Sudoku@Cloud

André Guerra 86382
Pedro Custódio 86496
Pedro Bernardo 86500

Instituto Superior Técnico
MEIC-A Cloud Computing and Virtualization

1. Architecture

The essential components of the system's architecture is illustrated in the diagram of Fig.1. The system has the Load Balancer and the Auto Scaler running in independent EC2 instances, such as each Web Server. Given that all the system is associated with an EC2 account, they communicate in a private network, using the private DNS of each other. When starting the system, the Load Balancer will sign up on the Auto Scaler that is already running and consequently the AutoScaler responds with the WebServer instances that are available. By default the AutoScaler, launches a WebServer on initialization. As the AutoScaler launches or terminates instances, it will inform the Load Balancer of the new event after the grace period has been reached and the WebServer is ready to handle requests.

After initialization the system is ready to receive requests. The request is received by the Load Balancer and it will forward the request to one of the WebServer instances available, using our distribution algorithm that will be discussed later. Given that the Load Balancer is multi-threaded, it will be waiting for the WebServer to respond with the solution, and then send it back to the client. If there is a WebServer fault, both the LoadBalancer and the AutoScaler have a mechanism to detect this fault and the LoadBalancer will re-distribute the request without the client realising as it is explained on *section 7 (Fault Tolerance)*.

The Auto Scaler and the Load Balancer will periodically retrieve data from Metrics Storage System in order to update their data structures needed for executing correctly their algorithms. The Metrics Storage System will be continuous updated by the Web Servers running.

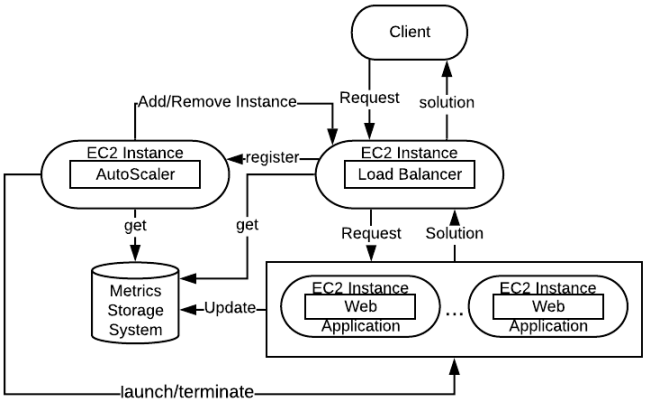

Figure 1. System Architecture

2. Instrumentation Metrics

We instrumented the solvers to output the number of , memory allocation instructions, loads and stores (both field and regular), branch outcome statistics and dynamic properties (instructions, basic blocks and method calls.

To find the best metrics, first we consider that the number of instructions is the closest to the real expected work, and subsequently, execution time of each request. Considering only the metric closest to real load of the request, we would chose the number of instructions but that would impose a considerable overhead on the nodes.

Given this, we tried to find the metric that had the best balance between its relation to the number of instructions, according to the possible variation of the request parameters that we can know in runtime (strategy, puzzle size, number of unassigned entries), and the overhead.

We noticed that the number of regular stores was a great candidate because it had an almost 1-to-1 ratio to the number of instructions both when varying the number of unassigned entries, the puzzle size and the solver strategy, but this metric still has a very high overhead.

By further analysing the data, we noted that the number of method calls was very good at estimating the number of instructions for the DLX and BFS strategies (Fig3, Fig2), being very precise in the first case, and underestimating a bit in the latter.

For the CP strategy, we observed the the number of field loads tended to overestimate the number of instructions by roughly the same amount that the number of method calls underestimates it, so we chose to use the average of these two metrics (Fig4).

The estimation of the number of instructions for each metric is calculated by multiplying it by a fixed ratio. This is so our metric results are normalized between strategies and sizes, and comparisons are always fair. To achieve this, the Load Balancer will keep a ratio for each metric and for each strategy, in a total of 6 ratios.

Also, it is important that these metrics also translate to a good estimation of the work to be done for different numbers of unassigned entries. We compared the variation of both of these metrics with the variation of the instructions, for all provided puzzles and strategies, and both of them are very good approximations, meaning they vary in almost the same way as the number of instructions do.

