17CS352:Cloud Computing

Class Project: Rideshare

Building fault tolerant and highly available mini DBaas using Node.js, Dockers, RabbitMQ, and ZooKeeper

Date of Evaluation:
Evaluator(s):
Submission ID: 1275
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Aditya Sivaram	PES1201700012	6 H
2	Athul Sandosh	PES120170110	6 G
3	Gaurav CG	PES1201700989	6 H
4	DR Sai Praneeth	PES1201701094	6 H

Introduction

The project deals with the building of mini DBaas using Node.js, dockers, RabbitMQ and Zookeeper. This DBaas is used to manage all the database operations which are required by the RideShare application which has separate user and rides entries.

Our infrastructure consists of AWS application load balancer which routes to either rides or users target groups based on the path requested. These target groups consist of respective rides and user instances which are REST endpoints. These endpoints make use of the mini DBaas for all its db operations. We use Node.js as the backend endpoint for our db orchestrator. We also use dockers for creating multiple worker containers to handle the requests. Rabbitmq is used for queuing these db requests and zookeeper is used for monitoring the worker containers. We used respective Node.js APIs to interact with docker (dockerode), rabbitmq (amqp-lib) and zookeeper (node-zookeeper-client).

Related work

Below are the list of references used for the development of the project.

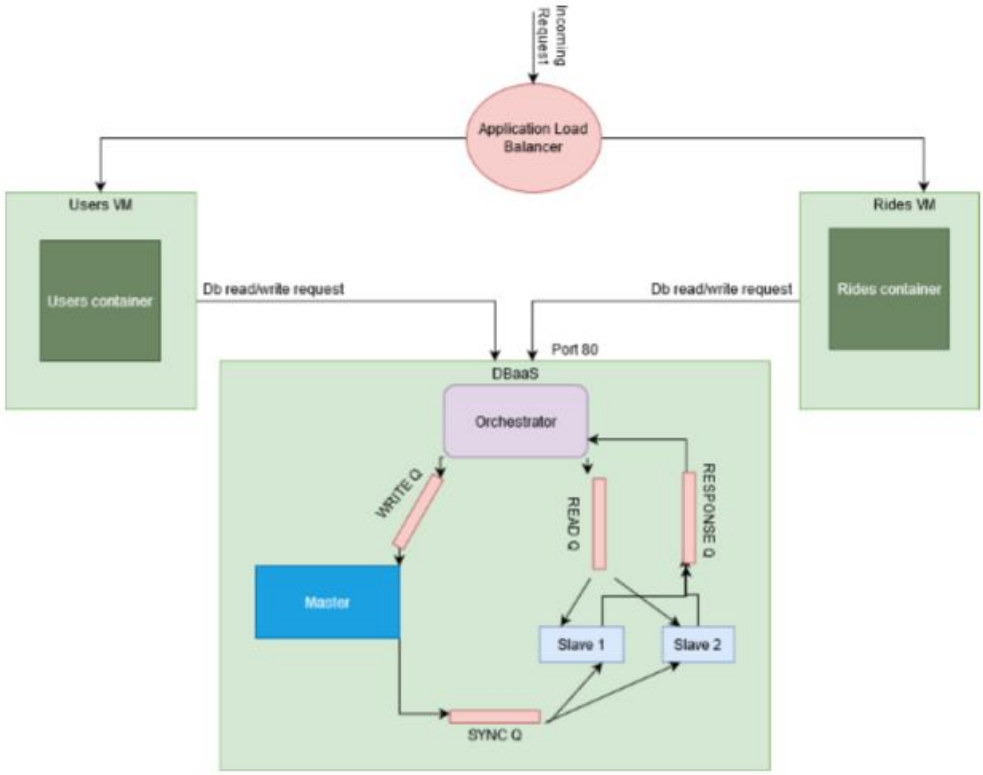
1. https://www.youtube.com/watch?v=00XYLzuucwE&list=PL55RiY5tL51q4D-B63KBnygU6opNPFk_q - For building Node.js endpoint.
2. https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html, https://www.rabbitmq.com/tutorials/tutorial-two-javascript.html, https://www.rabbitmq.com/tutorials/tutorial-three-javascript.html and https://www.rabbitmq.com/tutorials/tutorial-six-javascript.html - For implementing the RabbitMQ queuing system for both orchestrator and the workers.
3. https://www.npmjs.com/package/dockerode - for accessing the docker server from Node.js.
4. https://www.npmjs.com/package/node-zookeeper-client - for accessing the zookeeper server from Node.js.

