17CS352:Cloud Computing

Class Project: Rideshare

Cloud-Based Application for Pooling Rides

Date of Evaluation: 18/05/2020
Evaluator(s): Prof. Usha Devi BG, Prof. Sanjith Athlur
Submission ID: 443
Automated submission score: 10

SNo	Name	USN	Class/Section
1.	Swathi M	PES1201700256	6E
2.	Prakruti Prakash Rao	PES1201701126	6E
3.	Sneha Nemadi	PES1201701333	6E
4.	Pushpavathi K N	PES1201701347	6E

Introduction

Mini DBaaS Architecture For RideShare Application.

This project is focused on building a fault tolerant, highly available database as a service for the RideShare application. We have worked with Amazon EC2 instances to make it cloud-based. We have used users and rides VM, their containers, and the load balancer for the application. In addition now, we have enhanced our existing DB APIs to provide this service.

The db read/write APIs is now exposed by the orchestrator. The users and rides microservices are no longer using their own databases, they will instead be using the "DBaaS service" that we created in this project.

We have implemented a custom database orchestrator engine that will listen to incoming HTTP requests from users and rides microservices and perform the database read and write according to the provided problem specifications.

Related work

Some references that were very helpful to us in this project are

1. https://www.rabbitmq.com/tutorials/amqp-concepts.html
2. https://www.rabbitmq.com/getstarted.html
3. https://kazoo.readthedocs.io/en/latest/
4. https://docker-py.readthedocs.io/en/stable/
5. https://apscheduler.readthedocs.io/en/stable/

ALGORITHM/DESIGN

The entire RideShare Application consists of the following components:

1. **Rides Microservice**

   This microservice consists of all the APIs related to creation, deletion and modification of rides.

2. **Users Microservice**

   This microservice consists of all the APIs related to creation, deletion and modification of users. also includes the APIs related to users joining rides.

3. **DBaaS**

DBaaS:

Each part of our project consists of a unique design and implementation to make DBaaS fault tolerant, scalable and highly available.

The entire DBaaS system consists of the following components:

- Master Worker
- One or more Slave Worker(s)
- Orchestrator
- RabbitMQ
- Zookeeper

All of these components run on the same VM as an individual container and together constitute DBaaS. Design and implementation of each of the above components is as follows:

- **Master Worker :** The master worker has its own copy of the DB. It receives messages from the Write Queue and writes them into its DB along with sending them to the fanout exchange for sync.

- **Slave Worker:** The slave workers are used in a round robin fashion to respond to read requests. On receiving a message from the Read Queue they send a response back to the Response Queue. Each slave worker has its own copy of the DB. Copy of the DB is done from the master to the slave each time a new slave is created. Each slave receives messages from the fanout exchange through a temporary sync queue and writes them in its DB.

- **Orchestrator :** Orchestrator is the most important part of our DBaaS. It consists of multiple functionalities which are as follows:

    - **Read API:** Orchestrator has the endpoint for the Read API.
    - **Write API:** Orchestrator also has the endpoint for the Read API.
    - Creates the initial master container with Docker SDK.
    - Creates the initial slave container with Docker SDK.
    - **List Workers API :** Gives a list of the PIDs of all the current workers.
    - **Crash Slave API :** Crashes the slave with the maximum PID.

    - **Slave Fault Tolerance :**

      This is a main functionality of the orchestrator. It has been implemented using Zookeeper. An ephemeral znode is created under a common parent directory each time a slave worker is created. A watch function is placed on this parent directory. The childrenWatch API is used along with the watch function in the orchestrator to keep a watch on all the children znodes of this parent directory. The watch function is triggered each time a znode is created or deleted under the parent directory i.e when a slave is created or deleted. As a ephemeral znode is used, it is created/deleted when the slave is created/deleted.

When the watch function is called on deletion of a znode, it indicates the crash of a slave and a new slave is created using Docker SDK. We identify whether the watch function has been called for the creation of a znode or for the deletion of a znode by keeping track of the number of children znodes in a global variable. Then this global variable is compared with the number of children in the list of children each time the watch function is called. If it is greater, then a znode has been deleted i.e a slave has crashed.

Scaling down however has not been treated as a failure. It has been handled by setting a global variable to true each time scaling down is happening, and when that variable is set to true and the watch function is called on deletion of a slave, a new slave is not created.

- **Scalability :**

  This is another main functionality of the orchestrator. It has been implemented as follows. The number of slaves currently running is stored in a global variable and is initialized to one when the system is started. Another global variable is used to keep track of the number of read requests received by DBaaS and is incremented each time a read request is received in the Read API.

