Advanced Concepts of Cloud Computing

October 10, 2021

Software Engineering Department

LOG8415E

1st laboratory

Cluster Benchmarking using EC2 Virtual Machines and Elastic Load Balancer (ELB)

Presented by:

Yanis Toubal (1960266)

Estefan Vega-Calcada (1934346)

Guy-Hermann Adiko (2005119)

October 10th 2021

Abstract

We were asked to launch virtual machines on AWS, connect them as a cluster, use a load balancer to distribute the workloads, and then perform an analysis over their performance as a whole. The workloads will be sent by a docker container, which hosts an application, to the Application Load Balancers and then a benchmark will be performed from the data collected.

1 Introduction

AmazonWeb Services (AWS) is a leading infrastructure as a service Cloud provider. One of their products, the Amazon Elastic Compute Cloud (Amazon EC2) is a web-based service that allows customers to run application programs in the Amazon Web Services (AWS) public cloud. Amazon EC2 allows a developer to spin up virtual machines (VMs), which provide compute capacity for IT projects and cloud workloads that run with global AWS data centers. The overall goals of this lab assignment are:

- Creating 2 clusters of 5 EC2 instances each and an Application Load Balancer (ALB) for each cluster
- In each single instance, we have to deploy a Flask application
- Use Docker to perform our test scenarios locally
- Report the results

In the following, we will conduct this experimentation by firstly describing our procedure to deploy Flask Application on each and every EC2 instance, secondly we will define our cluster setup ALB, thirdly we will report our benchmark result and we will conclude with the description to run our script.

2 Flask Application Deployment Procedure

Flask is a small and lightweight Python web framework that provides useful tools and features that make creating web applications in Python easier. It gives developers flexibility and is a more accessible framework for new developers since you can build a web application quickly using only a single Python file.
After successfully creating our EC2 virtual machines, where we allowed, in the security group, the traffic on port 80, we connected to the instance and did the following procedure for the Flask deployment:

- Updated the packages
- Installed the package management system pip
- Installed Flask package
- We then created a directory where we edited a file called ***app.py*** to write our application function.

- Inside that file, we imported the Flask class,
- Kept in a ***variable*** the *ec2metadata –instance-id* which returns the instance ID of the VM
- We create an instance of the Flask class
- We then used the ***route()*** decorator to tell Flask what URL should trigger our function
- We created a function that returns the message we want to display in the user's browser. In our case the message is "***variable*** *is responding now*", where ***variable*** holds the instance ID of the EC2.
- Since Flask runs by default on port 5000, we therefore change it to port 80 in the ***app.run()***
- Finally, we ran the Flask application on the instance

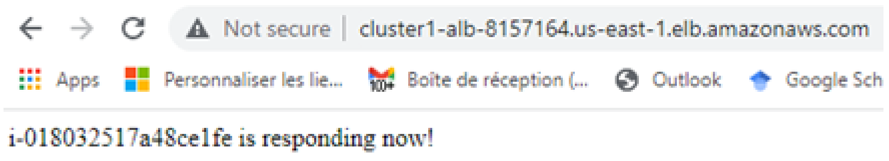
3 Cluster Setup Application Load Balancer

A load balancer serves as the single point of contact for clients. The load balancer distributes incoming application traffic across multiple targets, such as EC2 instances, in multiple Availability Zones. This increases the availability of your application.

An Application Load Balancer (ALB) makes routing decisions at the application layer
(HTTP/HTTPS) which is layer 7, supports path-based routing, and can route requests to one or more ports on each container instance in your cluster. Application Load Balancers support dynamic host port mapping. The ALB allows for the redirection of traffic based on the content of the request. Creating an ALB consists of 3 majors parts: The **Listener**, The **Target group** and the **Rules**.
A listener checks for connection requests from clients, using the protocol and port that you configure. A target group is a set of instances that can handle the requests. The rules that you define for a listener determine how the load balancer routes requests to its registered targets. Each rule consists of a priority, one or more actions, and one or more conditions. When the conditions for a rule are met, then its actions are performed. You must define a default rule for each listener, and you can optionally define additional rules.
Each target group routes requests to one or more registered targets, such as
EC2 instances, using the protocol and port number that you specify. It's possible to register a target with multiple target groups. You can configure health checks on a per target group basis. Health checks are performed on all targets registered to a target group that is specified in a listener rule for your load balancer.
When creating the ALB for one specific cluster, we select:

