17CS352: Cloud Computing

Class Project: Rideshare

Date of Evaluation: 19-05-2020
Evaluator(s): Usha Devi BG and Sanjith
Submission ID: **572**
Automated submission score: **10.0**

SNo	Name	USN	Class/Section
1	AMIT KUMAR	PES1201701295	B
2	AMOGH R. DESAI	PES1201700180	B
3	APOORVE GUPTA	PES1201700038	F
4	SIDDHANT SAMYAK	PES1201700247	C

Introduction

This project truly represents the work that went into developing a full-fledged fault-tolerant application and deploying it successfully on Amazon Cloud service (AWS).

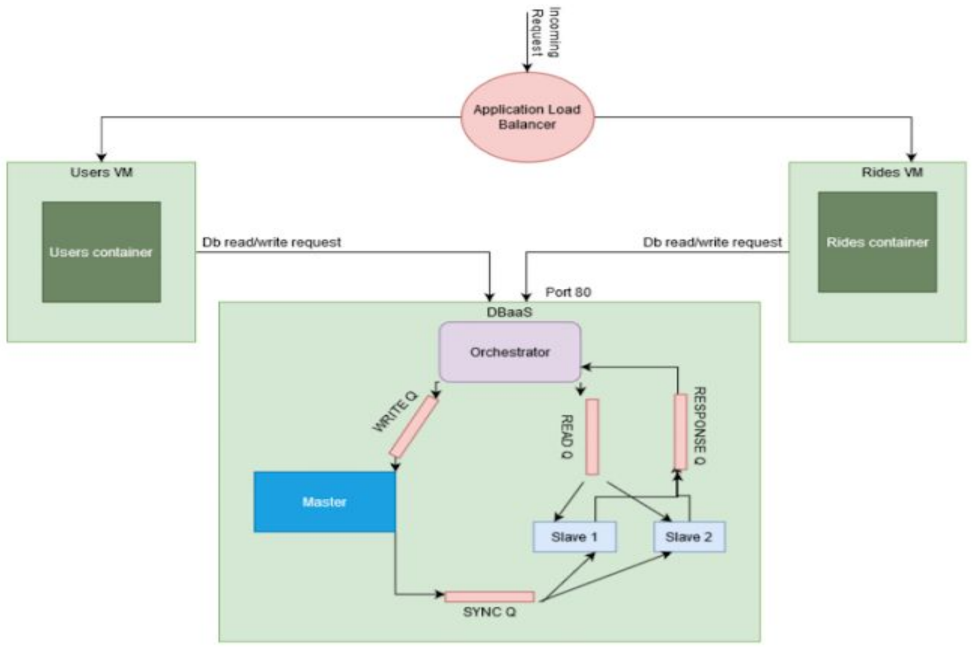

Fig 1: Architecture of the application

The application allows a user to register with a valid SHA1 encrypted password and the required validations are carried out at the server's end. Once registered, the user can go ahead and create a ride between a valid source and destination. He can also share a ride with other users. Certain checks are made as and when the user makes a request to the application and an appropriate response is sent back based on the action.

The following are the main services that the Rideshare application provides:

- User registration
- Remove a user
- List all users
- Create rides
- Fetch details of the rides scheduled between a particular source and destination
- List ride details for any particular ride
- Join an existing ride that was created by another user
- Remove a created ride

Key features of the application:

- Load balancing:
    - For the purpose of load-balancing, the users and rides services were deployed on different AWS instances (virtual machines) where the users' services were exposed to port 8080 while the rides' services were exposed to port 8000. An AWS Application Load Balancer (ALB) was used for this purpose. Security groups were set too with appropriate inbound rules.

- Database-as-a-Service (DBaaS):
    - The database service was set up on a different AWS instance. This lets the application access a database system that uses Master-Slave architecture to deliver a fault-tolerant, consistent and robust database service.

