17CS352:Cloud Computing

Class Project: Rideshare

REPORT SUBTITLE

Date of Evaluation: 17-05-2020
Evaluator(s): Rachana BS, Deepthi
Submission ID: 1389
Automated submission score: 10

SNo	Name	USN	Class/Section
1.	S Sriya	PES1201700048	6G
2.	Deepika P Karanji	PES120170103	6G
3.	Lavanya L	PES1201700798	6G

Introduction

The final project is focused on building a fault tolerant, highly available database as a service for the RideShare application on the Amazon EC2 instances. The project includes the use of a users and rides VM on separate containers, and the load balancer for the application.

The orchestrator implemented in the project is the endpoint for the DB APIs which include writing to and reading from the DB. This forms the DBaaS orchestrator. The term DBaaS (Database-as-a-Service) refers to software that enables users to setup, operate and scale databases using a common set of abstractions (primitives), without having to either know nor care about the exact implementations of those abstractions for the specific database. The database used in this project is sqlite3.

The users and rides microservices do not have their own databases, but will be using the DBaaS service. Instead of calling the db read and write APIs on localhost, those APIs will be called on the IP address of the database orchestrator. The database orchestrator engine listens to incoming HTTP requests from users and rides microservices and performs the database read and write according to the given specifications.

RabbitMQ, a message broker in order to implement AMQP(Advanced Message Queue Protocol). The queue in RabbitMQ is used to send/receive messages to/from between the orchestrator and workers and also among workers. The four main message queues implemented are readQ, writeQ, syncQ and responseQ.

ZooKeeper's architecture supports high availability through redundant services. Every node in a ZooKeeper tree is referred to as a znode. Znodes maintain a stat structure that includes version numbers for data changes, acl changes. The watch function in zookeeper maintained the high availability of the database. Clients can set watches on znodes. Changes to that znode trigger the watch and an event is called which executes a specific code .

Related work

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

https://zookeeper.apache.org/doc/r3.5.7/zookeeperProgrammers.html

https://www.oreilly.com/library/view/zookeeper/9781449361297/ch01.html

https://kazoo.readthedocs.io/en/latest/api/client.html

https://www.youtube.com/watch?v=AS5a91DOmks

https://blog.container-solutions.com/docker-inspect-template-magic

https://forums.docker.com/t/get-a-containers-full-id-from-inside-of-itself/37237/5

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

https://kazoo.readthedocs.io/en/latest/api/protocol/states.html

https://www.rabbitmq.com/channels.html

https://hub.docker.com/_/zookeeper/

https://hub.docker.com/_/rabbitmq/

https://kazoo.readthedocs.io/en/latest/

ALGORITHM/DESIGN

- The worker.py is the python program that the worker container runs.
- Znodes are used to keep track of the details of each worker. The data of each znode stores the process id (pid), container id (cid) and a string saying if it is the master or the slave.
- Each worker has its own database which is named after the pid of the worker container so that the names of each database is unique.
- Each znode created is ephemeral as they exist as long as the session that created the znode is active. When the session ends the znode is deleted.
- The znodes are created along the path “/worker” and the path is ensured or created if necessary by using zk.ensure_path(“/worker”).
- The first container that comes up is made the master as it is the container with the lowest pid. This znode is created at the path “/worker/master” along with its details.
- The containers that come up after this the master are created at the path “/worker/slave”, to ensure that the path of each slave is unique the path is followed by “/slave<pid>”.
- The znodes of the slaves are created only if the path “/worker/master” exists.
- After each znode is created a watch on its path is set. This is done so as to bring up worker containers in case of slave/master crash.
- The functionalities of the master are separated from the functionalities slave by comparing the cid at the path “/worker/master” and the current container's cid.
- The code for bringing up a new worker is written in the watch function for both the slave and the master, if the event type is “DELETED” .
- The new worker spawned has the znode with the data mentioned above at its respective path and a watch is set on this path. The database of this new worker is a copy of the database of the master to ensure consistency.

TESTING

1. Initially, we got DB Clear failed, score zero. We then changed the port on which the orchestrator is running from 5000 to 80
2. Then we got inconsistent ride information, score 1. We were returning code 400 instead of 204 for listing upcoming rides.
3. Then our auto scaling failed, scoring 3. Then we changed the docker volume binding path to the aws root user path, and scored 10.

CHALLENGES

1. Making the queues durable, using RPC so as to ensure consistency in the data and figuring out how to send the JSON data in the queues.
2. Figuring out how to differentiate between master and slave workers - did it through znode data.
3. Writing the watch function for spawning a new worker in case of master crash or slave crash in the correct file.
4. Fetching CID and PID of the container in which the code is running - used docker inspect sdk
5. Master crash - a major revelation that came to us was that a container with a particular CID gets another new PID on container restart. This was leading to unpredictable results during the master crash.
6. Getting new containers to spawn properly during slave crash and run the worker code - the deletion of the znode and container.kill() were async calls, so the old slave container used to die before it birthed the new slave successfully. Solved this with a sleep function.
7. For auto scale, killing container without spawning new container - we placed a data watch and if event.type == CHANGED, then we didn't spawn a new instance

Contributions

1. Sriya S - rabbitMQ, master fault tolerance, master/slave worker, zookeeper, performing data replica/sync, auto scale
2. Deepika Karanji - slave fault tolerance, master slave db replication, master/slave worker, zookeeper, auto scale
3. Lavanya L - load balancing, slave replication, auto scale, worker list api, master/slave worker

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