17CS352: Cloud Computing

Class Project: Rideshare

Database as a Service

Date of Evaluation: 20-05-2020
Evaluator(s): Prof. Srinivas KS
Submission ID: 909
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Anuj Tambwekar	PES1201700056	H
2	Anirudh Maiya	PES1201700170	H
3	Anshuman Pandey	PES1201700255	H

Introduction

Rideshare is a backend cloud application hosted on AWS servers, which can potentially be used to pool rides. It provides various microservices APIs to create, delete and manage users, rides etc. It runs on top of highly reliant, fault tolerant database as a service application, which handles any database read or write requests. The DBaaS application runs on separate server. It uses RabbitMQ as AMPQ messaging broker to handle internal data transfer and Apache Zookeeper which handles any internal crash, making the application fault tolerance. The DBaaS also support scaling the application to handle too many requests and at same time makes sure that resource do go unutilized.

Related work

RabbitMQ:

- https://www.rabbitmq.com/tutorials/
- https://pika.readthedocs.io/en/stable/

Zookeeper

- https://zookeeper.apache.org/
- https://kazoo.readthedocs.io/en/latest/
- https://kazoo.readthedocs.io/en/latest/api/recipe/election.html
- https://www.tutorialspoint.com/zookeeper/zookeeper_leader_election.htm

Docker

- https://docs.docker.com/network/host/
- https://docs.docker.com/engine/examples/postgresql_service/
- https://dev.to/jibinliu/how-to-persist-data-in-docker-container-2m72
- https://docs.docker.com/storage/volumes/
- https://gist.github.com/beeman/aca41f3ebd2bf5efbd9d7fef09eac54d#file-remove-all-from-docker-sh-L16

Docker SDK

- https://docker-py.readthedocs.io/en/stable/index.html
- https://docs.dockser.com/engine/api/sdk/

Apart from these, references were made to stackoverflow for debugging purposes.

ALGORITHM/DESIGN

The Application consists of 3 instances – Users, Rides and Database. The User Instance handles all the user related API calls i.e. create/view users and the Rides Instance handles the rides API calls. The Load Balancer redirects all the traffic to Rideshare application to the respective instance group: Ride or User group (The group consist of instances of Rides or Users) based of URL path of the API. Both the Ride and User Instance make API calls to the database instance to read or write the data.

The Database Instance consists of 3 parts: Orchestrator, Master and Slave.
Orchestrator:

Orchestrator handles all the request coming to the Database. Depending upon the URL of the request, the orchestrator performs appropriate action.

- Read Request – Orchestrator used a RabbitMQ queue, read queue and response queue to a make RPC (Remote Procedure Call) call to slave and retrieves the data from the database.
- Write Request – Orchestrator uses write queue to send data to the master to write to the database.
- Crash Master/Slave – Here the orchestrator uses docker SDK to connect to docker. It then finds the list of all containers which are running, iterates through this list to identify master and slave, by find then appropriate keyword in the process's command running on the containers.
- List Workers – This also uses docker SDK to connect to docker and find the list of containers which are running. It then identifies the slave workers. It finds it PID of the top process for each slave containers, sorts it and return the list of PID of the slave container.

Zookeeper Part in Orchestrator:

The orchestrator watches over the slave znodes, which are ephemeral that is they are deleted when the session closes. When the slave znode is deleted or the slave crashes, the orchestrator creates a new slave with update option set in system variable. It then waits for the node to be created, before it continues. Or else it run into infinite loop due to delay in creation of a new container.

Auto Scaling:

The orchestrator starts a counter upon the first read request. This counter calls a check_usage function every 2 minutes. The orchestrator keeps a count of number of read requests made and the number of slave instance running. The check_usage function compares request _count with running instance, and create/deletes the required number of slave instances (scaling up or down) such that there is a slave container running of per 20 read requests.

Worker (Master/Slave):

The worker takes two parameters during start up, first is type of node – master or slave and second is the update parameter. Each worker has its own database and run on a docker container of its own.

Master:

The master listens to write queue and update queue and performs write when request is made to the write queue. For each write request master writes to the sync queue.

Slave:

The slave listens to read queue and send the required data through the response queue. The orchestrator makes RPC call to the Read function, to get the data. Apart from this, when a new slave is created, the update parameter is kept set. If the update parameter is set, then the slave makes RPC call to master through the update queue and retrieves the entire database through the update_send queue. The slave listens to sync queue and makes a write to its database.

Zookeeper Section:

The slave watches over the master znode, which ephemeral. Then all the slaves stand of election to become the master. The one with the lowest PID gets elected to become the master. In which case the rest of the slaves opt out of the election to continue. The Elected master then reruns the python script as slave and a new slave container is also created by orchestrator to take the place the slave which turned.

RabbitMQ handles the queues. There are 6 queues running: write, read, response, update, update_send, sync. The master, slave and orchestrator use these queues to handle data transfer.

To check our design and development, we used Postman in order to run unit tests for the various components, and checked the responses as well as the database states accordingly.

TESTING

The testing challenge was

- Load Balancer issues: The request body parameter of database call did not match with database parameter, this issue got created during the porting of the database of Rideshare application.

CHALLENGES

The major challenges faced were

- Setting up individual database of each container. Using postgres image or python image showed persistent errors about connection not been established, connection refused and so on. This error was resolved by using a separate image and installing python and postgres separately and configuring it accordingly.
- Setting up the docker SDK. Docker SDK requires docker.sock socket file to make client connection to docker, which is not presents it default docker images. This is issue was resolved by mounting a volume of location of docker.sock file i.e. /var/run/docker.sock to the containers. The challenge was the discover of this issue as docker kept refusing the connection without any message.
- Zookeeper Maintenance Issues. Kazoo was used to ensure the when a worker crashes, a new worker is created, but kazoo's watch gets triggered every time any change is made

to znode, due to which every time a node was created and kazoo would create a new node which was not needed. This issue was fixed keeping a count of instance and checking it against number of children in the node to identify crashes from scaling.

- Changing a Slave to master. Upon a master crash a slave with lowest pid becomes the master. Now converting a slave to master was involves closing rabbitmq connection and resting ever queues, which was very difficult and show many errors. This issue was resolved but finding a way around it and rerunning the script again with different sys variables.
- Code reusage. Code was written by all of us at various points of time and reused from the older assignments. This lead to miscommunication and issues because of the way the older files were written and not being adapted properly for the new requirements

Contributions

The project is a collaborative effort.

Anshuman Pandey:

Design and Debugging of orchestrator, workers, handling AWS instances and connecting ride and user to DBaaS. Handling fault tolerance, auto scaling and data consistency.

Anuj Tambwekar:

Design and Debugging of workers and handling AWS instances and connecting ride and user to DBaaS. Setting up load balancing and endpoint APIs

Anirudh Maiya:

Assist and Debugging in the design of orchestrator and handling AWS instances.

CHECKLIST

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