  Once the first read request is received by the Read API, a job scheduler is invoked which calls a job function every two minutes. The job function calculates the number of slaves which are required to be running based on the number of read requests which have come in the last two minutes. Then it compares the number of slaves required to the number of slaves currently running. If the number of slaves currently running is less than required, then it creates as many slaves as required. Else if the number of slaves currently running is greater than required, then it kills many slaves as required. Hence DBaaS is made highly scalable.

- **Zookeeper :**

  Zookeeper container is used as the decentralised system to retrieve from and store the ephemeral znodes which are used in slave fault tolerance as explained above.

- **RabbitMQ :**

  RabbitMQ has been used for management and implementation of queues. A total of 6 queues have been used in our project. Their usage is as follows:

  - ***Write Queue :*** This queue is used to take all the write requests and send them to the master worker. So, the publisher for this queue is the orchestrator and the subscriber is the master worker. Every write request sent to DBaaS is received by the Write API in the orchestrator is sent as a message on this queue and received by the master and written into its database. Default exchange with a named queue has been used.

  - ***Sync Queue :*** Each time a message is added to the Write Queue, in the callback function of the Write Queue, the same message is added to a fanout exchange for sync. Each time a slave is created, it also creates a temporary queue which receives messages from the fanout exchange. Hence each slave worker has a temporary queue associated with the fanout exchange, and on every write in the master, the same message is passed to the slaves through the sync queue via the fanout exchange for sync to happen in each slave. Hence 'eventual consistency' is achieved in each slave.

- ***Read Queue and Response Queues:*** Both these queues have been implemented along with a RPC. Each slave worker is an RPC server which serves the response to a read request and the orchestrator is the RPC client. So the orchestrator on receiving a read request in the Read API, sends a message to the Read Queue. The Read Queue messages are sent to the slave in a round robin manner, which has been implemented by making the Read Queue a task_queue. The slave which receives the message from the Read Queue reads from its DB and puts the response in the Response Queue with the appropriate correlation ID. The orchestrator receives this response from the Response Queue.

- ***DB Copy Request and DB Copy Response Queues:*** Both these queues have been implemented along with a RPC. Here each newly created slave worker is an RPC client and the master worker is an RPC server. These queues have been used to develop a method to asynchronously copy the database to a newly created slave. A newly created slave worker sends a RPC request message to the DB Copy Request Queue for the DB and the master worker sends all the data of its DB as a response in the DB Copy Response Queue.

TESTING

Some of the major testing challenges we encountered were :

1. As RabbitMQ, DockerSDK and Zookeeper were all new technologies that we learnt for this project, understanding what error messages each one of them gave for what was challenging.
2. Containers created by Docker SDK run as a background process hence debugging errors in them was a tedious task.
3. As there were multiple components involved in this entire project, when an error occurred, figuring out which component of the project was throwing the error was a challenging task.

We did not have any issues with the automated submission and got a 10 on the first submission.

CHALLENGES

1. Understanding and learning each new tech (i.e RabbitMQ, Zookeeper, DockerSDK etc.) from scratch.
2. Figuring the exact relationship between each component of the project and how exactly they should interact with each other for DBaaS to work.

3. Creation of a container using DockerSDK, and understanding the attributes that should be passed to create it.
4. Figuring out an async method to copy the DB on creation of a new slave.
5. Debugging each other's code and collaborating offline was a major challenge and learning outcome of this project.

Contributions

1. Swathi M (PES1201700256)

   Setting up the DB for workers, setting up the RabbitMQ connections, implementing Write Queue, Read Queue with Response Queue, implementing DB copy using 2 queues and RPC, figuring out the usage of zookeeper, documenting the code and report.

2. Prakruti Prakash Rao (PES1201701126)

   Basic setting up of all the containers in the compose file, implementing DB Sync, figuring out the usage of Docker SDK to creation and deletion of new containers during scaling, modifying Read and Write APIs to work with queues in the orchestrator, accessing the PID of a slave worker from the orchestrator, Zookeeper for slave fault tolerance and recovery, load balancer and report.

3. Sneha Nemadi (PES1201701333)

   Keeping count of the incoming read requests in the Read API, figuring out how to use low level DockerSDK API for killing containers, figuring out scaling logic and code in the orchestrator to create/kill slave worker containers as required using BackgroundScheduler, crash slaves API and crash master API, list worker API based on their PID.

4. Pushpavathi K N (PES1201701347)

   Setting up the DB for workers, setting up the RabbitMQ connections, implementing Write Queue, Read Queue with Response Queue, figuring out usage of Zookeeper, figuring out the usage of BackgroundScheduler to keep track of incoming requests and do the scaling accordingly.

CHECKLIST

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