17CS352:Cloud Computing

Class Project: Rideshare

DBaaS

Date of Evaluation: 16-05-2020
Evaluator: Prof. Srinivas K S
Submission ID: 351
Automated submission score: 10.0

SNo	Name	USN	Class/Section
1	Goutham R	PES1201701025	H
2	Monish Reddy BS	PES1201701666	H
3	Srujan V	PES1201700132	E
4	Khamaroddin Sheikh	PES1201701017	H

Introduction

- This project focuses on building a fault tolerant, highly available Database as a Service(DBaaS) for the RideShare application. The Rideshare application consists of two microservices - rides and users each hosting a flask application with API endpoints to create user, create ride, delete user, delete ride etc. These two microservices communicate with DBaaS to perform db operations.

Related work

> Rabbitmq: https://www.rabbitmq.com/getstarted.html

> Docker sdk docs: https://docker-py.readthedocs.io/en/stable/

> Zookeeper Kazoo docs: https://kazoo.readthedocs.io/en/latest/

ALGORITHM/DESIGN

Initially, DBaaS contains 7 containers - orchestrator, master, slave, master_db, slave_db, rabbitmq, zookeeper. All these containers lies in a docker network "ubuntu_backend".

Orchestrator:

- A flask application is hosted with the following API endpoints:
  - /api/v1/db/read - Updates the file "requests_count.txt" with no. Of requests received. Uses a global variable to determine the first request. If it's a first request, then it starts the scalability script asynchronously. Receives read requests from rides and users microservices and writes them to the readQ(using RPC - request-reply pattern). Then it reads the response from the corresponding reply queue.
  - /api/v1/db/write - receives write requests from rides and users microservices and writes them to the writeQ(using RPC - request-reply pattern). Then it reads the response (whether the db write operation was successful or not) from the corresponding reply queue.
  - /api/v1/db/clear - write the names of the mongodb collections to the writeQ. Then it reads the response from the corresponding reply queue whether, db clear operation was successful or not.
  - /api/v1/crash/master - kills the master container and its corresponding mongodb container using docker sdk and sends the pid of the master container as response.
  - /api/v1/crash/slave- finds the slave which has the lowest pid; kills that slave and its corresponding mongodb container using docker sdk and sends the pid of that slave container as response.

- /api/v1/workers/list - fetches the pids of all the worker containers using docker sdk and send the sorted list of all these pids as response.

- Docker sdk
  - The docker server is on the host. So the orchestrator container cannot access the /var/run/docker.run file of the host to initiate the DockerClient. So, we used a tool called socat, which would forward all network packets received at the specified port(in our case 4444) to the /var/run/docker.sock file. So, basically docker server has become a TCP service now. Now the orchestrator DockerClient can connect to the docker server via "<gateway_ip>:4444". In our case gateway ip is 172.17.0.1
- Fault tolerance - A zookeeper watch is initiated on the children of "/worker"
  - The watch is triggered whenever:
    - A node is created under the path "/worker"
    - A node is deleted under the path "/worker"
    - Data on the node under the path "/worker" is changed.
  - The watch function maintains memory of the list of children just before the current watch trigger event. The new list of children is compared with the old list of children to determine whether a node was deleted or a node was created or the data on a node was changed and which node was deleted or created. If it determines a node was deleted, it will read the name of that node to determine if it's a master node or slave node.
    - If it's a slave node, we check the exit code of that corresponding slave container; if it's 137, then we remove that container and it's corresponding mongodb container; and start a new slave container and it's corresponding mongodb container.
    - If it's a master node, we remove/delete the master container and it's corresponding mongodb container. Then we use docker sdk to get a list of all slave containers and it's corresponding pids. Then we send a message "You are now the master" at port 23456 to the slave with the lowest pid. Then the newly elected leader will behave as master. A new slave container along with it's corresponding mongodb container is started.
- Scalability
  - The script is scheduled every two minutes. A thread is created which reads the "requests_count.txt" file to get the no. Of requests received in the previous 2 minutes. Then reset the requests count to 0. Based on this number and the no. Of slaves currently present, we determine how many slaves should be stopped or how many new slave containers should be spawned. Action is taken accordingly.