At this point, we had a reasonable approximation of the real load of a request while having a small overhead since these metrics do not grow to very large numbers compared to the number of instructions. However, looking for better predictions, we looked deeper at more precise metrics such as the previously mentioned

number of stores (store_count) and the number of instructions itself (instr).

We instrumented the Solver code to not only show us the final value of these metrics, but where in the code they are most affected. With this analysis we were able to, without looking at the source code, select the parts of the code that really make a difference in the total number of instructions and set aside the classes that always provide constant values, such as classes related to setting up the solvers and argument parsing.

We were able to get to a point where we only needed to instrument one basic block present in the run method of the Solver class and a few other methods of the classes related to solving with the DLX algorithm to get an approximation of the number of instructions within approximately 99% precision, which we were very happy with, but in still at the cost of around 50% overhead measured in computing time on our own computers (which are much faster than the EC2 instances this code will run).

For the previous metrics we weren't able to cut so much on the amount of instrumented code, but we still eliminated some sources of constant and negligible values.

In the end we opted to stick with the original metrics (field loads and method calls) since they still provide a reasonably good approximation to the real load, while imposing an almost negligible overhead on the Solver's performance.

3. Data structures

With the data stored in this node, the Load Balancer must be able to infer:

- How loaded is each server?
- What is the progress of each individual request?
- How long did the request take?

This means that each request has be to identified by a unique key, we chose the format the format:

$$< pl|s|sid|tid >$$

Where pl is the puzzle label provided in the request, s the strategy used to solve the puzzle, sid the server identifier and tid is an identifier set by the Load Balancer, which in our case is the id of the thread that handled the request, which will be unique for each pending request.
So, the entries stored are of the form:

Metrics
Server name
Running
Puzzle size
Puzzle label
Puzzle Un
Timestamp

Where we keep our chosen metrics (*number of methods called* and *number of field loads*), that are used to calculate the progress of a request. As well as, the $Server\ name$ to allow for server based queries, a $Running$ variable to allow for queries based on running puzzles, $Timestamp$ to allow us to query for the more recently logged puzzles, and finally the puzzle related values, so we can establish a correlation between puzzle characteristics and performance metrics in order to feed our cost estimation puzzle component.

In order to maintain efficiency, reduce MSS access related costs and guarantee client request completion, the Load Balancer needs to keep some structures in memory.

The first ones to mention are related to fault tolerance. The LoadBalancer keeps a map of <Query, Puzzle> alongside a map of <Server, HttpExchange[]> as well as <Server, Thread[]>. These are used to recover the client's HTTP request information so it can be replayed and responded to, and the last one serves to interrupt the Threads that are still waiting for a specific server's response in case we detect its failure.

To reduce MSS access, we keep the estimated total cost of each server, which is the sum of the estimation of each puzzle running in a server. To increase the granularity of our data, we also keep the estimated cost of each running puzzle and the number of puzzles running on each server. This will allow us to keep making decisions based on load without having to access the MSS, since we know what is running, how long it is expected to run for and we are informed when it stops running.

For all of this to work, we also keep a list of the active servers and a class responsible for cost estimation that also holds special structures, which we will discuss in the next section.

As for the Auto Scaler, our implementation simply keeps a list of active instances, and pending instances which are the ones already launched but not yet ready to handle requests.

4. Request Cost Estimation

Due to previous analysis we conclude that two requests have very similar cost if they are using the same strategy, size of the puzzle and number of unassigned entries, so the request cost estimation algorithm has to reflect that. That being said, the implemented solution calculates the average of different requests, if they use the same strategy and include a puzzle of the same size and similar unassigned entries, in what we call *Buckets*, incrementally with the help of a counter.

Whenever the Load Balancer needs to estimate a cost, it just needs to find the correct bucket. Each bucket contains an interval of values for the *unassigned* parameter that generate the same puzzle. This was pre-calculated only up to the puzzle size 25x25 because it is very slow, but it saves some memory and gives us better precision in our estimations. For puzzles above that size, then we simply keep a bucket for each possible value of *unassigned*.

Were the bucket responsible for the current request to be empty, the Load Balancer has to find the greatest value less than or equal (lower bound), as well as the least value greater than or equal (upper bound) to the current request and them computing the average, if both values can be found, otherwise no need do the average and return the found bound. Which will result in a worse, but still usable cost estimation. To find these values, first the system searches for similar requests, only changing the number of unassigned entries, if it returns unsuccessfully, it then goes through the different puzzle sizes. All these searches are done in

a log(n) time complexity, since it is being used a *TreeMap* for the implementation.

