HillClimbing@Cloud

Cloud Computing and Virtualization

Project - 2018-19

MEIC - IST - ULisboa

João Cardoso, 81361
Instituto Superior Técnico, Lisboa
joao.lourenco.cardoso@tecnico.ulisboa.pt

Mário Cardoso, 83523
Instituto Superior Técnico, Lisboa
mario.j.c.cardoso@tecnico.ulisboa.pt

Luís Loureiro, 94070
Instituto Superior Técnico, Lisboa
luismaloureiro@tecnico.ulisboa.pt

1. Overview

The goal of this project is to design and develop an elastic cluster of web servers that can execute a simple e-science related function: to find the maximum value on simplified height-maps (maps)[1], representing elevations with colours (for appeal), on-demand, by executing a set of search/exploration algorithms.

The system receives a stream of web requests from users. Each request is for finding the maximum on a given map, providing the coordinates of the start position and a search rectangle within the height-map. In the end, it displays the height-map and the computed path (in grayscale) to reach the maximum, using hill-climbing[2].

2. Architecture

The implemented system is comprised of six main components: The Web Server, the Instance Controller, the Instances Manager, the Auto-Scaler, the Load Balancer and the Metrics Storage System (MSS). The *Web Server* is the component responsible for performing the escape path solving and returning the result. There will be a varying number of *web servers* receiving and handling the requests. Each Web Server has also been instrumented to be able to count the number of basic blocks used when handling the requests. An EC2 instance is represented by the Instance Controller (IC), which is not only able to send requests to the Amazon client for creation and deletion of a instance but also check the health of that instance. The Instances Manager (IM) is responsible for keeping track of the available and idle instances, the total load of the cluster and act as a mediator between the instance controllers and the load balancer and auto-scaler. To decide which of the available *web servers* should handle a HTTP request that is received by the system, there's the *Load Balancer (LB)*. This component is the only entry point of the system. The number of available *web servers* is controlled by the *Auto-Scaler (AS)*, that detects when the system is overloaded and starts new *web server* instances or shuts down idle *web server* instances when the load decreases. Storing the metrics associated with each request handled by a Web Server is the MSS's responsibility.

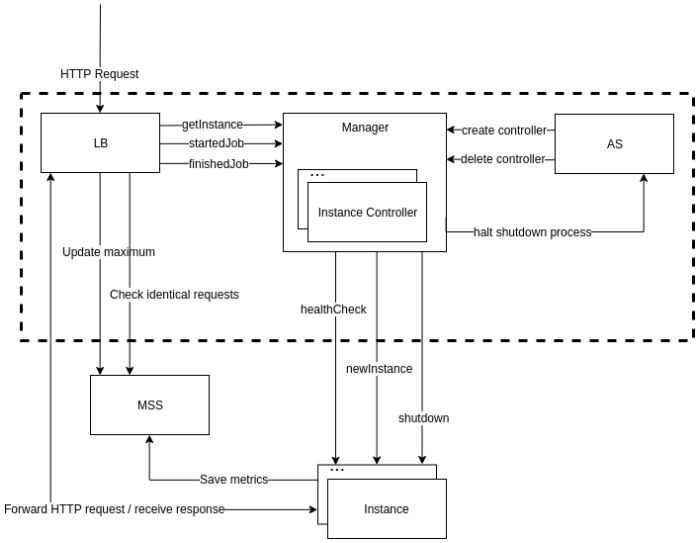

Figure 1- System Architecture

[1] https://en.wikipedia.org/wiki/Heightmap
[2] https://en.wikipedia.org/wiki/Hill climbing

2.1. Instrumentation

In order to extract performance metrics from the execution of the hill-climbing solving, this solution uses BIT, a Java instrumentation tool that allows building customized tools to instrument *Java Virtual Machine* (JVM) bytecodes before they are loaded by the VM. The BIT tool created for this solution is only instrumenting the *Solver* class, that is the main class involved in the requests processing. The metric we decided to extract was the number of basic blocks in the executed request, as it provided a good trade-off between overhead throughput introduced from bit and significant differences between requests. We make sure that, since the server is multi-threaded, each request and corresponding metrics are associated with a specific thread ID so that other threads execution don't interfere with BIT measurements.

2.2. Data Structures

The previously mentioned extracted metrics are associated with a specific request and stored to an AmazonDynamoDB table. This table stores a POJO class that represents the request (and all its parameters) with the corresponding metric. This POJO class utilizes a composite primary key constituted by the dataset used (partition key) and a unique request ID (sort key) attributed to each request by the Load Balancer and sent to the instances via the HTTP request URL. We chose this implementation because items with the same partition key are stored near each other, and since we often query for a specific dataset, this provides a welcome speed-up.

