HillClimb@Cloud
Cloud Computing and Virtualization
MEIC-IST

83531 - Miguel Belém - miguelbelem@tecnico.ulisboa.pt
83567 - Tiago Gonçalves - tiago.miguel.c.g@tecnico.ulisboa.pt
83576 - Vítor Nunes - vitor.sobrinho.nunes@tecnico.ulisboa.pt

Abstract

This report explains the second phase of the Cloud Computing and Virtualization course project. This project makes use of code instrumentation to obtain metrics from a running worker so we can estimate the cost of a request on a Load Balancer and Auto-Scaler.

1. Introduction

This project consists on developing a Load Balancer and Auto-Scaler for a program running on Amazon Web Services that searches for the maximum value of a map using different search strategies (A*, BFS, DFS). For that, we instrumented the code so that we could produce metrics about the running instances and adapt the system to get the maximum value of the instances running in AWS at a minimum cost.

Although this project uses a specific program to load the instances its objective is to be able to generalize the procedure in a way that the Load Balancer and Auto-Scaler would work even if the given program is completely different given that the code running on the instances would be correctly instrumented.

2. Architecture

We have two types of Instances. The Worker Instance, which runs the Web Server that solves the requests made by the users to the Load Balancer. The Load Balancer (Auto-Scaler and Metric Storage System) instance, which has the Load Balancer that distributes the requests from the clients to the running Worker Instances. It also implements the Auto Scaling functions, such as launching (scale-up) and terminating instances (scale-down). This instance is also runs the Web Server for the MetricStorageServer which is the point of communication with our table on DynamoDB.

2.1. Types of Instances

2.1.1 Worker Instance

We created a Linux Image based on the Linux AMI AWS - t2.micro image, which is the one eligible for free tier usage on AWS. It has a single core (up to 3.3 Ghz) and 1 GB of RAM. The Linux distribution was updated and extra packages needed for running the Web Server and BIT were installed. We load the Web Server that runs the solving code on port 8000 and altered on-boot scripts to start up the Web Server every time that the system is booted.

We created an AMI so that we could replicate the instance and use it on a Load Balancer.

2.1.2 Manager Instance

The Manager Image has the same properties as the Workers' Image, but instead of running the Worker Web Server on boot, it executes the Load Balancer (from now on called LB) Web Server and Metric Storage Server (from now on called MSS). The LB receives its requests on port 8001 and the MSS on the port 8002. All communication between the Web Servers is made using HTTP requests using different contexts to execute distinct operations.

2.2. Worker

The Worker's Web Server has 3 contexts to receive requests:

- **Climb:** receives the parameters for the search algorithm and runs it when the request is finished

it sends the metrics captured to the MSS and returns the result (an image) back to the LB so it can resend it back to the client.

- **Ping:** answers with a *"Pong!"* message to whoemever sends a ping request.

- **Progress:** returns the progress of all jobs running in the instance (jobId percentage).

2.3 Manager

2.3.1 Load Balancer

The Load Balancer runs on a separate machine from Workers along with the MSS. By default, it runs on port 8001 and forwards the users' request to worker that, uppon complete the task will respond to the LB and then the LB forwards the output to the user. The LB has several Timer Tasks running, namely:

- **Auto Scaler:** executes every 30 seconds, whose goal is to decide to either scale up or down instance deppending on the global system load.

- **Starter:** executes every 10 seconds, whose goal is to start instances and make sure that the LB only consider them available after the operating system and the Web Server are running. [1]

- **Terminator:** executes every 25 seconds, whose goal is to shutdown instances but only those that are marked as safe for delete. [2]

- **Test Timer:** executes every 10, debug timer to display state of all available instances.

- **Job Percentage Task:** executes every 15 seconds, periodically collects the progress of current jobs in all available workers.

The LB stalls the users' requests if currently there is no worker available. The LB also **redirects the users' request** to a different worker in case of the first one has crashed without responding to that request. This guaranteees that the client will eventually receive the reply. Is up to the LB to terminate the crashed worker because even if the Web Server on the Worker is not operating, the instance is still up and wasting resources.

[1] Periodically sends ping to the new worker to test liveliness.

[2] An instance is considered safe for delete when has no jobs running.