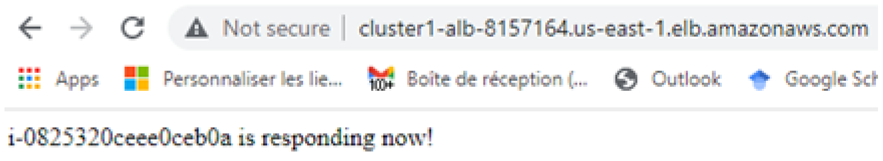
- Internet-facing and IPv4, this will be the gateway so the client can send the request to the ALB. An internet-facing load balancer routes requests from clients over the internet to targets
- Then we select the 3 availability zones of the concerned cluster, and we set the listener to port 80
- We add a target group to the ALB containing the 5 EC2 instances of the cluster
- We repeat all those steps for the second cluster

Below are some screenshots we got when using the ALB DNS name. We can see that a different EC2 instance (of the cluster) is responding everytime we refresh the browser.



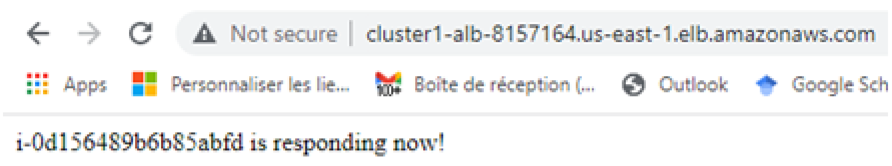
i-018032517a48ce1fe is responding now!

Figure 1: Cluster 1, Instance 1 responding



i-0825320ceee0ceb0a is responding now!

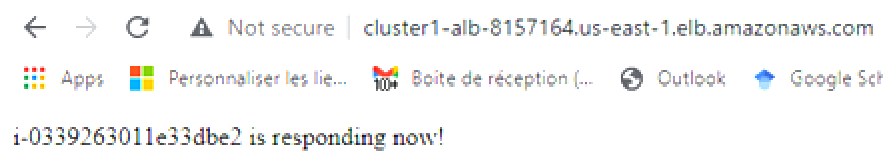
Figure 2: Cluster 1, Instance 2 responding



cluster1-alb-8157164.us-east-1.elb.amazonaws.com

i-0d156489b6b85abfd is responding now!

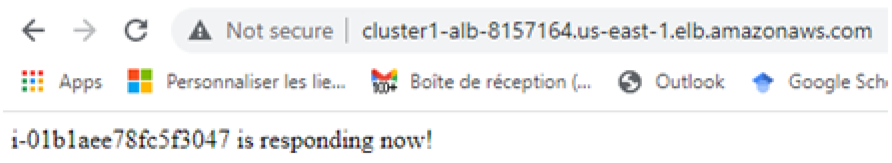
Figure 3. *Cluster 1, Instance 3 responding*



cluster1-alb-8157164.us-east-1.elb.amazonaws.com

i-0339263011e33dbe2 is responding now!

Figure 3: Cluster 1, Instance 4 responding



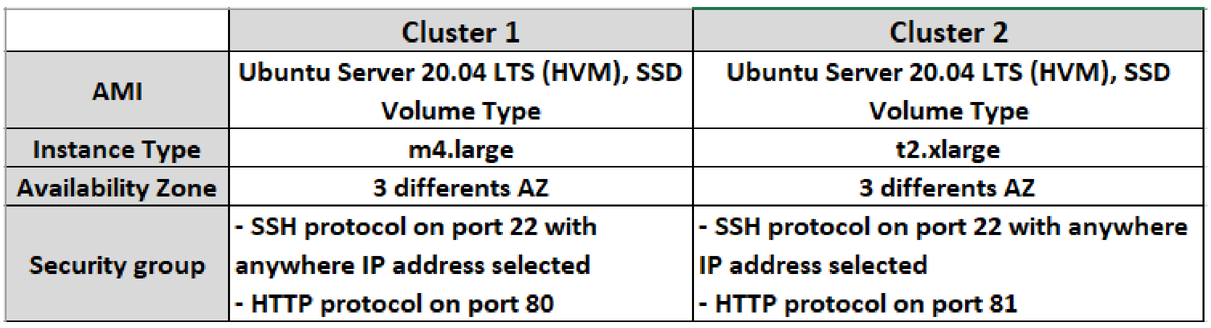
cluster1-alb-8157164.us-east-1.elb.amazonaws.com

