Introduction

Rideshare Application where We can add a user, delete the user, create a ride for the user, the user can also wish to join an existing ride, can get details about the user existing, rides created from a particular source to destination.

A custom database orchestrator engine (Dbaas aka database as a service) is implemented that listens to incoming HTTP requests from the users and rides microservices and performs read/write according to given specifications.

Advanced Message Queue Protocol using RabbitMQ is used as a message broker.

The orchestrator is responsible for publishing the incoming message into the relevant queue (read, write, sync, response) and bringing up new worker containers as desired.

2 types of workers are implemented – slave and master where slave worker is responsible for read requests and master worker for write requests.

Related work

- https://www.rabbitmq.com/tutorials/tutorial-six-python.html
- https://www.rabbitmq.com/tutorials/amqp-concepts.html
- https://docker-py.readthedocs.io/en/stable/client.html
- https://readthedocs.org/projects/kazoo/downloads/pdf/2.5.0/

ALGORITHM/DESIGN

RabbitMQ Implementation

Json data containing the table name, column, values and condition clause are sent to the read dB function in orchestrator which extracts

the details and places them in the read queue to be passed to the slave for it to execute. The slave executes the query and passes the relevant response to the orchestrator using response queue.

Here RabbitMQ is used to build RPC system: a client and scalable server.

When the client starts up it creates an anonymous callback queue. The client sends an RPC request message to RPC queue. The RPC worker on receiving the requests executes the query and send the response message back to the client which in turn returns the response to the application. Prefetch count value is set to run more than one server process to spread the load equally over multiple servers.

Similarly, the request made to insert or delete the data is sent to the write dB function in orchestrator which places it in write queue and sends to the master for it to execute the query. The master sends all the write requests to another queue called syncQ. SyncQ is responsible for synchronizing the database of master with all other slave workers, so that all the databases are up to date.

Replication

For replication of data, all the changes made to the database through master is placed as a query data to the sync queue which is picked up by the slave worker and slave copies all the changes to the replicated database. Shared dB is created to maintain consistency of data when spawning of new workers takes place.

Auto Scaling

Timer starts the auto scaler with the first request. Initially when request count is 0, no of slaves is 1. If the request count reaches 20, 40, so on, no of slaves increases by 1.

For other cases, no of slaves = int(request_count/20)+1

Every read request starts with calling the trigger timer function. After every 2 minutes, scale-timer function is called which increases/decreases the no of slave workers based on read request count. Docker SDK is used to increase/decrease the containers dynamically. If the worker containers running<=no of slaves, increase the container count by 1 by dynamically creating a new slave worker. If container running > no of slaves, decrease the container count by 1 by stopping the old slave worker.

Fault Tolerance

If a slave worker fails: A new slave container is started in its place via docker SDK. A shared_db container is maintained to provide database consistency to the newly spawned slave worker.

New API's Implemented

- Route: api/v1/crash/slave
  Functionality: Get the list of containers running and their PID. Find the slave worker with the highest PID and stop it. Implemented in orchestrator.

- Route: api/v1/worker/list
  Functionality: Display the PID's of the containers currently running. Empty list returned in case of absence of workers. Implemented in orchestrator.

TESTING CHALLENGES

- DB clear database wasn't able to communicate with the orchestrator properly. My condition data was empty so the query wasn't properly getting executed, so adding another suitable condition in write DB fixed the issue.
- Didn't enable TCP connection in the security group in the beginning so communication couldn't happen.
- Load balancer showed 503 during submission due to wrong configuration and wrong target groups, so had to make a new one again.
- We had initially used private IP of orchestrator, changing it to elastic IP in the code fixed the issue.
- Result showed inconsistent ride information, as join ride showed 405 even for correct method. Issue was fixed with some changes to the code.
- We got the wrong container count during initial testing. Issue was fixed by removing master and slave containers from docker-compose.yml in orchestrator since we were already starting them manually inside the orchestrator using docker SDK.