CCV Report

Alexandru Pena - 98742; Bruno Freitas - 98678; José Oliveira - 87678

Instituto Superior Técnico

Introduction

The goal of the CCV project is to design and implement a simulated radar scanning service hosted in an elastic cluster of web servers provided by Amazon Web Services. The service will execute a computationally-intensive task on demand with user chosen parameters and input data. The system should be able to scale and maintain good performance and efficiency, keeping in mind the cost balance. The final architecture of the project is composed by the following services: the web servers, load balancer (LB), auto-scaler (AS) and metrics storage system.

Load Balancing Strategy

For the LB to be able to cost efficiently maintain the system working, a strategy was defined in order to make those decisions.
The strategy is based on attributing to a set of metrics a certain cost and depending on that cost the LB can take 3 different actions depending on the current state of the system:

- 1º One of the web servers has enough resources to process the request: In this case the LB will pass the request to that server.
- 2º No Web server has free resources but one of the web servers is about to complete an operation and free the necessary resources: This scenario happens when for example an Web Server is operating at a, or slightly above defined threshold capacity (e.g. 90% CPU) but the instance was queried earlier by the LB, and the LB determined that one of the requests was almost done processing. If this is detected the LB can decide that its better to send the request to the (almost full) instance than going through the overhead of creating another instance, as the instance could free faster the resources.
- 3º No web server is available and it will still take time for any request to complete: This scenario is similar to the above but with the difference that no request is going to complete in a reasonable time. When this happens the only choice the LB has is between waiting for some web server to free the resources or creating another instance. If the request received has a low-medium cost that does not exceed instance capacity, the request will be put in sleep for some time. If that time is exceeded and still no VM is available then the load balancer will check how many instances are currently serving requests, if the maximum defined fleet of instances is already achieved, then the load balancer knows that the system is overwhelmed and can't do anything else except sending the request to the server with lowest capacity (but still past the defined threshold). If the maximum fleet size is not yet achieved then a new instance is created and the request distributed to this new instance. High cost requests that surpass instance capacity will always create a new instance if its possible, or overwhelm inevitably the instance with highest free capacity at the moment.

Metrics and Costs

The core of the strategy explained above is in the capacity of the LB being able to know the cost of execution of a certain request. To obtain the cost of a request the LB will query the metrics storage system. The metric storage system will return the cost as a long value (how this long value is determined it will be explained in this section).
For the LB to select which cost of the MSS to return it will have to compare the arguments of the request with previously executed requests and gathered metrics. The most similar cost will be returned, how this cost is selected will be explained in the implementation section, but one important factor to keep in mind now is that some parameters of the request are more critical than others, specifically VIEWPORT, which in the case that even if all other parameters of the request are identical but the VIEWPORT different, then the cost will have a correction applied to it, so it can reflect the closer to reality value.
Consider the following scenario where the LB receives 2 GRID SCAN requests:

- 1º SIMPLE-VORONOI-2048-2048-1 with viewport 256x256.

  ```
  Basic Blocks: 64 530 860
  Instructions executed: 190 550 027
  Branches checked: 28 138 901
  New Objects: 0
  Memory Reads: 0
  Memory Writes: 0
  ```

- 2º SIMPLE-VORONOI-2048-2048-1 with viewport 1024x1024.

  ```
  Basic Blocks: 1 807 842 099
  Instructions executed: 912 314 687
  Branches checked: 754 970 998
  New Objects: 0
  Memory Reads: 0
  Memory Writes: 0
  ```

As it can be observed the number of executed instructions is very different. The larger the viewport the more executed instruction it will have, therefore the appropriate cost must be corrected. (How this value is corrected will be explained in the Cost correction sub section)
In the initial stage the MSS will be filled with manually gathered metrics to be able to handle the initial decisions.

