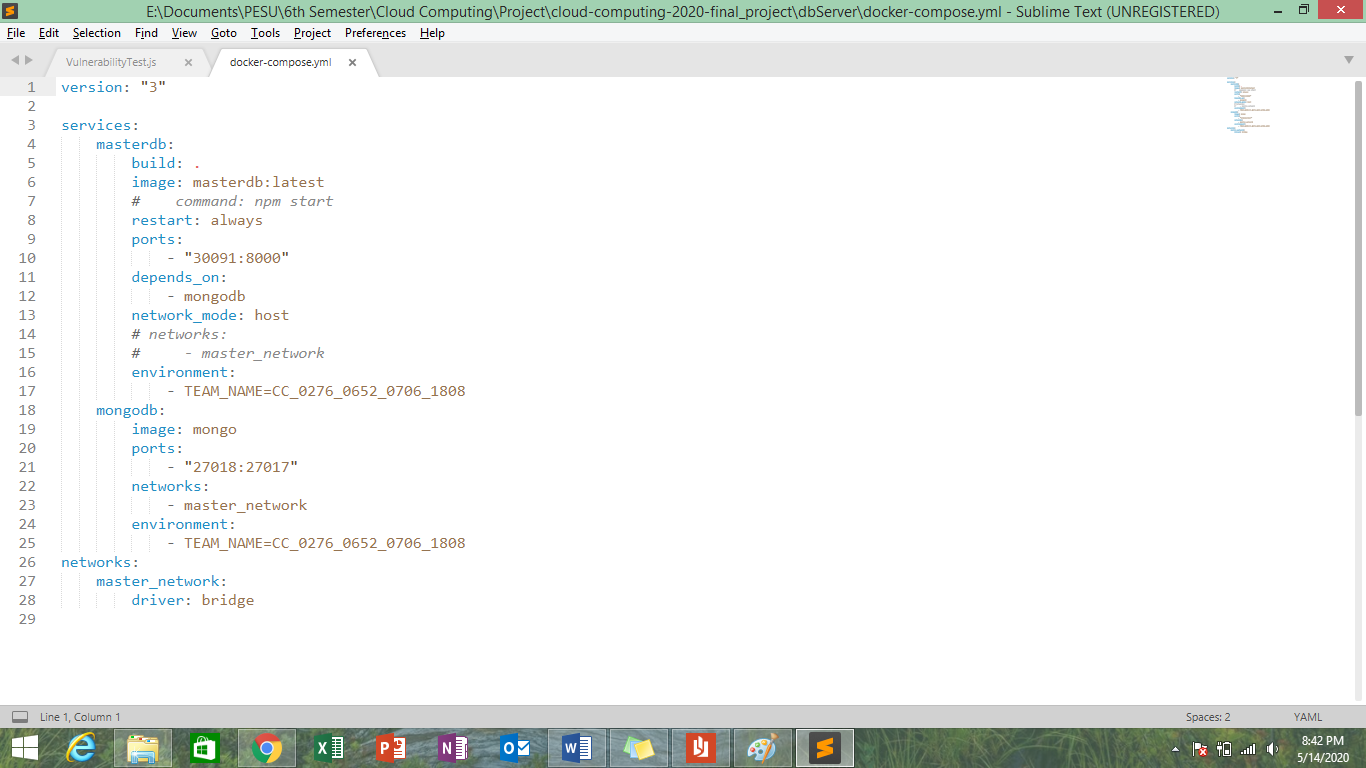


In the above image, each of the APIs, denoted in yellow color, can be independently implemented and tested. Routers ensure that none of the APIs need be in the same file.

### Monolithic to Micro service

Docker is a set of PaaS functionalities that uses OS-level Virtualization to create containers (Wikipedia). A container is an independent executable unit. The process of creating a micro service involved splitting users and rides into two groups and bundling each within a container.

Initially, Docker requires specifying the application’s services, network requirements, database specifications and dependencies. The compose file looks something like this



