17CS352 Cloud Computing

Class Project: Rideshare

MINI DBaaS FOR RIDESHARE

Date of Evaluation: **16 - 05 - 2020**
Evaluator(s): **Srinivas K**
Submission ID: **1194**
Automated submission score: **10**

SNo	Name	USN	Class/Section
1	Tanvi P Karennavar	PES1201700646	6B
2	Bhavya Charan	PES1201700032	6B
3	Aditi M Manohar	PES1201700210	6G
4	Arya Rajiv Chaloli	PES1201700253	6B

Introduction

The project is focussed on building a fault tolerant, highly available database as a service for the RideShare application. The *users* and *rides* microservices will be using the DBaaS service that is created in this project instead of using 2 separate databases. We would be implementing a custom database orchestrator engine that will listen to incoming HTTP requests from the *users* and *rides* microservices and performs the database read and write according to the given specifications. High availability (using Zookeeper) and scalability (implemented in the orchestrator) has been the prime objective.

Related work

The following are the key material that we have read and referenced to implement this project.

Zookeeper:	http://zookeeper.apache.org/
Kazoo:	https://kazoo.readthedocs.io/en/latest/
RabbitMQ:	https://www.rabbitmq.com/getstarted.html
AMQP:	https://www.rabbitmq.com/tutorials/amqp-concepts.html
Docker SDK:	https://docker-py.readthedocs.io/en/stable/

ALGORITHM and DESIGN

Overview

The *orchestrator* manages the whole system. It listens to requests and routes them to the appropriate components. The *readQ*, *writeQ*, *syncQ* and *responseQ* are the queues maintained for the transfer of information and requests across the different components (*workers* and the *orchestrator*). There are two types of *workers*, the *master* and the *slave*. In the scope of this project, it is limited to one *master* with multiple *slaves* (which is scaled up and down according to the need). The *master* only takes care of the writes into the database, whereas the *slaves* cater only to the read requests. All the *workers* and the *orchestrator* run on their respective containers.

When the *orchestrator* encounters a read request, it publishes the request to the *readQ*. One of the *slaves* (the *slaves* are constantly listening to this queue), picks up the request and publishes the response in the *responseQ* which is read by the *orchestrator*. As and when the number of requests vary, the number of *slaves* are proportionally scaled up or down by the *orchestrator*. Newly created *slaves* start off by replicating the database of the *master*.

When the *orchestrator* encounters a write request, it publishes the request to the *writeQ*. The master that is constantly listening to this queue picks it and updates its database. All the new database writes are published in the *syncQ* by the *master* after every write that it does, which is picked up by the *slaves* to update their copy of the database to maintain consistency.

To maintain high availability, new containers are started whenever a *master* or a *slave* fails. If a container acting as a *master* goes down, a new *master* is chosen from the currently running *slaves* and when a *slave* goes down, a new *slave* container is started.

The following sections elaborate on the implementation in further detail.

Orchestrator

The *orchestrator* is a flask application, which upon receiving a call to specific APIs will publish relevant messages to the respective queues. The redirection of the read, write, crash (both *master* and *slave*), and list APIs are implemented here. The scaling of *slaves* is also handled by the *orchestrator* .

Replication

Newly created slaves start off by asynchronously replicating the database of the master. We have used "*SyncQ*" in order to maintain consistency. Whenever a new write request comes to the orchestrator, through fanout exchange we are publishing the request to all the queues. Hence all the workers will consume from this "*SyncQ*" and the consistency is maintained.

Syncing

In our case we have multiple *workers* running simultaneously. Each *worker* (*slave* and *master*) will have its own copy of the database. The database is not being shared among the *workers*. Having separate databases creates the problem of maintaining consistency between the various replicas of the same database. In order to solve this issue, we have used an *eventual consistency* model, where after a write has successfully been performed by the *master* worker, it is eventually consistent with all the *slave* workers. For implementing *eventual consistency*, the *master* writes to the new database on the *syncQ* after every write that *master* does and this is picked up by the *slaves* to update their copy of the database. All the *workers* will run in their own containers but they will connect to a common *RabbitMQ*.

Scalability

The *orchestrator* keeps a count of all the incoming HTTP requests for the database read APIs. The auto scale timer begins after receiving the first request. After every two minutes, depending on how many requests were received, the *orchestrator* proportionally increases or decreases the number of *slave* worker containers. Every two minutes, based on the count of the number of requests, for every increase in twenty requests, the number of *slave* containers that are running is incremented by 1. Similarly, for every decrease in twenty requests, the number of *slave* containers that are running is decremented by 1. After altering the number of *slaves* containers, the counter is reset, and the number of requests are checked again in two minutes.

Availability

The *zookeeper* is being used to monitor all the *workers*. Whenever a *master* fails, amongst the existing *slaves*, the *slave* whose container has the lowest pid, is elected as the new *master*. Also, a new *slave* is created to replace the *slave* that was elected as *master*. If a *slave* worker fails, a new *slave* worker is started and all the data is copied to the new *slave* asynchronously.

This is implemented by creating a main node called */election* which acts as a parent node. All the children nodes are created under this (Eg - */election/node_1*). The main node is created in the orchestrator and the children nodes are created in the workers. The data of the main node is the PID of the master container and the data of the children nodes is the PID of the corresponding container.

On the main node, a watch is set on all its children using *@zk.ChildrenWatch("/election").* This ensures that whenever a child has been killed, a new container can be started in place of it. Whether to start a master or slave is done by checking the list of all the children nodes and their corresponding PIDs. Then the minimum PID is found out and compared with the data of the main node. If the data matches, then we can say that a slave node was killed. But if the data is different, i.e. we have found a new smallest PID, then the master must have been killed so a new master is elected by setting the data of the main node with this smallest PID. All the children nodes maintain a data watch on the parent using *@zk.DataWatch("/election")*. Thus when the data of the parent node changes, this triggers a watch on all the children and it allows the children to check their own PID with that of the main node. If the data of the main node matches the PID of the worker, it realises that it is now a master and thus runs the master code. And if the data is different, then the worker behaves as a slave and runs the slave code.

TESTING and CHALLENGES

While building the various components of the project, each of them were built separately and tested on a small scale to check if the basic functionality was working. They were then combined one by one and tested at every step to make sure they all worked fine together.

The issues we faced during the automated submission were:

The auto scaling was failing since it was taking time to crash the slaves ,so we resolved this by simply adding sleep(10) in our code.

Contributions

Tanvi P Karennavar:	Orchestrator implementation with RabbitMQ
	Replication and Sync
Bhavya Charan:	High availability using Zookeeper
Aditi M Manohar:	Scalability
Arya Rajiv Chaloli:	Scalability

Checklist

Sl. No.	Item	Status
1	Source code documented	Done
2	Source code uploaded to private github repository: https://github.com/BhavyaCharan/Cloud_Computing	Done
3	Instructions for building and running the code. Your code must be usable out of the box.	Done