If there is no comparable request already computed, the system is in the cold start state where it cannot estimate the request cost. So, some default values are set at the start of the Load Balancer from the analysis of a subset previously provided. This way, there is always a possible comparison and an approximated cost for every request.

In terms of the system's architecture, this request cost estimation could be it's own instance, which would add some communication time, but would make correcting the system's learning tendencies easier by adding,removing or modifying buckets in run time and without adding more load to the Load Balancer.

Overall, even though the first requests cost estimations are not as accurate has desirable, it is constantly learning and computing more accurate results as more requests arrive to populate the buckets.

5. Load Balancing Algorithm

Regarding to the Load Balancing algorithm, we made some updates to our distributing algorithm. Now we are giving low significance, to low cost requests taking into consideration the estimation previously explained.

Below are the algorithms that we used to distribute the requests. Note: the Load Balancer has a background thread that updates the list of healthy servers

Algorithm 1: DistributeRequest

```
Input: request
Data: bestServer
requestCost = predictCost(request);
if requestCost > THRESHOLD then
    bestServer = serversLoadList.getMin();
else
    bestServer = serversThreadList.getMin();
return bestServer;
```

For a given system, there is a given amount of computation that does not need to be accounted for in terms of load balancing. This is translated in very small puzzles, or puzzles with a very low *unassigned* value, which are puzzles that are practically solved or require negligible work. For theses puzzles it is not worth going through the effort of load balancing since they will have little to no impact on the server itself and, in some cases, this logic can even take longer than actually solving the puzzle.

To identify these puzzles we analysed the EC2 instances' performance when solving several puzzles of different sizes in parallel, and compared these times with the time it takes to solve each puzzle in an instance by itself. This is relevant because EC2 instances only have one logical core. With this analysis we were able to define a threshold of estimated number of instructions for which those puzzles will have little overhead by running on servers with other puzzles being processed concurrently.

Also, even though the EC2 instances used only have access to one logical core, we can still benefit from running more than one puzzle in parallel since there should always be some downtime due to I/O operations that would be wasted otherwise.

In this updated version, the algorithm will first check the estimated cost for the incoming request. After obtaining the estimated cost, the Load Balancer will do different choices if the request is above the threshold or not. If the request is above the threshold we know that probably that request will have a high cost so we want to distribute it to the minimum loaded server. If the cost estimate is below the threshold we know that request's cost will be very small so we just distribute it to the server executing less requests at the time. If there is a server that is only processing one request, it will have priority over other in receiving these smaller puzzles. This is to make use of the property mentioned above and not waste the instance's CPU otherwise idle time.

The data structure ***serversLoadList*** presented in the algoritmh above is calculated and updated by a background thread that periodically scans the Metrics Storage System for running requests in the servers and retrieves their current metrics.

Algorithm 2: updateServersLoad

```
Data: activeServers
Data: estimation
Data: THRESHOLD
Data: loadedServers
running = MSS.getRunningPuzzles();
for server : activeServers do
    load = 0;
    for puzzle : running[server] do
        if requestCost > THRESHOLD then
            load += estimation.predict(puzzle) -
              puzzle.nrInstructions;
    end
    serversLoadList[server] = load
end
;
```

In the algorithm above we are calculating the load of each by server, by summing the estimation of instructions left by each running request. We are ignoring the low cost requests currently running.

The data structure ***serversThreadsList*** is incremented for each server whenever a request is sent and decremented when a solution is received from the server.

For predicting the cost of arriving requests the Load Balancer has a estimation model as the one described in the previous section. This model is initially trained with data retrieved from our metrics analysis to some requests (combinations of strategy=BFS,DLX,CP and size=9X9, 16X16, 25X25 and maximum number of unassigned for each size). After that, the Load Balancer will have a background thread that will be continuously updating the model. The update to the model is done by scanning the Metric Storage for recently concluded puzzles and train the model with the data. To make sure that there are no empty scans the Load Balancer just scans the MSS after receiving 3 responses from the Web Servers. Furthermore to avoid repeated scans the Load Balancer will keep in memory the last timestamp that went to the MSS and then scans for concluded puzzles with timestamps greater than that value. After the scan is completed the timestamp is updated.