2.3.2 Auto Scaler

Firstly we created an Auto Scaling Group on AWS that increases the number of instances by 1 if the average CPU utilization is over 60 % in a period of 5 minutes and decreases the number of instances by 1 (to a minimum of 1 instance) if the average CPU utilization of all instances is under 40% for a period of 5 minutes. But this values have no real correlation with the metrics obtained.

So **we improved the AS in order to take account the average cost of all instances** instead of CPU utilization. To get the average cost of all instances (basically the load running on the Workers). We keep updated on the Job Object the percentage already done by the Worker which is updated by making a HTTP request to the progress context of the Worker (running on Workers) which is done in the TimerTask **JobPercentageTask**.

We use the cost estimation of the Job and percentage already done by the Worker to get the current load of the instance. We do this for every Job on that instance.

$$Cost_{Instance} = \sum_{job}^{Jobs_{OnInstance}} job.Cost \times \frac{100 - job.Percentage}{100}$$

After calculating the cost of all instances we divide it by *n-0.5* (n is the number of instances running). [3]

In order to evaluate if we want to scale up or scale down we want to take into account 3 situations:

- *Situation 0* - There are no alive instances.

- *Situation 1* - **Cost over 600 000** and the number of instances is bigger than 0 and lower than the defined MAX_NUMBER_INSTANCES

- *Situation 2* - **Cost under or equal to 400 000** and the number of instances is bigger than 1.

We use two counter to track this situations. They are increased based on the following function:

$$Counter+ = \lfloor CostAverageInstances \div \#Instances \rfloor$$

In the first situations the task simply launches an instance. But in a situation where we already have an instance running we have counters to track the number of times that a specific situations occurs. If the *Situation 1* occurs 3 times in a row it scales up and launches a new instance (or if an instance is marked for deletion

[3] We subtract *0.5* so we don't enter a situation where we launch a new instance and imediately close it because we didn't receive enough load to mantain it.

Algorithm 1 Scale Algorithm

```
1:  procedure AUTO SCALER
2:      cost ← average cost of all instances
3:      up_counter ← 0
4:      down_counter ← 0
5:      if #Running_instances > 0 then
6:          Launch one instance.
7:      if cost > 600000 then
8:          down_counter ← 0
9:          up_counter ← up_counter + ⌊cost ÷ 600000⌋
10:         if up_counter > 3 then
11:             if #Marked_instances > 0 then
12:                 Unmark one instance for deletion.
13:             else
14:                 Launch one instance.
15:             down_counter ← 0
16:     else
17:         if cost ≤ 400000 then
18:             up_counter ← 0
19:             if #Running_instances > 1 then
20:                 down_counter ← down_counter + 1
21:                 if down_counter ≥ 5 then
22:                     Mark one instance for deletion.
23:                     down_counter ← 0
24:         else
25:             up_counter ← 0
26:             down_counter ← 0
```

it unmarks it and reuses the instance that was going to be terminated) Marking an instance to delete makes it impossible to receive more Jobs and if not unmarked that instance will be terminated.

This allows us to scale up in a situation of continuous heavy load while not scaling down instantly if we get a slight pause of incoming requests.

We want to be careful while shuting down instances as they take some time starting up and if we get a temporary decrease and terminate them and then we might get back to a state where we can't deal with all requests that turns to a higher response time to those requests.

2.3.3 Metric Storage Server

The MSS runs on the same machine as the LB but on port 8002. It has two HTTP contexts:

- **RequestMetric:** receives a query which the server will try to match with another metric on DynamoDB. If not it will calculate an estimation and send it to the one who requested it.

- **PutMetric:** after a worker finishes its job it makes a *putMetric* request and sends the metrics to the MSS which put it in the DynamoDB.

3. Instrumentation

We used BIT (Bytecode Instrumenting Tool) presented in the laboratories to to alter the Java Class Byte Code of the program so we could measure the metrics that we wanted in a way that would help us decide the cost of replying to a certain request.

This metrics need to be heavily weighted as they might give a big overhead to the the original program. To store this metrics we created a class called Metrics, which stores all of the counter for each type of metric. We guarantee that each thread only counts its own metrics by using a ConcurrentHashMap to store all the metrics, where the key is the Thread ID and the Value is the Metrics Object (containing all the metrics). At the end of each request we store permanently the results of the instrumentation and remove the Metrics file from the HashMap, this allows to reset the counters for that specific thread.

With the metrics it's also stored the parameters given to program so we can associate those with the load it creates and in the future compare them with new requests.

