Soduko Cloud
Project Checkpoint Final
Cloud Computing and Virtualization

Gonçalo R. A. Faria, João Marques
Instituto Superior Técnico
University of Lisbon
Lisbon, Portugal
joao.a.marques@tecnico.ulisboa.pt, goncalorafaria@tecnico.ulisboa.pt

Abstract

This report details the design and development of an elastic cluster of web servers that is able to execute a computationally-intensive task to discover the solution of a sudoku puzzle on-demand.

1. Introduction

The goal of this report is to detail the design and development of an elastic cluster of web servers that can execute a computationally-intensive task to discover the solution of a sudoku puzzle on-demand. The system was designed to receive a stream of web requests from clients. Each request contains a description of a sudoku board as well as the name of one of a predefined list of available solution methods. In the end, the solved soduko boards are sent back to each of the clients.

The project was proposed by the faculty of **Cloud Computing and Virtualization** in a statement delivered to students. Accompaning these project's statement, there was an implementation of a single web server in java byte code and an HTML client.

To implement the functionality required for an elastic cluster of web servers, given that We were suposed to use the provided byte code, the original byte code was instrumented using the BIT library(0).

The main functionality implemented by instrumentation was data collection. The continously collected data, serves as the ground truth for the load balancing and resource allocation policies.

The system was designed to run within the Amazon Web Services ecosystem.

The first checkpoint comprises the software infrastructure for the system and a detailed exploration of what metrics are more usefull for the load balancing and resource allocations policies while keeping the ovearhead low.

2. System Architecture

Figure 1 contains a high-level illustrative description, as delivered by the faculty, of the target system architecture. The client interacts with the system by submitting an HTTP request to the load balancer. The load balancer decides which virtual machine will serve the request.

Figure 2 contains simplified class diagram of the project's code base.

2.1. Load Balancers

Since hypothetically, an extremely large number of web-servers could be started to take care of an increasing demand

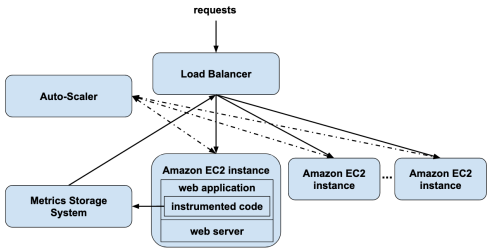

Figure 1: Diagram of the soduko cloud architecture.d

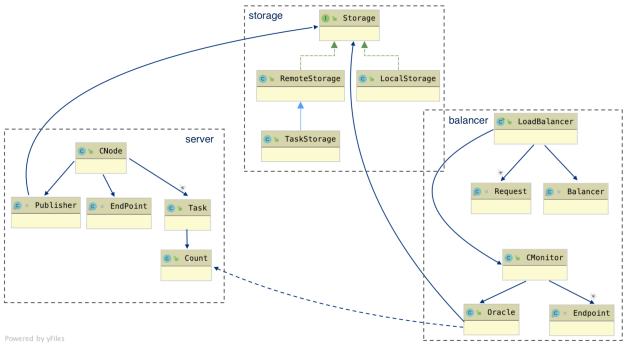

Figure 2: Classe diagram of the codebase of this project.

of requests, We can conclude that the load balancer will be the bottleneck of the entire system.

For this reason, initially, the main design decisions behind the load balancing implementation were motivated to free resources in this crucial component. However, this made the handling of web server faults a responsability of the client application.

Therefore, We decided to use HTTP redirection as the mechanism to avoid directly answering the client from the load balancer. It is important to note that, We refer to temporary HTTP redirection. This means that subsequent requests have to go again through the load balancer. This does not provide client affinity, which is not a particularly useful property for soduko solving. One of its clear costs is the additional round trip time for redoing the HTTP POST to the web servers.

With this approach, rather than keeping 2 connections in the load balancer per request, one for the client and other for imitating the client for the selected web server, I'll have neither.

