LOG8415E : Advanced Concepts of Cloud Computing

Assignment Final Project: Scaling Databases and Implementing Cloud Design Patterns

Handover documentation

Brandstaedt Jules, 2314495

December 27th, 2023

I) Introduction

As cloud computing has become more and more important, it is now very valuable to know how to use a cloud provider computing service. This assignment aims to get familiar with Amazon Web Services, a leading infrastructure as a service Cloud provider, and more specifically with Amazon Elastic Compute Cloud (Amazon EC2). The first goal here is to use Boto3 and MySQL to deploy multiple instances:

- three are running as workers;
- one is running as a manager;
- one is running as a stand-alone server.

This configuration will be tested on Sakila by using Sysbench.

The second goal is to implement two common cloud patterns known as **Proxy** and Gatekeeper. Each pattern should be implemented using its specificity: high level of security for the Gatekeeper, and three different implementations for the **Proxy**.

The public repository for this final assignment can be found at: https://github.com/JBrandstaedt/LOG4715E_final_project.
As you will see there are two contributors; however both are my accounts. The one the 'Brawlekk' as a pseudo is my personal one, and I forgot to change it back to 'JBrandstaedt' when I wanted to push the assignment from my second computer.

II) Benchmarking MySQL stand-alone vs. MySQL Cluster

a) Introduction

First we deploy all five instances by using functions our team created for the previous assignment. You can find them in the file `utils_create_instances.py`. Each instance is created as a `t2.micro`. In order to interact with these I created a `ServerClient` class in `server_client.py`. This class implements a few methods that are needed. First, the initialization instantiates a SSH client to be able to log on the instance using `paramiko`, 4 methods are added to the class:

- `exec_command`: uses SSH to execute a given command;
- `close_connection`: closes SSH connection;
- `download_file`: download a remote file on a local machine using SSH;
- `upload_file`: upload a local file on a remote machine.

This class will also be used in the following pattern creation part.

By using this class, we can upload on every instance a few files that will indicate the configuration used and the job of the instance. Files I upload this way are bash files that can

be found in the `config` and in the `setup/bash` directories, for the SSH connection, port 3306 will be used.

Once every instance is created and configured correctly, we can run the benchmark and retrieve the file. These files can be found in the `results` directory. I also created a directory to save results in images.

b) Benchmark

The benchmark I created is mainly focused on two attributes: latency and number of transactions. Both stand-alone and cluster are using a table_size of 100000 of the Sakila database, during 30 seconds, with 6 threads available.
First, we see that given these parameters the cluster latency is faster (14.2 ms) than the stand-alone (19.85 ms) by a 40% difference.

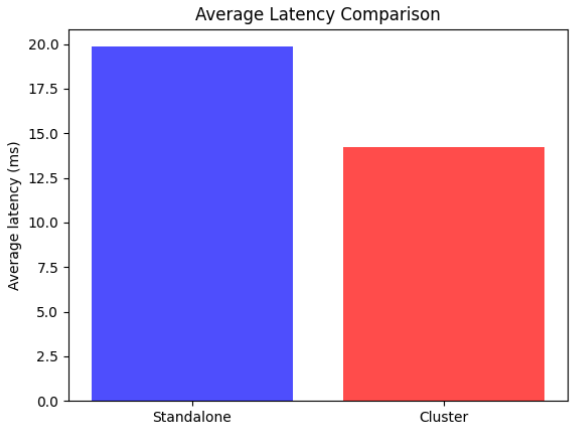

Figure 1: Latency comparison between stand-alone and MySQL cluster

We can see in the second figure that, once again, the cluster has answered more (12040 queries) than the stand-alone server (8611 queries). It represents an increase of 58.3% of the cluster score.

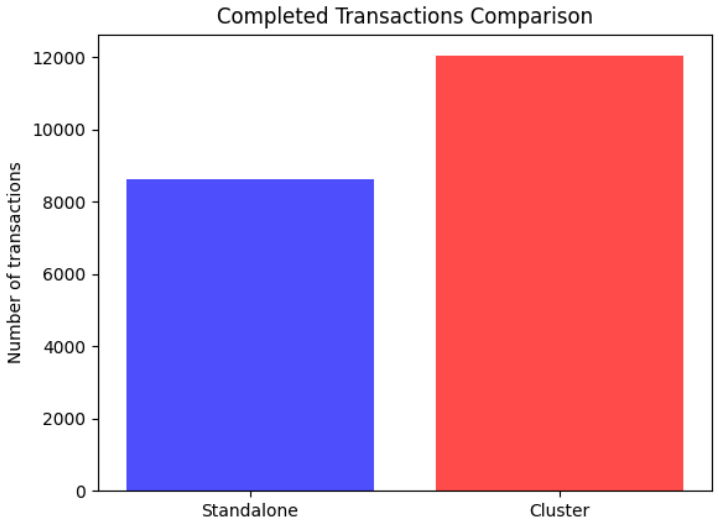

Figure 2: Completed transaction comparison between stand-alone and MySQL cluster

III) Implementation of the Proxy pattern

Our aim is to implement a **Proxy** pattern for our architecture. The **Proxy** should route the requests and provide read scalability on a relational database. Reads to the data should be handled by the workers and write requests should be handed to the manager that replicates it on workers.
For this pattern we had to create three different implementations:

