17CS352:Cloud Computing

Class Project: Rideshare

Mini DBaaS for RideShare

Date of Evaluation:
Evaluator(s):
Submission ID:
Automated submission score:

SNo	Name	USN	Class/Section
1	Sparsha P	PES 1201700226	6 'A'
2	Shashank MG	PES 1201700298	6 'A'
3	Shreyas BS	PES 1201700956	6 'A'
4	Abhaay S	PES 1201701554	6 'A'

Introduction

- A RideShare application is built which services ride requests from various users. The application supports addition of users, rides, option for users to join other rides and fetch their respective data either according to the RideID or the source and destination of their rides. Separate “users” and “rides” micro services are employed to respond to the queries made. This application is coupled with a highly available and fault-tolerant database service which ensures that consistent data is always available to the user, providing a smooth experience.

Related work

- **Zookeeper**: http://zookeeper.apache.org/
- **Leader election**: https://zookeeper.apache.org/doc/r3.6.0/recipes.html#sc_leaderElection
- **RabbitMQ**: https://www.rabbitmq.com/getstarted.html
- **AMQP**: https://www.rabbitmq.com/tutorials/amqp-concepts.html
- **RabbitMQ Channels**: https://www.rabbitmq.com/channels.html
- **Docker SDK**://docker-py.readthedocs.io/en/stable/

ALGORITHM/DESIGN

The application is hosted on AWS(Amazon Web Services). The “users” and “rides” micro services run in containers on different instances. These micro services are Flask applications which consist of various APIs made available to the client. A call to these APIs perform the desired action. Every incoming request to the application first reaches an application load balancer, which distributes traffic across multiple targets and routes the query to either of the two instances based on the URI of the request.

In order to achieve this, listeners are configured with rules and added to the load balancer. The listeners are then integrated with target groups, which comprise the “users” and “rides” instances. The incoming traffic is forwarded to the corresponding target group by the listener.

The APIs in these instances access data by making read and write calls to the database, which is handled by the DBaaS system. It functions as follows.

The first point of contact for all read/write calls to the database is the orchestrator. It is a Flask application running in a container, and establishes connection with the RabbitMQ and Zookeeper containers. These latter containers are built from images directly pulled from the Docker Hub registry. The read/write APIs exposed by the orchestrator consume the requests and write them to their respective message queues, which follow the AMQ(Advanced Message Queueing) protocol. The three message queues connected to the orchestrator are “writeQ” ,“readQ” and the “responseQ”. RabbitMQ is used as the message broker.

The other end of these queues are the worker containers. These containers are each connected to a MySQL database container. The master container performs the actual write to the database, according to the query written into the “writeQ”. To achieve consistency, the master creates a new message queue called the “syncQ” and copies into it the messages in “writeQ”. The master also writes all the queries in “writeQ” into a replica file, the use of which is explained later. All the slave containers listen to the “syncQ” and update their respective databases by performing writes. The slave containers are mainly responsible for handling the read requests written into the “readQ” by the orchestrator. The slaves read from the database according to the query in “readQ“ in a round-robin fashion and write the results into the “responseQ”, which is finally consumed by the orchestrator and returned back to the client. This prevents overloading on a single slave.

High availability of the workers is ensured through the Zookeeper. The orchestrator first creates a znode called “election”. Every worker on getting spawned creates its own ephemeral znode under “election”, named according to its PID. The orchestrator sets a watch on every znode under “election”, which enables it to detect a missing znode in case of a slave container failure. The orchestrator then spawns a new container and copies the replica file made by the master into it. This file is executed by the container, which makes its own database consistent. The container then runs the slave code.

Every worker container consists of both the master and slave code, divided into functions. Initially, the container having the smallest PID runs the master code and every other container runs the slave code. The worker containers in turn have watches set on the znodes occurring just before them in order of their PIDs. This is useful during leader election. In the general case where some container fails, the container watching it checks if it is the znode with the smallest PID. If yes, it runs the master code. Else, it is made to set watch on the znode which comes just before it in order of PIDs and continues running the slave code. It follows from this that the container watching the master container is next in line to become the master, in case the original master fails.

The orchestrator is also made to perform autoscaling for efficient responding. By keeping track of the number of read requests made, the number of slave containers are either increased or decreased.

Since spawning of new workers takes some time, the orchestrator becomes unresponsive for this duration, defeating the idea of high availability of the application. In order to combat this, a threaded model is employed where every new worker spawn is run in a separate thread, so that the orchestrator can continue responding to requests made by the client.

TESTING

During the beta testing, the autoscaling feature did not pass and the number of requests getting counted were less than the actual number of incoming requests.

We figured out that the counting mistake happened due to the usage of files to keep count. Every time a request was sent, the data in the file was read, incremented and written back to the file. Unfortunately the speed at which writes and reads happened were not as fast as the speed at which requests were being sent by the tool, which gave inconsistent count of requests. We then used a global variable for incrementing the read requests which solved the problem. The autoscaling then worked normally.

CHALLENGES

We faced the following challenges:

- Since we used MySQL database, running the application and the database service in the same container for the workers was not possible. This challenge was overcome by creating a separate container for the database and linking it to the application container via the "mysql.connector" module.
- Making the orchestrator highly available was a challenge due to the time taken to spawn a new worker. This was overcome by making the orchestrator a threaded model, where every new worker spawn was run in a different thread.

Contributions

- Sparsha P: Worked on fault tolerance of slave containers, syncQ ,responseQ
- Shashank MG: Worked on leader election, readQ, writeQ, syncQ and fault tolerance of slave containers
- Shreyas BS: Worked on leader election, readQ, writeQ, responseQ and autoscaling
- Abhaay S: Worked on the new APIs, autoscaling, setting up the orchestrator

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.	