CCV Checkpoint 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.

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 request was almost done processing (in terms of execution power). 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 queuing the request or creating another instance. If the request received has a low-medium cost it will not be created another instance just to process that request because it is not cost efficient. The LB will queue requests until a certain amount of cost, that is justified to create an web server instance is reached. (Or a queue time limit exceeds). If the request is medium-high cost a new instance will be created to handle that request.

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 MSS to select which cost 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 with a possible cost correction in case the identical request is not found.
Consider the following scenario where the LB receives 2 requests:

- 1º SIMPLE-VORONOI-2048-2048-1 with viewport 256x256 and algorithm GRID-PORT.

```
Basic Blocks: 64 530 860
Instructions executed: 190 550 027
Branches checked: 28 138 901
```

- 2º SIMPLE-VORONOI-2048-2048-1 with viewport 1024x1024 and algorithm GRID-PORT.

```
Basic Blocks: 1 807 842 099
Instructions executed: 912 314 687
Branches checked: 754 970 998
```

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 2 most important factors are executed instructions (counted per basic block) and the number of checked branches.
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.

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.
For that reason to calculate the cost we associate to each metric a different weight.
The weight of an executed instruction is "1" and the weight of a checked conditional statement "2".
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
```

The cost would be $308725*1+21558*2=351841$.
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 a maximum cost of 41 980 790. This value results from the arithmetic mean of the cost of each test performed. For each test an analysis of the metrics was made and with that result, the cost of that request was calculated. The maximum cost would be the cost of a request * the maximum number of requests supported by the VM until it used 100% of its CPU.
For example, the group knows that for a request using the GREEDY algorithm, with with a 256x256 viewport for the SIMPLE-VORONOI-512x512-1 the instance can handle up to 100 requests before getting close to 100% CPU intensity, therefore we can consider, ignoring thread pool capacity of the web server, context-switch, etc. if the request cost is 351 841, the instance processing capacity is $100*351841=35184100$.
Based on that calculated capacity and each request cost, the LB will be able to make it's decisions.

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.

Done to this point

For the initial checkpoint delivery the group managed to do the following:

- Create the instrumentation tool.
- Instrument in a multi-concurrent way for different threads different requests.
- Make the Web Server access the BIT tool code to save the metrics locally.
- Create the deploy code for an Instance, Load Balancer and Auto-Scaler using java AWS SDK.
- Define the strategy for the LB, cost calculation and cost correction.

To be done

For the final delivery the following must still be concluded:

- Deployment of the MSS.
- Make the Web Server store the metrics in the MSS.
- Implementation of the strategies described above for the Load Balancer + Auto-Scaler and MSS.