i-01b1aee78fc5f3047 is responding now!

Figure 5. *Cluster 1, Instance 5 responding*

4 Results of your benchmark

__In order to perform the benchmark, we finally deployed 4 EC2 instances in Cluster1 and 4 in Cluster 2. Actually we did create 5 in every cluster, but, for some unknown reasons, AWS Educate often terminates 1 or 2 instances unexpectedly. Here is the recap for the configuration of the clusters composed of EC2 instances:



	Cluster 1	Cluster 2
AMI	Ubuntu Server 20.04 LTS (HVM), SSD Volume Type	Ubuntu Server 20.04 LTS (HVM), SSD Volume Type
Instance Type	m4.large	t2.xlarge
Availability Zone	3 differents AZ	3 differents AZ
Security group	- SSH protocol on port 22 with anywhere IP address selected - HTTP protocol on port 80	- SSH protocol on port 22 with anywhere IP address selected - HTTP protocol on port 81

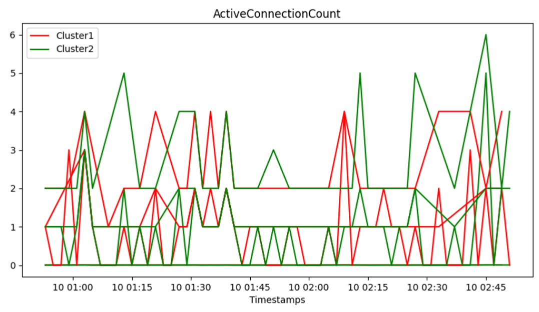
Figure 4: Specifications of the two clusters

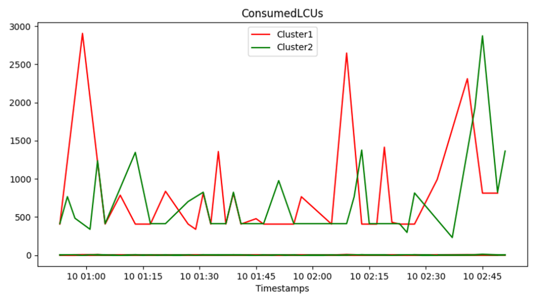
Regarding the metrics, we selected the ones for the Application Load Balancer and those for the Targets groups. **Other metrics exist, but there was no data, so the metric was ignored.** Here are the metrics selected and their explanation:

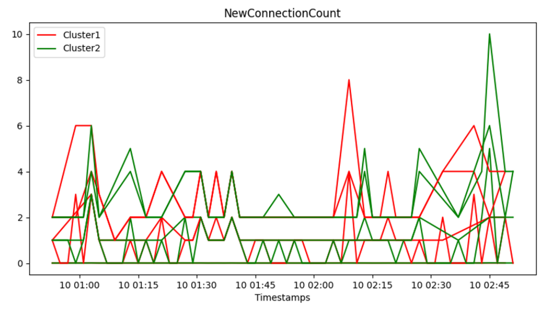
Elastic Load Balancer Metrics

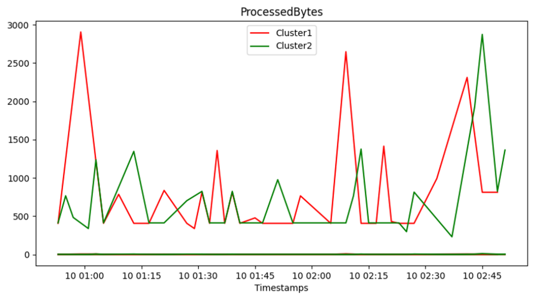
Table 1 - Elastic Load Balancer Metrics

Metric	**Description**	**Results**
Requests (Count)	The number of requests processed over IPv4 and IPv6. This metric is only incremented for requests where the load balancer node was able to choose a target. Requests rejected before a target is chosen (for example, HTTP 460, HTTP 400, some kinds of HTTP 503 and 500) are not reflected in this metric.	The Y axis is the number of requests processed and X axis is time. Both clusters behaved quite similarly. The spikes did not happen at the same time, but the amount of spikes, the average and the fluctuations are very similar when looking at the graphs. Cluster #2 reached a maximum of 5 requests and Cluster #1 reached a maximum of 4 requests.
Active Connection Count (Count)	The total number of concurrent TCP connections active from clients to the load balancer and from the load balancer to targets.	Very volatile, the graphs show that it is not stable for both clusters. On the y axis is the number of active connections and the x axis is time. The y value fluctuates a lot due to the fact the active count of TCP connections varies quickly. Overall we can see that the cluster #2 reached a higher maximum of Active TCP connections simultaneously (6vs4 connections)
New Connection Count (Count)	The total number of new TCP connections established from clients to the load balancer and from the load balancer to targets.	Y axis is the number of new TCP connections and X axis is time. Once again, very volatile for both clusters. The number of connections fluctuates a lot and it is very difficult to compare the results.
Processed Bytes (Bytes)	The total number of bytes processed by the load balancer over IPv4 and IPv6. This count includes traffic to and from clients and Lambda functions, and traffic from an Identity Provider (IdP) if user authentication is enabled.	We can see that the cluster #1 had more major spikes through time (4 spikes vs 1 spike). Both clusters reached a maximum of approximately 2800+ and a minimum of around 500 consumed LCUs. By simply looking at the graphs the average of consumed LCUs is higher for Cluster#1 due to the amount of spikes, but the difference is nothing major. Cluster#1 had different major spikes through time, but cluster #2 only had one major spike towards the end of the time axis. Y axis is the number of LCUs consumed and X axis is time.
Consumed Load Balancer Capacity Units (Count)	The number of load balancer capacity units (LCU) used by your load balancer. You pay for the number of LCUs that you use per hour.	We can see that the cluster #1 had more major spikes through time (4 spikes vs 1 spike). Both clusters reached a maximum of approximately 2800+ and a minimum of around 500 consumed LCUs. By simply looking at the graphs the average of consumed LCUs is higher for Cluster#1 due to the amount of spikes, but the difference is nothing major. Cluster#1 had different major spikes through time, but cluster #2 only had one major spike towards the end of the time axis. Y axis is the number of LCUs consumed and X axis is time.









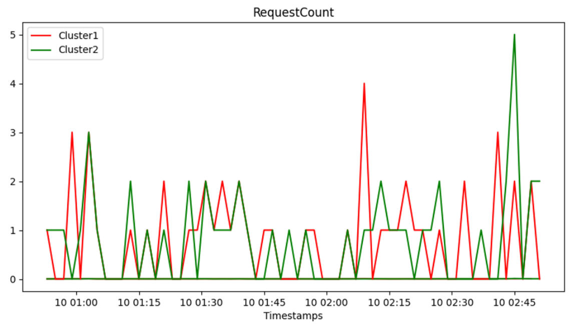
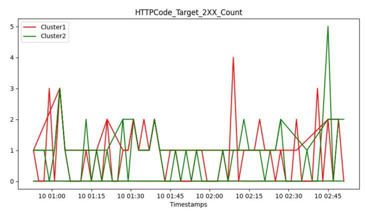
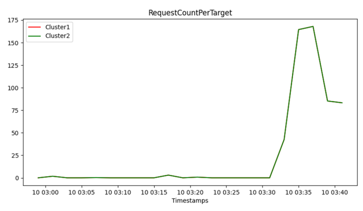