Finally, the Load Balancer could be further optimized by using a cache with a LRU policy to hold a couple of responses already computed by the instances and, to avoid even more unnecessary computation, it could recognized if some server is already working on the same request for another client and using it's response.

6. Auto-Scaling Algorithm

As described in Fig1, our AutoScaler is able to communicate with both the LoadBalancer and the WebServer EC2 instances.

At startup, the AutoScaler launches instances up to a programmer defined constant which we will call MIN_AVAILABLE. This constant represents the number of available instances our AutoScaler will aim to have at every point in time, so that the LoadBalancer always an available server to forward client requests.

We opted to take this approach due to the nature of the EC2 instances and the amount of time required to launch them and for them to finish processing requests. We also defined a threshold that will limit the load of a given WebServer instance. When the expected number of instructions left to execute by a server surpasses this threshold that server is deemed *loaded*. They will still be assigned requests by the LoadBalancer if no more instances are available, but will no longer be considered available by the AutoScaler. The AutoScaler will keep track of the number of loaded instances in relation to the total number of instances and it will add/remove instances in order to maintain the MIN_AVAILABLE number of instances at all times. It is also important to mention that every time an instance is added (available) or removed the AutoScaler will inform the LoadBalancer with it's private DNS name through an HTTP request. Bellow is a rough pseudo-code representation of our algorithm to calculate the number of loaded servers:

Algorithm 3: calcLoadedServers

```
Input: activeServers
Data: estimation
Data: THRESHOLD
Data: loadedServers
estimation = updateEstimation();
AutoScaler.running = MSS.getRunningPuzzles();
for server : activeServers do
    load = 0;
    for puzzle : AutoScaler.running[server] do
        load += estimation.predict(puzzle) -
          puzzle.nrInstructions;
    end
    if load > THRESHOLD then
        loadedServers += server;
end
return loadedServers;
```

Notice the calls to *updateEstimation* and *MSS.getRunningPuzzles*. The first will make a request to the LoadBalancer and it will respond with it's up-to-date *Estimation* class instance that is able to be trained with observed values, and predict the number of instructions it takes to execute a given puzzle based on the values it has seen before.

The latter is a call to a wrapper class that interacts with the MSS (Metrics Storage System) component and, in this case, returns the currently running puzzles in the form of a map <Server, Puzzles[]>

Bellow is the algorithm we use to make scale up/down decisions:

Algorithm 4: autoScaling

```
Input: activeServers
Input: pendingServers
Data: MIN_AVAILABLE
Data: emptyServers
loadedServers = calcLoadedServers(activeServers);
if acitveServers.size() == loadedServers.size() then
    if pendingServers.isEmpty() then
        launchInstances(MIN_AVAILABLE);
else
    availableServers = loadedServers ⊕activeServers
    for server : availableServers do
        if AutoScaler.running[server].size() == 0 then
            emptyServers.push(server);
    end
    while emptyServers.size() - MIN_AVAILABLE > 0 do
        removeInstance(emptyServers.pop());
    end
```

The *launchInstance* method takes the number of instances to launch as an argument and uses the AWS SDK to launch that amount of instances, placing them in a *pendingServers* list. It will then launch a thread that waits the programmer defined grace period and then moves these instances to the *activeServers* list.

Both the *launchInstance* and *removeInstance* methods inform the LoadBalancer when an instance is added or removed.

The AutoScaler also has a background thread performing healthchecks periodically on the active instances so it can detect when a server fails and compensate for it by launching a replacement.

7. Fault-Tolerance

Since it is not possible to differentiate between a slow network connection and a failed Web Server, a perfect failure detector cannot be implemented. Therefore, a polling approach was adopted. This way, both the Load Balancer and Auto Scaler can keep track of the active servers, by sending requests to the endpoint $/status$ in intervals of 10 seconds, which is handled by a background thread present in each Web Server. So that the Auto Scaler does not try to scale down nodes that have already failed, nor will the Load Balancer forward request to such nodes.

However, there is a possibility that a server fails mid computation, resulting in the lost of the client request. To avoid this situation, the Load Balancer keeps in memory for each server what are the threads currently handling it's requests, the connection to the clients and the puzzle board for a given query, eliminating all this information after responding to client. This lets the Load Balancer redo the request, only needing to choose the server with the less load, and removing the previous entries from the MSS.