The only work in the load balancer is receiving the request, putting it in a queue, and then a 'balancer'(server worker) decides where to re-route the request. In this way, after the re-routing, every responsibility regarding the request is transferred to the web servers. What is left is one TCP connection for each of the web servers. These connections are the medium for continuously maintaining a representation of each of the instances inside the load balancer. This state representation will have valuable information that will be used for the autoscaling and load balancing in conjunction with the metric storage system(queue size, expected time to complete, etc).
However, if transparent(to the client) fault handling is a requirement, the load balancer can be configured to provide it. This is achieved by specifying true as the argument of the load balancer application. When this option is chosen, instead of using Http redirection the load balancer uses an RPC(Remote Procedure calls) based architecture.
The choice of where to route a particular request is always based on sending the incoming request to the server with the lowest load. The estimation of the server load will be presented in the following sections.

2.2. Auto Scaler

The auto scaler is particularly simple assuming we have an accurate estimative of the remaining load in every server. Essentially, periodically, the average load per machine is computed. If this value is above 90% of the ideal load a new virtual machine is created. If the average load per machine is bellow 20% of the ideal load then the machine with the lowest load is deleted.

2.3. Web Server and Code instrumentation

The Webserver was intrumented for two porposes. Firstly, metric collection, to allow for counting the number of branches taken and other metrics that will not be used for the final appllication but that were crucial to study the computational caracteristics of requests with different parameters(particularly because the source code was not given) and secondly to keep a connection with the load balancer.
The first one proved to be challenging, given that to obtain metrics associated with a particular request the code has to intercept when a new thread is created, which parameters the request has and to start a new metric counter for that specific thread.
For this porpose, We extended the functionality of the high-BIT library to allow for the execution of a desired function before or after the execution of particular routine with the arguments of the routine. In this way, it is possible to acess values of variables contained in the running application.
This was used to execute code before the argument parser so as to associate the thread to specific request parameters.
Aditionally, as mentioned, the connection with the load balancer is used to pass valuable information. Particulary, load reports, metrics from the execution of request, webserver state information in the case of load balancer went down.

2.4. Persistent Storage

The persistent storage for saving the performance metrics of the request was a table using the Amazon DynamoDB. The schema of the table is just a key with the format "<solver type>:<unassinged entries>:<board format>" and a count object which keeps a running mean, variance (0) and count of the metrics of the specefic request.

2.5. Oracle and Request's Metric Cache

The Oracle is the component that obtains the expected load for each of the incoming requests. In order to reduce the number of requests performed to the Amazon DynamoDB, the Oracle was implemented using a request cache.
The cache was created with an update and remove policies. The remove policy decides which items to remove from the cache. The update policy decides when to get new entries or update old ones.
The entries need to be updated because each entry contains a running mean and variance for the expected load of a specific request. Ideally, we would update the entries frequently when the estimate was obtained from a few samples and is frequently accessed and eventually for the other cases.
The solution we've found for the policies is using the UCB(Upper Confidence Bound)(0) algorithm for each solver and board configuration pair and also for each individual request.

2.5.1. UCB Algorithm

The UCB algorithm was introduced has the solution for Bandit Problem in Statistics. However, it is commonly used as a heuristic for the exploration vs exploitation dilemma(exploit the current knowledge or explore in the present for more exploitation in the future). In this application, exploitation is using the available values in cache and exploration is going to DynamoDB in order to obtain a recent version of the same value.
The algorithm is based on the principle of optimism in the face of uncertainty, which is to choose your actions as if the environment (in this case bandit) is as nice as is plausibly possible. It has many forms depending on the assumptions regarding the particular bandit problem(here we assume gaussianity).
Essentially, here the agent has two actions, hit or update. When an action is performed we get a reward, which comes from a stationary distribution. We use this value to update a running average of the reward associated with each of the actions. So instead of picking the action that has the maximum reward estimate, we pick the one with the maximum value of the following expression.

$$\hat{R}_a + \sqrt{\frac{2log(t)}{N_a}} \tag{1}$$

The symbol $\hat{R}_a$ is the estimate of the reward for action a, N_a the number of times the action was taken, and t the number of times the agent had to decide. Therefore, an action that is rarely picked becomes more appealing has time goes on.
The rewards in this cache update environment are essentially fixed for each action, a value k for hit, and 0 for update. Therefore, updating will always be the worst action, however, it will sometimes be preferred for the sake of exploration.

