IST - MEIC - 2018/19
Cloud Computing & Virtualization
Group 14 - Final Report
HillClimbing@Cloud

93967	83448	83475
Diogo Almiro	Dorin Gujuman	Hugo Guerreiro

May 2019

1 Introduction

HillClimbing@Cloud causes long periods of computation just from a single request, therefore, to maintain this service running it is necessary to scale the system accordingly to its current load. We must also be able to deal with a continuous flow of requests as well as bursts of requests, all this while maintaining stability and fault tolerance.

Having in mind the costs of scaling a system up or down, the proposed system aims to deal with the above cited requirements.

2 Architecture

Each request to our system takes a lot of CPU power to process, doing all this in only one instance of a Web-server would not be feasible for the overall system performance. In order to mitigate this problem we divided the project in two main components, namely the `web-server` module and the `load-balancer` module.

2.1 Web-server

The web-server module contains a web-page application and implements the different algorithms needed to do hill climbing on different images. This code is then instrumented using the `BIT tool` which give us metrics about the execution of the program. After the instrumented code is executed, the main web server application will gather the generated metrics and send them to a DynamoDb table for storage.

2.2 Load-balancer

The load-balancer module itself is composed of many other sub-modules that when all brought together generate the system responsible for load-balancing, auto-scaling, fault tolerance and metrics gathering. All this components run in the same AWS EC2 instance, this way we can share all the metrics collected between the different modules and we also avoid consistency problems between the load-balancer and the auto-scaler. As aforementioned, this module is composed of many sub modules, namely:

2.2.1 Reverse proxy

The reverse proxy is the main entry point to our system. It provides an endpoint with an identical API to the web server instances. Each request made to this web server is then mapped to an instance indicated by the load-balancer. After the response from the instance the reverse proxy forwards it to the original client. In the case of any error the request is mapped to another instance, more on that in the section 9, Fault-Tolerance.

2.2.2 Load-balancer

The load-balancer is responsible for yielding an instance where to send a specific request (with a specific complexity), the algorithm used to actually perform this load balancing will be discussed

later on the report in section 7, Task Scheduling Algorithm.

2.2.3 Auto-scaler

The auto-scaler is responsible for creating or destroying worker instances in order to maintain good response times without allocating too much resources, the algorithm used to achieve this auto-scaling will be discussed in section 8, Auto-Scaling Algorithm.

2.2.4 Monitors

The monitors are tasks specific to keep *AWS* information about instances fairly updated without causing an overhead of *AWS Api* calls. They are overall responsible for monitoring the system state at any time. The different monitors available in our system will be discussed in section 6, Auto-Scaling Algorithm.

3 Metrics

The service was instrumented using the BIT tool in order to provide runtime metrics, these metrics are stored in various DynamoDB tables and used in the Load Balancer and Auto Scaler in order to estimate the complexity of future requests. These estimates permit us to manage our resources more efficiently.

The Solver class, BFS, DFS and ASTAR Strategy classes instrumented, this corresponds to the hot code present in the service.

In the checkpoint, we started by saving the number of instructions, basic blocks and methods calls. However, since we noticed that all these values had a linear relationship, we kept counting just the number of method calls for each algorithm (nothing of note could be inferred from instruction count nor basic block count that couldn't be done with method count only).

After running many tests, and counting the time it takes for the service to complete a request in both a instrumented and a non instrumented version of the service reveals that the performance overhead is negligible.

4 Data Structures

We created a singleton class `LocalStorage` that stores a map from a thread id to the method calls count. Before the worker starts the search algorithm, we reset the count for that thread. When the worker finish the metrics are saved in the DynamoDB table associated with the request parameters. We have a table for every combination of strategy and image present in the dataset containing the metric and the associated parameters. The usage of this data is explained in the next section.

5 Request Cost Estimation

One metric corresponds to a structure containing the parameter of the past request: x_s, y_s, x_0, x_1, y_0, y_1 and the number of method calls, $mCount$.

This implementation has two options to estimate the cost of a request.

5.1 Option 1

We search for past metrics in the same map with the same strategy and similar coordinates. To define *"similar"* can configure a threshold for how close a coordinate has to be for it to be considered similar. If many similar requests are found we return their average, if a singe one is found the estimate is exactly that metrics cost.
- In other words:
Query the table corresponding to the map and strategy for all metrics where:

$$(x_s - T_1, y_s - T_1) < (r_{xs}, r_{ys}) < (x_s + T_1, y_s + T_1),$$

$$(x_0 - T_2, y_0 - T_2) < (r_{x0}, r_{y0}) < (x_0 + T_2, y_0 + T_2),$$

$$(x_1 - T_2, y_1 - T_2) < (r_{x1}, r_{y1}) < (x_1 + T_2, y_1 + T_2)$$

With the results of the query, calculate estimate:

$$r_e = \frac{\sum_{i=0}^{n} mCount_i}{n}$$

$T \leftarrow$ Configurable threshold;
$x \leftarrow$ past metric;
$r \leftarrow$ current request;
$r_e \leftarrow$ current request estimate;
$mCount \leftarrow$ past metric method count;
$n \leftarrow$ number of found metrics in our threshold.

5.2 Option 2

However, in the case that no past metrics are found we can only assume the cost based on the area and starting point on a map of all past data. First, for the area estimation we do a linear regression with $x \rightarrow area$ and $y \rightarrow metric$ (method count) and infer based on the model. Second, we simply find the metric in the same map that has the closest starting point. Finally, we return the average between these two heuristics.

Note that the calculations for Option 2. are only done if Option 1. does not return any values. We prefer real values to estimating our own values.