3.1. Metrics Used

We decided to try a few simple metrics. Instructions run, Basic Blocks, Methods Called, Branches Queries, Branches Taken and Branches not taken. We ran the instrumented Web Server on an instance of AWS and made some requests (one at a time) while tracking the time it took to reply to each request. We obtained a table with all the values and concluded that some metrics grew linearly with the time to reply a request. Those metrics were Instructions Run, Basic Blocks, Branches and Branch not Taken.

Methods Called and Branch Taken were inconsistent because they depended on the search algorithm being used and do not necessarily mean a higher cost to reply to a request as different methods could have different number of instructions which could mislead the cost of the request.

As the other metrics scaled linearly with the time to reply to the request we can associate a higher number on this metrics to a more CPU intensive task. We decided to stick with **Basic Blocks** and **Branches Not Taken** due to its linear grow matched with response time to the request and also because they provided the lower overhead to the running code.

3.2. Storing the metrics

3.2.1 Real Cost Calculation

In order to use the two metrics (Number of Branches Not Taken and Number of Basic Blocks) we designed a cost function based on these two values.

$$Cost(\#BB, \#BNT) = \frac{\frac{\#BB \times \#BNT}{\#BB + \#BNT}}{1000}$$

This was created because branch not taken and basic blocks growth in different ways and we wanted to have the cost calculation based on growth of both without giving more importance to one then the other, so we tried to create a formula to minimize this disparity. The division by 1000 is just to minimize the occurrence of overflows.

3.2.2 Cache

We noticed some latency when performing multiple requests to DynamoDB, in order to avoid some of this latency, we created a local cache in the MSS. This cache has N entries (for testing proposes is set for $N = 20$, in a real system the value should be reviewed), it implements last recently used policy, so when 21st request arrives, the 1st request put is deleted from the cache. With this if a request arrives to MSS and gets a hit, we do not need to consult DynamoDB and the cost is returned directly to the Load Balancer.

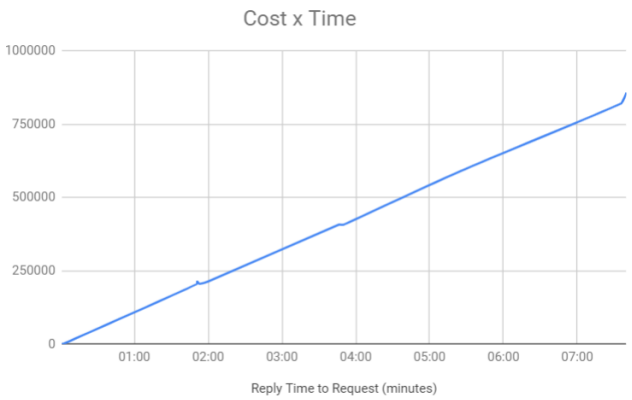

Figure 1. Relation between Cost and Time

3.2.3 Cost Estimation

When a new request arrives and we already have information in the DynamoDB, we will try to estimate the cost of the new one, based on the ones previously stored. To this we try to fetch from DynamoDB the closest ones. The following filters are applied to classify if an entry in the DynamoDB is similar to the new request :

- Same search algorithm;
- Same image;
- The initial point must be the same with a margin of $+$ or -10;
- search area must be the same with a margin of $+$ or -2000. With all the requests matched we perform an average of their costs and use it as the estimated cost for the new request.

3.2.4 Retrieving Cost

When the Load Balancer makes a *requestMetric*, several verifications are performed, such as:

1. We check if the request is in cache, if it is we instantly return it;
2. If we have a cache miss, we will ask to the DynamoDB if it has already stored that request and if it is we return it to the Load Balancer;
3. If we can't return the cost we ask DynamoDB for similar requests that it has stored and perform the average of all costs, then the value is returned to the Load Balancer;

4. If all previous options failed it is returned a default value, equivalent to medium size request.

3.2.5 Storing in DynamoDB

None of the estimated costs are stored in DynamoDB. A metric is only stored when a worker sends *putMetric* request, after we write to the DynamoDB and tries to put it cache (might not put it if it already exists in cache).

