LOG8415E: Advanced cloud computing

Autumn 2024

Cloud Design Patterns: Implementing a DB Cluster

Stéphane Michaud 1904016

Abstract

In this individual final project, I implemented a database cluster using both the proxy and gatekeeper design pattern. The full implementation run on aws services using script from start, setting up the resources and machines, to finish cleaning up the resources. All work, except from function taken from the previous assignment, was done by myself. I was able to implement all of the asked features. The code is done in python and can be found here: https://github.com/StephaneMichaud/LOG8451E-FinalProject. Sufficient instruction has been puts into the README.md file to replicate my results. This report is also paired with a video going through the code and the different execution steps which can be found in the repository under **docs/Report** under the name **Presentation.webm**.

Contents

1. General instances implementation

Similar to the work my team and I did on TP1 and TP2, the user data given at the ec2 instance creation will contain most of the instructions to setup correctly the machine. However they are a few modifications. To start, the pythons librairies as well as the flask script to run were not imbedded in the user data directly to have an easier time to modify them. Instead, they are kept in a separate folder and are upload to an S22 bucket. The user data contains instruction to download its specific assets from the the bucket.

```
# Install Python libraries
...
aws s3 cp s3://{s3_bucket_name}/instances_assets/db_instance/requirements.txt ./
requirements.txt
sudo pip3 install -r requirements.txt


#TODO RUN FLASK
...
aws s3 cp s3://{s3_bucket_name}/instances_assets/db_instance/main.py ./main.py
sudo python3 main.py
```

Listing 1: User data instruction to download python assets from an S3 bucket

Since this is an aws service and the machine require permission to access this service, in each user data I injected the session credentials. Usually this would instead be done through the use of Roles that we could assign the machine but the AWS academy setup does not allow us to create roles or use the IAM service, this compromise was found. I would however caution of using this technique in a real world scenario and this alternative only serve to mimic the usage of and IAM role.

```
sudo snap install aws-cli --classic
sudo apt install amazon-ec2-utils -y

...
# Mimic IAM role
aws configure set aws_access_key_id {aws_access_key_id}
aws configure set aws_secret_access_key {aws_secret_access_key}
aws configure set aws_session_token {aws_session_token}
aws configure set region {region}

# Save AWS credentials in .env format
cat << EOF > .env
AWS_ACCESS_KEY_ID={aws_access_key_id}
AWS_SECRET_ACCESS_KEY={aws_secret_access_key}
AWS_SESSION_TOKEN={aws_session_token}
AWS_DEFAULT_REGION={region}
EOF
```

Listing 2: User data instruction to download python assets from an S3 bucket

For both code blocks above, all variables are formatted at runtime before being given to the different ec2 instances.

Finally, one problem we ran into in both previous TPs was that it was difficult to estimate when the user data instructions had finished running. In the first TP, we waited a fixed amount of time. In the second TP, we used the generated keypair to see if the files created by the user data were correctly created, Since the first method is unreliable, as the installation time varies from runs to runs, and the second one would be not really possible since most of our machines are private, I instead decided to user Tags to indicate the installation status. For each machine, we crate them with two tags. One is role, who indicates what purpose the machine serves (DB_MANAGER, DB_WORKER, PROXY, ...) and the other is STATUS who indicates the progress of the different commands in the user-data bash script.

```
...

aws ec2 create-tags --region {region} --resources $instance_id --tags
Key=STATUS,Value=INSTALL:MY-SQL

...

aws ec2 create-tags --region {region} --resources $instance_id --tags
Key=STATUS,Value=INSTALL:Sakila

...

aws ec2 create-tags --region {region} --resources $instance_id --tags
Key=STATUS,Value=INSTALL:PYTHON

...

aws ec2 create-tags --region {region} --resources $instance_id --tags
Key=STATUS,Value=INSTALL:PYTHON-LIBS

...

aws ec2 create-tags --region {region} --resources $instance_id --tags
Key=STATUS,Value=READY
```

Listing 3: Example of the usage of the STATUS tag in the database manager user script to indicate the progress of installation and setting up the machine.

It is then easy on the local machine to know when the different instances have finished their user-data script and when we can start the benchmarking.