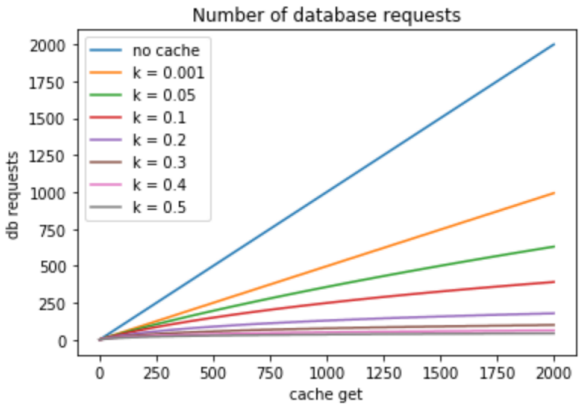

Figure 3: Experiments maid using the ucb algorithm applied to the cache replenishment problem.

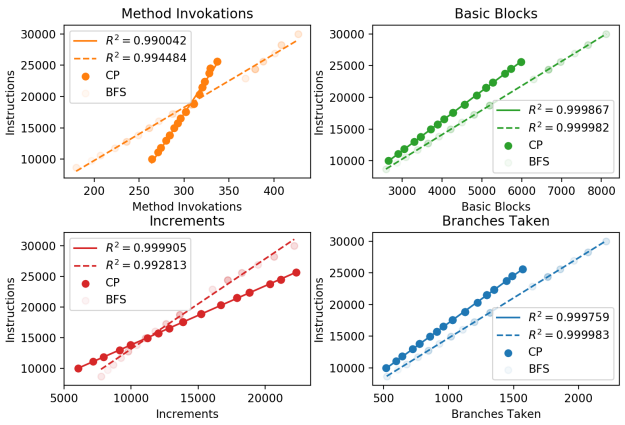

Figure 4: Experiments maid using the provided client for 9x9 soduko board with CP and BFS solvers.

To implement this algorithm, for this two actions case, we need only two integers, to count the number of times each action was performed.

Figure 3 contains experiments using different value of k. We chose the value of 0.1 in the implementation.

2.5.2. Cache implementation

The cache is a rose tree concurrent data structure. The root has 3 nodes, one for each of the solvers. Each of the solvers has a variable number of nodes one for each board type(9x9,16x9, etc.) present in the cache with the particular solver. Each board type has a variable number of nodes corresponding to the requests with the same solver and board type but a different number of unassigned entries. Upon receiving a request, there is a first-level bandit problem for the pair solver and board type, named **Group**. It is of deciding, when the request is not in the cache, whether to go query the database.

When the number of elements in the cache surpasses a predefined threshold, for each number request in excess, a Group is chosen, then a request within that Group is removed.

The Group with more elements is always the one chosen(in case of a draw, the one with fewer updates is picked). Within the particular Group, the request with an estimate with fewer samples is removed.

2.5.3. Load Estimation

The main functionality of the Oracle giving an estimate for the load a request will ensue in the server. The load is designed to be a measure of CPU time. Ideally, it would be independent of the solving method, which is crucial for comparing different requests.

The load is a function of a metric collected using instrumentation. The metric and the particular function for each solver type will be subject of study in the following sections. However, most of the time, the metric mean for a particular request won't be in the cache. Therefore, we designed functionality for estimating the metric mean, using the metric from requests within the same group. This is accomplished using interpolation. When there are no elements for the particular group a default value is used.

2.6. Fault detection and Handling

The webserver faults are detected by a timeout mechanism for the load reports. The load balancer faults are detected by an acknowledgment mechanism after every load report.

If the webserver fails the load balancer terminates the virtual machine and creates a new one. If the load balancer fails the webserver waits for a new connection from the same load balancer or another.

As mentioned before, the system was designed primarily to do fault detection and client-side fault handling. However, it is possible to configure the load balancer to also do fault handling in a way that is transparent to the client(just by starting the application with the argument 'true'). This comes at a cost of additional threads active in the load balancer and sockets open(which can be configured at load balancer startup).

When implemented using HTTP redirection, the load balancer loses the connection with the client, hence there's no possibility of handling any fault.

3. Metric Selection

3.1. Predictive capacity

To determine what types of metrics would be valuable for determining each request's load(which We considered to be the number of instructions) We tested a set of metrics. Particularly, We experimented with the number of 'basic blocks executed', 'methods invoked', 'branches taken' and 'increments performed'.

