TÉCNICO LISBOA

Cloud Computing & Virtualization

2017/2018

58543 Dário José Da Silva Ornelas

78895 Luís Miguel Paiva Sampaio Santos

91062 Tiago Alexandre Serafim Monteiro

ABSTRACT

We describe the architecture of our implementation for the MazeRunner@Cloud application. Our report includes discussion of the employed instrumentation metrics, data layouts for the Metrics Storage System(MSS), algorithms for routing requests based on request load estimation from historical data, and how to dynamically adjust the number of workers based on up-to-date load information. Additionally, we discuss how to improve the overall reliability of the system with fault tolerance.

KEYWORDS

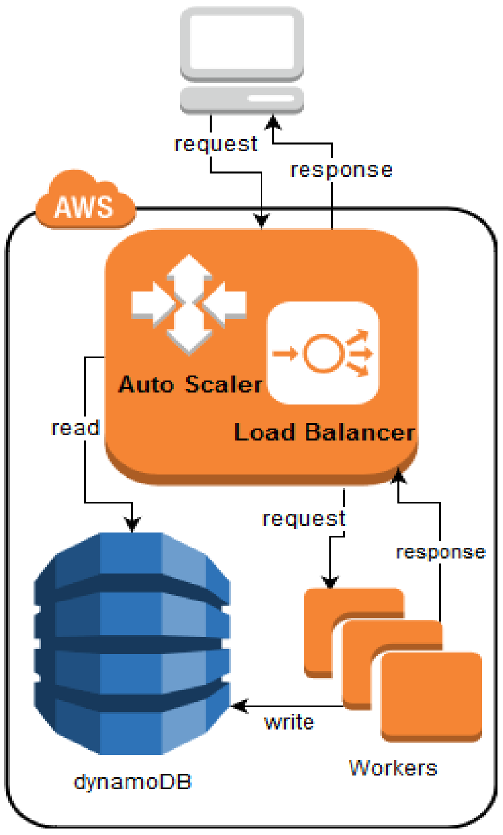
Load balancing, instrumentation, auto scaler, scheduling, aws, amazon ec2, amazon dynamodb, mazerunner

1 INTRODUCTION

Our task was to develop a solution for load balancing requests to a Java application which solves escape routes from predefined mazes. Users submit requests providing the maze configurations: start and end positions, velocity, maze size and strategy (either BFS, DFS or A*). The amount of work to solve the maze depends on the different request parameters, with larger mazes in general requiring more time to solve. In order to improve performance and availability, the application is deployed distributed across a set of worker nodes. The users communicate solely with a load balancer, which acts as the front end in the solution, the load balancer distributes requests to the workers based on instrumentation metrics extracted at the workers. An auto-scaler decides how many workers are required at a given time to process the influx of requests, automatically turning on and off machines to meet the expected demand. Finally, communication between the workers and load balancer/auto-scaler occurs indirectly via a Metrics Storage System. In this setting, the workers act mainly as producers of metric data, while the load balancer and auto-scaler operate as consumers of this data.

2 ARCHITECTURE

The implementation of our solution was made on top of AWS EC2 for the computing nodes and dynamoDB was chosen for the storage layer. The load balancer/auto-scaler runs in a single EC2 instance. The same worker code is deployed across all workers, additionally each individual worker is assigned to an EC2 instance and manages a thread pool which assigns threads to user requests. The code instrumentation is performed via BIT, which generates metrics about the dynamic behavior of worker nodes. Since each request is assigned to an operating system thread, to ensure isolation the instrumented metrics are thus locally stored independently for each thread. User requests take the form of HTTP GET requests, which are routed at the load balancer to a particular worker node. After maze completion, the response with the solved maze is sent back to the user through the load balancer node. The auto-scaler maintains a list of all the workers which are currently available for processing requests and sends a ping to the workers on this list at regular intervals. If any of the workers fails to respond within a grace period then we remove that worker from the list of available workers and terminate the EC2 instance. In order to initialize the application we first launch the load balancer, which at startup time sets up the dynamoDB databases (if these are non-existent) and loads the required configurations, namely the worker AMI name and the minimum/maximum number of workers. With these configurations, the auto-scaler fetches information from AWS about the running worker instances and starts new instances to meet the minimum number of instances. When a worker is ready (after instance startup and a running application) the auto-scaler adds that worker to the list of available workers so that the load balancer may accept and forward requests from clients(browsers) to that particular worker instance. Workers send regular updates to dynamoDB each time a fixed number of Basic Blocks are executed for each request. After request completion, the worker updates the cache in dynamoDB with the end result, and the response is sent back to the client.



