17CS352:Cloud Computing

Class Project: RideShare

REPORT SUBTITLE

Date of Evaluation: 16/05/2020
Evaluator(s): Rachana B S, Deepthi B
Submission ID: 663
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Vidhaan Mishra	PES1201700055	A
2	Hitesh Kumar	PES1201701511	A
3	Nishchal MN	PES1201701523	A
4	Suraj MM	PES1201701551	A

Introduction

This project developed by our team serves as the backend for a cloud based RideSharing Application. The RideSharing app helps the users to create a ride for travelling from a point A to point B. Further it carries out some basic functionalities such as - adding a new user, deleting a user, create a new ride, delete a ride, etc.

The approach for achieving the desired end product was gradual. Initially we had the users and the rides running on the same instance but as different microservices. This process was further enhanced by deploying the rides containers and users containers into two different AWS instances. This could be achieved by the help of a Load Balancer that made use of path based routing.

In the final stage of the project we worked on something known as the DBaaS(DataBase as a Service). All along the initial development phases of the project the databases were in separate containers for the rides and the users. What we have tried to implement is a fault-tolerant and a highly available DataBase as a Service for this application. This means that the rides and the user containers no longer use their “own” databases but use this DataBase as a Service.

Now the RideShare App is implemented using three microservices for - users, rides and Database management. All these microservices are deployed on AWS EC2 instances. APIs are implemented using flask and the database service is done using postgres. The load balancer directs the requests to the users or rides instances based on the requests.

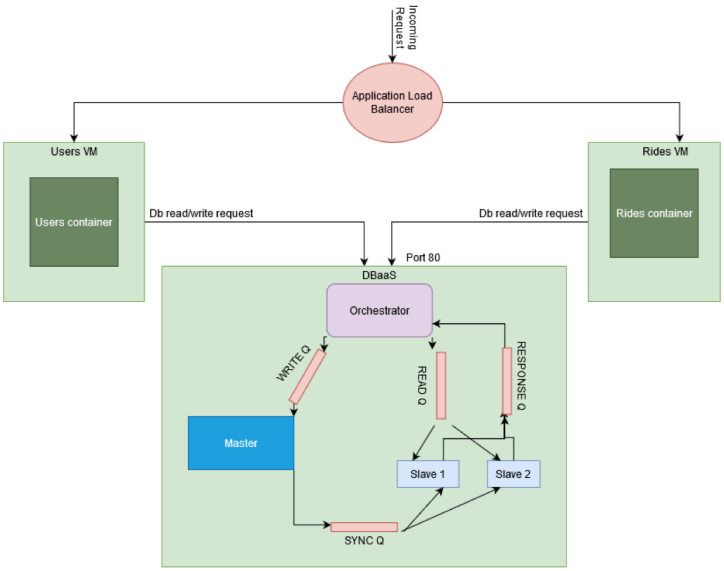
Related work

- RabbitMQ
    - https://www.rabbitmq.com/getstarted.html
    - https://www.rabbitmq.com/tutorials/amqp-concepts.html -
    - https://www.rabbitmq.com/channels.html
    - https://hub.docker.com/_/rabbitmq/
- Zookeeper
    - http://zookeeper.apache.org/ - Basic Overview
    - https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php - Implementation
    - https://kazoo.readthedocs.io/en/latest/ - Python binding for zookeeper

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

ALGORITHM/DESIGN

The basic architecture and the entire workflow is explained in the diagram given below. This was the basic structure we kept in mind while developing the DBaaS.



As it is clearly visible in the above diagram, the incoming request is directed towards the users or the rides container based on the number of requests. We have implemented a DBaaS orchestrator that essentially handles all the major functionalities of the DataBase. The orchestrator listens to the incoming HTTP requests and identifies the requests as read or write based on the endpoints of the requests.

