17CS352:Cloud Computing

Class Project: Rideshare

The backend functioning of a scalable, fault tolerant and highly available Ride sharing application.

Date of Evaluation: 17-05-2020 (3:00 PM)
Evaluator(s):Ms. Pushpa and Mr. Nitin
Submission ID: 850
Automated submission score: 10/10

SNo	Name	USN	Class/Section
1	Ketan Panwar	PES1201700195	6D
2	Subhranil Das	PES1201700309	6D

Introduction

– This is an implementation of the backend of a ride share application. Using the implementation users can register themselves, book a ride, can join an existing ride, can view ride details. The application also supports deletion of individual entities like users and rides and also facilitates deletion of the entire database separately. For administrative purpose the number of requests made to individual micro service can also be retrieved. The application is fault tolerant and scalable based on traffic volume.

Related work

- The theory taught as a part of the curriculum was very helpful to get started with the basic concepts. Everything has been built upon those basics. Also the reference materials provided along with the assignments and the project specifications were very helpful and sufficient. Also we cited the official documentation of python libraries like pika (https://pika.readthedocs.io/en/stable/) (for AMQP implementation) and kazoo (https://kazoo.readthedocs.io/en/latest/) (for Zookeeper implementation) to get started with the new concepts.

ALGORITHM/DESIGN

- This is an implementation of the backend of a ride share application on AWS platform. The entire application has been developed as two individual micro services namely users and rides. Both the micro services are running on different AWS instances inside separate containers. In order to have access to both the micro services on the same public IP address and same port we have a load balancer, which based on the URL sends the requests to their respective micro service. The load balancer used in the project is an application load balancer that works in the seventh layer of the OSI model (Application Layer) having the two micro services (rides and users) as its two target groups. We chose an Application load balancer over the Classic load balancer because we wanted path based routing but a classic load balancer provides us with load balancing without any consideration of the content on different servers. The user's direct interaction is with the load balancer. Once the micro services receive requests they need their database queries to be addressed. For this purpose we have a database service called DBaaS having a common database for users and rides. We are using one mongoDB database with two collections namely users and rides. Within DBaaS we have an orchestrator listening at its port 80 for database requests from rides and users micro services. We also have two types of workers running in DBaaS, master and slave. The master performs the database write operations and the slave worker will address all the read requests. All the workers will have their own individual databases.

- For communications within DBaaS we use Advanced Message Queue Protocol using RabbitMQ as a message broker. We are using two named queues for the internal communication :-
  1) Write Queue (RPC used)
  2) Read Queue (RPC used)

  And one exchange:-
  1) Sync exchange (Fan-out exchange)

  The orchestrator puts all the write requests to the write queue which has a master sitting on the other side listening for write requests. The master on receiving a request will perform that operation in the database and sends the reply back to the orchestrator using the 'reply_to' field queue of the write queue and then places the same request in the sync exchange for the slaves to consume it and update their respective databases. All the slaves have an individual queue bind to consume from the sync exchange. Since the sync exchange is a fan-out exchange, it will serve to any number of slave queues attached to it. Also the master will write the write commands in a text file named commands.txt which will be used to update the database of a newly spawned slave.

  When the orchestrator receives a read request, it puts them in the read queue. We have all the slaves subscribed to the same read queue. Using the default property of RaabbitMQ we have all the read requests distributed among all the slaves in a round robin fashion, meaning on average every slave will get the same number of messages. Once a slave has the results of the read request it sends the reply back to the orchestrator using the 'reply_to' field queue of the read queue.

  We assume that the number of read requests is way more than the number of write requests so the scaling part of the application works on changing the number of slaves in the application based on the number of requests counted every two minutes. We use docker SDK commands to dynamically spawn new slave as per the requirements.
  To update the database of a newly spawned we have an API in the orchestrator which reads the command.txt file written by the master and will pass all the commands in that file to the slave. On receiving those commands the slave will execute them and then go ahead with its regular function.