Metrics. To be able to maintain efficiency, the group decided that applying instrumentation to the instruction granularity would be a bad idea as the instrumentation code would be executed for each instruction. Therefore all our metrics are gathered at the basic block level (except the conditional instruction count but this is not executed on all instructions).
As per the metrics in specific the group decided that the most important factors are: executed instructions (counted per basic block), the number of checked branches, the creation of new objects, memory reads and memory writes.
The choice for the number of executed instructions is necessary as other counts like number of executed methods doesn't tell much about how work intensive an request is, but the number of executed instructions will tell us how much work its done. (The group understands that a bigger number of executed instructions doesn't necessarily mean that a request will be more CPU intensive, further details for the cost calculation in the cost subsection).
The second gathered metric is the number of conditional statements, as conditional statements means that the CPU pipeline may need to be flushed because of an incorrect branch prediction and instructions be calculated again.
The final metrics (new and memory accesses) are important for the PROGRESSIVE and GREEDY algorithms, as these algorithms have a lot less executed instructions than the GRID SCAN algorithm but are a lot more memory intensive. If we didn't count this metrics our judgement of the requests costs would be incorrect as we would be ignoring very heavy operations. For example, each NEW operation requires finding a free heap block of memory, allocation that block and then writing to that block and many other operations done when the heap block is freed after.

Cost. It would be an unfair comparison to consider the processing of a request slower by just comparing the number of executed instructions, an program with 1 000 000 instructions and 0 branches will probably run faster than an program with 500 000 instructions but 100 000 branches or 100 000 memory allocations.
For that reason, to calculate the cost we associate to each metric a different weight.
The weight of an executed instruction is "1", the weight of a checked conditional statement "2", the weight of memory operations: "15", "10" and "10" respectively for NEW memory allocations, memory reads and memory writes.
With those values we are able to calculate the cost of a request, for example for the following metrics:

```
Basic Blocks: 88 915
Instructions executed: 308 725
Branches checked: 21 558
New objects: 7000
Memory write: 10 000
Memory read: 20 000
```

The cost would be $308725 * 1 + 21558 * 2 + 7000 * 15 + 10000 * 10 + 20000 * 10 = 756841$.
Based on previous tests executed on the EC2 t2.micro AWS instance, the group also knows for some arguments and data sets how much an instance can take before becoming completely overwhelmed. With the study made by the group, it was possible to conclude that a VM has a capacity of 100 000 000 according to the metric rules defined in this paper. This value results from the observation and correlation of amount of different requests being processed while observing cloudwatch cpu usage statistics, further information about this observations in the analysis section.

Cost correction. As previously mentioned, choosing the most similar request can be very inaccurate if the viewport argument is different. For that the group based on different tests reached the following equations to correct the cost of a request:

- Function for greedy and progressive after the breakout point (512): $Y = 0.0001256 * X - 31.67$
- Function for grid after the breakout point (256): $Y = 0.0003815 * X$
- Function for grid for the values 64 - 256: $Y = 0.0003790 * X + 0.2667$

Where Y is the cost in percentage and X the viewport area (e.g. for 256x256 its 65 536). In this context the breakout point, is the upper limit before the cost of a request dramatically increases.
For example, let's assume that the maximum VM cost was 20 000 000 and that a 512*512 viewport request had a cost of 5 000 000 (which means the VM supports 4 requests of this type). If the LB receives a request for a greedy or progressive algorithm and the viewport is 768*768 which has a area of 589 824, it will get the closest request in the MSS, which in this example would be the one with viewport 512*512. First, would calculate the deviation of the cost per viewport size with the equation $Y = 0.0001256 * X - 31.67$, with X = 589 824, getting Y = 42.41%. Then, it would calculate the expected cost with $Cost = (5000000 * (1 + 0.4241)) = 7120500$. Where 5 000 000 is the cost of 512*512 viewport request.

