RenderFarm

CNV group number: 5
Gonçalo Rodrigues (78958), Nuno Afonso (79035), Diogo Silva (79039)
Instituto Superior Técnico
goncalo.alfredo.rodrigues@gmail.com, nuno.c.afonso@ist.utl.pt, diogo.m.r.silva@ist.utl.pt

Abstract

We were initially given a centralized ray-tracer code and we had to develop a web server elastic cluster on Amazon Web Services (AWS) around it, which is called RenderFarm.

The system is composed of Web Servers running instrumented ray-tracer code, a Metrics Storage System (MSS) to persistently store state information and an instance responsible for hosting both the Load Balancer and Auto-Scaler components.

The instrumentation code produces little overhead in comparison with its initial version. Scaling decisions take into account application specific information, meaning that the system can quickly adapt to the current and expected loads. In case some Web Server fails, there is a fault tolerance mechanism to redistribute its load among the others.

The system's entry point is a single instance responsible for balancing the load and scaling the machines, which means it may become a bottleneck whenever the requests flow is high.

1. Introduction

The goal of this project is to develop a web server elastic cluster on AWS that renders 3D images on demand by executing a ray-tracing algorithm. This algorithm was originally developed for a centralized environment, which meant that its invocation should be adapted.

The system receives a stream of web requests. Each request is for the rendering of a rectangle (of specific variable size) that is part of a full, much larger scene with specified resolution. There are different scenes, all with different complexities.

As showed in Figure 1, requests arrive to the same machine. It will act as both Load Balancer and Auto-Scaler. The Load Balancer component is responsible for redirecting those requests across all the existing machines, so that they do not overwhelm specific machines. The Auto-Scaler part is going to adapt the number of machine to the current system needs.

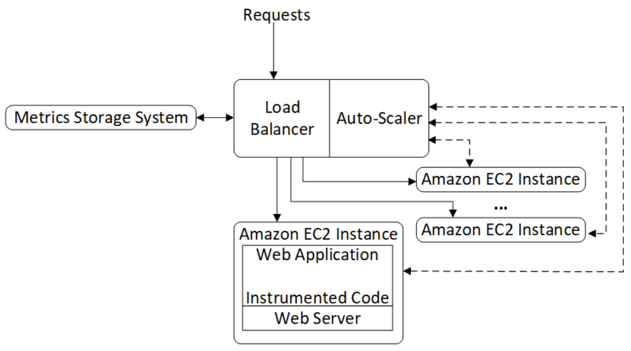

Figure 1. Architecture of the system.

The Load Balancer communicates with the MSS, in order to store a backup of the most updated information. It is used to predict the complexity of an incoming request and it is based on the recovered metrics from each Web Server.

Because all request go through the same machine, sometimes it can be a bottleneck of the system. When there is large increase of the number of requests, their offloading may be delayed, even if the Web Servers are not overloaded.

Furthermore, the system is protected against the crash of Web Servers. Every time a machine goes down, their load is redirected to the others. This guarantees that the request will be eventually completed, in spite of taking a little bit longer.

Regarding the instrumentation metrics, we found that the number of instructions is proportional to the number of *dot* method calls. This allowed us to have a lightweight instrumentation with a comfortable degree of confidence.

In the following sections, we will go deeper into each system component's characteristics, the way they are related to each other and the evaluation of the system.

2 Architecture

In this section we describe each component of our system. We start by explaining the instrumented Web Server, the Load Balancer, the Auto-Scaler and, finally, the MSS.

2.1 Instrumented Web Server

As mentioned before, the Load Balancer will redirect the requests to a Web Server based on each Web Server's current load.

In an attempt to predict this, some dynamic metrics from the instrumented ray-tracer classes will be gathered at runtime and sent back to the Load Balancer. Then, it will update its internal data structures and possibly save these to MSS. The metrics chosen were thought out with overhead-/accuracy trade-off concerns.

Because of overhead issues, we decided to only instrument methods, not looking to each basic block or instruction. After a few tests, we found out that the number of executions of *dot* method was nearly proportional to the total number of instructions. This relation can be seen in Figure 2.

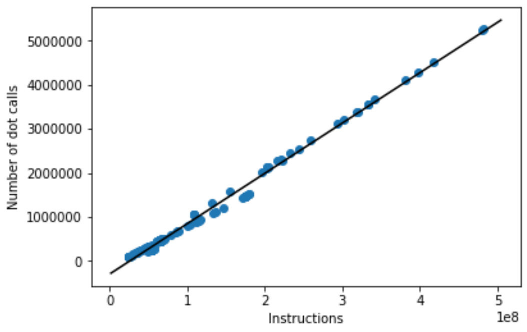

Figure 2. Relation between *dot* calls and number of executed instructions in 100x100 windows in different regions of different images.

