EcoWork@Cloud

Group 04

Diogo Miguel da Silva Santos
ist195562

João Pedro Antunes Aragonez
ist195606

Vasco Miguel Cardoso Correia
ist194188

1. Introduction

In this report, we will be discussing our implementation of a service hosted on AWS to support the EcoWork@-Cloud workload. We include a detailed description of the adopted Task Scheduling and Auto-Scaling Algorithms, as well as how we assure that clients requests are always answered within a short time span.

2. Architecture

2.1. Components

The architecture is composed of a monitoring Web server containing both the load balancer, the auto-scaler, and the health checker, a group of workers running on Elastic Compute Cloud (EC2) instances, a DynamoDB database to store the metrics, and a Lambda function for each endpoint.

2.2. Workflow

Upon receiving a request, after the load balancer decides whether the request is going to be processed by an EC2 instance or a Lambda function, it waits for a reply and forwards it to the pending client. The process of deciding where a request goes relies on an estimator and some empirical data collected by running offline tests, which we will describe further ahead.

3. Instrumentation Metrics

Empirical testing revealed that the most accurate metric in measuring request complexity is the instruction count, so it was our choice.

There is an overhead associated with the instrumentalization of code since we are injecting extra byte-code.

4. Data structures

In this sections we describe the relevant data structures and objects used throught the system.

4.1.1. Javassist

We instrumentalized the relevant function calls for each operation by calling a `createMapping` function that creates a *threadID* to *Statistics* mapping. *Statistics* objects store the number of instructions and basic blocks. When the function ends, we free the mapping, allowing that *threadID* to serve other requests, and store the *Statistics* object in a permanent *URI* to *Statistics* mapping if an entry mapping from the same *URI* doesn't already exist. Additionally, when a new entry is added to the previous mapping, the (*URI*, *Statistics*) pair is added to a queue that is periodically loaded into DynamoDB by the worker.

4.1.2. Request

Contains all the information of a request, such as *URI*, endpoint, parsed arguments, the payload (for *POST* requests), the cost of the request and the *HttpExchange* connection to the client.

4.1.3. InstanceInfo

Stores data about a given instance, such as current running requests, number of missed health checks and the *Instance* object from the AWS SDK.

4.1.4. AWSInterface

For the monitoring webserver to interact with AWS, we designed a wrapper class, `AWSInterface`. It implements the necessary remote procedure calls for the Load Balancer and Auto-Scaler to function, such as `createInstances`, `queryCPUUtilizationHomeMade`, `callLambda` or `getFromStatistics` to access the DynamoDB. It stores a set of alive instances, which represents the EC2 instances available for processing requests; Auto-Scaler also stores a set of suspected instances ('InstanceInfo'), which is used by the Health Checker to temporarily store instances that might not be working properly. Apart from the instance data, it also keeps a cache used to maintain the statistics of the requests that were recently loaded from the dynamoDB. To make the cache as time-efficient as possible, we have a queue limited to a size of 512 requests, with the goal of keeping only the most recent request, and a map containing the <Key, Value> pairs as <URI, Request_Statistics>, for fast access to the request's statistics. Due to the small size of the cache, having two separate data structures doesn't degrade memory efficiency. Credentials and connections to AWS (EC2, CloudWatch, and LambdasClients) are also stored here.

4.1.5. Load Balancer

No data is explicitly stored in the load balancer, but every request received is stored on the respective `InstanceInfo` kept by the `AWSInterface` in the aforementioned set of alive instances, which allows the load balancer to know how much load each instance has, leading to a better distribution. Upon replying to the client, the request is cleared.

4.1.6. Auto-scaler

The Auto-scaler doesn't explicitly keep any special data structures but it updates the aforementioned set of alive instances in the `AWSInterface` when it decides to scale up or down the system.

5. Request Cost Estimation

The load balancer contains an "estimator", which is the entity responsible for computing the cost estimate of every request. Since we instrumentalized the worker to measure instruction count, we will be using it as the cost of each request. To understand each request, we created a Python script to test multiple combinations of arguments and see how each one affects the resulting instruction count.