ALGORITHM/DESIGN

Our infrastructure consists of an AWS application load balancer, that forwards incoming requests according to the path of the request to the respective target group. All requests with path /api/v1/users are forwarded to the users target group; and all the other requests are forwarded to the rides instance. We have two target groups : the users instance and the rides instance. The API endpoints on these target groups send database read/write requests to the database orchestrator instance.

In the database orchestrator instance, there are two types of workers defined. A master worker to handle write requests and slave worker for read operations. Since we assume that the RideShare application is a read-heavy application, we use only one master and multiple slave workers. Each worker will run in its own containers. Multiple slave

containers are scaled in/out based on the incoming requests. The below layout describes the infrastructure used.



The database orchestrator consists of 7 containers running initially : RabbitMQ, Zookeeper, master container, master db container, slave container, slave db container, and the orchestrator container.

We used amqp-lib to connect to our RabbitMQ container from the orchestrator container. We followed the official examples (1,2,3 and 6) on the RabbitMQ website to achieve the above design.

For scale in/out, we use the middleswares defined in the orchestrator which update the incoming requests and append it to a file. We had a function defined which after every two minutes read the file for the count and then reset it and scale the slave containers accordingly using the 'dockerode' library in Node.js. This function is set initially when the first read request arrives using the setInterval method in javascript.

For high availability using Zookeeper, we used the 'node-zookeeper-client' library of Node.js. Every new worker container upon creation, connects to the zookeeper and creates their unique path(Ephemeral znode) and this unique path is checked by the orchestrator upon creation of the worker container using the getData method of the above library. Using this we set an event function which is triggered upon any change of the path

of the worker container. Every time a worker container crashes, the event function is also triggered and we get the unique path which is used to inspect the respective worker container using 'dockerode' library. If the exit code is 137 or 139, we can assume that the container crashed and then we can spawn a new container in its place. If the container was stopped by the orchestrator during the scale in/out process the exit code of those containers are 143. This was used as a differentiating factor. Hence the event function permanently removes the crashed/stopped containers but spawn only if the container is crashed.

TESTING

We tested the API endpoints using Postman. To test for scale in/scale out, we changed the scale in/out conditions such that a new slave is created if more than 2 requests were counted instead of 20 and manually sent those requests from postman. This was changed back to normal conditions after our testing.

On the submission portal, we faced the following challenges.

1. When we first submitted, we got 0.0. We realized that the problem was the container start up and that the order was not set properly. We then used a sleep function to get the desired order.
2. At second submission, we got 2.0, this was due to the fact that we were initially supposed to check for the first slave container's znode, hence there was no initial watcher set which caused the program to crash.
3. Upon fixing that error we were able to get 10.0 on our third submission.

CHALLENGES

We initially faced the problem of consuming from both readQ and syncQ in the slave worker. We read that only one queue can be consumed in a channel. In order to get around this problem we used the async property of Node.js where the queue.consume was made async and thus the slave worker was able to consume from two queues.

We faced the problem of updating a newly created slave. To solve this issue we used the mongodump and mongorestore tools which are provided by mongoDB. When a new slave container is created we first create the slave db and use dockerode's exec function to execute a command. In this command we dump the master db and restore it to the new db. Upon execution of this command the new slave is created.

Setting up the availability using zookeeper was the hardest challenge of them all. Because of lack of understanding the documentation of the 'node-zookeeper-client'. We tried many alternatives in order to watch the node paths. We initially used the 'zookeeper-watcher' library which provides path based watching and we were unable to implement it properly. Then we went through the document multiple times and found that getData function can be used to add the watcher for the unique node paths. Using this we were able to complete the project.

Contributions

The Contributions are as follows:

1. Aditya Sivaram - RabbitMQ listen queues set up for worker containers.
2. Athul Sandosh - Replication and Syncing of the master and slave containers.
3. Gaurav CG - Orchestrator and high availability implementation using zookeeper.
4. DR Sai Praneeth - Scale In/Out of slave containers.

CHECKLIST

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