Cloud Computing and Virtualization Report

Henrique Almeida 84725 | Tiago Luiz 93976 | Tomás Oliveira 84773

April 25, 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.

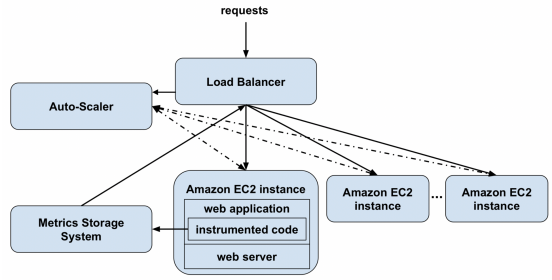

Figure 1: System architecture

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 group so that it can suggest to launch another instance (more about this further). Regarding the auto scaling group, it only communicates with the system's instances, constantly evaluating, along other things, the CPU usage, so that it can decide if it needs to launch another instance. The instances run a slightly changed WebServer class, that after every request sends an order to the instrumentator to save the request's metrics on the database.

1.1 Instrumentator

The delivered instrumentator is based on the BIT's sample *StatisticsTool*, where instructions' opcode and type are being used to decide which counters are incremented. The instrumentator registers seven metrics, the number of allocations (NEW), the number of arrays that were created (newarray), the number of all types of stores (STORE_INSTRUCTION) and loads (LOAD_INSTRUCTION), the number of virtual and special invokes (invokevirtual and invokespecial) as well as the maximum stack depth, as suggested on the project's statement. To collect the stack depth, it is used a auxiliary variable *level* that is incremented before a routine/method is called, being in that moment verified if it is bigger than the maximum currently registered, and decremented after a return of any type (areturn, lreturn, etc).

These metrics were chosen based on the group's knowledge regarding the functioning of an operating system and also on an ITHare article [1]. It is assumed that an instruction that uses more clock cycles and/or demands access to memory, potentially triggering a hard disk read/write, better reflects the computational cost of an operation. To send the metrics to the Amazon Dynamo DB, the Web Server was changed so that at the end of each request, calls a method at the Intrumentator that is responsible for the publication of the collected metrics.

1.2 Load Balancing

For this delivery, it is used the load balancer available at AWS, configured as in the labs. Since the metrics model is not done yet because there is the need to collect as much metrics from requests as possible, it was unnecessary to already implement a handmade load balancer, given that it couldn't predict properly a request cost.

The load balancer that we plan to implement is capable of estimating the cost of each request it receives, distributing them between the available instances, based on their CPU usage and on the type of requests that are already being processed or on the queue. Regarding the distribution of work between instances, one possibility is to reserve some instances to respond only to small requests (low computational cost) and others that accept also big ones (high computational cost), thus prioritizing the conclusion of small requests, another one is to distribute both small and big requests to any instance. Both strategies have their flaws. The first one makes the big requests take a bit longer, since there is no filtering between light and heavy request, which means that these requests can get stuck behind a long queue of fast and light requests, delaying its processing. Concerning the latter, the problem arises when all instances are occupied doing big requests so both small and big requests need to hold at the load balancer or at the queue of each instance, this strategy will also make the auto-scaling group more prone to create new instances.

To help the load balancer job, there is a plan to change the instrumentator to report back to the load balancer upon reaching 70% of the request estimated cost, so that the load balancer can redirect work for that instance again.

1.3 Auto Scaling Group

The auto scaling group currently creates up to four instances, launching a new one each time the group's average CPU usage surpasses 80% for a period of 5 minutes, deleting when it reaches 20%. On the handmade version, there is a plan to create a link between the load balancer and the auto scaling group, since the first knows how many instances there are, the work they are doing and its respective cost, it can predict the need for more instances. The load balancer will then be able to suggest to the auto scaling group to create an extra instance, and if a given threshold of suggestions is hit, then a new instance is created.

2 To Do Yet

Until this moment we have been using the load balancing and auto scaling provided by the Amazon Web Services as implemented in class. This implies that the system herein proposed does not yet take into account the available metrics to estimate the cost of a request, thus, redirecting that request to the most adequate instance. In the next and final delivery, both load balancing and auto scaling group will be explicitly implemented, to enable a better decision making based on the metrics that are being constantly collected.

References

[1] "No Bugs" Hare. Operation costs in cpu clock cycles. URL: http://www.ithare.com/infographics-operation-costs-in-cpu-clock-cycles.