3 INSTRUMENTATION

We started the design of our solution by studying the provided code and experimenting with some BIT instrumentation samples to the effect of analyzing the tradeoffs between runtime overhead and information gain. Since our application is CPU-bound, we do not instrument any metrics which would make sense in an IO-bound workload. Originally we instrumented the number of method calls for each method (e.g., method getContent() had 500 invocations for request X), which permitted us to store the five most called methods for each request. In the end we chose to remove this metric, since its results were tightly coupled to the source code, and if the method names were eventually changed the data stored previously would lose its value. Besides, it could happen that a method X were called N times for a short request and the same N times for a long request, and in this scenario further analysis would be required to assess the internal complexity of each method.

We resorted to extract the number of executed basic blocks for each request, and the number of instructions, which we could obtain without additional overhead. We do not extract the number of method calls per request as it would require an extra overhead without much information gain over the number of basic blocks.

4 DATA STRUCTURES

4.1 Workers and Metrics

All the metrics gathered during the computation of the requests are stored in a class Metrics, that keeps the basic block count and instructions count.

In order to support multi-threading, there was also the need to create an HashMap that keeps track of the metrics for each request.

At each 1000000 basic blocks, an update is sent to dynamoDB that stores the values in table ***Metrics.***

At the end of the thread execution we also send the final metrics to the CACHE table.

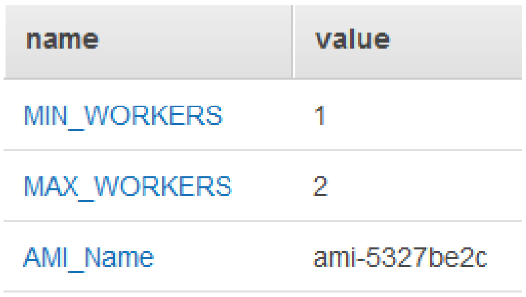
4.2 External Storage System

All the storage is kept in tables in dynamoDB. The used tables were:

- **MyConfig** TABLE
  - Holds the configuration needed to run instances

name	name of configuration
value	value for configuration

Ex:

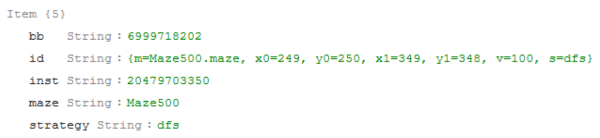


name	value
MIN_WORKERS	1
MAX_WORKERS	2
AMI_Name	ami-5327be2c

- **CACHE** TABLE
  - Holds basic block and instruction count for finished mazes

id	String of the request
strategy	Strategy used
bb	Basic blocks done in request
instr	Instruction Count in request
maze	Maze

Ex:

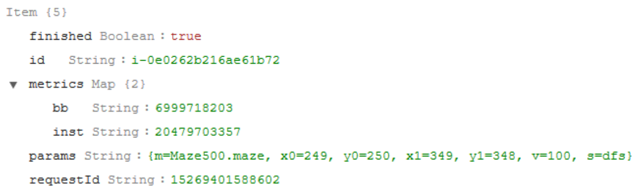


```
Item {5}
   bb   String : 6999718202
   id   String : {m=Maze500.maze, x0=249, y0=250, x1=349, y1=348, v=100, s=dfs}
   inst String : 20479703350
   maze String : Maze500
   strategy String : dfs
```

- **Metrics** TABLE
  - Holds information about metrics

id	Instance Id
requestId	Request Id
finished	If already finished the request
metrics	Map with current bb count and instr count
params	maze params

Ex:

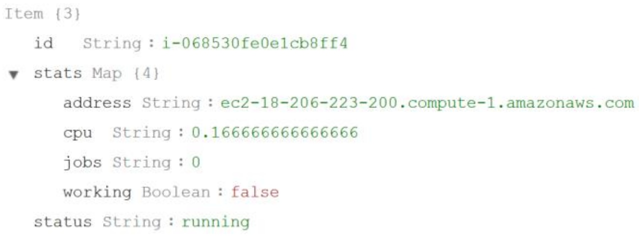


```
Item {5}
   finished Boolean : true
   id   String : i-0e0262b216ae61b72
 ▼ metrics Map {2}
      bb   String : 6999718203
      inst String : 20479703357
   params String : {m=Maze500.maze, x0=249, y0=250, x1=349, y1=348, v=100, s=dfs}
   requestId String : 15269401588602
```

- **WORKERS** Table
  - Hold information about workers

id	instance Id
stats	Map with address, cpu use, quantity of jobs and working status
status	machine status, can be pending, running or dead

