Final Project: Mini DBaaS for Rideshare

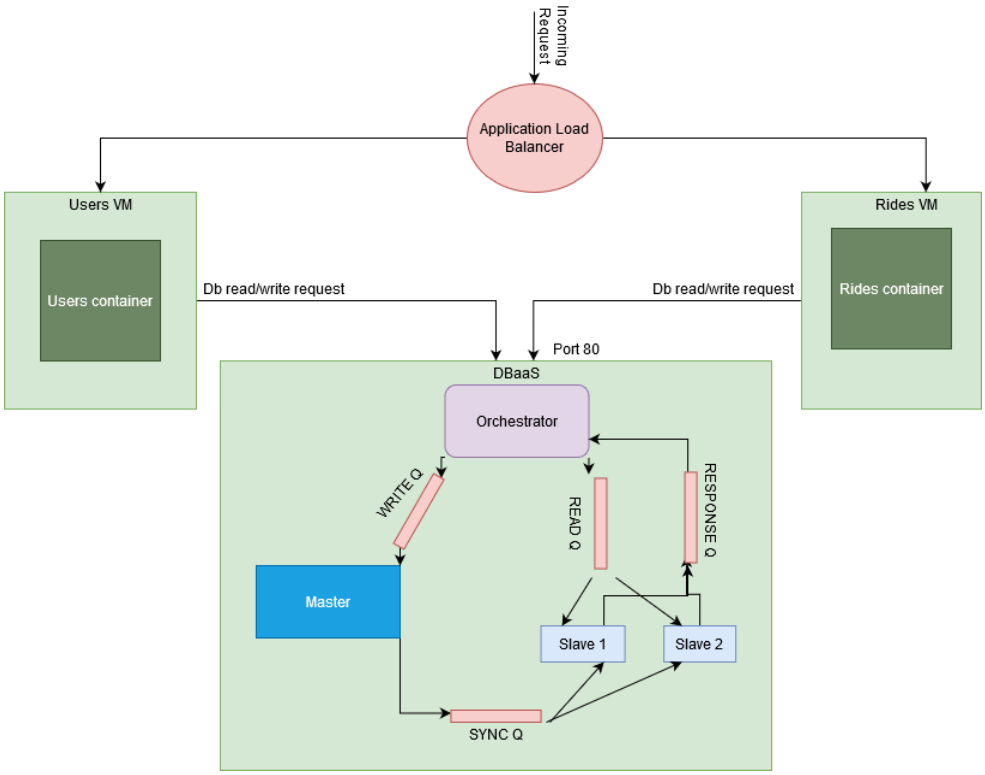
17CS352 - Cloud Computing, PES University

The final project is focused on building a fault tolerant, highly available database as a service for the RideShare application. You will only be working with Amazon EC2 instances. You will continue to use your users and rides VM, their containers, and the load balancer for the application. In addition now, you will enhance your existing DB APIs to provide this service.

The db read/write APIs you had written in the first assignment will be used as endpoints for this DBaaS orchestrator. The same db read/write APIs will now be exposed by the orchestrator.

The users and rides microservices will no longer be using their own databases, they will instead be using the "DBaaS service" that you will create for this project. This will be the only change that has to be made to the existing users and rides microservices. Instead of calling the db read and write APIs on localhost, those APIs will be called on the IP address of the database orchestrator.

You will implement a custom database orchestrator engine that will listen to incoming HTTP requests from users and rides microservices and perform the database read and write according to the given specifications.



Architecture of DBaas:

- The orchestrator will listen to incoming requests on port 80.
- You will be using AMQP(Advanced Message Queue Protocol) using RabbitMQ as a message broker for this assignment.
- The orchestrator is responsible for publishing the incoming message into the relevant queue and bringing up new worker containers as desired.
- There will be four message queues named “readQ”, “writeQ”, “syncQ” and “responseQ”
- All the read requests will be published to the readQ.
- All the write requests will be published to the writeQ.
- There will be two types of workers running on the instance, “master” worker and “slave” worker.
- The master worker will listen to the “writeQ”.
- It will pick up all incoming messages on the “writeQ” and actually write them to a persistent database.
- The slave worker is responsible for responding to all the read requests coming to the DBaaS orchestrator.
- The slave worker will be listening to the readQ, and picks up any messages on that queue.
- Upon querying the database based on the request, the slave will write the output to the “responseQ”, which will then be picked up by the orchestrator, using the right type of exchange and correctly using channels. (See RPC example of RabbitMQ, instead of implementing in manually)
- If there are multiple instances of the worker running, then the messages from the readQ must be picked up by the slave workers in a round robin fashion. (See. Worker Queues example of rabbitMQ, instead of implementing it from scratch.)
- Each worker (slave and master) will have its own copy of the database. The database will not be shared among the workers.
- Separate database creates the problem of maintaining consistency between the various replicas of the same database.
- To solve this issue, you will use an “eventual consistency” model, where after a write has successfully been performed by the master worker, it must eventually be consistent with all the slave workers.
- For implementing eventual consistency, the master will write the new db writes on the “syncQ” after every write that master does, which will be picked up by the slaves to update their copy of the database to the lastest.
- All the workers will run in their own containers, they will connect to a common rabbitMQ.
- The orchestrator will also run in a container.
- DBaaS has to be highly available, hence all the workers will be “watched” for failure by zookeeper, a cluster coordination service.
- In case of the failure of a slave, a new slave worker will be started.
- 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, using zookeeper’s leader election. And upon that, a new slave node will be brought up by the orchestrator.