The graphs present in this section will all have the same labels: a label indicating what the axis is, a title, and an equation representing the linear regression of the points.

5.1. Foxes and rabbits

Foxes and Rabbits has three possible arguments: *generations*, *world* and *scenario*. Given that the number of worlds and scenarios is small, we can test all combinations while varying the number of generations.

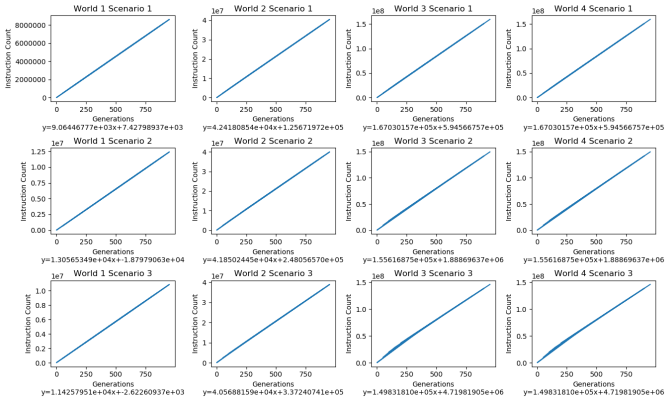

Figure 1: Foxes and rabbits endpoint graphs

In Figure 1, each vertical line represents all the scenarios for a given world. The important finding here is that varying the scenario within the same world only affects the instruction count by a small margin. This means that we can average the slope of each equation with the same world and use that to estimate the cost, resulting in three different equations per world.

The values found were:

- W1: $\text{instructionCount} = 11182.2659 * \text{Generations}$
- W2: $\text{instructionCount} = 41612.3819 * \text{Generations}$
- W3: $\text{instructionCount} = 157492.947 * \text{Generations}$
- W4: $\text{instructionCount} = 157492.947 * \text{Generations}$

5.2. Insect Wars

Insect Wars has three possible arguments: *max*, *army1*, *army2*. We found this request to be very hard to estimate, for multiple reasons:

- The *max* argument only sets a ceiling for the number of iterations the war will run, meaning that it can end earlier, leading to an excess in the estimative
- It's hard to find the weight that each argument has on the final cost since they are all dependent.

Initially, we tried to find the weights in this equation:

$$\text{instructionCount} = a * \max + b * \text{army1} + c * \text{army2}$$

However, this equation doesn't take into account the difference between army sizes, which in our tests was a driving factor for the cost. For example, having armies of similar sizes tends to have higher costs than having a small army fight a bigger army.

This leads us to a new equation:

$$\text{avg} = \frac{\text{army1} + \text{army2}}{2}$$

$$\text{diff} = (\text{army1} - \text{army2})$$

$$\text{instructionCount} = a * \max + b * \text{avg} + c * \text{diff}$$

To find the weights a, b and c, we tried multiple combinations and concluded that $a = 0.51, b = 0.48$ and $c = 0.01$ gave us the smallest error when compared to the real values. These values don't match our expectations since they attribute a very small weight to the armies differences, but in a constrained time window, they were the most precise alternative.

5.3. Compression Image

Compression Image is a *POST* request but we can say it has three arguments: *imageSize*, *targetFormat* and *compressionFactor*. This endpoint is not very well instrumented since it uses native code to do the compression, and as a result, it doesn't count towards our instruction count metric. Nevertheless, we decided to try to measure it and retrieve some results so some type of estimate could be computed.

We decided to measure the instruction count across different compression factors for the same target format.

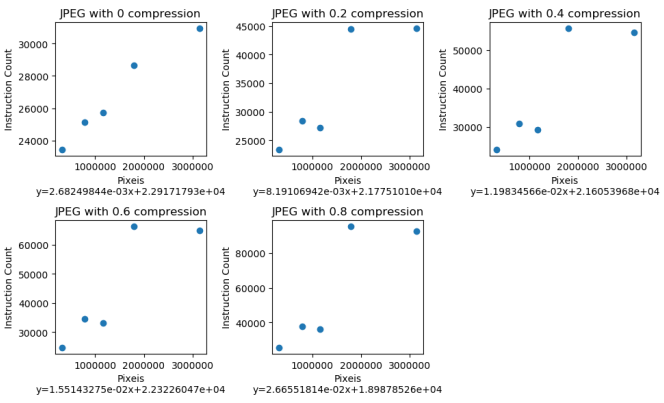

Figure 2: Compression PNG endpoint graphs

In Figure 2, we noticed that a bigger compression factor resulted in a bigger cost.