To implement this in a parallel environment, we started with a HashMap indexed by the thread identifier responsible for the request. However, this led to a huge overhead on the total execution time, which was not acceptable. So, we changed to a very large array indexed by the thread identifier modulo the size of the array. To maintain correction, as soon as some thread starts running the ray-tracer, we lock its corresponding entry. It is only unlocked after finishing.

The probability of a thread waiting for a lock during long periods of time is negligible, if the array has a big enough size. The overheads associated with the instrumentation are described in Table 1.

	No Instrum.	Array	HashMap
Instr. (Millions)	3778	3977	3977
Time (s)	1.69	1.76	2.76
Overhead on instr.	0	0.052	0.052
Overhead on time	0	0.041	0.633

Table 1. Comparison between the number of instructions and execution time for the same request with no instrumentation, instrumented code using an array and using a HashMap. Times were taken as an average of 20 runs.

For a while, we were comparing our metric to the number of instructions, not taking into account the execution time as it is not very reliable. However, when we started implementing, we noticed that there was a big difference between the number of instructions and the execution time for different images, which we show in Figure 3.

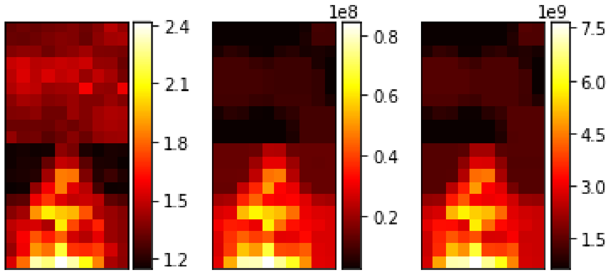

Figure 3. Computational work of test01 (top) and test04 (bottom) measured in execution time in seconds (left), total number of calls to the *dot* method (middle) and number of instructions executed (right) for every region in each image. Each image is 1200x1200 and each region 120x120.

This is probably due to some complex Java byte codes being much more costly than some other simpler ones and we did not instrumented any library outside the ray-tracer classes. We tried to find out which methods could be using these expensive byte codes more often, but we were unable to find a good candidate.

We noticed, however, that within each image, the relationship between our metric and the execution time was pretty decent. So, we just multiply the metric by some image-dependent constant calculated beforehand, allowing us to estimate the machine's load.

2.2 Load Balancer

In order to better predict the computational demands, the Load Balancer stores a square cost matrix nxn for each one of the input files. Each entry is the computation cost per pixel of a specific region with an associated precision. This matrix is kept in memory and saved to MSS every 15 updates.

When the user performs a request, it has to provide the region of the image she wants to compute. We can map this region into the correspondent cells of the prediction matrix, which we will call mapped cells. The precision of this request is said to be $\frac{1}{|mapped\ cells|}$. The cost associated with this request is measured by our instrumentation and divided by the number of pixels (we have noticed that the cost of the request is proportional to the resolution of the request, as seen in Figure 4).

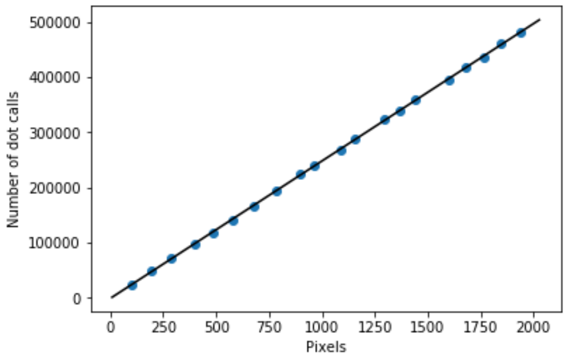

Figure 4. Relation between complexity of a request and its resolution. All data points correspond to the same region of the same image, but with varying window sizes.

The algorithm then picks the elements of the mapped cells with smaller precision than the metrics result. Then, it subtracts the sum of the costs of these more precise entries with the one from the request, splitting the difference between the remaining elements. The precision of these elements is set to the precision of the request. The corresponding Python code is in Listing 1.

The cost estimation is done by adding the values of the requested cells from the prediction matrix and multiplying it by the total number of pixels in the request. The corresponding Python code is in Listing 2.

Looking at Table 2, we concluded that the Load Balancer should have a 40x40 matrix. We did not consider bigger ones, because the complexity of the insertion algorithm increases with the size of the table.