For optimization purposes, the Load Balancer stores a small map to serve as a cache. This cache stores previously executed requests associated with a specific dataset. In order to avoid storing a massive cache in memory, we only store the last 10 requests for each dataset. Whenever a new request arrives at the Load Balancer, the cache is analysed and, if the incoming request matches a request stored in the cache, the associated metric value is normalized and returned. This value is then assigned as the complexity of the incoming request. This approach avoids having to query the database for each incoming request.

2.3. Instance Controller

This class abstracts the ec2 instance and is responsible for communicating the manager requests to the actual ec2 instances.

These requests can include instance creation or deletion, marking the instance for shutdown or reactivating it and sending a health check to the instance.

When the manager requests that an instance is created, this class sends a request to the Amazon client for creation of an instance. It will then mark the instance as pending and create a timer which every 10 seconds checks whether the instance status on the Amazon client has become available. When that happens, the controller can obtain its IP address and mark the instance as available. With this information the manager is then informed that the instance has been created and is available to receive requests. Deletion of the instance is also requested by the manager and this class will create an Amazon client request to do so.

Health checks are periodically requested by the manager and the controller will send a request to the instance accordingly.

2.4. Instances Manager

This component acts as a middleman between the instances and the LB and AS. The IM is also responsible for keeping a list of all instances that are active, pending creation or marked for shutdown.

As the LB receives requests it queries the IM for the best instance. The instance manager will then sort the instances in descending order with regards to their available load. Afterwards, it will choose the first instance and check if it was marked for shutdown. If it was, it will try to find another instance that is unmarked and is able to handle the current request. In order to better distribute requests and speed up responses, the IM will analyse the current load of the best unmarked instance found and check if it's already processing requests. If affirmative, the first idle instance gets assigned to the new incoming request and the shutdown is halted. If there are no unmarked instances able to handle the request, the IM will unmark the previously chosen instance and notify the AS to stop the termination process. If there were no instances that were able to handle the current request, independently of their status, the manager will create a new instance. The IM will now notify the LB of the best instance that it should forward the request to and add that request to the instance in the corresponding Instance Controller.

With regards to the AS communication, the IM is responsible for executing the creation and termination of instances when the AS commands it to do so according to its algorithm. The IM must also be able to provide it with the list of idle instances and the cluster's total available load and mark the requested instances for shutdown.

The IM also manages the periodic health check on each instance. On instantiation, the IM creates a new thread with a timer that every 20 seconds sends a health check to each instance. This time interval between health

checks seemed, after some testing, the best value considering the trade-off between having to send an HTTP request for each instance versus not keeping the cluster status updated as often. The IM will then update a map that lists how many failed health checks each instance has. If an instance fails three health checks consecutively then it is considered unreachable and is dropped from the list of available instances in the IM. If the instance was executing requests, the threads responsible for those requests in the load balancer receive time-outs or HTTP error codes and the requests are sent again to different instances.

2.5. Auto Scaler:

The elasticity of the system will be maintained by the auto-scaler (AS). This component will thus be responsible for creating and shutting down instances as they become necessary or obsolete, respectively.

A few parameters need to be present in the auto-scaler for it to function properly. Namely, a maximum and minimum number of active instances, a minimum amount of computing power that the system must always have available and how often it will run, checking if scaling up or down is needed. Our maximum and minimum number of instances are 1 and 10, respectively and the minimum amount of computing corresponds to the minimum number of instances times the maximum complexity of a request (in our case it will be 1 * 10)

The auto-scaler will run every 60 seconds. When this happens, it will first check if there are at least the minimum number of instances available, which at the time present, is one. If there are less instances than the minimum, then the auto-scaler will create a new instance.

Provided that the minimum is met, then it will first check if scaling up is necessary. In order to accomplish this, the AS will query the instances manager for the cluster's available load. If the available load is smaller than the minimum load that the cluster must always have, then the AS will check if the maximum number of instances has been reached or if an instance is already being created and if not, scale up.

After this step, the AS will check if it did scale up. If it did, then the AS will wait for the next cycle, if not it will check if scaling down is necessary. It does this by querying the instances manager for the idle instances. The AS will then iterate through each and check whether they have already been marked for shutdown. If an instance has not been marked, then the AS will check if deleting it would mean having less instances than the minimum or would bring the cluster's available load below the minimum required. If both are false, then the AS will ask the manager to mark the instance for shutdown and start a timer for 30 seconds, after which it will ask the manager to terminate the instance. At any time during these 30 seconds, the marked instance can be reactivated by the manager, if it is thought to be the best instance to send a request. Should that happen, the AS will cancel the timer and, by extension, the instance's termination.

