17CS352: Cloud Computing

Class Project: Rideshare

A app deployed on AWS cloud capable of fault tolerance and load balancing

Date of Evaluation:
Evaluator(s): Rachana BS, Deepthi Bargav
Submission ID: 1128
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Tushar Raj	pes1201700221	A
2	Sagar Ratan Garg	pes1201700913	B
3	Kashish Oberoi	pes1201700113	B
4	Vivek Aditya	pes1201701125	B

Introduction

- The app serves the purpose of basic back-end for online cab service. The app is based on containers and is deployed using flask servers and uses microservice architecture for its deployment. The main component of this app is the DB service which is fault tolerant and is capable of providing very responsive read and write operations.
- Below is a list of main services provided by our Ride-Share App:
    1. Create and delete users
    2. Display all the users
    3. Create rides
    4. Join a user to a list of upcoming rides
    5. Get ride details by their id
    6. Get all the rides between a given source and destination
    7. And various other DB related services
- To provide all these services the app is basically divided into three components:
    1. Users API
    2. Rides API
    3. DataBase As A Service component (DBAAS)
- The users API provides all the endpoints required for managing users. All the APIs make internal calls to read and write API provided by DBAAS component as and when required to store and retrieve information.
- The rides API provides all the endpoints for managing rides. Similar to users API it also makes internal calls to read and write API provided by DBAAS component as and when required to store and retrieve information.
- The DBAAS component is the main part of this app. It is responsible for handling all DB servers. Initially on start-up the DBAAS component creates two DB servers: master and slave, the master performs all the write operations and the slave performs all the read operations, as they receive requests for these operations from Users and Rides API.

- The load-balancer works by scaling the slave DB servers up and down based on the number of read requests the DBAAS component receives.
- The fault tolerance nature or high availability of service is provided using Apache Zookeeper, a distributed coordination service, which creates znodes for all master and slave workers and keeps a watch on them to handle faults.
- All the master and the slave workers communicate with the orchestration service and with each-other using AMQP 0-9-1 protocol implemented using RabbitMQ.

Related work

- https://www.rabbitmq.com/getstarted.html
- https://www.rabbitmq.com/tutorials/amqp-concepts.html
- https://pika.readthedocs.io/en/stable/
- https://docker-py.readthedocs.io/en/stable/index.html
- https://zookeeper.apache.org/doc/r3.4.6/zookeeperProgrammers.html
- https://kazoo.readthedocs.io/en/latest/

DESIGN

- Our APP has basically six components:
    1. Users
    2. Rides
    3. DBAAS orchestrator
    4. Workers
    5. Zookeeper
    6. RabbitMQ
- The Users and Rides component act as first layer in handling all the requests made to our app. They perform all the primary checks regarding the request body and then perform internal calls to read and write APIs of DBAAS orchestrator. Both of these services run in their separate container managed using Docker services.
- The DBAAS acts as the DataBase management console for the whole app. On start-up of the app this service only spawns up two workers, one of them working as master and the other as slave. It is also responsible for keeping the response time as low as possible by scaling the slave worker up and down as and when required based on number of requests.

- Fault tolerance nature or high availability of the workers and the slaves is also managed by DBAAS orchestrator by using Zookeeper using kazoo as their communication client. Now Zookeeper comes into picture to provide this property to our app, whenever a new worker spawns up, a new znode is created and is associated with that worker. Now a data-watch is created on the parent of the znode created handles all the faults.
- RabbitMQ is used as the message broker for our app and is used for establishing communication between workers and orchestrator.

IMPLEMENTATION

- ### RabbitMQ

  The message broker is implemented using AMQP-0-9-1 messaging protocol. Our messaging has basically 3 queues and 1 fanout exchange. One random queue is generated per slave worker which gets binded to fanout exchange.

  The read and write API of DBAAS orchestrator receive requests from Users and Rides microservice, the requests body is then pre-processed and then passed to readQ or writeQ based on whether it is read or write operation.

  The master worker has a consumer consuming messages on writeQ and storing data on the DB persistently.

  The readQ is a modified RPC implementation using RabbitMQ tutorial 6. DBAAS orchestrator acts as a client and a producer while all the slave workers act as servers and consumers. Since they act as producers and consumers and auto-acknowledgement is disabled readQ messages are automatically load-balanced using Round-Robbin method.

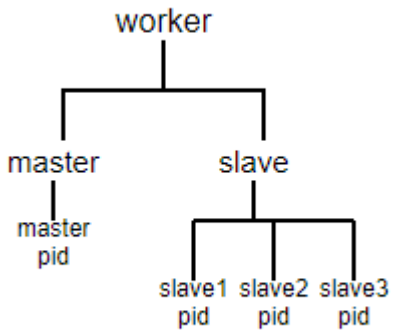
  Eventual consistency model used in our app so when the master worker is done with write operation it forwards the write request to all slave workers on the fanout exchange.

- ### Data Synchronization between Master and Slave workers

  For data synchronization we have used a separate database called syncQ. So whenever a write request comes to a master, it stores the request in the syncQ DB with an always increasing id. Now if a new slave comes up, it sends a sync request on the writeQ, the master responds by sending all the write requests stored in syncQ DB and based on id of each sync request sent the slaves update themselves. The slaves also update their own syncQ DB because it might be possible that they can become master later on.

- ## Zookeeper

  We have used Zookeeper for fault tolerance. We have created znode created in three levels.



  We have a datawatch at the second level of master and slave. So whenever a master or slave crashes data is changed at the second level, then the datawatch gets activated and creates a new slave or converts a slave to master and creates a new slave. The conversion of slave to master happens by sending a "change_designation" message to all the slaves and only the required slave responds as the message contains the pid of the acting slave also.

- ## Scaling Up/Down

  A timer runs in the background which gets activated when the first read request is received by the orchestrator and keeps count of all the requests received. The timer resets the read request count after every 2 minutes.

  It scales up or down the slave containers based on the following rule:

  - 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

While testing we encountered 204 No Content error, even when there existed a ride between given source and destination. This error occurred because we were using different data-types for storing and querying the database. We overcame this issue by changing data-type to int everywhere.

CHALLENGES

- Creating a separate database for each worker, we overcame this by installing mongoDB in the workers image itself.
- Scaling up and down as the respective znodes were not created, we overcame this by creating znodes asynchronously.
- Converting slave to master was a big challenge as pika is not thread-safe, so we converted a slave to master by sending a message on syncQ containing the PID of the slave. When a slave with the same PID receives this message, it calls change_designation function which converts it from slave to master.

Contributions

Tushar Raj: communication with orchestrator (RabbitMQ, Zookeeper), setting up queues and workers, Replication/sync of data, clear DB API, fixed AWS issues during submission

Vivek Aditya: Master Crash Handle, fixed AWS issues during submission, API development

Sagar Ratan: Crash Slave API, fixed AWS issues during submission, API development

Kashish Oberoi: Auto scaling, fixed AWS issues during submission, API development

CHECKLIST

SNo	Item	Status
1.	Source code documented	
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