2. Databases cluster implementation

For these machine, we first install MySql and Sakilla with the commands given in the project statement. We then run the benchmarks using the following commands:

```
#run mysql benchmark
aws    ec2    create-tags    --region    {region}    --resources    $instance_id    --tags
Key=STATUS,Value=INSTALL:SQL-BENCHMARK
sudo apt-get install sysbench -y
sudo sysbench --test=oltp_read_write --table-size=10000 --mysql-db=sakila --mysql-
user=root --mysql-password="" prepare
sudo    sysbench    oltp_read_write    --table-size=10000    --mysql-db=sakila    --db-
driver=mysql --mysql-user=root --num-threads=6 --max-time=60 --max-requests=0 run >
standaloneBenchmark-_$instance_id.txt
sudo sysbench oltp_read_write --table-size=10000 --mysql-db=sakila --db-driver=mysql
--mysql-user=root cleanup
aws     s3     cp     standaloneBenchmark-_$instance_id.txt     s3://{s3_bucket_name}/
{benchmark_upload_path}/standaloneBenchmark_$instance_id.txt
```

Listing 4: Commands to run the MySql benchmarks on the instances of the database cluster.

We can note that the benchmarks are save in a txt format and then uploaded to the s3 bucket so that they can be downloaded by the local machine later.

As for the code, I run, like all other instances, a flask application so that the proxy can communicate to the database workers. I implemented two functions to mimic read and write operation to our database. The first one read the top 5 actors that were the most recently updated. The write function add a new actor and expect a first name and last name as inputs.

Finally, the manager is also responsible of assuring that the data is replicated to the workers. This is why in each write request, assuming that it is successful, the manager will communicate to the workers the same operation so that they are up to date. This is done again using http query.

3. Proxy pattern implementation

The proxy take the form of a flask application. To communicate with the database cluster, it uses the ROLE tag to query, using boto3, the machines private ip.

```python
# Get private IP of first instance with the tag ROLE = DB_MANAGER
response = ec2.describe_instances(Filters=[
    {
        'Name': 'tag:ROLE',
        'Values': ['DB_MANAGER']
    },
    {
        'Name': 'instance-state-name',
        'Values': ['running']
    }
])

if response['Reservations']:
    db_manager_private_ip = response['Reservations'][0]['Instances'][0]['PrivateIpAddress']

# Get private IPs of all instances with the tag ROLE = DB_WORKER
response = ec2.describe_instances(Filters=[
    {
        'Name': 'tag:ROLE',
        'Values': ['DB_WORKER']
    },
    {
        'Name': 'instance-state-name',
        'Values': ['running']
    }
])
if response['Reservations']:
    db_workers_private_ips = [res["PrivateIpAddress"] for res in response["Reservations"][0]["Instances"]]
```

Listing 5: Instructions in the proxy application to query the database clusters private ip.

Theses addresses are then use to communicate with the Flask application running on the database workers and manager.

Finally, to switch between the various load balancing mode, I added an additional post path.

```python
async def switch_lb_mode(mode: int):
    """
    mode:
    0 = manager only
    1 = random
    2 = least response time
    """
    global lbmode
    if mode not in VALID_MODE:
        return {"error": "Invalid mode"}
    else:
        lbmode = mode
        return {"message": f"LB mode set to {VALID_MODE_NAMES[mode]}"}
```

Listing 6: Function to switch load balancing mode in the proxy application.

3.1. Write operation

All write operation are directed to the manager who will also take care of updating the workers.

3.2. Read operation

3.2.1. Manager only load balancing

For this load balancing mode, we simply direct all read request to the manager ip address.

```python
...

 if lbmode == 0: # manager only
      response = requests.get(f"http://{db_manager_private_ip}/read").json()
...
```

Listing 7: Manager only load balancing code.

3.2.2. Random load balancing

We choose a random instance between the manager and the workers.

```
...

 elif lbmode == 1: # random choice
         choice = random.randint(0, len(db_workers_private_ips)) #if we select 0, it the
manager, else some worker
        if choice > 0:
            choice = choice -1
            response = requests.get(f"http://{db_workers_private_ips[choice]}/read").json()
        else:
            response = requests.get(f"http://{db_manager_private_ip}/read").json()
...
```

Listing 8: Random load balancing code.

