17CS352:Cloud Computing

Class Project: Rideshare

Database as a Service Project Report

Date of Evaluation: 18/05/20
Evaluator(s): Prof. Pushpa, Prof. Nitin
Submission ID: 156
Automated submission score: 10

SNo	Name	USN	Class/Section
1.	Raunak Sengupta	PES1201700072	6 G
2.	Sai Prashanth R S	PES1201700206	6 E
3.	Yash Pradhan	PES1201700262	6 G
4.	Vishwas B Rajashekar	PES1201700704	6 H

Introduction

RideShare is a fault-tolerant, highly available cloud based application which offers the facility of allowing users to pool rides. It allows the users to create a new ride if they are travelling from Destination A to Destination B. Other functionalities the app offers are adding a new user to the ride, deleting an existing user, creation of a new ride, joining an existing ride etc. The entire application is deployed on Amazon Web Services. In order to take on a large number of read requests, the system automatically scales up depending on the number of requests. It is also highly available, it respawns workers if they crash. The database is eventually consistent - powered by AMQP.

Related work

https://www.rabbitmq.com/tutorials/tutorial-six-python.html

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

https://apscheduler.readthedocs.io/en/stable/

https://www.allprogrammingtutorials.com/tutorials/leader-election-using-apache-zookeeper.php

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

https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html

ALGORITHM / DESIGN

The requests to the application are **load balanced** based on route to either the users or rides microservice.

The application has **two microservices (users, rides)**. These two microservices will be making use of a **shared database as a service** (DBaaS). DBaaS consists of a database **orchestrator** engine which listens to the incoming HTTP requests from the users and rides microservices to perform the database operations such as read and write. It is the orchestrator's responsibility to **forward the message to the right queue** and **spawning new master/slave workers** based on requirement.

RabbitMQ is used as the message broker, we made use of **Pika** (a **RabbitMQ python client**). This implementation makes use of 3 named message queues:

ReadQ : read requests published here

WriteQ : write requests published here

SyncQ : special queue that is bound to a **fan-out type exchange** in order to distribute each message to all the consumers (slave containers).

In addition, there are **anonymous queues** defined for **Remote Procedure Calls** (RPC) returns.

There are two types of workers being used here i.e. **master, slave**. The master worker listens to the WriteQ and makes all those changes in the Persistent database which in our case is stored in a separate container called **Persdb**. The slave workers on the other hand are responsible for all read requests coming to the orchestrator, and after querying the database response is written to an **anonymous reply queue**. The read requests are **load balanced** among all the slaves currently running. **Eventual consistency model** is used to ensure consistency between databases of master and slaves. This is implemented using **SyncQ**. All the write requests are forwarded from the **master** to the Persdb, and then **SyncQ** shares this with all the workers. A **subprocess** runs on each of the worker containers that listens to the SyncQ and performs a **synchronisation** operation to make the database consistent.

The Orchestrator also offers:

Scalability: We monitor the incoming HTTP requests made and increase / decrease the number of slave containers to manage the load. Scaling is done for every additional 20 requests. Check this over a duration of 2 minutes post which the timer is reset. The timer is implemented using the Advanced Python Scheduler (APScheduler) Library, which is perfect for executing jobs periodically. The current slave count is maintained in zookeeper in the znode **'/slavecount'**

1 slave container for 0 - 20 requests

2 slave container for 20 - 40 requests   and so on

High Availability: Upon **failure** of a slave worker, a new slave worker is spawned and if the master worker fails, one of the slave worker nodes is **elected** to be the master. Criteria used for leader election is the least container process ID.

The master and slave workers run the same code which responds to a **role assignment/reassignment**. When the orchestrator spawns a worker, a role is assigned to it. This role assignment triggers a **subprocess** spawn which executes the master or slave specific task of writing to or reading from the database. A role can be **reassigned** by killing the subprocess and starting a new one.

When the application starts, two containers are spawned with jobs master and slave. When the containers are created, their PIDs are obtained by taking the **PPID** (parent PID) of the **root process of the container** returned by **container.top()** function in the docker SDK. This is the **containerd-shim** process's PID in the host, which is in fact the container's PID.