When the Load Balancer needs to match a request to a Web Server, it simply chooses the one with most load up to a small threshold. If there are no Web Servers below that threshold, it chooses the one with the least load. This allows us to provide low latency when many machines are available.

```
def update(x1, y1, x2, y2, matrix, metric):
  insertLevel = (abs(x1-x2)+1)*
   (abs(y1-y2)+1)
  knownCost = 0
  nrLessPrecise = 0
  lessPrecise = []
  for x in range(x1, x2+1):
    for y in range(y1, y2+1):
      if(matrix[y][x].level < insertLevel):
        knownCost += matrix[y][x].cost
      else:
        nrLessPrecise += 1
        lessPrecise.append(matrix[y][x])

  costToSplit = metric - knownCost
  if costToSplit < 0:
    costToSplit = nrLessPrecise

  if nrLessPrecise > 0:
    eachCost = costToSplit/nrLessPrecise
    for e in lessPrecise:
      e.cost = eachCost
      e.level = insertLevel
```

Listing 1. Matrix update algorithm.

```
def estimateCost(x1, y1, x2, y2, matrix,
 proportion):
  estimate = 0
  for x in range(x1, x2+1):
    for y in range(y1, y2+1):
      estimate += matrix[y][x].cost
  return estimate * proportion
```

Listing 2. Cost estimation algorithm.

Then, it computes the cost of the request, adding it to the Web Server's load. Upon receiving the result, the load is removed. During this time period, it periodically asks each one of the Web Servers for an update and they respond with a mapping of their current requests to each corresponding current metric values. This allows us to continuously update each worker's load as such: $load(w) = \sum_{r\ in\ w} max\{0, estimatedCost(r) - currentMetric(r)\}$, where w is the worker and r are the requests being currently processed by it.

There is also a maximum threshold of load that each ma-

	test01	test02	test03	test04	test05
10x10	11025	14241	10138	126438	39447
20x20	12409	18144	9652	120077	42177
30x30	8297	14389	6042	85163	31097
40x40	7084	9893	5951	75636	26090
50x50	11512	17814	8496	117088	36549

Table 2. Average difference between the number of predicted and real *dot* method calls for each file with several matrix sizes. Image resolution of 100x100 and sample size of 20 continuous client requests with 20 different paths and 20 samples in each client step.

chine can take, not allowing more requests that surpass it. Instead, they will be put on hold until a machine with low load appears. These requests are called *pending requests* and their summed estimated cost is *pending load*.

When the number of Web Servers is reduced, the Load Balancer will detect it when trying to contact the corresponding machine (which it does periodically to update the load as described before). This checks have a five second period.

2.2.1 Fault Tolerance

If a request to a ray-tracer Web Server fails (usually by a socket exception), then the request will be re-fed to the Load Balancer, without notifying the user.

In case a machine is dead, the ping will not be answered in a timely manner. This results in all of its requests being treated as new ones (as explained in the previous paragraph).

2.3 Auto-Scaler

The Auto-Scaler has two concepts: *current workload* and *current workflow*. They are what drives both our upscale and our downscale policies.

The total workload of the system, W, is defined as the average of the load of all workers, including the unborn ones. The total workflow, F, is defined as the total load that came into the system per second in the last minute, also divided by the number of machines. Of course, the load that we are mentioning is estimated using the mechanisms described before. These values are updated every minute, using a decaying mechanism, as we can see in Equation 1 and Equation 2, respectively.

$$W_t = 0.2W_{t-1} + 0.8\frac{\sum_{w \in workers} load(w)}{|workers|} \tag{1}$$

$$F_t = 0.2F_{t-1} + 0.8\frac{\sum_{r \in latestRequests} estimatedCost(r)}{|workers|} \tag{2}$$

The upscaling and downscaling policies are only checked and executed once a minute.

2.3.1 Upscale Policy

To decide when to add a new machine to the system, we simply check if the current workload of the system, W, is almost reaching the maximum threshold of load. If it is, then we create one worker. Other possibility is that the threshold has been surpassed, leading to the creation of as many machines as necessary for the new instances not going over the maximum threshold.

2.3.2 Downscale Policy

To decide when to remove a machine, we check if the current workflow of the system, F, is below a certain threshold. If this happens five times in a row and the following workload can be distributed between the remaining workers without overloading them, then we mark the worker with the least load for termination. No requests will be forwarded to that machine and it will be terminated after its last request finishes.

2.4 Metrics Storage System (MSS)

The MSS is represented by AWS's DynamoDB.

To compress the stored information, the Load Balancer stores its prediction and precision matrices as strings. Each matrix's entries are separated with tokens, which led to the creation of dedicated functions for the type conversions.