1. Design of the DB Write Request:

- After the request has been identified as a write request, we use RabbitMQ for RPC(Remote Procedure Call). We write the requests into a queue called as writeQ and wait for a response from the master. To avoid any mismatch between the requests and response message, we compare the id's of the response and requests. If a mismatch exists, the message is discarded.
- The master listen to all the requests on the writeQ and writes them onto the database. To be in accordance with the "eventual consistency" model, the master needs to ensure that whenever the it updates the database, the slave workers should also update it with the same value. To ensure this , whenever the master write the DB successfully, we pass it to a fanout

exchange. Now, each of the active slave workers are connected to this fanout exchange and have a queue attached to it so that they can take the message from that queue and update their respective databases.

- The master also writes all the successful DB writes to a queue called syncQ. The purpose of this queue is to maintain the consistency amongst the new slave worker need that are spawned. When a new slave worker is formed, we need to ensure that it is provided with a consistent database for it to read it and update its database accordingly. This is achieved using the syncQ. As soon as the slave worker reads the messages from the syncQ and updates its database, we also set the acknowledgement of these messages within the syncQ to False. When the entire syncQ is read, all the messages are unacknowledged. What this means is that the RabbitMQ observes all the unacknowledged messages and makes them readable again thinking there was an issue. Hence all the messages in the syncQ become readable again for a new slave worker that will be spawned.

2. **Design of the DB Read Request:**

- After the request has been identified as a read request, we again create an object using the RPC from RabbitMQ to write a request to the readQ. To handle the fact that multiple slave workers can request from the readQ, we have scheduled the reads in a Round Robbin fashion. We also ensure that the slave worker can only read the next message from the readQ only when it has properly executed and acknowledged the first message.

- After reading the message from the readQ, the slave does a read operation on the database and writes the response back to the responseQ. The orchestrator receives the response from the slave and again does a matching to avoid any mismatches with the request messages.

3. **Designing a Fault Tolerant, scalable and highly available System:**

We have made intensive use of Zookeeper to implement a scalable, fault tolerant system. There are two cases that we considered - what if the master crashes? And what if the slave crashes?

- We have used Zookeeper to keep a watch on all the worker nodes so that we find out when a slave worker node crashes. The algorithm we used for implementing this “watch” over the nodes was that each node keeps a watch on the immediate next node having the Pid of one less than the node itself. That means that a node with Pid of 4 will keep a watch on the node with Pid of 3.

- Whenever a worker is started, a script runs within the container and creates a Znode. Using zookeeper we keep a watch on the nodes using the algorithm described above.

- The watch function in each worker is triggered when the corresponding lower pid worker crashes. Using the event.type attribute we check if the node has been deleted, indicating a crash. Now if the current worker has no other lower pid to keep a watch on, the current worker is converted into a master.
- However if there exists a node with a lower pid than the crashed node, the current worker will start watching the node with Pid -1, where Pid was the Pid of the crashed slave.
- We keep a watch on all the worker nodes, As soon as a worker crashes, an event is triggered in the orchestrator.

4. **Leader Election:**

- After the script has created the Znodes, and there is no lower Pid to be found for the slave worker created, we assign that as the master by killing the subprocess and starting it again as the master, as it was initially started as a slave.

5. **Scalability**:

- 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 we 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.

TESTING

The testing criteria kept in mind were:

- The new APIs
- Ensuring the old APIs still work
- Load Balancer
- Auto Scaling
- Fault Tolerance for the slave

All the testing was done on AWS instances for easy deployment.

- We used nginx to test our APIs.
- Used the Collection Runner feature of the Postman to test for autoscaling. This sends multiple requests at the same time.

CHALLENGES

There were quite a few challenges that we faced as team. A few of them were:

- Debugging was very time consuming but thanks to StackOverflow and really informative discussions on Piazza, we could overcome this.
- Spawning containers dynamically was difficult to understand using DockerSDK.
- Obtaining the Pid using the low-level APIs
- Ensuring there were no ThreadPool Errors while testing for auto-scaling.
- Implementing the Zookeeper design into the Service was pretty challenging for us but with continued efforts we accomplished it.

CONTRIBUTIONS

The project was entirely a group effort. However in times of a pandemic and following social distancing norms posed a challenge to us, we made extensive use of technologies such as hangouts call, etc. in order to achieve the desired end product. Piazza discussion forums and the links provided for RabbitMQ, Zookeeper and DockerSDK played an immense role in the completion of this project as per the guidelines.

CHECKLIST

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