Analysis

On this section we strive to explain how we achieved the conclusion that the capacity of an instance is "100 000 00" and why each metric (instruction, branch, memory read...) have their associated numeric cost.
The group started by sending the same request with equal parameters, except the algorithm used to understand how each

algorithm is processed and how expensive it is. The first conclusion from the gathered metrics was obvious, the GRID SCAN algorithm is a lot slower and expensive than GREEDY and PROGRESSIVE, no matter the parameters used, and the metrics reflected that, the number of instructions executed for the same request with different arguments is completly different. Take in cosideration the following results:

```
GRID_SCAN with viewport 512x512:
Number of basic blocks: 326085275
Number of executed instructions: 923811478
Total number of branches to check:132959000
New count: 0
Field load count: 0
Field store count: 0

cost = 1189729478
```

```
PROGRESSIVE_SCAN with viewport 512x512:
Number of basic blocks: 415980
Number of executed instructions: 1460365
Total number of branches to check:106475
New count: 17578
Field load count: 265424
Field store count: 0

cost = 4591225
```

We then took some of the requests and started to study the CPU usage for different quantities of the same request. For example we concluded that an EC2 Instance can only process up to 3 requests of GRID-SCAN algorithm with viewport 128 before reaching 100% CPU usage. Then we repeated this test multiple times with different instances to actually decide if this behavior was repeatable.
We proceeded by testing different requests with different algorithms at the same time and observe if the behavior was as expected, by that meant the CPU usage would be a close value to what the sum of the individual CPU usages were. And in fact that behavior was observable.
Finally we tried to correlate the different requests to obtain a similar "base ground" of capacity. As previously said 3 requests of GRID SCAN with viewport 128x128 would get really close to 100% CPU. And 100 requests with the GREEDY algorithm with viewport 256x256 would also get close to 100%. Given this information we gave the weight 15,10 and 10 to the respective memory operations of NEW, Memory Read and Memory Write in order for the costs to approximate both the capacity of 100 000 000.
This way we could find a "base" instance capacity.

Auto Scaler

The auto-scaler is responsible for the following operations: synchronizing instances, terminating and adding new instances according to system status, sending health checks to the instances, sending queries to obtain information on requests being processed and observing CPU Usage with cloud watch.

Increasing and decresing fleet size. Fleet in this context refers to the group of EC2 instances running the web server. Before actually adding or removing instances the auto-scaler first gathers information on the instances he has knowledge of, this information reports how many instances are overwhelmed, underwhelmed or with normal capacity. With this information, depending on the current fleet size different actions can be taken.
If the current fleet size is at the minimum fleet capacity defined then the auto scaler checks if the number of instances above threshold minus the total fleet size, is below the minimum free instances defined that we want, if it's below then an increase fleet command is issued. The minimum-fleet-capacity is different from minimum-free-instances, one defines the minimum instances that the system tolerates (2 in our case), if less than that value exists then more instances will be created and added. The other value defines the desired number of "free instances" to tolerate burst of requests (also value 2). If the minimum free instances was 1, then every time a request that exceeded an instance capacity was distributed another instance would be created, to keep with the desired minimum to process other requests, which would slow down the system as new requests would have to wait for the startup of the fresh instance before being able to be distributed and processed. In case the fleet instances are below threshold defined activity, nothing happens as the fleet size is at the desired minimum capacity.
If the fleet size is above minimum fleet capacity then it checks if it can terminate or add more instances according to the number of overwhelmed or underwhelmed instances, always without exceeding the maximum fleet capacity defined (6).
In any of the cases above there is always the possibility of no action being done if the system status seems to be normal.
This process is done every 5 seconds.

Health check. Every 30 seconds the auto-scaler will send an HTTP GET request to the endpoint "/health" of the webserver. This endpoint was added as it did not exist. This endpoint only answers with "I am alive". If the message "I am alive" is received, then the auto-scaler knows everything is okay with the web server. If no message is received then the auto-scaler increases an health check failed attempts counter, when this counter reaches the value 3 (three failed health check attempts), then the auto-scaler considers this instance as dead and removes it from the fleet. The next round, when the auto-scaler wakes up again, it will add or not a new instance if needed. If the health check failed for example two times but the third succeeded, the failed attempts counter is restarted.