The read and write capacities are set to one and five, respectively. The Load Balancer will only read once, after booting up its machine. However, the write frequency depends on the amount of requests that change the matrices. So, if a sequence of requests has increasingly higher precision, the updates will be much more common, resulting in higher needs. The chosen write capacity supports a very demanding workflow, leaving some leverage space for inconveniences.

3 Evaluation

To evaluate our system, we considered two scenarios. The first is putting it under a constant workload. The other is with sudden increases of the workload. Before starting, the system was warmed up with a set of requests, in order to calibrate the Load Balancer's matrices. This set's processing time was well below one minute.

We started our tests with only one machine and a flow of one request per second for the image test04 with arguments sc = wc = 400, sr = wr = 300 and coff = roff = 0. Then, we increased to two requests per second at time stamp $7335 + 1.49529e9$ and further to three requests per second at 7580 + $1.49529e9$. Each request takes about two seconds in a single machine, so one request per second cannot be maintained in a single machine, triggering our Auto-Scaler.

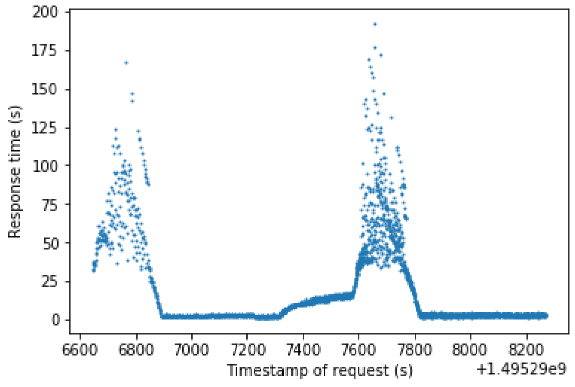

Figure 5. Evolution of latency over the first 30 minutes.

In Figure 5, we can see that the initial requests start getting on hold, because there are just too many of them. As the Auto-Scaler triggers, it creates one more instance, which relieves the workload and steadily decreases the response time. When we increase the load suddenly we can always see a period where the response time is high until one or more machines are deployed. After, the system stabilizes, matching the response and request times.

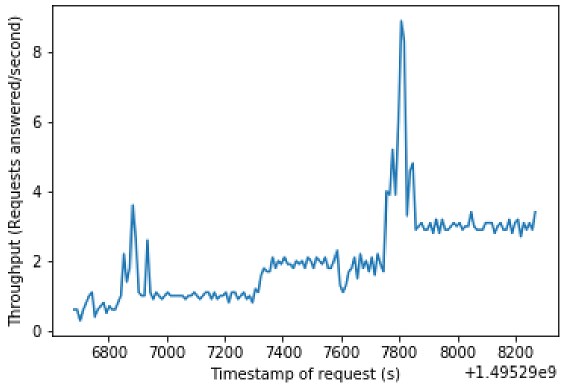

Figure 6. Evolution of throughput over the first 30 minutes.

We can see in Figure 6 that the system scales to different workloads. Specifically, as we increase the workload, the throughput increases to match it, even though it might take several seconds. When there are too few machines, the response time increases. But, soon after, the increase on the number of machines translates into the increase of the throughput.

4 Conclusion

RenderFarm is a web server elastic cluster on AWS that renders 3D images on demand by executing a ray-tracing algorithm. It has four main components: the ray-tracing Web Server, the Load Balancer, the Auto-Scaler and the MSS.

The Web Server is instrumented so that we can know the number of *dot* method calls a given request has. This number will be approximated to the total number of instructions.

Using the previously recovered metric, the Load Balancer will estimate the cost of a request by converting its cost matrix to the size of the request. Then, it adds the mapped cells and multiplies the sum by a image-dependent constant. To update its information, it does the matrix conversion and changes the cells that have higher precision. When choosing a worker, it tries to give the requests to the same machine until it reaches a certain threshold. After, the Load Balancer changes the target and applies the same principle. Sometimes, requests are put on hold until the current work is done. As a fault tolerance mechanism, it is periodically pinging the Web Servers and, in case something goes wrong, the Load Balancer redistributes the faulty machine requests.

The Auto-Scaler runs in the same instance as the Load Balancer and decides the number of machines a system should have. Its upscale policy takes into account the system's current workload. To reduce the number of workers, it looks to the system's current workflow.

For the MSS, we used AWS's DynamoDB. It stores the Load Balancer's data in a persistently manner. During the system lifetime, it will only be read once, when the Load Balancer starts. However, the number of writes is related with the number of updates in the Load Balancer's matrices. So, there may be times when they are frequent and others when they are none.

According to ours tests, the system can quickly adapt to load changes. After some time, both the throughput and latency stabilize, meaning that the system is balanced.