17CS352:Cloud Computing

Class Project: Rideshare

Fault Tolerant and Highly available DBaaS

Date of Evaluation: 20th May, 2020
Evaluator(s): Rachana B S & Deepthi
Submission ID: 917
Automated submission score: 10

SNo	Name	USN	Class/Section
1	Inumella Sricharan	PES1201701114	6H
2	Nishikanth C.S	PES1201701411	6H
3	Sharath V	PEs1201701727	6H

Introduction

- The project is to build a DBaaS for Rideshare application, which is fault tolerant, highly available.The DB read/write APIs from the previous assignments are made as the endpoints in the orchestrator.The users and rides will no longer have their own DB.We have used AMPQ (rabbitmq) to serve requests by workers.The DBaas follows a master slave architecture, with master serving write requests and slaves serving read requests.
- Scaling of worker containers according to the incoming requests using Docker SDK.
- Implementation of Slave Fault Tolerance using Zookeeper and Docker SDK.
- Implementation of Leader Election using Zookeeper : In case of failure of master, existing slave container with the lowest pid is elected as the master.

Related work

The rabbitmq official documentation, Docker SDK documentation, Zookeeper documentation, stackoverflow.

kazoo - https://kazoo.readthedocs.io/en/latest/

Docker SDK - https://kazoo.readthedocs.io/en/latest/

ALGORITHM/DESIGN

The DBaas system has an orchestrator, which listens to read and write requests and publishes into appropriate queue.Workers will elect a master, and remaining workers will act as slaves, they run the same code.The master listens to write queue, the slaves listen to read queue.

The flow:-

```
1.Check if the worker is a slave or master.
2.If the worker is master
        a.check if there is a request to sync a new slave and serve the request.
        b.check if there is a request to write to DB
        c.Repeat from step 1
```

```
3.If the worker is slave
        a.If a new Slave
                -Sync the db from master
        b.Else
                -Serve if any read request present or sync the db with new rite
                .
        c.Repeat from  step 1
```

We have used many queues, read_ queue, write_queue, read_response_queue, write_response_queue, read_response_queue.

Orchestrator publishes a read request to read_queue and one of the slaves serves the read request and returns the response to read_response_queue.Orchestrator publishes write request to write_queue and returns the response to write_response_queue.The orchestrator picks up the responses and returns to user or ride instance, RPC is used for this.

Each slave has temporary queues to sync from the master, the master publishes the sync messages to sync_queue and slave reads from the sync _slave to update it's DB.

Queue	Publisher	Consumer	Exchange for the queue
Read_queue	Orchestrator	Slave	orchestrator_exchange, type=direct
write_queue	Orchestrator	Master	Orchestrator_exchange, type=direct
read_response queue	Slave	Orchestrator	default, type=direct
write_response _queu	master	orchestrator	default, type=direct
temporary queue for each queue	master	respective slave	sync_exchange, type=fanout

request_sync_queue	slave	master	slave_sync_exchange
request_sync_queue	master	slave	slave_sync_exchange

ZOOKEEPER SETUP

- In the orchestrator code, we create two parent nodes, which are namely, /***slave_ft*** and /***election***, both of which are sequential and ephemeral.
- /***slave_ft*** is used for appending the znode of slaves for the purpose of slave fault tolerance and /***election*** is used for the Leader election.
- Each worker node which is not master creates a znode under the parent node /***slave_ft.*** If a worker is elected as master then its znode under /***slave_ft*** is deleted.
- Each worker including master creates a znode under /***election.***
- When creating a znodes, its data is assigned with the pid of current container in which the worker resides.

Znode Name	**Flags**	**Data Section of Znode**
/election/	SEQUENTIAL | EPHEMERAL	PID of container
/election/master	EPHEMERAL	PID of the current master container
/slave_ft/	SEQUENTIAL | EPHEMERAL	PID of container

SCALING

We implemented a set_interval function which routinely spawns/shuts down workers according to the read API at the end of every two minutes since the first request and resets the counter.

When scaling out, the worker connects to the zookeeper server by starting a zookeeper client session and creates a znode under the parent node /***slave_ft.***

SLAVE FAULT TOLERANCE USING ZOOKEEPER

When crash API is called, the slave container with the highest PID is killed and the corresponding watch gets triggered which starts a new container which in turn creates a znode under /***slave_ft*.**

When a slave gets elected as leader, its node under /***slave_ft*** is deleted. This is done so that the master container doesn't get terminated when crash slave API is called.

LEADER ELECTION WITH ZOOKEEPER

Algorithm: Leader Election

Note: We maintain a boolean global variable called **switch2master** in each worker code to notify the container whether it has to execute the master or slave section of the code.

Step 1: Initially the first container to execute worker code, creates a znode under /**election.** It becomes the master and sets the variable **switch2master** to True so that it executes the master section of the code in the worker. It creates a /**election**/**master** node and stores the pid of the container in the data.

Step 2: The subsequent containers that are spawned create a znode with data as the pid of its container. It also sets a watch on the znode that has a pid that is just smaller than the current znode's pid.

Step 3: When the master is killed by master crash API, /**election**/**master** is also deleted since it's an ephemeral znode.

Step 4: When the watch is triggered it checks if the current pid matches with the lowest pid of all the containers. If it matches then it elects itself as the master by creating /**election**/**master** znode with data as its pid and sets **switch2master** to True.

The watches can be illustrated using a dependency graph which may be disconnected graph.

TESTING

Prob:We had missed a test case in create ride API.

Sol: We could infer that we were creating duplicate rides which was not supposed to happen.

Prob: In leader election, if the spawned container had a pid smaller than all the existing containers. Since there are no containers' pid smaller than this there is no watch set from this znode and by doing so it will never become the master.

Sol: To tackle this problem we had to routinely call a function which checks if the current znode pid matches with the lowest pid of all the containers. It is elected as the master only if the **/election/master** does not exist.

CHALLENGES

1. To sync a slave and to serve a read request by slave.we made use of threads to overcome this.
2. How to transfer the whole master db to a new slave?
we had to convert the whole db of master and send it over rabbitmq to the new slave
3. While scaling out there is significant delay after creating the znode and that reflecting in the zookeeper. Setting watch on the znode with highest pid didn't work because of the delay in the server. Introducing the sleep function every time after spawning a new container will impact the total time of the scaling function so it was necessary to set the watch in crash API.

4. What if the newly spawned container has pid smaller than all the existing containers? If the watch must be set on itself, then how does it become the master?

We routinely call a function to check if this znode pid matches with the lowest pid of all the containers. It is elected as the master only if the **/election/master** does not exist.

CONTRIBUTION

Nishikanth : Rabbitmq part.

Sharath : Zookeeper & Docker SDK.

Sricharan : AWS & Load Balancer.

CHECKLIST

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