3.2.3. Least busy load balancing

For this load balancing mode, we ping each instances and use the one who responded the fastest.

```
async def read_db():
 ...
 elif lbmode == 2: # least busy
        least_busy_ip = get_least_busy_ip()
        if least_busy_ip:
            response = requests.get(f"http://{least_busy_ip}/read").json()
        else:
            return {"error": "Cannot ping any db"}
def get_least_busy_ip():
    global db_manager_private_ip
    global db_workers_private_ips

    all_ips = [db_manager_private_ip] + db_workers_private_ips
    response_times = []
    valid_ip = []

    for ip in all_ips:
        try:
            response = requests.get(f"http://{ip}/ping")

            if response.ok:
                response_times.append(response.elapsed.total_seconds())
                valid_ip.append(ip)
            else:
                logger.warning(f"Failed to ping {ip}: {response.status_code}")
        except requests.RequestException as e:
            logger.error(f"Error pinging {ip}: {str(e)}")

    if response_times:
        least_busy_ip_index = response_times.index(min(response_times))
        return valid_ip[least_busy_ip_index]
    else:
        logger.error("No responsive IPs found")
        return None
```

Listing 9: Least busy load balancing code.

4. Gatekeeper pattern implementation

The most important component of this design pattern in the isolation and security of the different instances. Only the gatekeeper can be exposed to the internet. The trusted host alongside the prox and the database cluster must be kept private. To accomplish this, we first create a VPC. To this VPC, we attach an internet gateway. After that, we create a public and a private subnet. Since the instances in the private subnet will need resources like python libraires or the Sakilla database we also create a NAT gateway. This will allow them to download theses resources from the internet in a safe manner. Once this is setup, we can assign a subnet for each instances. As mentioned, only the gatekeeper instance will be assigned the public subnet while all of the other instances will use the private subnet.

```
...
print("Creating PROXY...")
private_instance_proxy= launch_ec2_instance(
    ec2,
    config["key_pair_name"],
    group_id,
    private_subnet_id, # <-- GIVE PRIVATE SUBNET TO PROXY
    instance_type=config["instances"]["proxy_instance"]["proxy_instance_type"],
    image_id=config["instances"]["proxy_instance"]["proxy_instance_ami"],
    public_ip=False,
    user_data = get_proxy_data(s3_bucket_name=config["s3_bucket_name"]),
    tags=[("STATUS", "BOOTING-UP"), ("ROLE", "PROXY")],
    num_instances=1)[0]
```

Listing 10: Proxy instance creation in main.py. We can see that the private_subnet_id is given alongside the other parameters.

For the actual application running on both the gatekeeper and trusted-host instance, it is rather simple. The trusted-host simply redirect all communication to the proxy. The gatekeeper does mainly the same thing but has some additional sanity check when writing new data. Usually the gatekeeper would also take care of authentication but for the purpose of this project, I omitted this part.

5. Results

5.1. MySql benchmarks

The benchmarks was computed using this command:

```
sudo sysbench --test=oltp_read_write --table-size=10000 --mysql-db=sakila --mysql-user=root
--mysql-password="" prepare
sudo  sysbench  oltp_read_write  --table-size=10000  --mysql-db=sakila  --db-driver=mysql  --
mysql-user=root  --num-threads=6  --max-time=60  --max-requests=0  run  >  standaloneBenchmark-
_$instance_id.txt
sudo  sysbench  oltp_read_write  --table-size=10000  --mysql-db=sakila  --db-driver=mysql  --
mysql-user=root cleanup
```

Listing 11: Commands used to compute the SQL benchmarks.

```
Running the test with following options:
Number of threads: 6
Initializing random number generator from current time


Initializing worker threads...

Threads started!

SQL statistics:
    queries performed:
        read:                            255416
        write:                           72968
        other:                           36485
        total:                           364869
    transactions:                        18241  (303.88 per sec.)
    queries:                             364869 (6078.34 per sec.)
    ignored errors:                      3      (0.05 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          60.0262s
    total number of events:              18241

Latency (ms):
         min:                                    3.91
         avg:                                   19.74
         max:                                  146.47
         95th percentile:                       33.12
         sum:                               360041.83

Threads fairness:
    events (avg/stddev):           3040.1667/17.70
    execution time (avg/stddev):   60.0070/0.01
```

Listing 12: SQL benchmarks for one of the three instances in the database cluster.

5.2. Read write benchmarks

I tested the load balancing mode in this order:

1. Manager only
2. Random choice
3. Least busy

I also added a wait time of 2 min between each test so that the statistics collected by cloudwatch would be clearer. As asked in the tp description, a 1000 simultaneous read and write request were launched for each proxy mode.