- To save yourself from the hassle of installing/configuring zookeeper, you can use the official existing “Zookeeper” docker image from dockerhub.
- Even for rabbitMQ, use the existing image.
- The orchestrator will expose the API endpoints ‘api/v1/db/write’ and ‘api/v1/db/read’

High Availability:

Zookeeper will be used to “watch” on each worker.

If a master fails:

- A new master is elected amongst the existing slaves. The slave whose container has the lowest pid, is elected as the master.
- A new slave worker is started and all the data is copied asynchronously to the new worker.

If a slave worker fails:

- A new slave worker is started.
- All the data is copied to the new slave. asynchronously

Scalability:

The orchestrator has to keep a count of all the incoming HTTP requests for the db read APIs. For now we will assume that most requests are read requests and hence the master worker does not need scaling out.
Do not count the request made to ‘crash API’ on the orchestrator.

The auto scale timer has to begin after receiving the first request.
After every two minutes, depending on how many requests were received, the orchestrator must increase/decrease the number of slave worker containers. The counter has to be reset every two minutes
For auto-scaling count only db read APIs.
0 – 20 requests – 1 slave container must be running
21 – 40 requests – 2 slave containers must be running
41 – 60 requests – 3 slave containers must be running
And so on.

Orchestrator:

- The orchestrator will be a flask application.
- The orchestrator will be serving the endpoints ‘api/v1/db/read’ and ‘api/v1/db/write’.
- Upon receiving a call to these APIs it will write relevant messages to the relevant queues.
- From the users/rides container, you will now be calling the db read/write APIs on the DBaaS Vm instead of localhost.
- You will also have to implement an API ‘api/v1/crash/master’ called over POST method
    - When this API is called the orchestrator will kill the master worker.
- Another API ‘api/v1/crash/slave’ also has to be implemented called over POST method
    - When this API is called, the slave whose container’s pid is the highest has to be killed.
- Another API ‘api/v1/worker/list’, over a GET method has to be implemented.
    - The response of this API should be the sorted list of pid’s of the container’s of all the workers.
    - Eg, [1000, 2112, 4330]

Hints

- Both the master and slave must run the same code, as any slave can be elected as the master.
- Use docker sdk to start and stop new containers programmatically
- Use the db read/write API implementation from the first assignment to implement ‘master` and the ‘slave’ workers.
- The ‘api/v1/db/read’ and ‘api/v1/db/write’ must only publish the request to the relevant queues and not write to db themselves
- Crash APIs should return 200 OK with message body as `[ "pid_of_container_killed" ]`
- For syncing a new slave with all the data, you can use multiple techniques, a few of them could be
    - Making the message “durable” so they are not removed from the queue
    - Not sending “ack” from the consumers so the messages stay in the queue, so when a new slave joins becomes a consumer, they can get all the messages.
    - Or any other algorithm you find appropriate.
- Other than the given 4 queues, you can use any number of temporary queues at your discretion.

Other points

You must keep the AWS setup that you created in your last assignment still running. The Users container must still be running on the Users EC2 instance, and the AWS Application Load Balancer must still perform the path-based load balancing before requests are ever received by the db container orchestrator.

The read write db requests from users/rides container will no longer go the 'api/v1/db/read' API running on the localhost, but instead must go to the public IP of the DBaaS VM.

- **The rabbitMQ tutorial is a great place to start.**
- Hello World, Worker Queues, RPC, Pub-Sub are few examples you must look at in the RabbitMQ tutorial, as you'll be using those techniques in this project.
- Initially you will start with 1 slave and 1 master worker.
- Zookeeper, RabbitMQ will both run in their own containers, their official image can be pulled from dockerhub.
- Hence initially you will be running 5 containers, for "zookeeper", "rabbitmq", "orchestrator", "slave" and "master" which will then be increased/decreased accordingly.
- Ideally you must be using "docker-compose" to orchestrate your containers.
- For the project you will need to know a little more about the intricate details of AMQP like, queues, exchanges, channels, brokers etc.

References:

Zookeeper: http://zookeeper.apache.org/
Zookeeper:https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php
RabbitMQ: https://www.rabbitmq.com/getstarted.html **(Please go through tutorial 1, 2,3 and 6 and try to accomodate similar designs in the project)**
AMQP: https://www.rabbitmq.com/tutorials/amqp-concepts.html
RabbitMQ Channels: https://www.rabbitmq.com/channels.html
Zookeeper Container Image: https://hub.docker.com/_/zookeeper/
RabbitMQ Container Image: https://hub.docker.com/_/rabbitmq/
Python binding for zookeeper: https://kazoo.readthedocs.io/en/latest/
Docker sdk to start containers programmatically: https://docker-py.readthedocs.io/en/stable/

Due Date: Week of 19th April, 2020
Marks: 25

Distribution of marks

Sl. no	Feature	Marks
1	Master/slave worker	5
2	Orchestrator	3
3	Setting up rabbitmq, with all the queues having correct producer and consumer	3
4	Perform data replica/sync	3
5	Scale out/in	3
6	Fault tolerance (slave + master) using Zookeeper with correct znodes and watch	5
7	Report - illustrating design and a list of all challenges your team had to solve	3

Suggestion:

- Step 1: Implement the orchestrator with RabbitMQ
- Step 2: Implement replication/sync
- Step 3: Demonstrate scale out/in
- Step 4: Implement High Availability using Zookeeper

This way, you will always have a working project, and get partial marks in case some of the features are incomplete.
Make sure that you have separate docker container images with each feature, to checkpoint your work.