Sudoku@Cloud - Final Report - Group 11

Vasco Morganho
Nº 81920

José Canana
Nº 82039

Diogo Cardoso
Nº 94024

Abstract

This report describes the final state of our sudoku@Cloud implementation. We start by explaining the full system architecture and main components in Section 2 followed by all the developed algorithms to help with the process, in Section 3 we proceed to explain our fault tolerance implementations and Section 4 concludes the report.

1. Introduction

Sudoku@Cloud consists of an elastic cluster of web servers with the main purpose of solving different types of sudoku puzzles. Following the previous checkpoint submission, we have developed all the proposed remaining system features while also adding fault tolerant measures to provide a more reliable and available service. These implementations and additions are as explained in the next Sections.

2. System Components

The system architecture and components are as explained in the previous report with the addition of an Metrics Storage System (DynamoDB), as portrayed in Figure 1.

The system receives HTTP requests through the Load Balancer module, which then chooses the most optimal Web Server instance to solve the request. This choice is based on the stored metrics available at the DynamoDB.

The Auto-Scaler module works along side the Load Balancer one. This module is in charge of monitoring each live instance and decides to raise or lower the current number of live Web Server instances in order to save resources while also maintaining service availability.

2.1. Web Server

The Web Server instances are in charge of solving the received sudoku puzzles and sending the solutions back to the users, via Load Balancer. However, this module also has the important responsibility of saving desired metrics used to optimize the Load Balancer decisions, and consequently, the overall system capabilities. This is possible by the previous instrumentation of the Web Server in order to record such metrics.

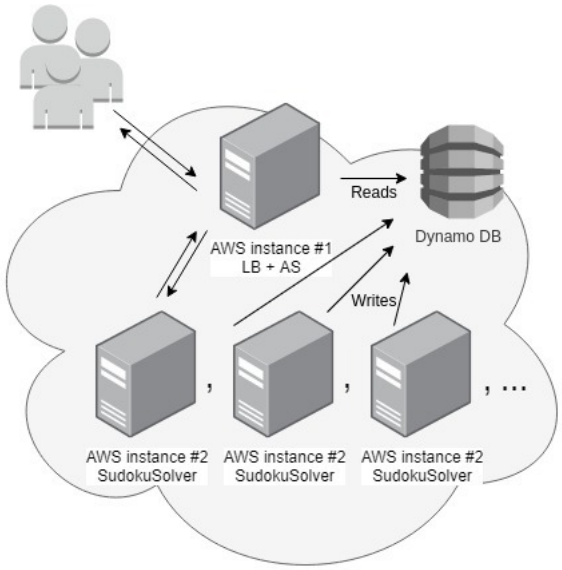

Figure 1. Sudoku@Cloud Architecture

2.1.1 Instrumentation

The given java Bytecode Instrumentation Tool, BIT, allows us to build customized tools to instrument Java bytecodes before they are loaded and executed by the Java Virtual Machine. These developed tools are able to retrieve several valuable information from the running java program that we can take advantage of in order to optimize our system.

To accomplish this task, the solver classes of the Web Server where instrumented with a tool developed for this purpose. Using the BIT java tool and our own SudokuTool (available at BIT/BIT/SudokuTool.java) we were able to dy-

namically evaluate the running program and save several metrics, including the total number of instructions executed and methods called. The extracted metrics are then used to calculate the cost of the received request and the information is stored in the Metric Storage System, making it available to the Load Balancer.

The instrumentation process is as shown in Figure 2.

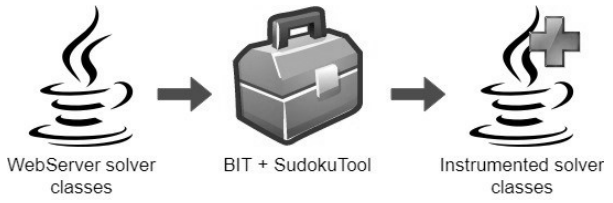

Figure 2. Instrumentation Process

2.1.2 Cost Calculation

Algorithm 1: Web Request Cost Calculation for DynamoDB insertion

$stage1 = \frac{unassigned \times 100}{puzzle\_size}$;

if *strategy = BFS* **then**
  $stage2 = 50$;
else
  $stage2 = 25$;

$stage3 = 0.5 \times \frac{\#instr \times 100}{max\_instr} + 0.5 \times \frac{\#methods \times 100}{max\_methods}$;

Result: $0.3 \times stage1 + 0.2 \times stage2 + 0.5 \times stage3$

To calculate the cost of each request, we have developed a three stage arithmetic function that takes into account several of the request characteristics, as explained in Algorithm 1, and returns a cost between 1 and 100. In the first stage, the puzzle data is considered, i.e. puzzle size and unassigned entries, and it is calculated the percentage of missing values. In the second stage the chosen solver strategy is considered. Using as reference the values from the testing phase of our work, we assigned the cost of the DLX and CP strategies the same and raised the cost of BFS requests. Finally, and the phase with most cost importance, the third stage takes into consideration the total number of executed instructions and methods called. This final calculation is based on the maximum possible number of methods and instructions, given by the biggest puzzle, with maximum number of unassigned positions and Brute-force solver strategy.

Once a Web Server instance finishes solving a given puzzle, its cost is calculated and all the request parameters and instrumentation metrics are saved in a new entry at the AWS's DynamoDB.

2.2. Load Balancer

The Load balancer functions as a bridge between the users and the solver instances. This module maintains a private array where is stores the id of the current Web Server instances and their corresponding weight, i.e. the total amount of requests being solved by each instance.

When a new request is received by the Load Balancer, it is transferred to the live instance with the least weight. It is worth noticing that weight does not correspond to total number of requests being solved, but to the total cost of them.