- Direct hit, the requests will be directly forwarded to the master node without any logic given to distribute data;
- Random, the requests will be handled to a worker chosen at random;
- Customized, the requests will be given to the worker with the lowest average ping among the three.

We will implement this pattern by using a `t2.large` instance. As stated in the previous chapter, we will use the `ServerClient` class to set up this pattern. We can upload all files that are needed. In particular, we upload a copy of our key, two Python files and a bash script.
I created two Python classes, the first one `SQLConnect` is used to connect the **Proxy** to MySQL and execute queries. The second one is `ProxyServer`, it is the class that implements all three **Proxy** behaviors and the methods they need, such as ping to the server and a **SSH Tunnel forwarder**. The bash script `setup_proxy.sh` is used to install all needed libraries on the instance and set up some environment variables for the main in `proxy.py`.

The setup requires the user to create a terminal and run two commands to finish it. Once they are done, you can run commands like the following one to send a query to the SQL cluster:
sudo python3 proxy.py \"SELECT COUNT(*) FROM actor;\"

It is also possible to run `proxy.py` so the code waits for any request sent to it from a given dns, sends it to the SQL server, waits for the response and sends it back.
sudo python3 proxy.py \"\" private_dns

This feature was made to be able to test the **Proxy** alone and to use it along with the Gatekeeper by giving the private dns of the **Trusted Host**.

IV) Implementation of the Gatekeeper

Our aim is to add a **Gatekeeper** pattern to our architecture. This pattern adds a level of security by reducing the exposed surface to attack. The **Proxy** should only get requests from the Gatekeeper and send answers to it as well. This pattern consists of two parts, a Gatekeeper and a **Trusted Host**, we will run them both on `t2.large` instances. The setup is similar to the **Proxy**, albeit it will forward requests to the **Proxy** rather than the SQL server. Also, the security must be stricter to keep it as high as possible. That is why the security group for the **Gatekeeper** is different from other instances.

The both Gatekeepers instances set up the environment by running a bash file. Then both use a dedicated Python code.

The **Trusted Host** waits for queries from the Gatekeeper. It forwards any query to a target that is linked by a SSH Tunnel forwarder at the initialization of the class `TrustedHost`.
The **Gatekeeper** uses a similar `Gatekeeper class.` Its link to the **Trusted Host** is always done during the setup. You can run the following command to forward any query to the Gatekeeper and the Trusted Host will forward it and send back the response:
sudo python3 gatekeeper.py \"SELECT COUNT(*) FROM actor;\"

V) Summary

a) Incomplete implementation

One part of the project is missing, the SSH connection between the **Gatekeeper** pattern and the **Proxy** pattern, as well as the connection between **Gatekeeper** and **Trusted Host**. The incomplete part is in the `SQLConnect.py` file. It is possible to connect with pymysql but not with paramiko.

In order to complete the missing part, I could repeat the previous code I used for the SSH connection in the main. Once the `SQLConnect` is completed I can modify the code in all three instances (`Gatekeeper`, `TrustedHost` and `Proxy`). I would have made it so that the class connects in SSH to the instance it needs to communicate with and run the next

python class in the chain. And additional information would have been needed: whether the query has to be forward or the answer has to be sent back. The whole structure is present but I could not make it work in time.

b) Results

Results from the stand-alone vs. cluster are explained in the previous chapter.
They can be found in the `Final_project_LOG4715E/results` directory and figures can be seen in the `Final_project_LOG4715E/graphs` directory.

c) How to run the code

Clone the given Github repository. Be sure to have an updated version of Python (3.10.13 is the one I can guarantee the usage). Please install all the libraries written in the `requirements.txt` file by running: ***pip install -r requirements.txt***.

Create a new private key named `my_key_pair.pem` using AWS and put it in a `/key/` directory in the same folder as the project.

Once the previous steps are completed, you can execute the `main.py` Python file from the `Final_project_LOG4715E/setup` directory. The code will run in a Python terminal. The benchmark is done once the SQL server and user are set up.

After it completes the setup for the stand-alone and cluster MySQL, you will need to create ***three*** other terminals and run a few commands to configure the SQL server/user, the **Proxy** and the **Gatekeeper/Trusted Host**.