Figure 7: Graph of Elastic Load Balancer Metrics for both clusters

Metric	**Description**	**Results**
Target Response Time (ms)	The time elapsed, in seconds, after the request leaves the load balancer until a response from the target is received. This is equivalent to the target_processing_time field in the access logs.	The Y axis is the time elapsed in seconds until a response from the target is received and the X axis is time itself. When comparing both clusters with the graphs, we can see cluster#2 performed better than cluster #1, by keeping the response time lower on average. Towards the end (02:45), the response time increased quite drastically for cluster #2, which corresponds to a major spike that can be observed on many other metrics. Same goes for Cluster #1, a major spike in ressources occurred around 02:10 and we can see the response time increased accordingly. Overall, Cluster #2's response time is low (0.002 seconds) the majority of the time. Cluster #1's response time is generally fluctuating between 0.002 and 0.004 seconds which would make it for a 0.003 seconds low-average.
HTTP 2XXs	The number of HTTP response codes generated by the targets. This does not include any response codes generated by the load balancer.	Consistently sends HTTP 2xx response codes. We can see that the graph fluctuates in a similar way to the RequestCount graph, which makes sense because most generated requests by the target are 2xx's. We can observe two spikes that occur at the same time as the RequestCount graph. Y axis is the number of HTTP 2xx's requests and X axis is time.
Request Count Per Target	The average number of requests received by each target in a target group. You must specify the target group using the TargetGroup dimension. This metric does not apply if the target is a Lambda function.	The number of requests per target is exactly the same for both clusters, this is why we only see one line in the graph. This makes perfect sense because the workload that was sent to both clusters is exactly the same due to the load balancer. Y axis the request count per target and the X axis is time.

Table 1: Target Group Metrics





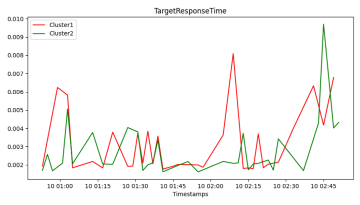


Figure 10. Graph of *Target Group Metrics* for both clusters

1. Instructions to run the code

Prerequisite:

- Having the source code ready on the machine
- Having docker installed on the machine
- Having a working aws environment with 10 instances (5 m4.large and 5 t2.xlarge), 2 target groups (1 for the m4.large instances and 1 for the t2.xlarge instance) and 2 load balancers (one per target group)

Before running the code, some configurations need to be done. There are essentially three pieces of information to input so the program can run smoothly. All of this needed information is to be added in the *main.py* file. Let's now see the changes to be made:
1. One must provide the urls of the public DNS of the two ELBs in the function *consumeGETRequestSync.* With those urls, it makes it possible to do the GET requests needed when running our scenarios for each cluster.
2. One must provide the **name** of the **LoadBalancer** and the **TargetGroup** for each cluster in the function *findNameValue.* It's purpose is to find the name of the **LoadBalancer** and **TargetGroup** for each cluster which are required to retrieve the metrics when requesting them.
3. One must provide the **aws credentials** when instantiating the *boto client.* This step can be done in the computer directly but doing it directly in the code is a better idea given the two other configurations to make. This step is necessary to provide access to **AWS** to the *boto client* so it can run operations such as requesting metrics.
After making these three changes, the rest of the process is pretty straightforward. Run the **bash script:** *script.sh* which will build and run the docker instance of the application. After that, simply wait for the docker instance to finish. Then, the next step would be to retrieve the generated figures from the container. To do that, we must run the command: *docker cp CONTAINERID:/images/ /path/to/destination* where the id of the container can be found with the command *docker ps -a.* A folder named *images* containing the figures for each metric should appear in the destination path. Voilà!
These steps can be summarized into the following points:

1. Configuration the three required informations in *main.py*
2. Execute the *script.sh* and wait until the docker process finishes.
3. Copy the generated figures from the docker container to the host machine

5 Acknowledgements

We would like to thank Amir for all the help he provided in the achievement of this assignment.

References

Amazon. (n.d). *What is an Application Load Balancer?.* AWS.
`https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html`

Amazon. (n.d). *CloudWatch metrics for your Application Load Balancer.* AWS.
`https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-cloudwatch-metrics.html`

`https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3`