Related work

The following documentations were referred for the project

Docker SDK: https://docker-py.readthedocs.io/en/stable/containers.html

AMQP: https://www.rabbitmq.com/getstarted.html

Zookeeper: https://kazoo.readthedocs.io/en/latest/basic_usage.html

Leader Election:
https://stackoverflow.com/questions/39125064/how-to-use-kazoo-client-for-leader-election

ALGORITHM/DESIGN

Leader Election: On starting up, the orchestrator creates a persistent Znode '/election' and initializes the data of the node with a very high value stored as bytes. When a worker container starts up it creates an ephemeral Znode for itself with the path as '/election/Worker Name' and thus is unique to the worker and the worker's PID (process id) is stored as data. In each of the workers, a DataWatch is set on '/election' which triggers the decorated function in every worker once the change occurs.

The initial initialization of '/election' triggers the watch function. In the function, a particular worker acquires a lock and compares its PID with the data of the '/election' node. If it is lesser, then this worker becomes the new master and sets the data of

‘/election’ as its own PID. This happens for every worker running at that very instance and thus the data of ‘/election’ stores the PID of the master. All the workers know at all times if they have to function as a master or a slave.

Fault-Tolerance: To achieve this a zookeeper watch is set on each worker Znode via setting ChildWatch on ‘/election’ and the watch function is triggered when a CHILD event occurs i.e if a child node (ephemeral worker node) is created or deleted. Therefore, when a container (slave worker) crashes, zookeeper deletes the ephemeral node created by it which in turn triggers the watch function in orchestrator which spawns a new worker if deletion occurred.

Database Replication: When a new worker spawns up, its database will be empty and thus this renders the database system inconsistent. To make up for it, a volume was created and shared between worker containers. Whenever the master writes its own database, at the same time it also performs the same write operation on the database present in the shared volume (this is persistent). Therefore, when a new worker spawns, it initially copies the database file from shared volume to the container. This confirms consistency.

Scaling: The orchestrator keeps count of all the incoming HTTP requests for read operations. Thus, based on the number of read requests received during the 2-minutes window, the slaves require scaling in or out. The threaded auto-scale timer begins after the first request and depending on how many read requests were received, the orchestrator increases the number of slave workers by spawning the required extra number or decrease the number of slave workers by killing the excess number of slaves in the decreasing order of their PIDs.

Advanced Message Queuing Protocol (AMQP): Since the workers handle the tasks related to the database there emerged a need for request forwarding from orchestrator to the worker containers. Using RabbitMQ as the message broker, queues and exchanges were defined for the purpose of Writing, Reading, or Synching the database to maintain consistency. For the purpose of reading from or writing to the database 4 queues were defined per worker following the RPC protocol: WriteQ, WriteResponseQ, ReadQ, and ReadResponseQ. Direct Exchange was used.

For the master to sync the writes performed by it among slaves, once the master was done it would publish the same request to all slaves using a Fanout exchange. This helped keep up the consistency of the database.

TESTING

During testing, the main goals were to achieve consistency during replication when a new worker is spawned, to achieve fault tolerance if a worker crashes and to be able to scale the number of slaves up and down based on the number of read requests sent in the 2-minute window.

Therefore, starting 2 workers (master and slave), we few write requests were sent out and the first level of consistency was achieved through the AMQP service. Batches 75, 26, 11, or 0 read requests were sent out in the 2-minutes window to check if the scaling took place properly. Our system did the job perfectly. The response of the read requests made once the system had scaled up verified that the replication was working properly. Thus, our database withheld consistency and fault-tolerance.

No issues emerged during the automated testing and the system aced a perfect score of 10 in the first go itself.

CHALLENGES

Reasonable number of challenges were faced during the course of this project which is related to DBaaS.

- Lack of proper python documentation present for most of the parts like AMQP, Zookeeper. But this also opened doors to learning and building everything from scratch because of the lack of any reference.

- The DBaaS system lacks modularity as one feature highly depends on the other and thus is not advantageous to a team of designers.

Contributions

Since the DBaaS system lacked modularity the entire project was done collectively.

Name	Contribution
Amit Kumar	Docker, Orchestrator, Worker, AMQP, Zookeeper, Scaling, Replication, APIs
Amogh Rajesh Desai	Docker, Orchestrator, Worker, AMQP, Zookeeper, Scaling, Replication, APIs
Apoorve Gupta	Docker, Orchestrator, Worker, AMQP, Zookeeper, Scaling, Replication, APIs
Siddhant Samyak	Docker, Orchestrator, Worker, AMQP, Zookeeper, Scaling, Replication, APIs

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