Query metrics of requests being processed. The second step of the load balancer strategy as explained in the LB section, consists of checking if any VM that has not enough resources to process a request, is about to complete some heavy request that will release the necessary resources, and it will avoid having to go through the penalty time of waiting the startup of a new instance. The process of checking if the request is about to complete consists of the load balancer checking the status of all requests associated with an instance and accumulating the freed capacity of all finishing requests. To get those values the auto-scaler will send every 5 seconds an HTTP GET request to the endpoint "/metrics" of the web server. How this operation works, and how the web server knows which request to read the executed metrics from, will be explained further in the implementation section.

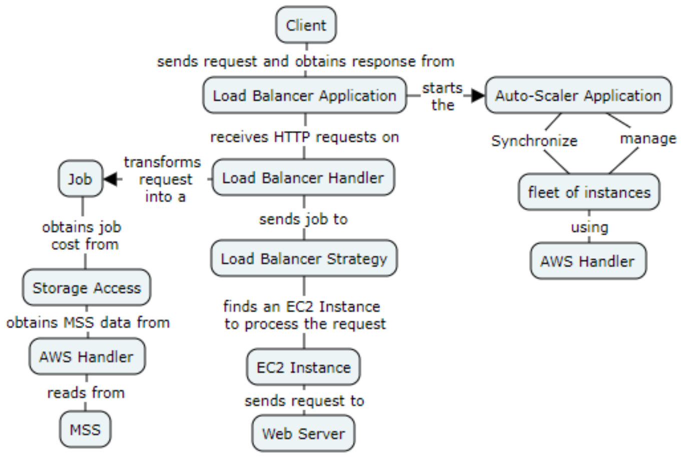
For a job to be considered almost complete it must reach 90% of the expected cost, for example if the expect cost of processing a request is "100 000" it must reach "90 000".

Synchronizing Instances. Sometimes the local status of the load balancer can become de-synchronized with the actual status of AWS regarding the existent instances. One particular case that this always happens, is when the load balancer is forcefully terminated. When is started again it won't have any information of the past instances that still exist on AWS. To solve this problem every 5 minutes (or on start-up), the auto-scaler through the AWS SDK, will send a request to obtain all running instances and then it will use that information to compare it with the local status. If some instance is not found locally but its found running remotely, it will be added back to the system and reuse it. This is specially helpful in the situation of the forced stop mentioned before, the load balancer won't have to recreate the instances every time the load balancer is terminated.

Cloudwatch CPU Usage. The LB strategy heavily relies on the cost of the request being correctly calculated for the instances to keep functioning normally. Though as seen previously because of many factors this decisions won't always be correct and actually negatively impact the performance of the instances. To solve this issue the auto-scaler, every 180 seconds will use the AWS SDK to send a cloud watch request and obtain the CPU utilization, on average, of the fleet instances. If the average CPU utilized by the instances constantly is above the CPU utilization threshold of 85%, then a cost correction variable will be increased by a factor of 50% with a base initial value of "1.0". For example, the normal cost of a request would be calculated as following $Cost = expected - cost * 1.0(basevalue)$, but if the auto-scaler decided that the CPU usage was constantly above expected and increased the base value by 50%, then it's new value would be "1.5" and the cost from on would be calculated with $Cost = expected - cost * 1.5$, in an attempt to lower the CPU usage. The opposite would also happen (lower the value) if the CPU usage was not being exceeded, with a minimum of base value "1.0".

Implementation

The implementation of the load balancer application consists of a multi-threaded web server that receives HTTP requests on the endpoint "/scan". The application on start-up creates a new thread to handle the tasks of the auto-scaler process. For the communication with AWS we use a maven dependency to import the AWS SDK.