2.6. Load Balancer:

As discussed above, the load balancer selects appropriate workers to distribute the load. For this to happen, the load balancer must assign a complexity value to each request and based on this value, forward the request to a worker that can handle the complexity.

To estimate this complexity a sequence of steps must be done when the request first arrives at the load balancer. First, the load balancer checks if similar requests have already been recorded. To achieve this goal the load balancer queries the *AmazonDynamoDB* for previously stored requests and associated metrics and considers only the requests which have the same dataset as the incoming request. If this condition is verified, a filter is then applied on all the remaining parameters. First, the strategy of the stored request must match the strategy of the incoming request. For the remaining parameters, if the distance between each pair of corresponding parameters from the incoming request and stored request falls within a 10% interval of the maps size, then they are considered equivalent. In that case we use the metric associated with the stored request. If more than one match is found, then the average of the metrics is computed. While querying the *AmazonDynamoDB*, in the occasion that a previously stored request matches exactly with our incoming request, the stored metric value is returned right away.

This metric value is then normalized using the maximum value found thus far. This normalized value becomes the estimated complexity associated with the incoming request.

If no matches are found for the incoming request, we estimate a complexity according to the following formula:

$$\frac{\frac{Map\ area}{Largest\ Map\ Area} + \frac{dist(start, worst\ case)}{Largest\ map\ dist}}{2} \times n$$

The biggest factor that influences a request's complexity is the search space size. This search space size can be divided into two sub-problems, the overall space and where the search begins. We considered both these problems equally important, so we compare each with their respective worst-case scenarios and then compute the average. Another important factor that impacts a request's complexity is the search strategy used. In order

to maintain a black-box, flexible model, we decided to discard this information when estimating complexities.

The *Map Area* is calculated using the following formula:

$$(|x1 - x0|) \times (|y1 - y0|)$$

dist(start point, worst case) represents a generalization of the problem in order to be able to calculate the complexity without assumptions. The worst case represents the point farthest away from the given start point. The distance considered is the Euclidean distance.

The *Largest Map Area* is the largest area found within the previously executed and stored requests.

The *Largest Map Dist* is the Euclidean distance from the top-left point (0,0) to the bottom-right point for the largest map found.

n is the normalization factor, it forces the complexity to be within the interval [0, *n*]. We chose n = 10 since it proved to be a good trade-off between different requests

To update the largest values found this far (largest metric result and largest dataset size), each corresponding parameter in the incoming request is analysed. If the incoming request contains a larger dataset or a larger metric result the corresponding maximum variables are updated. The largest dataset is analysed as soon as the request arrives, the largest metric is analysed after the request completes its execution. We chose this approach instead of hand-coding maximum values because it allows our model to flexible and maintain a black-box approach. This approach has the drawback of initial requests having maximum complexity (since the maximum seen is the request itself) but as more requests are executed, and more requests are stored in the database, these values quickly converge to the real maximums.

After associating a normalized complexity with an incoming request, the Load Balancer will ask the Instance Manager for the best instance given that complexity (following the process described in the IM section). If the IM successfully returns an instance, the LB retrieves its address, forwards an HTTP GET request with the appropriate URL and waits for the response. After a response is obtained, the LB reads the response's bytes and displays the solution to the user. If the IM is unable to acquire an instance, the LB halts it's services for a small period (10 seconds) and after that period is over, it will request another instance from the IM. This procedure is repeated until the IM can successfully return an instance.

The Load Balancer has been developed to operate multi-threaded, so it can handle several incoming requests in parallel. Due to this reason, the previously mentioned timeout period due to not receiving an instance affects only the specific thread handling that request, so it does not stop the overall LB execution.

3. Fault Tolerance

The system handles fault tolerances by using periodic health checks, using timeouts when sending or receiving requests and responses and by checking the responses HTTP codes.

Periodically, the instance manager checks the health of every instance and if an instance is unable to respond to three consecutive health checks, then it is deleted from the IM and the LB re-forwards the requests of that instance to other instances.

The Load Balancer deals with faults by checking for exceptions and the incoming HTTP response code. If the Load Balancer is unable to connect to a given instance, an exception is thrown, and a different instance is selected to forward the request to. If the instance receives the request but is unable to complete it for whatever reason, the Load Balancer checks the response HTTP status code and, if the code does not belong to the success family of codes (2xx), the request is forwarded to a different instance.

4. Conclusion

This report presents a AWS-based cloud system that is capable of handling web requests to an elastic cluster of worker instances that solve hill-climbing functions. This system can decide the best worker instance to forward its requests as well as when should it create or remove an instance. It was a big challenge to put all these components communicating correctly and to consider the extracted metrics in the balancing decision. This is a final result, but we are sure it could be improved. Despite that, we are really satisfied with what we accomplished with this project and the knowledge we obtained with it.