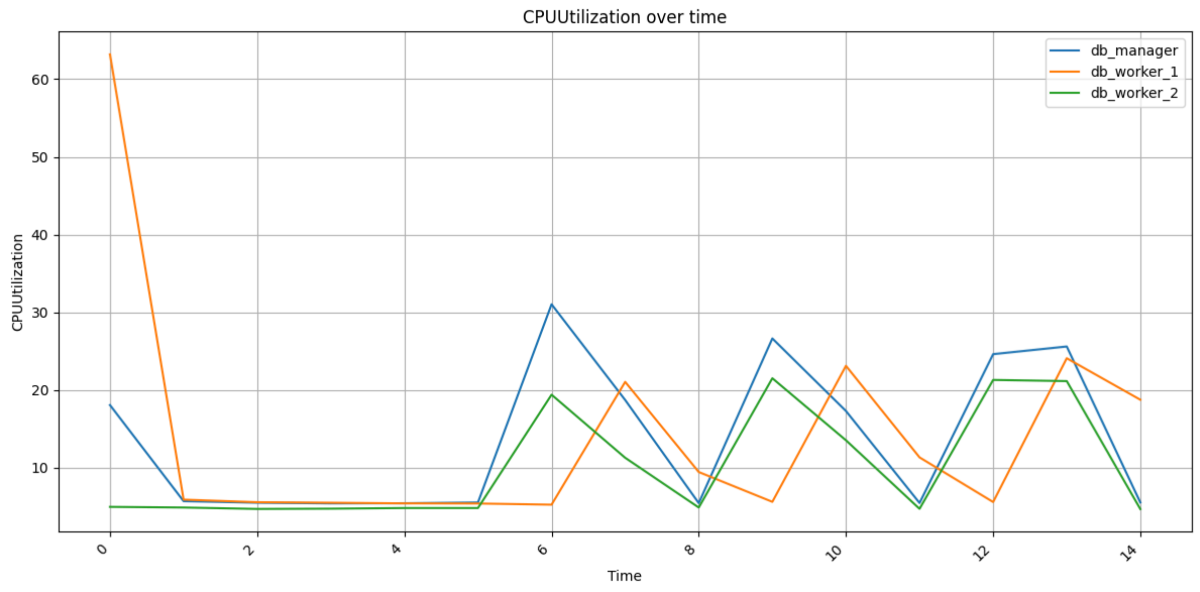

Figure 1: Cpu utilization over time for the 3 machines in the database cluster. Each spike correspond to the benchmark performed on a certain proxy load balancing mode. Metrics for the first worker are slightly off because of cloudwatch. We can see how a rerun of the benchmarks yield slightly different metric in the annex.

In the figure above, we can see that the cpu utilization for the manager stay at all time above the other two. This is due to that fact that the manager has to send a write request to the workers for each of the write operation. We can also see that the cpu utilization never goes down during the benchmark for every machine, even in the manager only proxy mode first spike). This is due to the write operation that are replicated on the two workers. We can however see in the first spike that the discrepancy between the manager and the worker is higher that in the two other spike, confirming that the manager is receiving all of the read requests. We can see also that the write operation are the most consuming in term of resources since the workers barely spike in utilization for the two to other mode. While one could think this is an issue with the read operation not properly going through, running the benchmarks with only the read request show that they correctly go through the workers instances:

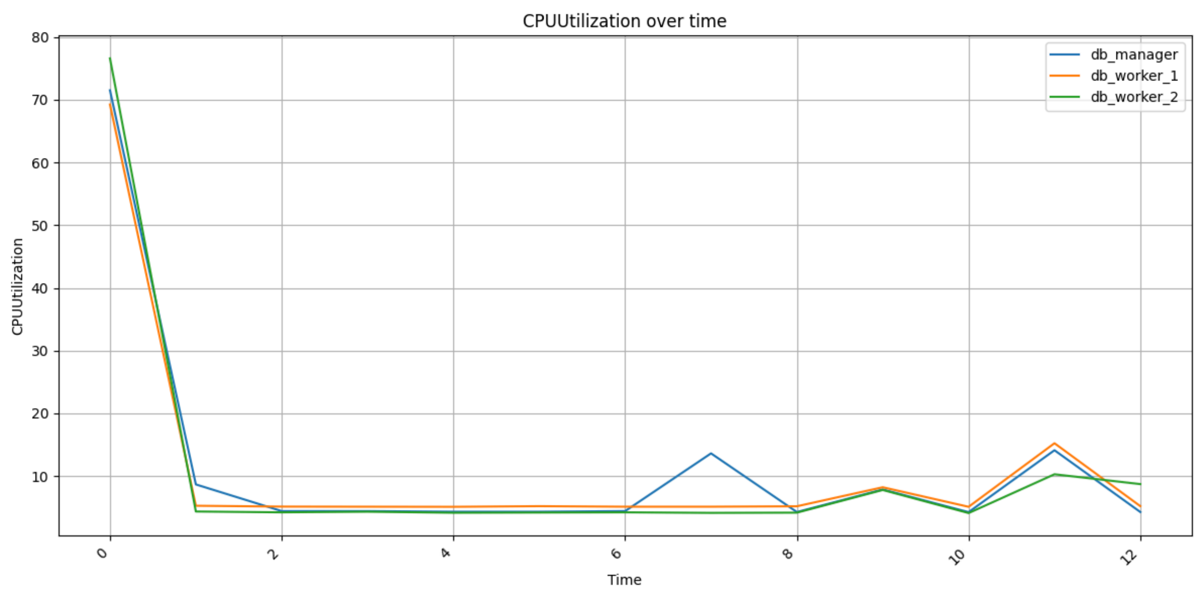

Figure 2: Cpu utilization over time for the 3 machines in the database cluster for read only benchmark. Each spike correspond to the benchmark performed on a certain proxy load balancing mode. Metrics for the manager are slightly off because of cloudwatch.

We also note that in the customized load balancing, the manager still seems to peak higher than the other machines when we include the write operation. While this mode is intended to get the least busy machine, I think that since our CPU utilization is rather low, the ping response is quite similar across all machine even accounting the additional responsibility of the manager. This might be an issue with the simplicity of the request and more heavier SQL query could be preferred.

6. Annex

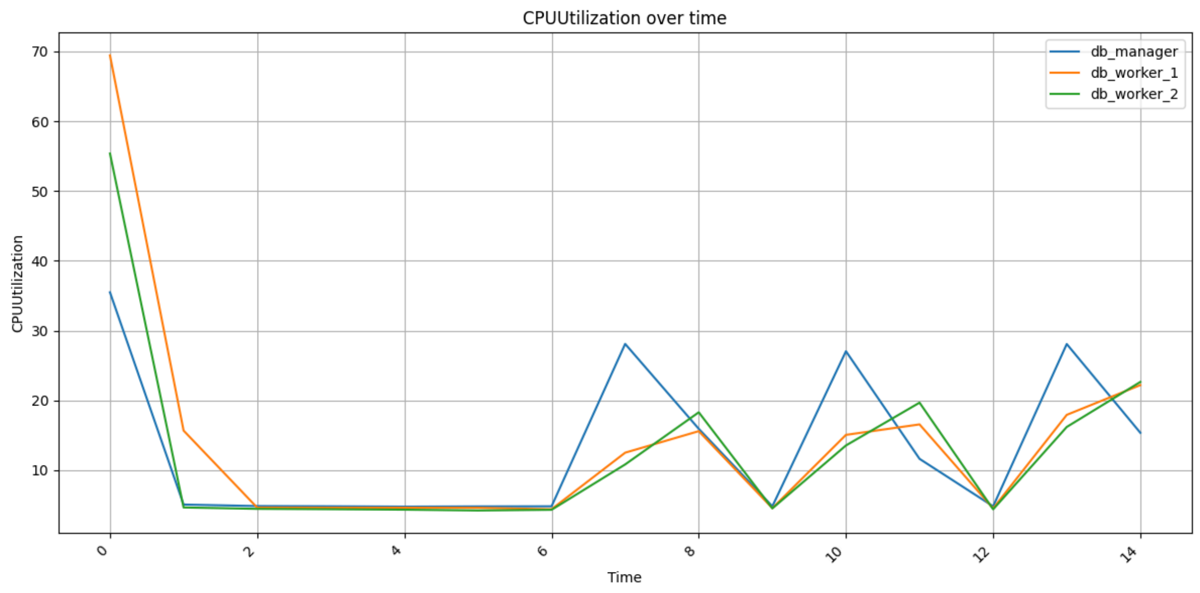

Figure 3: Other example of Cpu utilization over time for the 3 machines in the database cluster. Each spike correspond to the benchmark performed on a certain proxy load balancing mode. Metrics for the manager are slightly off because of cloudwatch.