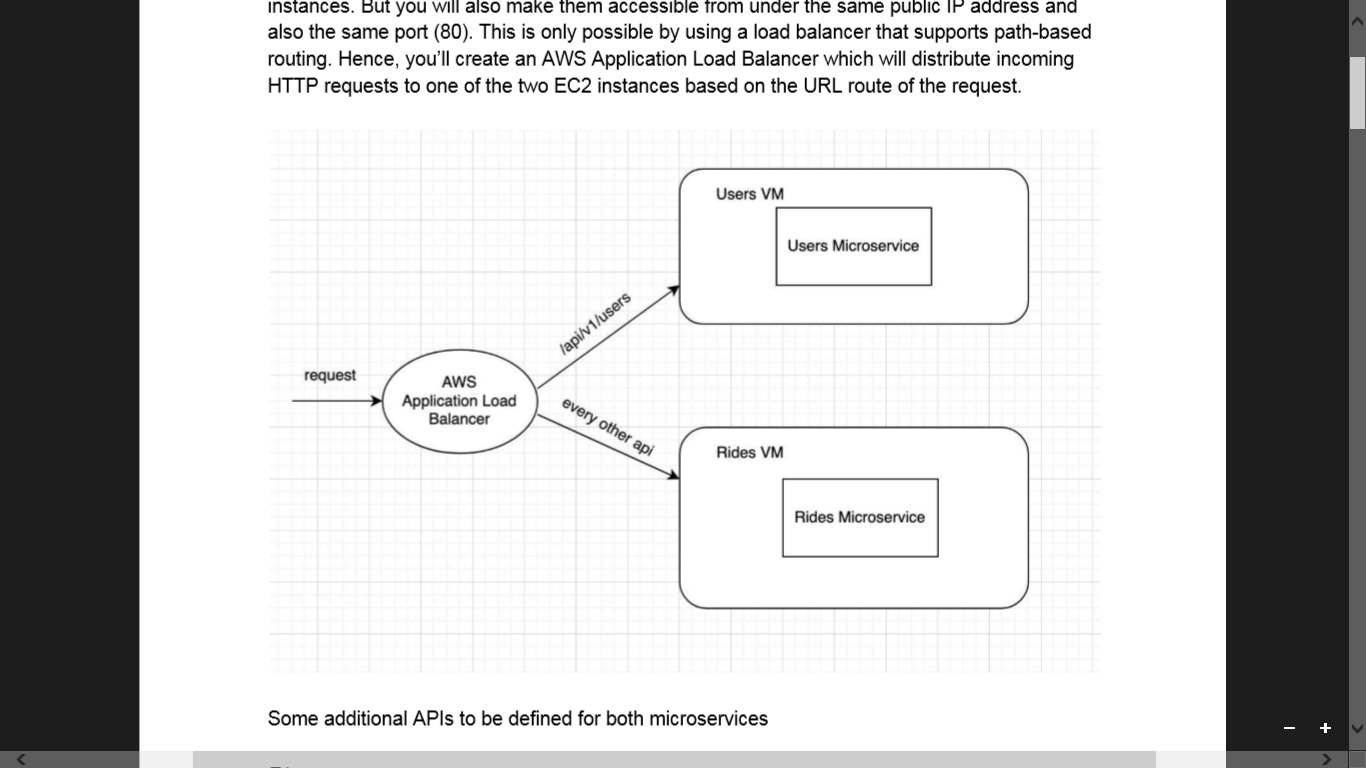
Once the dependencies are set, the next step is to create the application directory, set environment variables, install the required packages and expose the port on which the container can be accessed.

With this done, the containers can be deployed separately.

### Load Balancing

The third stage of the project was to run the containers on different instances and provide a common IP address to access them. An instance is a virtual server for running applications. An instance provides a computing environment, with an interface similar to a PC. It can be accessed through SSH.

The AWS load balancer is used for path based routing. Users access the APIs with the same public IP Address and port number. The load Balancer routes them into separate instances.

