CNV Project: Final Report - Group 7

Diogo Neves	Filipe Silva	João Moniz
95554	95585	83480

17 June 2023

1 Introduction

For the final delivery, our objective was to develop an auto scaler and a load balancer, both in the Java programming language and using the Amazon SDK, that could do a work similar to the auto scaler and load balancer encountered in the Amazon Web Services (AWS).

With this objective in mind, we were given a workload (EcoWork@Cloud, in the *webserver* folder) that created a *webserver* with three scenarios:

- Image Compression
- Foxes and Rabbits
- Insect Wars

Our objective was, then, to develop these applications, alongside scripts to create the images programmatically and deploy them in AWS EC2.

We will then, in this report, show our design decisions for each of the following modules of our work (in the git repo, each one is a folder located in the *src* folder):

- javassist
- lbas
- scripts
- webserver

2 Design

2.1 General

An overview of the steps taken to deploy this system is as follows:

1. Use the scripts to create the images for both the Load Balancer/Auto Scaler (*LBAS*) - image name *CNV-LBAS* - and the *Webserver* - image name *CNV-Webserver*.
2. Use the scripts to deploy an EC2 instance with the *CNV-LBAS* image, and run it.
3. This instance will deploy other EC2 instances with the *CNV-Webserver* image, according to the load it receives.
4. Also accordingly to the load, the *LBAS* instance will select an instance, using the parameters in the request and the metrics gotten from the *webserver* instances (that are running the workload provided with a custom Javassist tool to gather and send data to the Load Balancer).
5. It can also choose to launch a Lambda function, if it deems it necessary (like when a machine is starting up and cannot receive requests yet). It will only use these when necessary, due to their high cost relatively to EC2 instances.

2.2 Javassist

The Javassist module consists of two main parts: the *ICount* tool, modified to report the instruction count per thread (because the workload is multithreaded); and the *AmazonDynamoDBConnector* class, which consists of a custom connector to the Amazon DynamoDB, a key-value store, used to create and send items to DymanoDB. These will be used to keep certain information about the program we will run it with.

This module is then ran with the *webserver*, and allows to know how many instructions were ran for a specific scenario of it, by intercepting each one of the functions that correspond to a scenario:

- *Image Compression* - *process* function, with the following arguments: image, target format and compression quality.
- *Foxes and Rabbits* - *populate* and *runSimulation* functions: the first to get the world that was requested (important for the metrics); the second to get the number of generations to simuate.
- *Insect Wars* - *war* function, with the following arguments: size of each army and the maximum number of simulation rounds.

The number of instructions per thread is saved on a map, and then metrics are created according to each scenario, and then sent to DynamoDB in order to be read later by the Load Balancer, in order to choose an EC2 instance to send the request to.

In order to keep the number of requests to DynamoDB to a minimum, we only update the information kept there every *BATCH_SIZE* requests (default value is 5, but can be changed in the *ICount* class). The information is kept in a "cache" (a *Map* in the class for each scenario, alongside a *Float* for the *InsectWars* scenario: it has no separator between different types of workloads, such as a world or a type of picture format).

The data sent to DynamoDB is already processed as well (which means that we are sending to it only the computed statistics and not all the parameters), in order to reduce computation on the Load Balancer side.

The statistics sent to DynamoDB are the following:

- *Image Compression* - the metric chosen was

$$\frac{I}{width * height * f}$$

, where I is the number of instructions, *width* and *height* are the width and height of the image and f is the compression factor. This metric is then sent to DynamoDB for each format: PNG, JPEG and BMP.

- *Foxes and Rabbits* - the metric chosen was

$$\frac{I}{G}$$

, where I is the number of instructions and G is the number of generations. This metric is then sent to DynamoDB for each world (the generation doesn't matter much for this workload).

- *Insect Wars* - the metric chosen was

$$\frac{I * r}{max * (sz1 + sz2)}$$

, where I is the number of instructions, r is the ratio between the sizes of the two armies, *max* is the maximum number of rounds and *sz1* and *sz2* are the sizes of the armies. The ratio is kept in mind because it changes somewhat the number of instructions (a bigger disparity between armies leads to a quicker battle and less instructions ran).

It is also relevant to mention that the statistics are updated and not replaced every 5 requests: it follows an exponential weight, where the new request's metrics count for 50% of the new metric, and all the older metrics count for the other 50%. This gives us more adaptability to each workload according to the conditions.

2.3 *LBAS*

2.3.1 General

falar do endpoint de test

2.3.2 Load Balancer

Regarding the load balancer, our goal is to distribute the load as evenly as possible across all nodes, in order to keep the number of nodes spawned at a minimum. It can also choose to serve a request using a Lambda instead of a nodes: this option is more expensive, however it makes up for it for the speed in startup. Especially when the system is under load and more requests are being made while the LB spawns more nodes, Lambdas will be a good option to support users while the system is still trying to scale up.

2.3.3 Auto Scaler

Regarding the auto scaler, the metric we will keep in mind is the average CPU utilization of each node: if the system detects that more requests are being made and the CPU utilization is above a certain threshold, it will spawn a new node. Likewise, if the CPU utilization is below a certain threshold, it will destroy nodes, in order to keep the cost of running as low as possible. The algorithm will scale up by creating new nodes, and scale down by marking the nodes it wants to destroy for termination, however they can only be terminated once there are no more requests being served by them.

2.4 Scripts

The *scripts* folder contains the scripts to create the images and deploy the first instance of this system - the *LBAS*.

Here are the scripts alongside a quick explanation for each:

- *config.sh* - keeps all the variables needed to contact AWS.
- *create-image.sh* - creates the *webserver* and *LBAS* images. It is composed of three other scripts:

  install-vm.sh - Installs the needed packages and programs in an EC2 instance.

  launch-vm.sh - Launches an EC2 instance. Is used on *create-image.sh* but also as a standalone, to launch the *LBAS* instance.

  test-vm.sh - Tests that the instance is ready to go, using the */test* endpoint that is found in all programs used in this system.

Each instance (and script) needs to source the script *config.sh*, which keeps the environment variables needed for the scripts to work, relating to security groups, key pairs, etc.

2.5 *Webserver*

falar do endpoint de test

3 Things to improve

falar de load balancing e esperar para apagar instance

4 Conclusion