Worker:

- The worker container maintains 3 environment variables:
  - WORKER_TYPE - master/slave
  - DB_NAME - the hostname of it's corresponding mongodb container
  - NODE_NAME - zookeeper node name
- Using the DB_NAME environment, the worker connects to it's corresponding mongodb container.
- An ephemeral znode "/worker/<NODE_NAME>" is created with data as the hostname of it's corresponding mongodb container.
- Based on the WORKER_TYPE environment variable, the worker behaves either as a master or a slave
  - The master uses RPC(request-reply pattern) to consume on writeQ. It reads queries in the writeQ and performs write operation in it's corresponding database. Then it writes the message whether the write operation was successful or not onto the reply queue which is then read by the orchestrator. Then, to achieve eventual consistency, it sends the same query that it read on writeQ to the syncQ which is a fanout exchange. Since it's a fanout exchange, all the queues that is connected to syncQ exchange, get the queries.
  - In order to sync the database of the newly spawned slave with that of the master, the slave reads the node "/worker/master" to get the hostname of the master database container. Then it uses mongodump tool to dump the database of the master ; and mongorestore tool to update the database of the new slave with the dumped database. Then the slave creates two new processes.
    - Process 1 - Creates two threads:
      - Thread 1 - It uses RPC(request-reply pattern) to consume on readQ. It reads the queries on readQ, and performs read operation on it's corresponding mongodb. The results returned by mongodb is written on the reply queue, which is read by the orchestrator. If there are multiple slaves listening on readQ, each slave reads from the readQ in a round robbin manner.
      - Thread 2 - It uses publish-subscribe pattern. A queue is declared and then bound to the syncQ fanout exchange. Whenever a message is sent to the fanout exchange, all the queues, that is bound with it receives the message. The slave reads these queries and performs write operations on its corresponding database to achieve eventual consistency.

- Process 2 - Opens port 23456 and listens for incoming messages on that port. When the orchestrator elects a new leader. It sends a message to the newly elected leader to port 23456. Upon the slave receiving this message, process 2 terminates process 1 and hence, it is no longer consuming on readQ and syncQ. It changes the environment variable WORKER_TYPE and NODE_NAME to master. It creates a new zookeeper node /worker/master with data as it's corresponding mongodb hostname. Then the node created earlier by this node is deleted. Then the new leader starts consuming on writeQ. To resolve conflicts between scalability and availability, the newly elected master container is renamed as master.

TESTING

- The /api/v1/db/clear, api/v1/crash/master and /api/v1/crash/slave were implemented as DELETE methods. During the automated evaluation, we realized they should be POST.

- While scaling up, the new containers were created with parameter remove=True which automatically removes the container once it is killed or stopped. The zookeeper watch function was using docker sdk to find the exit code of the stopped or killed container. Since the container is removed just after killing or stopping the container, we encountered an error "No such container". We were able to fix this by setting the parameter remove=False.

CHALLENGES

- The docker server is on the host. By default, docker sdk connects to it via the /var/run/docker.sock file which is on the host. We had to communicate with the docker server from within a container (ie; orchestrator). After a long time, we realized that there is a unix tool called socat which can forward network packets received at a specified port to a specified sock file. Now, within the container the DockerClient communicates with the docker server using "tcp://172.17.0.1:4444"

- Whenever a znode was deleted or added, the watch event was triggered twice. After a long time we realized, that the flask app reloads the main function twice. We were starting zoowatch asyncronously from the main function of the flask app, so watch was triggered twice for the same event. By setting parameter use_reloader=False in app.run of the flask app, we were able to fix this issue.

- Making the newly elected leader to behave as a master was a challenge. Multiprocessing came to the rescue. Kill the process which is consuming on readQ and syncQ; and start a new process which consumes on writeQ.

Contributions

- Goutham R - master/slave worker, Eventual consistency, Zookeeper watch, Zookeeper leader election
- Monish Reddy BS - RPC client, RPC server
- Srujan V - Scalability
- Khamaroddin Sheikh - orchestrator APIs

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