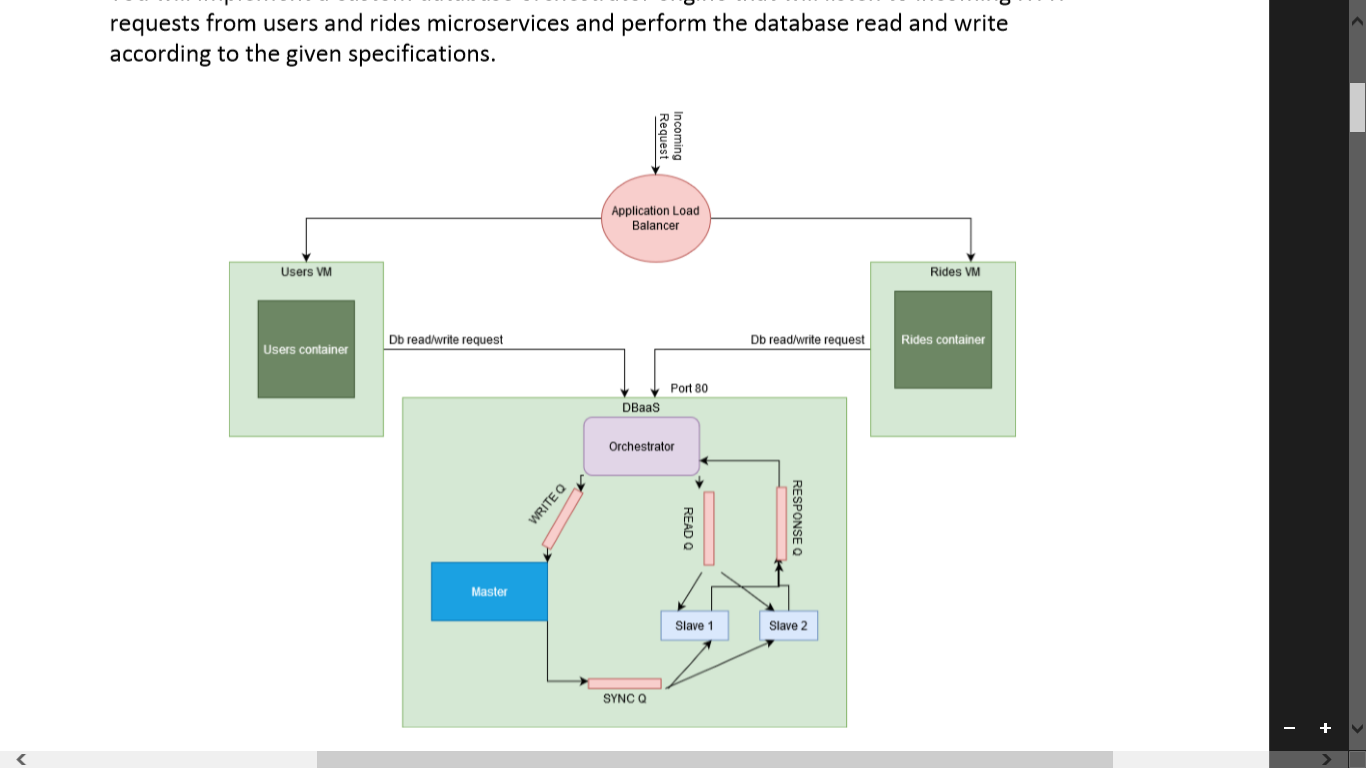


### Orchestrator and Zookeeper

The final stage of the project included

1. Setting up a master-slave architecture to handle database accesses
2. Scaling the containers based on the number of requests
3. Fault Tolerance using zookeeper and crash APIs.

#### Master Slave Architecture



#### Zookeeper

ZooKeeper is an always available and highly scalable network wide “tree” that can store information about our currently equipped and functioning containers. In addition to a regular tree, it has watchers and callbacks that make it extremely useful to keep track of running instances of our application without resorting to first hand inefficient methods such as polling or signals.

Our ZooKeeper setup consisted of the following:

A root node “/”, having five children. We are only interested in three of them:

* 1. “/election”,
  2. “/allNodes”
  3. “/liveNodes”.

The “/election” node is the primary node that we will be used to handle and manage the leader election process and keeo track of the nodes. Every dbServer container in our application starts off as a slave. Initially, two slaves are created.

As soon as the container runs, it first tries to connect to the ZooKeeper server, and creates an EPHEMERAL SEQUENTIAL node representing itself under “/election”.

Upon creating the node, it checks the list of all nodes to check for the oldest node. If it is the oldest node, it restarts itself as a master. Similarly, the second slave registers a node under “/election”, stores the oldest node as its leader and sets a watch on the election node.

As soon as the current leader goes down, due to the watch, a callback is triggered, and all the slaves check to see if they are the oldest node in the server. The next oldest node now proceeds to become the master (by restarting itself) and the other slaves store it as its leader. Simultaneously, a new worker is automatically started upon the death of the master. This cycle repeats forever and ensure high-availability.

## TESTING

What were the testing challenges

Expand node:

Expand scaling;

Expand readDB:

Explain how you fixed the issues on automated submission

Again, expand the last few days…

## CHALLENGES

A particularly troublesome challenge was to asynchronously call AMQP channel consume function within read database API to read the message queue. The asynchronous nature made it hard to send the data read from the database as the http response. The channel, when kept open, sent multiple responses every time it received data, causing the application to crash.

Some of the fixes we tried were

1. Use of global variables – Failed because due to the asynchronous nature of the consume function, the value was sent as the response before the channel even received the data from the database
2. Async/Await feature failed for reasons that could not be determined
3. Use of nested return statements worked the first time but sent no data on subsequent requests

Finally, the fix that worked was to close the channel immediately after receiving data from the database.

In ZooKeeper, one of the main issues we encountered is the callback function was so complex and nested that it resulted in a lot of confusion over the variables and their scopes that are passed in and out of several nested callbacks.

This was overcome by simplifying callback code by creating a clever recursion in the watch function, to automatically create a new watch and terminate the previous running instance of the function by returning.

## Contributions

Each individual team member must list their contributions towards it.

### Ravendra Singh

1. Assignment 1 – APIs 8 and 9
2. Assignment 2 – Setting up docker, splitting the rides and user APIs
3. Assignment 3 – Request counter API, reset counter API, setting up load Balancer
4. Final Project – Rabbit MQ and scaling

### Hrishikesh V

1. Assignment 1 – API 1,2
2. Assignment 3 – Count the number of rides
3. Final Project – Rabbit MQ framework, create container, delete container API, testing/debugging Orchestrator

### Aekansh Dixit

1. Assignment 1 – Full Node JS Backend framework, API 6,7
2. Final Project – Crash APIs, test/debug Orchestrator

### Varun R. Gupta

1. Assignment 1: APIs 3, 4 and 5, AWS Setup and submission.
2. Final Project: High-availability (ZooKeeper), List Workers API, Master-Slave Crash APIs.

## 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. | ✔ ✔ ✔ ✔ |