Finally, if the server is overloaded,the health check can timeout, which will result in a second request for a sanity check. If the server is still overloaded, the system will presume that the server is no longer up, and will proceed without it. A simple way to solve this would be to always try to remove the instance whenever both requests fail, this way if it is really down the Auto Scaler can catch the resulting exception, and if it was not, actually removing the instance.

8. Testing

First we wanted to confirm our Load Balancing algorithm was working as expected, so we setup an environment with 3 instances and fired various requests of different sizes at the LoadBalancer. It distributed the requests based on its estimation cost in a balanced way across all of the available instances and once they reached the AutoScaler threshold for number of loaded instances it launched more to compensate.

After these requests all finished, the AutoScaler removed the unnecessary instances until the programmer defined number of desired available instances.

After this, the tests we made revolved mainly around the Fault-Tolerance part of the system. We testing different scenarios in which an instance is suddenly terminated and everything behaved as expected.

Here are a few of the scenarios we tested:

- Two instances running with only a 25x25 puzzle in one of them. After killing that server, the LoadBalancer detected it and moved the puzzle to the other free instance.
- Three instances running, all of them containing puzzles and with only one of them not considered loaded. The other two servers had the same load which surpassed the "loaded" threshold. After killing the non-loaded instance the LoadBalancer re-distributed the puzzles equally between both the loaded servers. The AutoScaler relaunched an instance to compensate for this.

9. Appendix

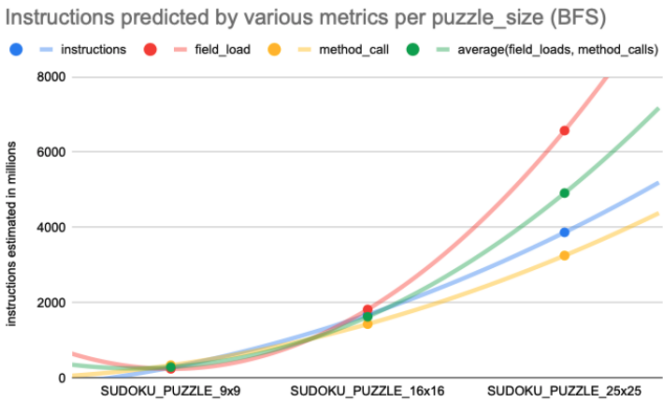

Figure 2.

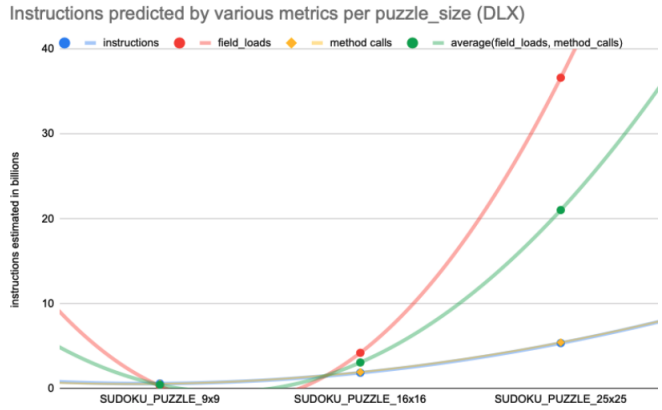

Figure 3.

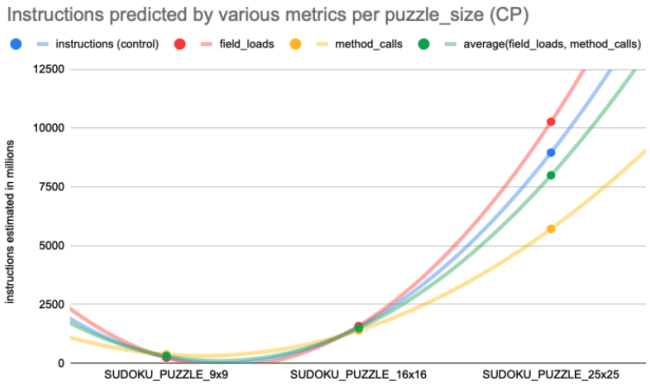

Figure 4.