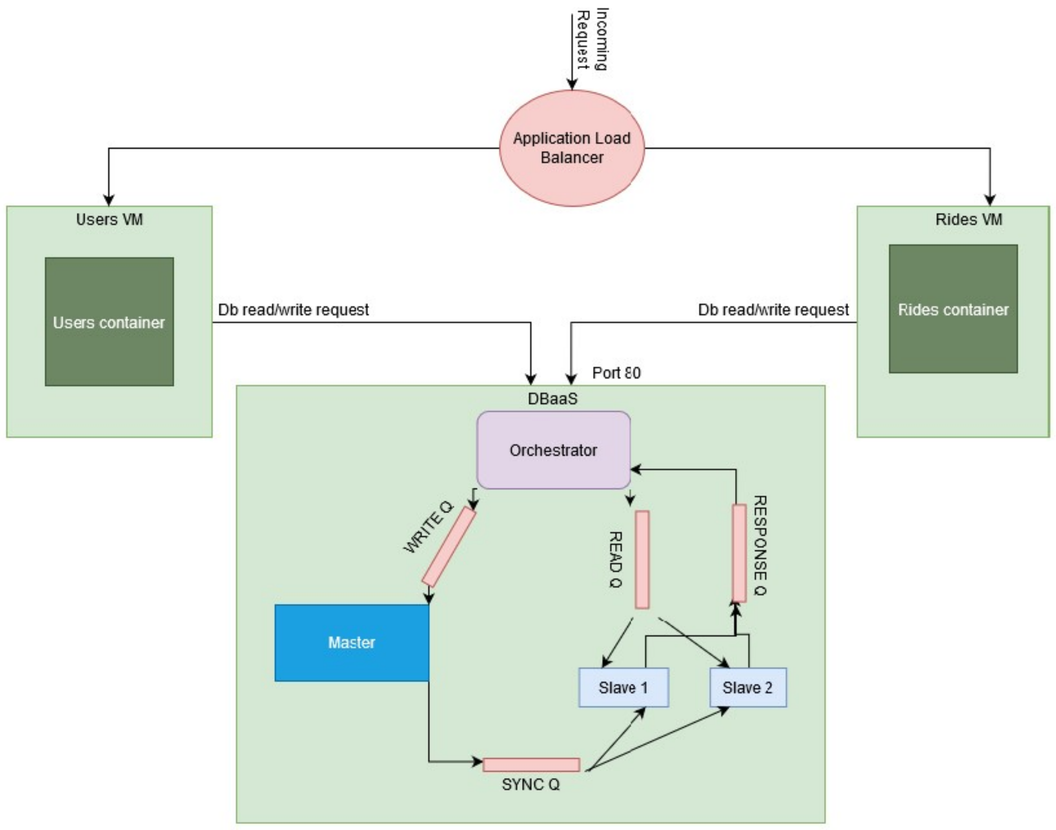
  Orchestrator and all the workers are running in their separate containers.

  To make the DBaaS service highly available we use the zookeeper service. ZooKeeper nodes store the data in a hierarchical name space, much like a file system or a tree data structure. Clients can read from and write to the nodes and in this way have a shared configuration service. So, we are keeping all the workers in the zookeeper hierarchy and if something goes wrong with any of the slave, the zookeeper watch spawns a new slave.

If the master fails then the slave with the lowest process id is chosen and is converted to the master by killing its current process and then starting a new python process as master.

Setting up RabbitMQ and zookeeper was easy, we simply pulled their respective images and did the port matching (2181 for zookeeper) and (5672 and 15672 for rabbitMQ). On running these container with above specifications everything worked well.

The following diagram explains the design and working of the entire application:-



TESTING

- Designing test cases which can cover everything according to the specifications was really tough. But together we managed to come up with all possible test cases. The entire testing was done using the Postman tool.
- To test the scaling in and out functionality we had to send around 20-200 requests every two minutes, so to deal with it we developed a python script which just took one request from Postman with number of read requests to be made to the application and then send that many requests appropriately.

- Automated submission:
  We didn't face much issues in the automated submission and scored 10/10 in our third attempt (submission ID : 308). In our attempts before that our rabbitMQ and zookeeper servers were in stopped state. Submission ID 850 was made to update the required zip file.

CHALLENGES

- Having an understanding of zookeeper was a challenge for really long time.
- RabbitMQ, was an entirely new and tough but yet interesting concept, was easy to understand but gave us a tough time to implement due to its minor details.
- Also fitting the code in from all the team members together created several dependency issues.
- Initially, every time we started our system, things behaved differently. That taught us to take care of dependencies before working on the applications.

Contributions

There is no differentiating line between the contributions of the teammates. Most of the times we worked together, we helped each other, we debugged codes for each other. So we can say its 50-50 for both of us. But we would like to appreciate the contribution of the teachers and TA's who helped us getting out of our problems with regular doubt clarification sessions.

CHECKLIST

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