Figure 4 contains 4 scatter plots that depict how the number of instructions varies as a function of the metrics for the CP(Constrained Programming) and BFS(Breath-First) solving methods. The data pertains to a 9x9 soduko map, for multiple values of unsigned entries.

As can be seen, all of the metrics correlate extremely well with the number of instructions for the tested solvers. It can also be seen that the slope for the regression line is different for different solvers.

Figure 5 contains the same experiment now performed with the DLX solver, under the same conditions.

Unlike, when using CP or BFS, the metric number of methods invoked, does not correlate well with the number of instructions.

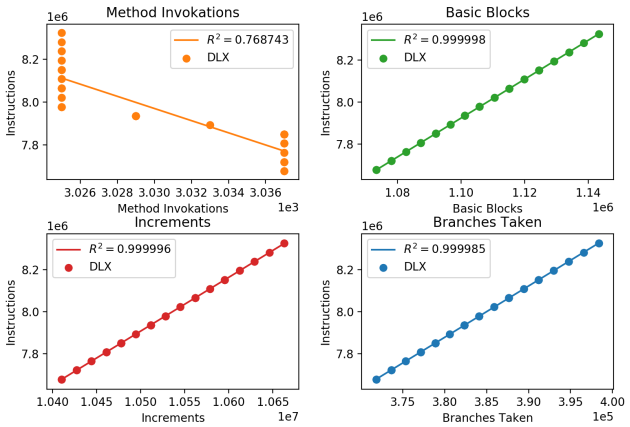

Figure 5: Experiments maid using the provided client for 9x9 soduko board with DLX solvers.

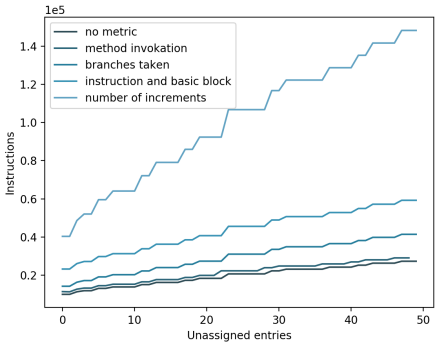

Figure 6: Overhead results from solving multiple configurations of 9x9 soduko board with the CP solver.

3.2. Metric Overhead

Given that almost all of the tested metrics predict reasonably well the number of instructions to decide which to use for the load balancing We decided to measure the number of additional instructions required for collecting each of them. Figure 6, 7 and 8 present an exploration of the metric overhead for the same 9x9 soduko board for the CP, BFS and DLX, respectively.

What We can conclude is that 'methods invoked' causes the lowest amount of overhead, followed by 'branches taken'.

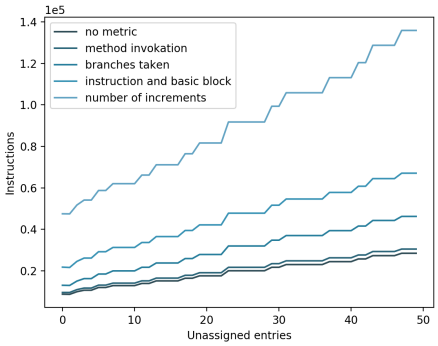

Figure 7: Overhead results from solving multiple configurations of 9x9 soduko board with the BFS solver.

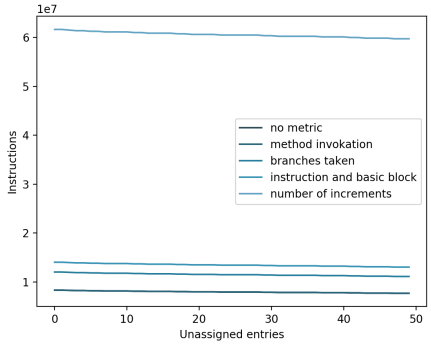

Figure 8: Overhead results from solving multiple configurations of 9x9 soduko board with the DLX solver.

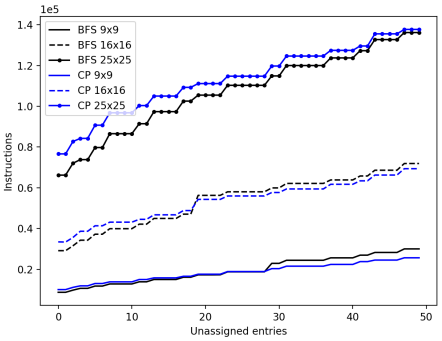

Figure 9: Comparison of number of instructions required for boards with differnet sizes with CP and BFS solving methods.

