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 the *ICount* tool, modified to report the instruction count per thread (because the workload is multithreaded). It also used to have 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. However, it has been moved to the *Webserver*, and we will explain this decision there.

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 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.

Note: the metrics are sent to DynamoDB just as is said above; however, they are sent using the *Webserver* and not this package. It used to be this way but not anymore (again, we will explain this decision on the *Webserver* section).

We chose to keep this section here because it made more sense to talk about the metrics collected in the section where we talk about getting the information used to create these.

So, in conclusion, this module is used to get the number of instructions, calculate the metrics, and then it sends them to a text file */tmp/dynamodb* every *BATCH_SIZE* requests, where the *Webserver* will collect and send them.

2.3 *LBAS*

2.3.1 General

An overview of the *LBAS* operation:

1. Creation of a *HashMap* to keep track of the EC2 instances and their respective CPU utilizations, alongside two *AtomicInteger*s to know how many instances we have (both created and available, as one can still be initializing).

2. Creation of the *AutoScaler* and *LoadBalancer*: while the load balancer will work as a webserver, responding to

endpoint requests, the auto scaler will be running every *DELAY_AUTOSCALER* seconds (default: 10 seconds).

2.3.2 *Load Balancer*

Our goal for the load balancer is to distribute the load as evenly as possible across all instances, in order to keep the number of new instances as low as possible.

As stated before, the load balancer works as an webserver (much alike the actual webserver we instrumentalized), and also has a */test* endpoint in order to do a sanity check of its operations.

Its process goes as follows:

1. Create endpoints with the same names as our workload.
2. For each request, analyse the parameters given, and calculate an estimation of the number of instances that request will take, using data gotten from DymanoDB.

   Note: we keep the metrics in a local cache (using Java structures as in the *ICount* class stated before), and only every *DYNAMODB_CACHE* (default: 5) requests do we fetch data from DynamoDB. This helps us reduce the latency introduced by going to the database every request.

   Note: if there are no metrics available, the estimation will be -1. See below for a better explanation of this scenario.
3. Using this estimation, select the instance to send the request to.
4. Act as a proxy: send the request and receive the response, then forward the response back to the client.

For the instance selection, we need to keep in mind two scenarios:

1. There are no metrics available yet: the system just started or is in its first requests. In this case, we will choose an instance using round-robin: we sort the instances by their CPU average usage (more on that in the *Auto Scaler* section), and then iterate through these as more requests come by: the instances with least CPU usage will be chosen first. Once the map containing the instances and usages is updated, so will this list.
2. There are metrics available: we apply a best-effort algorithm, where we sort the instances by the number of instructions (estimated) that it has already done, and go from the instance with more instructions ran to the one with least ones, trying to ”fit” these new instructions in it.

   We associate the instructions ran already with the CPU usage for each instance, and then estimate the CPU usage for that request: if the sum of the usage for the instance and the estimated usage for the request is under 100%, we assign the request to that instance. That will lead to a smaller number of instances being created.

On specific scenarios, we resort to using a Lambda function instead of a node: this option is more expensive, however it makes up for it for the speed in startup compared to the time an instance takes to start up.

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*

The auto scaler's goal is to respond to higher and lower loads than usual, adding and removing instances as needed.

The metric we need to keep in mind is the **average CPU utilization of all nodes**. Using this, we can define thresholds to scale up and down as needed. These are defined as follows:

- *MAX_CPU_USAGE* (default: 80%)

- *MIN_CPU_USAGE* (default: 20%)

Therefore, the auto scaler works as follows:

1. Iterate over every instance that is running. If it is running and the *AMI_ID* is the one for the image *CNV-Webserver*, we get its average CPU usage for the last 60 seconds. This average is only taken after the instance started up: we do not count the bootstrapping of the system).
2. Calculate the average CPU usage of all nodes.
3. Accordingly scale up or down:

   If there are no instances, start a new one. However, if one is in the *pending* state, do not start a new one just yet.

   If the average CPU usage is above *MAX_CPU_USAGE*, we start a new instance.

   If the average CPU usage is under *MIN_CPU_USAGE*, we stop the instance with lowest CPU usage. However, it is not stopped right there and then:

   - It is marked for termination (this is done by removing the instance from the map from where instances are chosen, effectively not allowing new requests to be made to it)
   - Only after *DELAY_KILL* (default: 60) seconds will the instance be terminated. This is because we want to make sure that the instance is not needed anymore, it might still be running requests. This way, the user won't notice a degradation of the system, as the requests will still go through to it.

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. Is created by the user but there exists a *config.sh.example* that acts as a template.
- *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.
- *create-lambda.sh* - creates the lambdas used in this system.
- *launch-deployment.sh* - creates all images, lambdas and launches the entrypoint to the system (the *LBAS* instance).

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*

The webserver is the program that will be run in the instances. It was provided by the faculty, however it has two main modifications:

- A new endpoint, */test*, used to do a sanity check on the system if needed. Always returns the parameters provided in the request.
- The addition of the *AmazonDynamoDBConnector* class and a subroutine to operate it, as written before. We will explain this decision now.

2.5.1 *AmazonDynamoDBConnector*

For the Image Compression workload, core Java packages are used to compress images. These are loaded by a different (primordial) class loader for security reasons, and agent's classes are not visible out-of-the-box. In order to have more insightful metrics, we need to instrument the core Java classes as well.

However, this caused problems with the ClassLoader, because the DynamoDB dependencies (the Amazon SDK for Java) were mixed in with the Javassist package.

In order to fix this while keeping all functionality, we adopted the following method:

- As stated before on the *Javassist* section, the class *AmazonDynamoDBConnector* was moved to the *webserver* package.
- Every 5 requests, the instance (using the Javassist tool) will output its data to a file (*/tmp/dynamodb*).
- The webserver, using a scheduled task, retrieves that data from the file every *DELAY_DYNAMODB* (default: 10) seconds and sends it to DynamoDB.

This is **not** an optimal solution, but it was the quickest way to keep both the metrics and DynamoDB interaction while allowing the Image Compression workload to run without error.

3 Things to improve

The program works as expected, however there are some things that could be done better if there was more time. Here we write some of them:

- The lambdas don't have instrumentation. This was due to lack of time, but an high-level overview of the feature would be as follows:

  The endpoints would receive a new parameter, that warned the instances that that request was to sent right away.

  The lambdas would get the data from DynamoDB and then update it, which would incur in some latency but with the tradeoff of having more data to work with. This is because lambdas are used in a one-off fashion, which means they cannot gather multiple metrics to batch in an operation to the database.

- In the *AutoScaler*, we could employ a better way of deleting instances, instead of just marking them for termination and waiting for the amount of seconds defined in a variable. We could make the instance return something that would indicate that it is ready to be deleted, and only then delete it.
- Also in the *AutoScaler*, we could scale more than one machine at a time. This could be done in a scale coresponding to a percentage of the number of instances already up (20%), for example, but due to a lack of time we did not implement this.
- The load balancing algorithm could be improved, to be more complex and aware of various metrics, such as the real instructions ran on each instance. This would lead to a more efficient system, but it would require more modifications to the *webserver* package.

4 Conclusion

This was a really interesting project to do, and allowed us to learn more about how these systems operate, and how hard it is to create a system that is reliable, scalable and efficient, without the end user being aware of this complexity.