6 System monitors

Our system has a set of different monitors that help it work correctly. These monitors are typically threads running their separate loop and that from time to time execute a certain task.

6.1 Auto-scaling monitor

This monitor is the most important monitor and is responsible for scaling up/down the system when it is needed. Periodically, it contacts the auto-scaler and sees if a scale up or down is necessary. If it is needed then he will trigger this scaling operation.

6.2 Liveliness monitor

The liveliness monitor is another operation critical component. From time to time he will query the AWS Api to check which instances are currently available (i.e running). After obtaining this list, it compares with the current available instances in the auto-scaler and updates it accordingly.

6.3 Metrics monitor

The metrics monitor is responsible for obtaining the most recent updates to the dynamo db tables. In order to not being constantly querying all the tables, at each time step it will query only one table (obtaining all the new values) and then wait for a certain amount of time before querying the next one.

6.4 Progress monitor

In order to know how far away from finishing a certain request an instance is, this monitor queries it from time to time and obtains all the current metrics of the running processes. Then, it updates the corresponding cost estimation on the auto-scaler so that the overall cost decays over time.

7 Task Scheduling Algorithm (Load Balancing)

The Load Balancer is responsible for selecting which instance will handle a given client request based on the overall group load. It makes this decision based on how many requests are pending for each machine, and the expected cost calculated as described in the section before. We allow the same instance to process multiple requests if we deem it to be the best choice since the load balancer always picks the instance with the least estimated load for the next request. The decisions made by the load balancer are as follows:

Request is received in the reverse proxy $\rightarrow$ Load balancer is notified with the request parameter $\rightarrow$ expected cost is calculated $\rightarrow$ see what instance contains the least load $\rightarrow$ notify the reverse proxy where to dispatch the request and what cost it will ensue for the system.

The load balancer must also communicate with the Auto Scaler in two moments: First it needs to know the current load on the instances and also when the system intends to scale down it must keep at least one instance empty so in the future we may terminate it.

8 Auto-Scaling Algorithm

The auto-scaler is responsible for ensuring that the system is never too overloaded (affecting the overall system performance) or underloaded (wasting infrastructure resources).

In order to achieve this behaviour, the autoscaler implements heuristics that help decide

when to create or destroy instances. The auto-scaler also manages which requests are being currently executed in a certain instance.

8.1 Data structures

The auto-scaler needs to hold information about the current distribution of requests per instance and their respective current cost estimation. For each instance we store the current running requests and their respective cost estimation at each instance. From time to time, the progress monitor will update the cost estimations of each running request with information obtained from the instances.

8.2 Scaling

In order to know how/when to scale up/down, the auto-scaler implements a set of heuristics for each case. Independently of the situation, there is a monitor (**auto-scaling monitor**) that is periodically asking the auto-scaler if a scale down/up should happen. In case a scaling operation is needed, the monitor will trigger it.

In order to scale, we need to first decide if we should or not scale. To do that, we implemented an heuristic based on the average cost estimation of our cluster of web servers and the number of current executing requests. When the computed averages passes a certain threshold, we decide that we should scale. For deciding if the system should scale or not we also take into account if we are already scaling up/down, in which case we might decide that it isn't necessary for the system shouldn't scale any further. We also take into consideration the number of instances queued to create and/or destroy.

After the auto-scaler decided that the system should scale up/down, the monitor will trigger the scale up/down operation. When executing this operations, the auto-scaler needs to choose a number of instances to create/destroy. We decided that the auto-scaler should gradually create/destroy one instance until it sees the system as stable. We made this design choice in order to avoid situations where we would have periodic bursts of requests and the system would be constantly scaling up/down without actually making use of the allocated resources.

When scaling down, we make sure that the instance being deleted is not currently running any requests.

9 Fault-Tolerance

We always ensure that an instance worker is not killed if he has any request pending. However, if for some reason the instance dies, the request is reissued to another available machine, and eventualy it's response dispatched to the requester.

To ensure a faster response time, the system it gets the information of the liveliness monitor and sends the requests to other alive worker. This avoids that the class hangs up until a HTTP Timeout Exception is thrown as it would be harder to set a good timeout for any request due to the inherent time it may take just to run one request. Internally it creates two tasks one to make the request to a worker instance (given by the load-balancer) and another to ensure that the worker is alive (using the liveness monitor information).

10 Evaluation

We rely heavily on the assumption that the request cost will correspond to the metric used, witch may not be the case in general. If, for example, the task the service was trying to accomplish a memory-bound task instead of a CPU-bound one it may be the case that method-calls are a very bad metric to estimate the complexity of it (it may take the same number of methods to load a 1MB file vs a 1GB file, even though the load is orders of magnitude greater). More research is needed to find a generic metric that fits all cases.

We noticed that the micro instances we are using for this project are also burst-optimised, meaning that if running at high load for too long a it becomes throttled and go from 100% CPU usage to 20%. For this kind of service it's the opposite of what we desire, especially since any request, no matter how small it is will spike the CPU to its max and chaining multiple requests even sequentially one after the other means the machine will eventually get throttled.

For the load balancer, sending the request to the least used instance is good, however a better scheme could be arranged. For instance, there

are certain requests that are so large that you might as well dedicate an entire instance to running it and have other instances run fast, and easy to respond requests.

In the end, despite the challenges this solution is good enough to provide a reliable service that is possible to adjust by tuning the constants present on the load-balancer and auto-scaler.

11 Conclusion

Amazon provides a auto-scaler and load-balancer that try to fit the general use case. However for certain cases, custom built solution provide much better performance, both in response time and throughput then the ready-made options. By exploiting instrumentation metrics we are able to get insight and more opportunities for monoitoring, controlling and optimizing our problem.