The application follows an object oriented pattern and maps all requests obtained to an object of class Job. A job consists of the requestId of the request which is a UUID generated on object construction, an expected cost that the request will require of a web server to be processed, it's arguments and the executed metrics which are updated every 5 seconds by the auto-scaler thread. The cost of the request is obtained from the class Storage Access which keeps in cache the records read from the MSS. The cache is updated lazily when a new request comes if 5 seconds passed since the last update.

This job object is then passed to an object of the class EC2Instance which represents an AWS instance running. The EC2Instance object keeps track of the jobs its processing, its ip, id, failed health checks and current capacity (sum of all jobs expected cost). The objects of this class send the job to be processed to the actual web server, obtain the response and send it back. EC2Instances also insert an additional url parameter "requestId".

The parameter requestId, is used by the web server to create an association between the thread that is processing the request (on the web server) and the actual request. This is necessary for the auto-scaler to be able to query the executed metrics of the requests. Also, not all requests metrics are queried but only requests that take more than 5% of the capacity of the instance, as sending an metrics request for all jobs would generate a lot of traffic.

Improvements and Optimizations

Unfortunately a couple details couldn't be fine tuned before the final delivery. We will use this section to describe some performance issues that the load balancer suffers from, and some not that good algorithm decisions.

Concurrency. One of the biggest sources of inefficiencies of the load balancing application, is the used locking mechanisms on certain critical operations, specifically the ones who involve the data structures who hold the EC2Instances. When a new job appears, an instance must first be selected, and for that on worst case scenario the instances may have to be iterated multiple times and then the job added. For that reason the access to the instances can't be concurrent, which means that the load balancing algorithm is synchronized, only one

job can search for an instance at time. This locking brings another problem, sometimes jobs that require a lot of capacity can't be instantly processed and will have to wait a bit, worst case scenario three seconds. Which means that smaller requests will remain blocked when they could very rapidly be distributed and processed.
A solution for this problem could be a priority queue, instead of having big jobs block the distribution they would be added to an priority queue and every time a new job appears, it would first check if the priority queue is empty and address waiting jobs if any. Regarding the locking of the whole distribution algorithm, it may be unnecessary if we lock each EC2Instance entry individually.
Another concurrency problem is again regarding the instances list, when the auto-scaler thread wakes up to do its tasks, it will compete for access to the instances with other requests. This can bring problems as there is no "priority lock access" mechanism implemented and it can be delayed.

MSS. One of the problems that currently exists in the way that writes occur to the MSS, is that its not being checked for repeated writes of the same request, therefore the MSS will have repeated entries. This means that when the load balancer updates its cache of MSS entries for the selection of the most similar request, it will download all repeated entries which can slow down the application, the more the MSS grows with repeated entries.

Algorithms. To speed up the development of the application during the implementation of some algorithms a couple corners were cut to speed up the process. Specifically the algorithms of cost correction depending on the viewport and the algorithm to find the most similar request. Regarding the viewport correction, the resulting values from the mathematical equations are not the most accurate or even unstable, increasing too much the cost.
With the algorithm to find the most similar request, we implemented it in a way that every time that a parameter matches, the request gets one point. The request with most points in the end would be selected as the most similar request. Though this is not accurate as we disregard possible approximate viewports for example. We only check if its the exact value and not more similar to one value that another. This can return requests that are not the most similar.

Wrapping Up

The group concluded this project by having implemented a load-balancer with custom balancing rules and algorithms, an auto-scaler that dynamically increases or decreases the amount of web server instances serving client requests according to the status of the system locally, and remotely observed with Cloud Watch, a metrics storage system that gathers executed requests and associated costs derived by analyzing the behavior of the instances under different circumstances (algorithms, parameters, inputs...) and an edited web server that is able to process requests, instrument the executed code, save those metrics to the storage system and able to answer metrics queries regarding the execution status of jobs being processed.