MazeRunner-Cloud Report

Luís Miguel Carmona Murta Mendes, n°76229

Henrique Fernandes Alves, n°87891

Matheus da Silveira Mello, n°89147

Gonçalo Alexandre Torrão Garcia, n°90869

I. Introduction

This report describes the first implementation steps for the MazeRunner-Cloud system. The current implementation features multi-threading and instrumented web servers. It relies on the Load Balancing and Auto-Scaling infrastructures provided by AWS.

II. Architecture

The system is composed of 4 sections:

1. Web servers
2. Elastic Load Balancer
3. Auto-Scaler
4. Metrics Storage System.

In the following subsections a description of the operations performed will be given for each component.

1. Web Servers

The web servers receive web requests to solve find-path problems and return the result. They were developed using the Java 1.7 version and have multi-thread capability in order to receive multiple requests from clients. Each one of them runs on a rented AWS Elastic Compute Cloud (EC2) machine. The nodes are using code that was previously instrumented using BIT, a Java instrumentation tool. The number of branchs executed are the metrics selected to be gathered during the execution of all the classpath files of the package 'pt/ulisboa/tecnico/meic/cnv/mazerunner/maze/'.

2. Load Balancer

The Load Balancer is entry point of our system. It is configured to be use one availability zone and it is responsible for fowarding all requests to the machines available to process the maze. The parameters for inbound and outbound traffic are the described in Table 1

LB Protocol	HTTP
Load Balancer Port	80
Instance Protocol	HTTP
Instance Port	8000

Table 1: Traffic routing configuration

3. Auto-Scaler

During the fowarding of the requests a Thread is running a scalling algorithm on the same machine as the Load Balancer to measure the necessity to leverage the quantity of machines in order the reduce the burden of the system and at the same time have a efficient utilization of workers. The Load Balancer has a list of active webservers that can process the maze.

3.1 Health Checks

It is important to detect if workers some failure in order to keep a updated version of machines that can do the processing. For this purpose, at the beggining of each iteration of the auto-scaler algorithm a set of threads perform a periodical health check on each worker. The configuration for the health check requires specifying values for timeouts and ping frequency. These values can be seen in Table 2

Protocol	HTTP
Ping Port	8000
Ping Path	/isAlive.htlm
Response Timeout	10000
Ping Interval	90000
Retries	3

Table 2: Health Check configuration

4. Metric Storage System

Dynamo is our choice for the Metrics Storage System due to its strong integration with the other services used in this project. Our implementation features a single table which will be used to store performance metrics that will enrich our load balancing and auto-scaling algorithms.

Table 3 demonstrates the topology for the Metric Storage System where the following parameters are stored: the Maze configuration, the coordinates (x0, y0, x1, y1) identifying the entry and exit locations, the velocity that the runners achieve inside the maze in this particular run, the strategy that they use to navigate (BFS, DFS, A*) and the respective number of branches taken executed while solving the maze. The partition key is defined as a composite of the Maze name and a SHA-256 digest of each of the fields. This ensures that the primary key will be unique for each combination of parameters.

To access records on this table, scan and filter operations are used. These are not as efficient as a regular query, but the limitations of Dynamo's implementation prevent us from querying multiple secondary indexes.

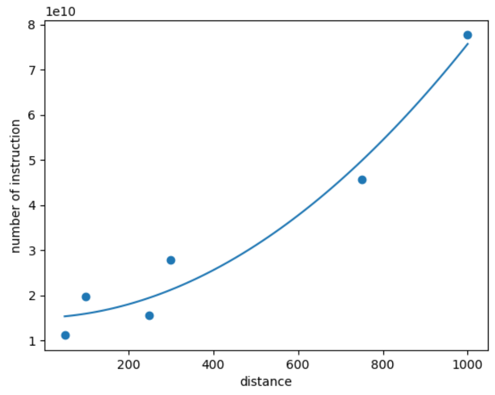

Figure 1: Number of instructions per different distances on the same maze

III. Web Server Instrumentation

To understand and differentiate between different request loads, the web server instances have to be instrumented. To minimize the impact of the instrumentation overhead, only the core java files were instrumented, such as the Maze.java and the solving strategies. The minimize the overhead even further, it was found that instrumenting only the number of branches taken yielded a very similar distribution as instrumenting every basic block and extracting from them the exact number of instructions, as seen in figures 1 and 2.

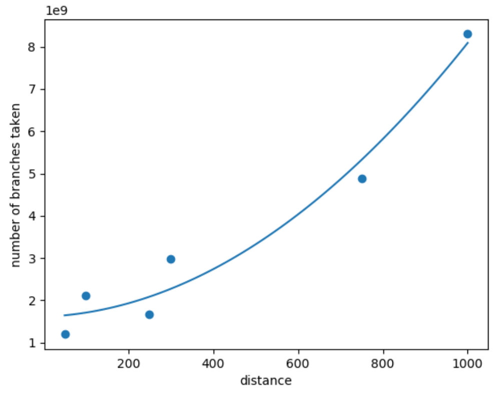

Figure 2: Number of branches taken per different distances on the same maze

IV. Load Balancing

The distribution of the requests that reach the load balancer is performed by minimizing the cost func-

Maze (Partition Key)	Hash (Sorting Key)	xStart	yStart	xEnd	yEnd	velocity	branches taken
String	String	Integer	Integer	Integer	Integer	Integer	String

Table 3: Topology of the Metrics Storage System table

tion

$$\Phi = \#i + \sum_i D + i_{new} \quad (1)$$

where $\#i$ is the number of instances on the AWS, $\sum_i D$ is the sum of perceived delays by the users for each instance and i_{new} is the cost associated with launching a new machine. The delay is obtained by first determining the ratio between the expected time for the task to complete, with other tasks running simultaneously, and its nominal value, and then awarding a weight based on the ratio value. The cost associated with launching a new machine is based on the difference between the nominal time of the new request and expected time for a machine to start(assumed to be 1min), and then assigning a higher weight if the difference is low.