Therefore, to estimate the cost, first we compute the slope associated with the given compression factor by starting from a *baseSlope* and since we observed that the compression factor affects the slope of the graph linearly, we then compute the *slope* by adding the difference between *maxSlope* and *baseSlope* multiplied by the compression factor.

From this, we decided to create the equation:

$$\text{diff} = (\text{maxSlope} - \text{baseSlope})$$

$$\text{slope} = \text{baseSlope} + \text{diff} * \text{compressionFactor}$$

$$\text{instructionCount} = \text{slope} * \text{pixels}$$

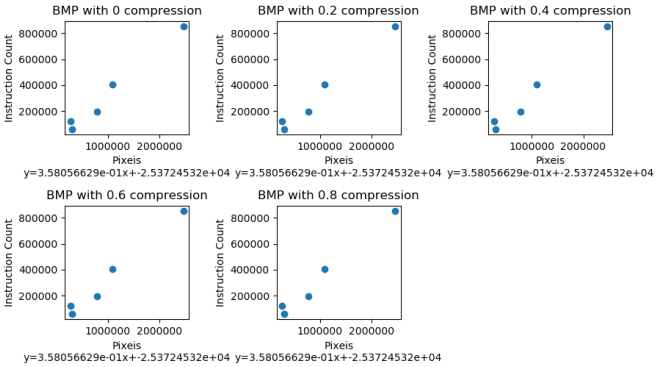

Figure 3: Compression BMP endpoint graphs

In Figure 3, the compression factor didn't change the estimated cost, so we simply used the slope from the linear regression to get the equation:

$$\text{instructionCount} = \text{slope} * \text{pixels} + b$$

where b is the intercept at the y axis.

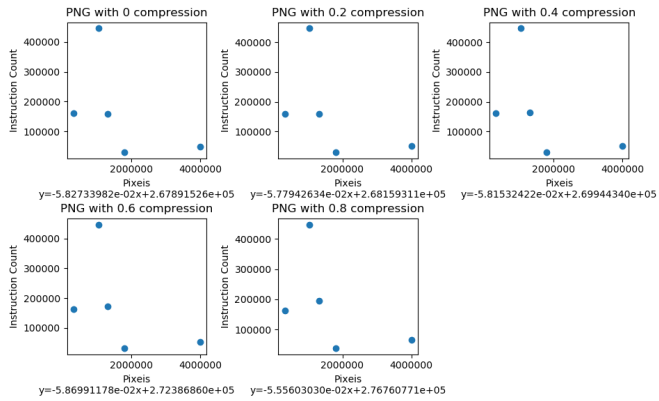

Figure 4: Compression PNG endpoint graphs

In Figure 4, the values did not follow anything close to a linear regression. This can either be caused by the fact that PNG has transparent pixels, which affect the cost in unpredictable ways, or bad input from our tests. Due to time constraints, we computed the average across those requests and estimated a constant value of 276958.771.

6. Task Scheduling Algorithm

The task scheduling algorithm can be divided into two distinct phases. First, we compute the request cost, and then we decide where to execute it.

6.1. Request Cost

Given that every request will have a "correct" cost after it is executed on the instrumented worker and uploaded to the DynamoDB, we aim to take advantage of this by fetching the cost from the database. However, waiting for the query to finish on every request will slow down the application, which led us to create a cache structure that stores a mapping of *URI* to *Statistics*.

Upon receiving a request, we check the cache for the request URI; if we find one, there is no need to estimate, and if we don't, we use the previously mentioned `Estimator` to get an approximate value. In the event of a cache miss, we also spawn a new thread to, in the background, query the DynamoDB. Initially, we would query for the exact request, but this wouldn't bring much value if the request was new, so we switched to a "range" query, where we retrieve requests that are similar (within a given threshold) and average their cost to get a much better estimate.