4. Appendix - Metrics Measurements

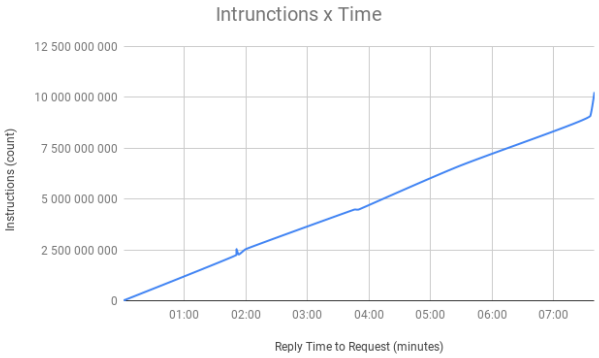

Figure 2.

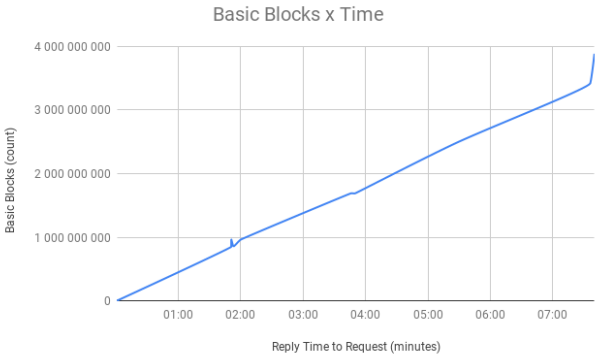

Figure 3.

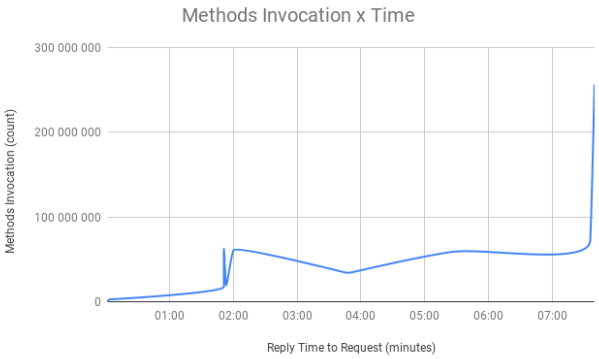

Figure 4.

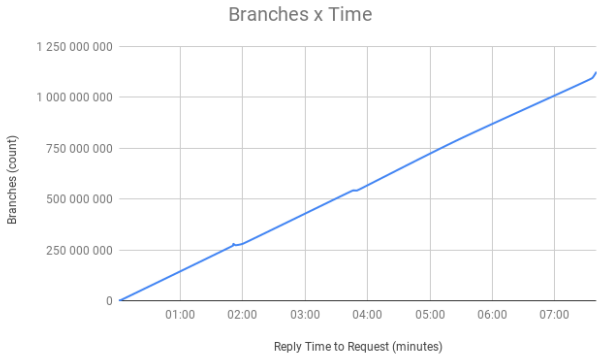

Figure 5.

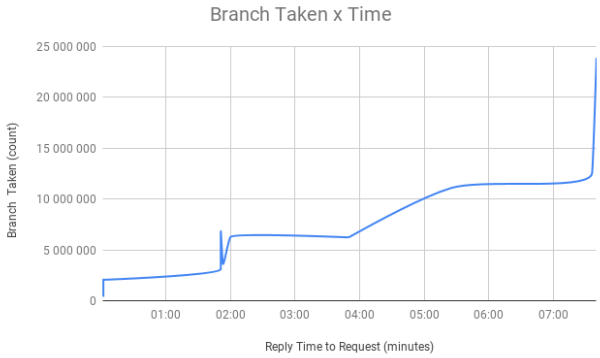

Figure 6.

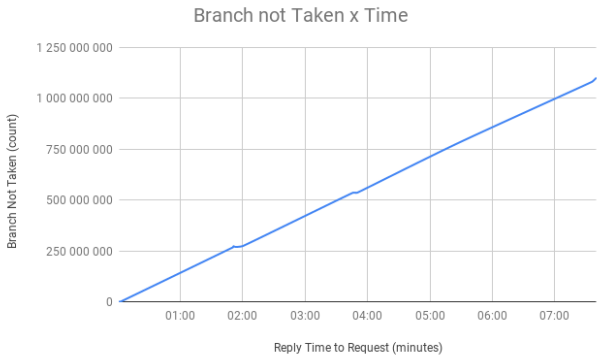

Figure 7.