Ex:



```
Item {3}
   id   String : i-068530fe0e1cb8ff4
 ▼ stats Map {4}
      address String : ec2-18-206-223-200.compute-1.amazonaws.com
      cpu  String : 0.166666666666666
      jobs String : 0
      working Boolean : false
   status String : running
```

5 REQUEST COST ESTIMATION

For the cost function we start by running a series of tests for each maze and strategy to try to find some pattern that would help the cost function. We could not find a pattern that would eventually help us but with these tests we managed to find two things:

1) Based on the tests we were able to define order types according to their size in basic blocks:

- longRequest more than 5MM basic blocks
- medium request between 20M basic blocks and 5MM basic blocks
- shortRequest less than 20M basic blocks;

2) We were able to make an expectation of the cost of a request according to its strategy in terms of initial cost and cost per distance in basic blocks based on some tests that we made. We consider distance the min (ABS (x1-x0), 1) * min (ABS (y1-y0), 1), and for each strategy they are:

	astar	dfs	bfs
Initial Cost	83241	90128	90139
Distance cost	88028	35929	60898

These values are still not very good, but it allows us to have a sense of cost.

These values are also only used if necessary because we have the objective of using a cache of previous results and the cost of the function will take into account this cache with the following algorithm:

```
let worstcase = (((Math.abs(x1 - x0) * Math.abs(y1 - y0) * DISTANCE_COST + INITIAL_COST) +
                ((Math.abs(x1 - x0) + 1) * (Math.abs(y1 - y0) + 1) * DISTANCE_COST + INITIAL_COST)) / 2);
if no cache available:
    return worstcase;
else:
    get from cache the result with less distance:
        for each cache result:
        let distance = Math.abs(x0 - cacheValue.getX0()) + Math.abs(x1 - cacheValue.getX1()) + Math.abs(y0 - cacheValue.getY0()) +Math.abs(y1 - cacheValue.getY1());
    compute expectedCase based on the distance from the cache selected result;
    return (expectedCase + worstCase) / 2;
```

In addition to the type of algorithm we also take into account the speed. For speed we can find a pattern that tends to help with the calculation of the expected cost based on speed. To find the pattern we decided to take a sample of tests and run at speeds of 10 in 10, thus obtaining 10 results per test. Then we decide to compute V (n) / V (n + 10) and this is how we get a pattern for bfs and dfs. In the case of the astar we conclude that the speed does not have any impact. The following table show us the ratio based on V(n)/V(n+10) and we use this ratio to calculate the expected cost with the speed:

	Ratio (bfs)	Ratio (dfs)
V(10/20)	1,745786269	1,94598237
V(20/30)	1,331722142	1,461898777
V(30/40)	1,197415342	1,297970488
V(40/50)	1,134977036	1,218769479
V(50/60)	1,101060126	1,174761305
V(60/70)	1,076816376	1,140720308
V(70/80)	1,057542583	1,110712347
V(80/90)	1,049745343	1,100321691
V(90/100)	1,040675452	1,085569101

6 TASK SCHEDULING ALGORITHM

At the beginning of each request we build a Job where we put the expected cost based on the cost function and the basic blocks processed equal to zero. These jobs are synchronized so that the processed blocks are updated according to what the workers have already processed.

With this information and assigning the jobs to workers we can know the progress of each worker and get how many basic blocks are missing and so choose in the load balancer the worker that has fewer basic blocks missing.

7 AUTO-SCALING ALGORITHM

The auto scaling algorithm has two parts, create instances and destroy instances. The decision to create or destroy worker instances is made every 30 seconds (through a timer schedule).

Create Instance

The decision to create a new worker instance is based on the following rules:

- The last worker instance was created more than 240 000 milliseconds (4 minutes) ago.
- The number of workers is less than the configured value of minimum number of workers
- The CPU average of all instances is above a CPU threshold of 60 percent.
- The total estimation of basic blocks to be processed is above 3 short requests plus a long request
- The total estimation of basic blocks to be processed is above 2 long requests plus a short request

Destroy Instance

The decision to destroy a worker instance is based on the configured value for the maximum number of workers. If the number of workers is bigger than the maximum number of workers configured we make the decision to destroy one instance based on the following:

- We choose the less relevant instance, based on the worker with less basic blocks to be processed
- If the instance doesn't have any job, we terminate it, if the instance is still processing requests we mark it to don't accept more requests.

8 FAULT TOLERANCE

In order to support Fault Tolerance, all we had to do was to check if the response from the worker to the load balancer is empty. If it's empty, we wait 5 seconds and resend the request to the best worker available.

We only stop doing this when we receive a valid response from the worker.