These thresholds depend on the request endpoint, and two requests are considered similar if: Simulation: generations±50 and (exact) world and (any) scenario War: max±30, army1±10, and army2±10 Compress: pixels±1000 and (exact) target format and compression±0.1

Due to the known benefits of caches in terms of space and time locality, we decided to also store these similar requests in cache, assuming that future requests would be similar to the one just received.

6.2. Where to execute

Upon computing the cost of the request (and noting that it may change due to the background task), we must decide where to execute it. Since every instance stores which requests they are running, it's easy to query them and find the lowest-loaded by summing the cost of each request. However, not taking into account the current request cost could lead to the overwork of certain machines, by, for example, sending a high-cost request to an already loaded instance, leading to CPU usages above 100%. To solve such an issue, we estimated the maximum number of instructions that a single instance can execute. This value was measured by sending requests with known costs and observing the CPU load, assuming that if the CPU usage was 100%, then the current number of instructions was the maximum supported. This value is highly empirical and can't be measured very accurately, but it enables us to detect if a given request can "fit" on an instance. Knowing this, we can then decide if the request is small (up to 20% of the maximum number of instruc-

tions a machine can execute) and can be offloaded to a lambda function or if it's too big and must wait until there is enough "space".

7. Auto-Scaling Algorithm

The auto-scaling algorithm is responsible for deciding when to increase or decrease the number of EC2 virtual machine instances available to process client requests. Before explaining our solution for this problem, it is necessary to clarify some terminology.

7.1. Concepts

Every time we refer to "scaling up", we mean increasing the number of instances by 10%, taking the ceiling of the result, and subtracting the previous number of instances from it. The resulting delta is the number of instances that will be created. When we refer to "scaling down", it is very similar; we take 10% of the number of instances, compute the ceiling, and terminate that number of instances.

7.2. CPU Measurements

Another useful clarification is how the CPU utilization averages are obtained. Initially, we were using Amazon's CloudWatch (and we still support it in the code), but the problem with that is the very large latency, meaning the numbers reported by CloudWatch are already quite "old" when we read them. To solve this, we read the 1-minute load average from `/proc/loadavg` and serve it on an endpoint from every worker instance. We never ran into any problem where the machine became overworked and didn't reply to this endpoint, but in case this happens, we can always fall back and use *CloudWatch*.

7.3. Solution

Going back to the main problem, it can obviously be subdivided into three smaller and simpler problems: deciding when to increase, decrease, and maintain.

7.3.1. Scaling Down

The simplest case is deciding when to decrease the CPU usage of all EC2 instances. If the usage is lower than 20% and there is more than one machine alive, we scale down.

7.3.2. Scaling Up

Scaling up is more complex. We want to make sure we're not creating instances without necessity, so we only decide to scale up if the average CPU usage is above 80% and there are still requests pending. This last part is to avoid creating unnecessary instances from a short burst of requests that were already served.

If none of these conditions are observed, we just decide to maintain the number of instances.

7.4. Implementation

In order to facilitate the sharing of data between them, the auto-scaling algorithm runs on the same machine as the task scheduling algorithm. They share the instance sets (alive, suspected, and pending), which allows for easier coordination. When the auto-scaler decides to scale (down or up), those changes are immediately visible to the load balancer.

8. Fault-Tolerance

Machines can fail and have unexpected errors. Therefore, we implemented fault-tolerance mechanisms at two distinct levels.

The first handles software-related errors, such as AutoScaler exceptions thrown by libraries or any 5XX HTTP code errors. The implementation is a simple retry with a maximum number of 3 tries, which, if reached, returns a 500 error code to the user.

The second handles EC2 instances errors by implementing a health checking mechanism. Upon starting the monitoring web server, a separate thread starts to run the Health Checker, and every 10 seconds it pings all alive instances with a GET request. If an instance fails to reply, it's moved to a suspected instance list, preventing the load balancer from forwarding any requests there. An instance that misses three health checks is terminated, and all their pending requests are sent to other alive instances using the scheduling algorithm previously mentioned.