When a new request comes in, the first iteration will be performed with the same number of instances, and calculating the delay for each instance with the new request in each, choosing the instance with the lowest increase in delay. On the second, one instance is added, as well as the new instance cost ($i + new$), and the delay is maintained as it was before the request came in plus the delay in the new instance, which is zero, as only the new request is into it. If the last cost is lower than the previous one, a new instance is launched and the new request goes into it. If it is higher, then the new request will go into the instance where the accumulated delay from all instances is lower.

1. Delay

The ratio between the expected running time (t_e) and the nominal time (t_n) is represented by t_e/t_n, with

$$t_e = (t_e^- - t_s) + t_s \cdot \#r \quad (2)$$

where t_e^- is the previously calculated expected time, equal to the nominal time in the beginning, t_s is the time shared with other requests running on the same machine machine, and $\#r$ the number of request on that instance. The shared time

$$t_s \leq min(t_e) \quad (3)$$

will always be considered ($t_s = min(t_e)$), and as such it will sometimes overestimate the actual shared time. Without real time information on the state of the ongoing requests it is the best guess without errors. Once a new request arrives or finishes, the shared and expected times are recalculated for every running task on that instance. This assumes that the CPU usage is divided equally between all tasks.

Time	BT	BT/second
1m48.049	1196359567	1.10723798184e7
3m12.217	2109393948	1.09740238793e7
2m31.600	1667327851	1.09982048219e7
4m29.645	2973148912	1.10261599955e7
12m29.070	8302106334	1.10832183026e7

Table 4: Statistics from different maze runs

T	Performance	D
Fast (0-5min)		
$1.5 \leq t_e/t_n$	Good	0
$1.5 < t_e/t_n \leq 2$	Acceptable	0.25
$2 < t_e/t_n \leq 3$	Tolerable	0.5
$t_e/t_n > 3$	Bad	2
Medium (5-30min)		
$1.5 \leq t_e/t_n$	Good	0
$1.5 < t_e/t_n \leq 2$	Acceptable	0.5
$t_e/t_n > 2$	Bad	2
Slow (30+min)		
$2 \leq t_e/t_n$	Good	0
$t_e/t_n > 2$	Bad	2

Table 5: Delay weight in function of t_e/t_n

Given that whether a request takes too much or too little time from the perspective of the user is measured in seconds, a conversion between Branches Taken(BT) and seconds is presented in table 4. The experiment was run on an Intel i5-6300U @ 2.4GHz, forced to use only one core, to replicate the response on the AWS machine. As it can be seen, it averages out to $\approx 1.10e7$ BT/s.

Because a delay being acceptable or bad and the boundaries between a fast or slow task are dependent on the perception of each individual, and after a small survey the responses were varied, table 5 shows an example of the possible configurations.

2. Nominal Time

The nominal running time for a given request is the branches taken for the most similar previously computed request divided by the BT/s constant.

The steps for finding the closest request are the following: first we query Dynamo for all requests associated with the requested maze. Then we find the starting point with the closest Euclidean distance to the request's starting point. Out of the filtered requests, we once again find the ones with the closest Euclidean distance, but for the end-point. The remaining requests are once again filtered, this time by strategy (If a request with the same strategy exists) and last by velocity.

In the end we will (guaranteedly) have a request with the closest Euclidean distance of starting and end-points, the closest velocity and possibly with the same strategy.

As the system is starting this may not be the closest estimation, but as time goes on and the number of requests increases, the estimations will become closer to the actual branches taken of the received requests.

3. Cost of launching a new instance

Due to the fact that launching a new instance on AWS is not immediate (based on our testing averaged at about 1min), then it must affect the cost function, especially for smaller and faster requests. The i_{new} will then be equal to

$$i_{new} = \begin{cases} 2 & \text{if } (t_n - t_i) \leq 5min \\ 0 & \text{if } (t_n - t_i) > 5min \end{cases} \quad (4)$$

with t_i being the time it take for an instance to initiate.

V. Auto-Scaling

When a worker ends all the requests that it was executing, a flag is set to true (if a new request comes in, it is again set to false). During the periodic run of the auto-scaler, it iterates through all workers and checks which ones are currently not under load. Until it reaches a threshold of the maximum units it can take down, so that it doesn't respond immediately to changes in requests, the workers are shutdown. There is also a minimum number of instances that are up at any time, and none will be shutdown if that value is reached.

VI. Data Structures

The Dynamo is used as our Metrics Storage System. It is used to hold the metrics gathered by the worker instances as already explained in subsection 4. In the Load Balancer we also save metrics in a class called Instance Manager, where the data is persistent and in internal storage. When the Load Balancer queries the Metrics Storage System for the cost of a request Section IV, if it returns a result, we also store the Rank in a file created by Dynamo so the next time the Load Balancer gets the same request, he doesn't need to query the Metrics Storage System for the cost of that request because he already has it in database so we save a big overhead.

We also have a system that set approximately the Rank value of a new request to save a big overhead system. Like mentioned before, we also store it persistently (in a file), this is because if, for some reason, the Load Balancer is shut down, when its turned on again, while initializing he reads the costs stored in this file and set up them to receive queries that will able to save our time, money and Web Server provider.