When a role is assigned to a worker, it creates a znode in Apache Zookeeper by the path **'/conts/cont_<its_pid>'**. Sequential znodes are **not** used so as to avoid race conditions where a worker with a higher pid is assigned a lower sequence number in its znode. The node created is marked **ephemeral** so when the container's session with zookeeper ends (when it crashes), the znode is deleted as well. The node creation is also set to **ensure_path** so the **'/conts'** node is created if it doesn't exist.

In the orchestrator, when the role assignment API of the worker responds, a **watch** is set on the newly created znode depending on the role assigned. The **master watch** is triggered when the master crashes. The **lowest PID container** is chosen among all the running slaves and its role is changed to master. In this process, its znode is **deleted** (triggering the slave watch on it) and **recreated**. Then a **new slave** is spawned to replace it. The **slave watch** sleeps for 1 second and **rechecks** if the znode exists. If yes, that means its role was changed to master. If not, it checks if the current number of active slaves is less than the number stored in **'/slavecount'** in zookeeper. If yes, then a slave has genuinely crashed and a new slave is **spawned**. If not, the slave was killed by the scaling function of the orchestrator. This accounts for the **contrast** between high-availability and scaling.

The structure of the znodes in zookeeper is:

```
                    /
            /              \
          conts          slavecount
      /     |     \
cont_113 cont_256 cont_518 ...
```

TESTING

Local testing was carried out using Postman. We developed a Postman Collection containing essential API calls and appropriate Postman environments for local and cloud deployment.

Load testing (for the scaling module) was carried out using Apache JMeter as well as Postman Runner.

API unit testing was done using a custom built Pytest module containing sample inputs, and expected outputs and status codes.

CHALLENGES

1. Requests placed in the ReadQ and WriteQ will immediately return if just normally added to the queue.
    a. Used RPC to allow operation to occur before returning response.

2. RPC causes infinite loop on a container when there is no available worker to service the request.
    a. Used a timeout for RPC. Returns 503 if there is no available worker.

3. Editing a container's role (master/slave) can be subject to a race condition.
    a. Used Mutex locks on the roles.

4. Synchronisation on startup of a new worker is not elegant with using rabbitmq since one queue cannot have multiple consumers receiving all messages.
    a. Used a function that retrieves the file from the persistent db before the container flask server starts up.

5. Retrieving a container object post start-up via the default container name was leading to weird behaviour.
    a. Used the container object's IP address instead

6. Getting the workers' container PID from inside the orchestrator was tricky.
    a. The container.top() function of the docker SDK returns a list of its inner processes (sorted ascending by PID). So the first process's PPID (parent PID) is taken. The PPID belongs to the containerd-shim process with the container ID in the host so it's accurately the container PID necessary.

7. Clash between the behaviour of scaling-in and high-availability caused slave workers to respawn when killed by the scaling functionality of the orchestrator.
    a. The current required slave count is stored in the race-condition free super-fast data store zookeeper in a separate node and retrieved by the slave watch to check if a container genuinely crashed.

8. Slave watch still persists on a slave reassigned as master.

a. When the slave's role is reassigned, its znode is deleted and recreated to remove the existing slave watch and set a master watch. The slave watch waits to see if the znode is not recreated to ensure a slave really crashed.

Contributions

Team Member	Contribution
Vishwas Rajashekar	RabbitMQ, named queues: ReadQ, WriteQ, SyncQ, fanout exchange for SyncQ, anonymous queues for RPCs, eventual consistency model
Raunak Sengupta	Zookeeper for master fault tolerance: slave leader election via lowest PID, slave fault tolerance: respawn slave if not killed by scaling (balance HA and scaling)
Sai Prashanth R S	Creating the new APIs for Crashing Slave, Crashing Master and Retrieving Worker List, comprehensive testing using scripts and tools like Postman.
Yash Pradhan	Implementing the Python Docker SDK, Dynamic Spawning of worker containers for scaling, background job scheduling for scaling, scripts for testing.

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