Cloud Computing and Virtualization Report

Henrique Almeida 84725 Tiago Luiz 93976 Tomás Oliveira 84773

May 21, 2019

1 System Architecture

The system herein proposed can be divided into five components, as shown in the Figure 1 below, there is the load balancer which receives the requests and redirects them to the instances, which are a component by themselves, the instrumentator, the database, where the metrics are stored, and the auto scaling group, which manages the number of instances.

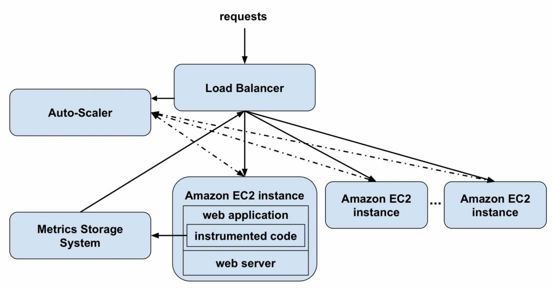


Figure 1: System architecture

As can be seen, there is a bidirectional communication between the load balancer and the instances, since there is a need to forward the tasks and its results between these two components in order to process the requests made by clients. The load balancer also has a reference to the auto scaling so that it can request an instance launch. Regarding the auto scaler, it is only used by the load balancer as a component that is capable of managing the instances' creation. The instances run a slightly changed WebServer class, that before a request starts to be attended sends some parameters to the instrumentator (more about this later) and after every request is completed, sends an order to the instrumentator to save the request's metrics on the database.

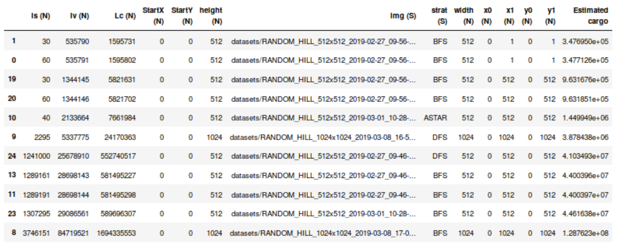
1.1 Instrumentator

For this phase of the project we have decided to revisit the instrumentator already done in the first phase and increase its efficiency by reducing the number of gathered metrics. We have opted to keep three metrics only, loads (LOADINSTRUCTION) and the number of virtual and special invokes (invokevirtual and invoke-special), since these metrics seem to be better correlated with the number of instructions executed, which we have obtained using BIT's Icount.

The Statistics Analyzer class also gained more responsibilities than it previously had. In this new version, it receives the load balancer's address, a double representing the estimated cargo of the request and the identifier of the request, these last two transmitted by the load balancer to the Web Server, which then reroutes them to the Statistics Analyzer. During the metric's gathering, the instrumentator is constantly evaluating the state of the request according to the formula presented at Equation 1, if we get a value bigger than 90% of the estimated cargo, an alert with the job identifier is sent to the load balancer (using the provided address in the request), in order to inform it that the request is almost attended and an answer will be given shortly.

$$\begin{aligned} LOADINSTRUCTION * 0.05 \\ + invokevirtual * 0.50 + invokespecial * 0.45 \end{aligned} \quad (1)$$

In this formula we weight the contribution of each of the three metrics being calculated. Although the number of loads seems to have a small contribute in the formula, the low value for its weight is explained by the fact that its order of magnitude is between 10 to 100 times bigger than the remaining metrics.



	Is (N)	Iv (N)	Lc (N)	StartX (N)	StartY (N)	height (N)	Img (S)	strat (S)	width (N)	x0 (N)	x1 (N)	y0 (N)	y1 (N)	Estimated cargo
1	30	535790	1595731	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-56-...	BFS	512	0	1	0	1	3.476950e+05
0	60	535791	1595802	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-56-...	BFS	512	0	1	0	1	3.477126e+05
19	30	1344145	5821631	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-56-...	BFS	512	0	512	0	512	9.631676e+05
20	60	1344146	5821702	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-56-...	BFS	512	0	512	0	512	9.631851e+05
10	40	2133664	7661984	0	0	512	datasets/RANDOM_HILL_512x512_2019-03-01_10-28-...	ASTAR	512	0	512	0	512	1.449949e+06
9	2295	5337775	24170363	0	0	1024	datasets/RANDOM_HILL_1024x1024_2019-03-08_16-5...	DFS	1024	0	1024	0	1024	3.878438e+06
24	1241000	25678910	552740517	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-46-...	DFS	512	0	512	0	512	4.103493e+07
13	1289161	28698143	581495227	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-46-...	BFS	512	0	512	0	512	4.400396e+07
11	1289191	28698144	581495298	0	0	512	datasets/RANDOM_HILL_512x512_2019-02-27_09-46-...	BFS	512	0	512	0	512	4.400397e+07
23	1307295	29086561	589696307	0	0	512	datasets/RANDOM_HILL_512x512_2019-03-01_10-28-...	BFS	512	0	512	0	512	4.461638e+07
8	3746151	84719521	1694335553	0	0	1024	datasets/RANDOM_HILL_1024x1024_2019-03-08_17-0...	BFS	1024	0	1024	0	1024	1.287623e+08

Figure 2: Example metrics

As we can see in Figure 2, the estimated cargo (calculated using the formula in Equation 1) augments with the growing complexity of the request reaffirming the relation between the two.

These instrumentation metrics are first stored in a Metrics type object (attributes of type long) where each thread has a different Metrics object. Then the metrics are written (asynchronously) to a DynamoDB table and are used for cargo estimation purposes.