If this metric also correlated well with DLX, it would be the best choice. Since this is not the case, We decided to use 'branches taken'.

3.3. Metric Choice

It is clear that for every solver 'branches taken' correlates almost perfectly with instruction number. However, for different solvers, the slope of the linear predictor is different. This means that increasing the "number of branches taken" does not lead to the same increase in the number of instructions. This suggests that, by itself, the number of branches taken can not be used as the criterion for comparing loads of different solvers. To this end, we need a linear predictor for each solver which, given the number of branches taken, outputs the expected number of instructions.

Despite being generally known that as the size of the board increases the problem rapidly becomes intractable(due to the exponential growth of possible combinations). However, for a fixed board size, the growth associated as a function of the number of unsigned entries seems close to linear. This can be seen in Figure 9. This proved extreamly usefull for load estimating in the Oracle component.

Additionally, given that some of the solvers shown to be slightly sensitive to board configuration, it will be useful to save the sample standard deviation along with the metric mean for each of the request parameters in the Metric Storage System.

4. Evaluation

To understand which are the best parameters for our system, we developed tests to evaluate the performance of the different options.

We already have a unified measuring unit to evaluate all our requests, as well as the load of each instance. In our methods below we indentify the optimal load for 1 instance and then scale up the procedure to test the Auto-scalar parameters (scaleUpThreashold, scaleDownThreashold).

Our objective is to maintain good server stability and reliable latency.

4.1. Methodology - Single instance load

The single instance test involves fixating a 20 request array with diverse properties and complexity. This array of requests is then executed one at a time, at a variable frequency. In figure 10 you can see the estimates, loads and latencies of each series of the tests.

Typically, with higher frequency of requests we would notice a load increase, but at frequencies higher than 60s the server had dificulties responding and handling the requests. We can safely assume that the Webserver can't handle loads bigger than that limit. Even at 80s interval between requests, erros were more frequent than acceptable.

At 80s frequency the typical load spike had a estimated load of 1000000units, and at 100s frequency, half of it, 500000units. The average load at 100s frequency was around 800000units. The ideal load seems to be a value between 500000units and 800000units.

One of our runs of tests with decreasing frequency of requests can be seen at figure 14. This figure also shows the credit usages that occured during the test. This run goes from 20s frequency until 60s frequency, ending at 17h. We can notice that we have to repeatadly use the max possible number of credits (5). This is not an acceptable position, because this credits eventually run out, and then we will be stuck with serveral folds less performance. From our expericence, if the average cpu % usage is close to 100%, we are most likely using CPU credits, because Amazon already tries to handle load spikes for us.

4.2. Methodology - Auto-scalar parameters

For the sake of testing the auto-scalar we have fixated an ideal load of 600000units. We now want to understand how much of a deviation from this load, is an appropriate estimator for terminating or summuning an instance.

We refurbished the previous test client of the single instance test, which sends a pre-determined set of requests at a specified frequency. But this time, at each iteration, new clients start using the service as well, all of them wanting to execute that same pre-determined series of requests, and then stop using the service.

The scale parameters are percentages of the ideal load (600000units in this case). If the scale up threshold is 0.9, we will summon a new instance when our instances have 90% of the ideal load. And similarly, we will terminate an instance when our load is below the threshold.

Our objective was to find the thresholds which allowed our system to be between 80% and 50% average CPU usage at

