17CS352: Cloud Computing

Class Project: Rideshare

A Service for pooling rides

Date of Evaluation: 18/05/2020
Evaluator(s): Prof. Srinivas K S
Submission ID: 499
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Keshav Kansal	PES1201700079	6E
2	Parshva B Jain	PES1201701336	6E
3	P Sai Krishna	PES1201701515	6E
4	Surya R	PES1201701873	6E

Introduction

Rideshare is an application which facilitates pooling of rides among users. In this project the Backend for the application has been developed. API's for managing 'rides' and 'users' are exposed to the users, These API's contain the main logic of the Application. The Data for the application is not stored on a trivial Database, but instead stored on the Database-as-a –Service (DBaaS). The read and write API's of this DBaaS is exposed to the main logic managing this 'rides' and 'users' which uses this Service as its Database.

The API's for managing 'rides' and 'users' are divided into separate services and are connected to a Load Balancer which distributes the Requests to the appropriate Service. The Address of this Load Balancer is exposed to the users. Each Service is a Flask Application which receives requests redirected from the Load Balancer, process the request and sends the appropriate response back.

The DBaaS which is the main focus of the project is a Highly Available and Scalable Service built to facilitate the read and write operations performed on the Database. This service comprises of two components Orchestrator and Workers.

The Orchestrator is a Flask Application which serves the read and write API's, upon receiving a request the Orchestrator sends the request to a Worker. Which performs the requested Database operation and sends the response back to the Orchestrator. Another vital task that Orchestrator performs is the Managing the number of workers.

Each Worker has a separate Database exclusive to the Worker, this ensures data replication. The main job of the worker is to receive requests from the orchestrator, perform the requested operation on its database and send a response back. There are two types of Workers - Master and Slaves. A Master is a worker which only processes write requests. At a time only one Master is present. A Slave worker only process read and sync requests.

Related work/References

1) Tutorials on RabbitMQ - https://www.rabbitmq.com/getstarted.html
2) Pika Documentation - https://pika.readthedocs.io/en/stable/
3) Kazoo Documentation - https://kazoo.readthedocs.io/en/latest/
4) Zookeeper Articles on Medium.

ALGORITHM/DESIGN

1) Internal Communication between the Orchestrator, Master and Slaves

Whenever a read or write request is issued to the Orchestrator ,it sends the requests to the Worker this kind of communication is established using queues a concept in RabbitMQ. In our implementation we have used totally 7 Queues:-

1) readQ – read request/Messages are published by the Orchestrator and are consumed by the Slaves.
2) responseQ – response Messages are published by the Slaves containing the response of the read request received from the Orchestrator
3) writeQ – write request/Messages are published by the Orchestrator and are consumed by the Master.
4) writeResponseQ – response Messages are published by the Master Containing the response of the write request received from the Orchestrator.
5) syncQ – write requests / Messages are published by the Master and Consumed by the Workers , Each worker has a separate syncQ connected to a fan-out exchange.
6) killQ – Messages are published by a Slave to itself to stop consuming messages/stop operating as a slave.
7) eatQ – This is a persistent Queue from which no messages are deleted. Messages are published by the Master and consumed by the newly spawn Worker to get to achieve a consistent database.

Request Lifecycle:-

When a Read Request is received by the orchestrator it publishes the message to the 'readWrite' Exchange with a unique id for identification and routing key as 'read' which leads the message to be published in the read queue. All Slaves consume messages from the readQ, however messages are distributed among the Slaves in a Round-Robin Fashion, So a single Slave receiving the request performs the read operation and sends the response back with the same unique ID to the 'readWrite' exchange which ends up in the responseQ. When the orchestrator receives this request it verifies that it is the response of the request it sent by matching the unique id, reads the data and sends back to the client.

For Write Requests almost the same procedure is used, but since Write requests are processed only by the Master, the requests are sent to the master which receives requests from the writeQ and sends the response back to the writeResponseQ. Additionally whenever a write operation is successful the master publishes the same write request to the fan-out 'sync' exchange. As mentioned before all the Slaves consume messages from an exclusive syncQ. All the Slaves receives this write message from the 'sync' exchange into their syncQs and they perform the updates on their Database. This ensures consistency between the database of the master and the Slaves.

A persistent queue called as the eatQ is attached to the 'sync' exchange. All the messages which are published to the eatQ are durable and hence are never removed from the Queue. This plays an important role when a new Worker is started up.

2) High Availability & Scalability

To achieve Highly Availability, Zookeeper is used. The Zookeeper nodes structure is the following:-

- /Container_pid
- /Election
    - Master
    - Slaves

1) Container_pid is managed by the orchestrator, it contains the information of all the Workers which the orchestrator knows are 'supposed' to be working. The orchestrator maintains a mapping between the containers and their PID in the child nodes, also mapping between the workers and their database container id is maintained here.

2) Election/Master is a node which contains the PID of the master worker.

3) Election/Slaves is a node where the each worker creates an ephemeral Child node

Worker Start-Up procedure:-

The Orchestrator manages all the workers when a new worker container is started up it creates a node in /Container_pid which containers the worker name, PID and the database container id. On startup the first thing a worker it does is to get in sync with the other present workers , it does so by reading all messages from the eatQ. It then fetches its PID from the same node and creates an ephemeral node in /Election/Slaves. It then looks if a master is present or not by checking the /Election/Master node , If it is present it assumes its role as a slave and sets a DataWatch on /Election/Master otherwise it creates the node and assumes the role as a master.

Worker Crash:-

When a worker crashes the ephemeral node created by it in /Election/Slaves is deleted. The orchestrator which has a ChildWatch set on the node gets notified. It then removes worker entry from the Container_pid and stops the mongodb container is initially running and creates a new worker.

Scalability:-

The Orchestrator keeps track of all the number of workers using the Container_pid node in zookeeper. It counts the number of read requests it received for 2 minute intervals appropriately stops or starts new workers.

Master Election:-

Whenever a Master crashes the ephemeral node /Election/Master created by the master gets deleted. All the Slaves are notified about this. The Slave then fetch the the PID of all running Slaves from 'Election/Slave' and find the lowest PID. They compare their PID with the Lowest PID if the PID is not the same they continue their role as a Slave otherwise they stop receiving read requests by sending a message to killQ and assume the role as a Master.

TESTING

There were not many problems in while testing the code on the Automates Submission Platform However problem encountered was regarding the Crash API, Our system did not allow crashing a Slave when only two slaves are alive. This check was not allowing a worker to be stopped and lead to failing the Test. Upon removing this check the test passed.

CHALLENGES

Two Major Challenges:-

1) Pika not being thread Safe – Pika the client used for RabbitMQ is not thread safe so while a Worker has to transition from a Slave to a Master the Pika Connection when closed from another thread (where the zookeeper watch is triggered) gives an error and the program crashes. So the killQ was implemented to overcome the problem and close the connection from the thread the connection was established from.

2) No Heartbeats from orchestrator – In our design a single Pika connection is used throughout so when the Orchestrator is idle, it is unable to send heartbeats and hence the connection is closed by RabbitMQ, Hence Heartbeats were removed for the orchestrator.

Contributions

Keshav Kansal – Worked on RabbitMQ, studied Pika documentation, design and implemented the Architecture, Docker.

P Sai Krishna – Worked on Zookeeper, studied the Kazoo documentation, implemented the Leader Election algorithm, design and implemented the architecture.

Parshva Jain –Worked on Zookeeper, studied the Kazoo documentation, Implemented the Leader Election, design and implemented the architecture.

Surya R – Worked on RabbitMQ ,Testing and debug, Docker, Made the Orchestrator API's, design and implemented the architecture.