2.2.1 Cost Estimation

Algorithm 2: Web Request Cost Estimation for solver instance selection

if *Request in cache* **then**
  $estimated\_cost = cached\_value$;
  Return;
else
  if *Request in DynamoDB* **then**
    $estimated\_cost = dynamoDB\_value$;
    Return;
  else
    $estimated\_cost = average\_of\_similar\_requests$;
    Return;

$stage1 = \frac{unassigned \times 100}{puzzle\_size}$;

if *strategy = BFS* **then**
  $stage2 = 50$;
else
  $stage2 = 25$;

$stage3 = 0.5 \times \frac{\#instr \times 100}{max\_instr} + 0.5 \times \frac{\#methods \times 100}{max\_methods}$;

$0.3 \times stage1 + 0.2 \times stage2 + 0.5 \times stage3$
Return;

Result: estimated_cost

At this stage of the process, the load balancer does not

yet know how many instructions and methods will be executed, so the cost can only be estimated at this point. The process is as demonstrated in Algorithm 2.

To accomplish this task the DynamoDB is used again, this time to gain information from the previous solved requests.

If the same request has already been processed, i.e. same puzzle size, unassigned entries and solver strategy, the metrics can be read from the database and the cost successfully calculated. Otherwise, the cost is estimated based on similar saved requests.

To estimate the cost, two possible scenarios are considered. If there are several similar requests saved, i.e. same puzzle size and solver strategy but different number of unassigned entries, the cost average is calculated and assigned to the request. If there are not enough similar previous requests, the number of instructions and methods are set to 50% of their possible max values.

With this implementation, a couple of advantages should be noticeable. If the previous request was already solved by the same instance there is no need to check the DynamoDB, which is a time consuming process. This means that the Load Balancer cost estimation process becomes faster and more efficient as time goes by.

2.3. Auto-Scaler

Algorithm 3: Live instances Auto-Scaling procedure

$lowestCPU = 200;$
$totalCPU = 0;$
$avrgCPU = 0;$

for *instance in current_instances* **do**
  $CPUutilization =$
   $getInstanceCPU(instance);$
  $totalCPU+ = CPUutilization;$
  if $CPUutilization < lowestCPU$ **then**
    $lowestCPU = CPUutilization;$
    $instanceToShutdown = instance;$

$avrgCPU = \frac{totalCPU}{current\_instances.size()};$

if $avrgCPU > 80$ **then**
  $launchNewInstance();$
else if $avrgCPU < 20$ *and*
 $current\_instances.size() >$
 $MIN\_INSTANCES$ **then**
  $terminateInstace(instanceToShutdown);$

The Auto-Scaling module runs in the same instance as the Load Balancer and is in charge of managing the total number of live Web Server instances, with a minimum of 2 instances at a time (for testing purposes) and an undefined maximum.

To help its decision of launching/killing instances, the average CPU utilization metric of each running solver instance is calculated and a decision is made, as shown in algorithm 3.

Every 2 minutes a new calculation is made. If the average is greater than 80% a new instance is launched, if it is lower than 20%, the instance with less utilization is terminated. Otherwise, the total number of live instances remains the same until the next calculation cycle.

With this implementation we are able to save valuable system resources while also maintaining the system available for large / small user concurrent computations.

2.4. Metrics Storage System

The Metric Storage System, implemented using Amazon's DynamoDB, is in charge of storing the saved metrics and making them available to the Load Balancer. Once a Web Server instance successfully solves a request, it writes a new entry to the database with the request parameters and retrieved metrics, as shown in Table 1.

This module makes it possible for the Load Balancer to correctly estimate the cost of new requests and consequently optimizes the overall system resources and capabilities

It is worth noticing that equal requests are not stored again.

#_instr	#_methods	strategy	puzzle_size	unassigned	cost
65862	578	BFS	16	15	12
19618	197	BFS	16	151	27
11122	130	BFS	16	256	40
18224	154	BFS	25	150	17
27017	212	BFS	25	552	36
17989	210	BFS	9	25	19
29311	312	BFS	9	40	25
67133	474	CP	16	120	19
53593	718	CP	16	256	35
11098	132	CP	25	100	9
3450	41	CP	25	50	7
12515	117	CP	25	500	29
16136	112	DLX	9	25	14
15302	123	DLX	9	70	31
14438	136	DLX	9	80	34

Table 1. DynamoDB stored data sample

3. Fault Tolerance

As a way to increase system availability we have developed two fault tolerant measures and incorporated them into

our Load Balancer.

- The maximum time for instance response is monitored by the Load Balancer. In case of long wait for instance reply larger than the maximum time, the request is sent to next instance in line and the first one is assumed stopped. In case a late reply is recorded by the Load Balancer, the solution is discarded but the instance is considered alive again and consequently re-placed in rotation.

  With this implementation we can detect stopped / failed instances and are able to restart the request elsewhere without the user ever knowing about the occurrence.

- The Load Balancer maintains a cache of all the previous web requests and their corresponding costs. This was initially implemented as a process optimization to access the DynamoDB less often but also works in case of DynamoDB unavailability.

  This implementation makes the system functional in case of DynamoDB failure and doubles as a time saving feature.

4. Conclusion

Due to lack of time to fully develop and test the system, mostly because of coincident deadlines, some features could have been better prepared and developed. It is our felling that the cost estimation / calculation algorithms could have been tweaked for better efficiency and some auto-scaling features could also be improved. Specifically, and as explained previously, the Auto-scaling module only takes into consideration the CPU usage of the AWS instances but it would have been interesting to also take into account information about the incoming requests to more consciously manage the scale-in and scale-out of our system.

Nevertheless, all modules implemented are currently working and ready to run at AWS instances on startup.