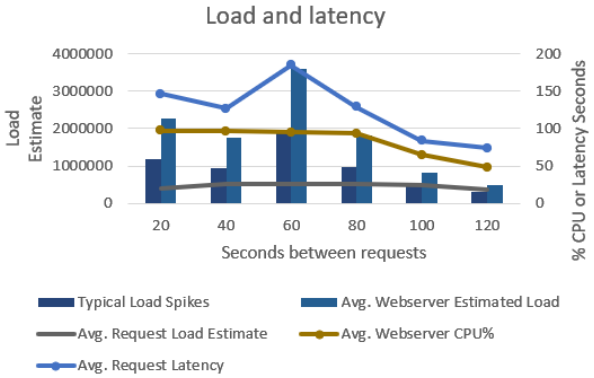

Figure 10: Single machine load test - Load and latency

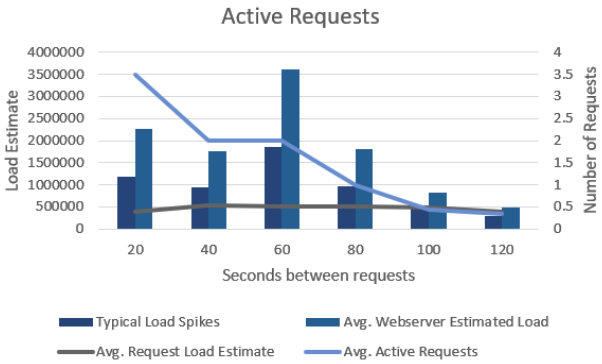

Figure 11: Single machine load test - Active Requests

all times. Because those were the values we found out to be correspondent to our ideal load.

The clients entering the service at each time-step for the measurement of the scale-up threshold: (2,4,8,2,4,8). The clients entering the service at each time-step for the measurement of the scale-down threshold: (16,4,2,2,1,1)

We can see in figure 12 and 13 the best runs we had, with the threshold of 0.8 for scale-up, and 0.3 for scale-down.

5. Conclusion

In this project, we developed the software infrastructure for the entire sudoku cloud system. The data collection, the load balancing and the resource allocation are operational. In order to effectivelly load balance we develped a system

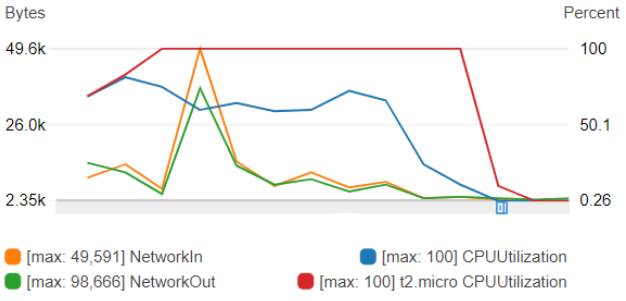

Figure 12: Load balancer scale up 0.8 threshold test: Red Line is the max %CPU usage during the last monitoring period; Blue line is the average %CPU usage.

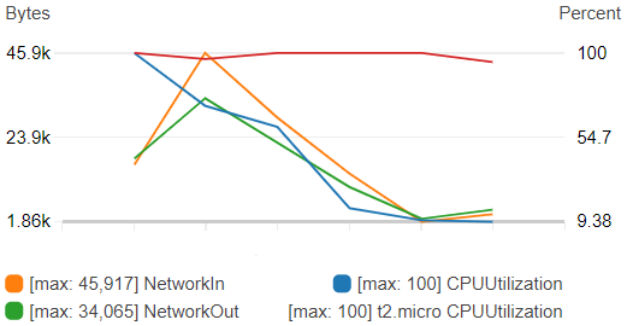

Figure 13: Load balancer scale down 0.3 threshold test: Red Line is the max %CPU usage during the last monitoring period; Blue line is the average %CPU usage.

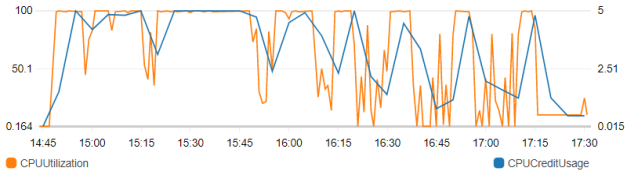

Figure 14: Average CPU usage and credit usage during the single machine load test (20s up to 60s

for estimating the load for each incoming request. It was implemented using a direct acess to the metric storage system or a cache. Additionally, we tested our system and studied the resource allocation policies for multiple parameters.

6. Bibliography

AGRAWAL, R. Sample mean based index policies by o(log n) regret for the multi-armed bandit problem. Advances in Applied Probability.

CHAN, T. F., GOLUB, G. H., AND LEVEQUE, R. J. Algorithms for computing the sample variance: Analysis and recommendations.

ZORN, B., AND LEE, H. B. Bit:a tool for instrumenting java bytecodes. Advanced Computing Systems Association.