A cloud based RideShare application, that can be used to pool rides. The application allows the users to create a new ride if they are travelling from point A to point B.

The features provided by the application are:

- Adding a new user
- Delete an existing user
- Creating a new Ride
- Search for an existing ride between a source and a destination
- Join an existing ride
- Delete a ride

Related work

Docker:

https://docs.docker.com/engine/docker-overview/

https://docs.docker.com/get-started/part2/

AWS Load-balancer:

https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html

RabbitMQ:

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

Docker-SDK:

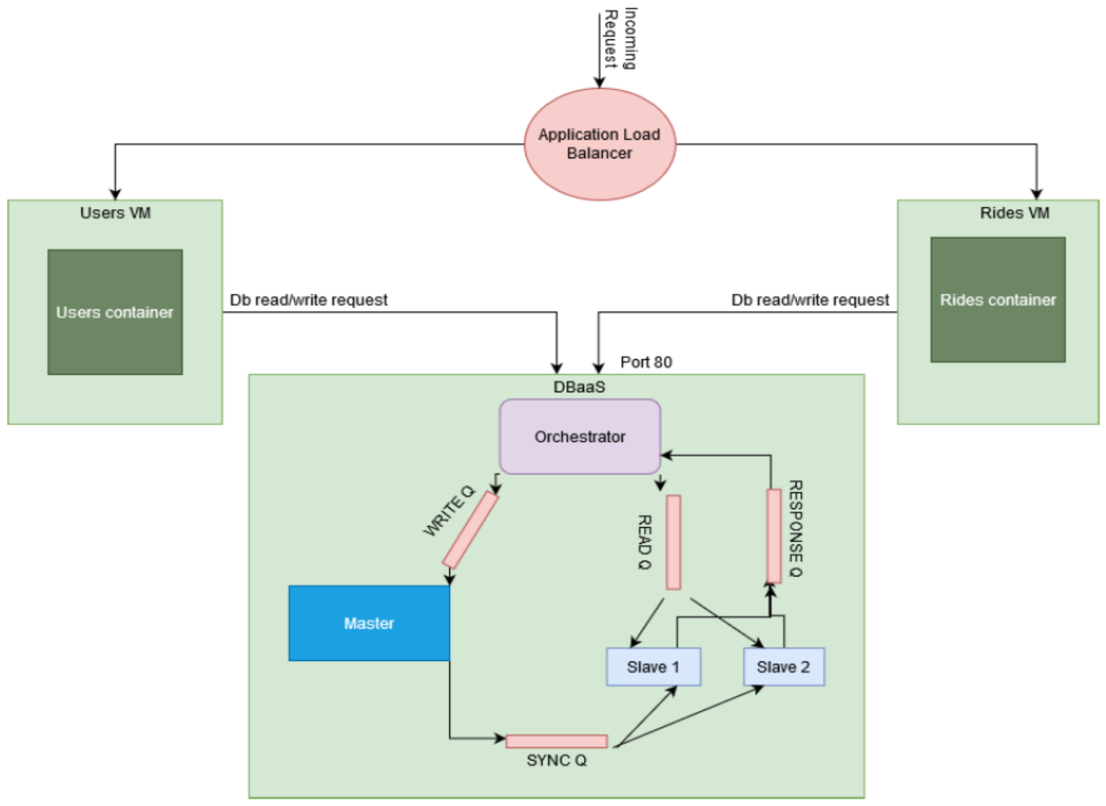
https://docker-py.readthedocs.io/en/stable/

Zookeeper:

http://zookeeper.apache.org/

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

ALGORITHM/DESIGN



- The orchestrator will listen to incoming requests on port 80 and publish the incoming message into relevant queues and **auto-scaling(scalability)** containers as required.
- The master worker will listen to the writeQ and pick up all incoming messages on the “writeQ” and write them to a database.
- The slave worker will be listening to the readQ, and picks up all the read requests on that queue. readQ is implemented using RPC.Default exchange is used which ensures the slaves read the requests in a round robin fashion.
- Multiple instances of the worker are created such that the messages from the readQ are picked up by the slave workers in a round robin fashion.
- For implementing eventual consistency, the master will write the new db writes on the syncQ(SQL Queries) after every write that master does, which will be picked up by the slaves to update their copy of the database to the latest.Made use of Fanout exchange to achieve the broadcasting of messages from master to all the slaves.
- DBaaS has to be **highly available**, hence all the workers will be “watched” for failure by Zookeeper.

● In case of the failure of a slave, a new slave worker will be started, and a fresh copy of DB is copied from the current master. This is made possible by using using the get_archive and put_archive methods of docker-sdk. The master's db is copied to the local system using get_archive and the same db is copied to the newly created slave using put_archive.

● In case of failure of the master, the existing slave, with the lowest pid of the container they are running in, will be elected as master and a new slave node will be brought up by the orchestrator. The container that became the master will change its running code accordingly.

TESTING

Testing challenges:

- Maintaining and checking correlation between test requests and auto-scaled containers.
- Zookeeper leader election triggers nodes for all kinds of events, but we needed to differentiate between creation and deletion events.
- Testing the containers bypassing the load balancer.
- Corner cases testing. Such as killing the master or slave as soon as it is up, sending variable amount of requests to ensure scaling of slaves etc.

Issues on automated submission:

- Fixed auto-scaling by changing the count interval.
- Taking care of bad requests.
- Had issues with using count manager and files. So we used global variables instead.

CHALLENGES

- RabbitMQ: Making the slave listen to both readq and syncq.
- Not losing a request when it is made to a non-existing worker.
- To bring up new containers from another container using Docker-SDK.
- To copy the master's database to the newly created slave.
- To keep track of existing/crashed containers using Zookeeper.
- Performing leader election.
- Implementing Master Crash. To ensure that the newly created container gets the database of the master only after a new master is elected.
- Changing the running process when a slave gets elected as master.
- Reducing the time to get a new container up and running.