1.2 Load Balancing

For this delivery, we have implemented the load balancer from scratch since we were previously using amazon's load balancer. The main objective of the load balancer is to redirect requests to instances running Web Servers according to the estimated cost of the request and the instance's workload. While attributing work to each instance the load balancer is also responsible for obtaining an estimated cargo for the job, if possible, using for that the data that is stored on Dynamo DB. If it is not possible to obtain an estimation of the request's cargo, then a default value is chosen.

When receiving a request, the load balancer will check the workload of every instance and select the one which has the lowest cargo and still space available in its queue for new jobs. If no instance is available the load balancer will call the auto-scaler and request an instance to process the request. This instance might be a new instance created for the purpose, an instance already created but that was still launching or an instance that, in the meantime, became available. When sending a request, if a connection refused exception is raised by the instance receiving it, the instance will be marked as unstable and the job rerouted to a healthy instance, the same thing happens if this exception is thrown while a request is already being attended.

While its running, the load balancer will keep a list of all its instances and will check its health by pinging each machine in the "/healthcheck" path every 30 seconds. If it fails to ping one of the instances, it sets the instance to unstable status avoiding that new requests are assigned to it. Although the instance is considered to be unstable, the load balancer will try to reach the instance for four more times and will also keep waiting for responses for the already sent requests. If it cannot establish connection after all the tries, the machine is terminated and the request is rerouted to an available healthy instance. During the health check process, if the load balancer finds that an instance has not attended any request for the last 2 minutes it will terminate it, in order to avoid wasting resources when there is a small amount of requests to attend for the available computing power.

1.3 Auto Scaling

As previously mentioned, the auto scaling is used by the load balancer to request new instances when there aren't any available (i.e., with enough space in its queue and capable of holding the estimated cargo), to do so, it sends to the auto scaler the request's identifier that has triggered the requisition and its estimated cargo. The auto scaler however, instead of launching a new instance right away, it follows the Algorithm 1. Firstly it verifies if there's already an available instance, and if there aren't any, then it verifies if there is an instance already being launched, appending the work to that instance and waiting for 10 seconds for its launch. When it returns from the wait, if that instance is not available yet, then it removes the work from it and starts again from the beginning. Although, if there aren't any instances available nor launching, then the auto scaler requests to Amazon a new instance.

```
while(true) {
    availableInstance =
      getAvailableInstance()

    if(availableInstance is not null)
        return availableInstance

    availableInstance =
      getLaunchingInstance()

    if(availableInstance is not null) {
        appendWorkToInstance()
        // Wait 10s for instance
        // to be launched
        autoScalerLock.wait(10 seconds)
        if (availableInstance is RUNNING)
            return availableInstance
        //If not running, try again
        // and remove the appended cargo
        removeWorkFromInstance()
    } else {
        //Create new instance if none
        // is launching nor available
        availableInstance =
          launchNewInstance()
        if(availableInstance is RUNNING)
            return availableInstance
    }
}
```

Algorithm 1: Auto Scaler's Request New Instance Pseudo-Code

After requesting a new instance to Amazon, the auto scaler will verify every 15 seconds if that instance is already running. While doing this verification, it will also keep scanning the already running instances, verifying if any of them became available to attend the request on hold. If there is any, the request is rerouted to it, avoiding the need to wait for a new instance to be launched, and increasing the speed of the client's response. In case the auto scaler has to wait for the new machine to launch, as soon as it able to reach it, it will wait another 15 seconds so that the services are properly initiated and attribute the request to the newly created instance.

If a request has an estimated cargo which surpasses the maximum cargo that an instance can have (given by the constant *MAX_CARGO_IN_QUEUE*), a new instance will be launched to attend that request. Note that we couldn't define a maximum cargo per instance so big that this case wouldn't occur because it would allow an instance to accumulate heavy requests, which would affect the system's performance.

1.4 Fault-Tolerance

In the proposed implementation, we've made some assumptions, namely that after requesting Amazon the creation of a new instance, it will launch even if it takes some time. Other assumption is that when a request is redirected to an instance, that instance has 60 seconds to establish connection with the load balancer but unlimited time to answer the request. If in the mean time, for example, that instance is terminated, the connection is automatically closed throwing an exception being selected another instance to attend the request.

2 Conclusion

In the previous delivery, our implementation used seven metrics. In order to gather these, some valuable resources were being consumed, which in a large-scale CPU-intensive system is undesirable. Since some of those metrics were considered redundant, we chose to only keep only three so that we could improve the Solver's and overall system's performance. The presented solution tries to keep up with the Amazon's load balancer and auto scaler functionalities, however enabling both to use the gathered metrics to take better decisions. We've successfully implemented a health check that verifies if an instance is healthy and therefore capable of receiving new requests or unhealthy. An instance can be marked as unhealthy up to five times, at the fifth time it is terminated and its jobs are redirected to another instance. The load balancer distributes the requests among instances based also on their estimated cargo, attributing a request to the instance which has less cargo at that moment. If there aren't any available instances, it requests the auto scaler an instance. For its turn, the auto scaler will evaluate if it there is the need to launch a new instance by checking first if there are any available instances, secondly if there is any launching instances, and just then if none of these conditions are not fulfilled, it creates a new instance. While waiting for the instance's creation, if another instance becomes available, it reroutes the request to that instance. This decision as well as most of the decisions taken during this project's implementation aim to deliver the best user experience by improving the system's performance.

One possible improvement to the current implementation which would be impactful on the user's experience is the introduction of a cache. This would improve drastically the response time if we had a cache hit, but at the cost of increasing the load balancer's memory consumption.