Instituto Superior Técnico

Cloud Computing and Virtualization

Project Checkpoint

Sudoku@Cloud

Taguspark Group 36

Iulian Puscasu
87665

Raphaël Colcombet
81957

April 26, 2020

1 Introduction

With the goal of developing an elastic load balancing cluster of web servers, which is used to execute a computationally-intensive task, we designed an architecture that is capable of efficiently respond to a large number of concurrent requests, while also being very easy to deploy, configure, manage and destroy cloud resources by using configuration-as-code tools and principles.

2 Implementation

This project was designed to work with AWS cloud infrastructure, but the design principles can be applied for any type of system, whether it is cloud resources from another provider or private resources.

Our implementation uses a modified Amazon Linux 2 AMI as a base for all of our nodes. The AMI is modified by adding some libraries used by all the modules, adding AWS credentials and provisioning by setting the java classpath and other relevant variables. By using a base AMI for shared configuration, we can reduce the amount of EBS storage utilized. To create the AMI specific to each module, we just copy and install the relevant code module.

To automate all of this work configuring AMIs, we use Packer. Then, we use Terraform to deploy all the infrastructure. This tool uses a configuration file where we specify all the configuration options we want, including the previously created AMIs.

For the checkpoint delivery, we are using off-the-shelf Amazon AWS load balancer and auto scaler. Some non functional load balancer code is implemented. No auto balancer code is done.

3 Architecture

The system is organized into four main components:

Web Servers

The webserver simply receives a query, computes the result and returns the result. What matters here is the following:

- the queries can be very computationally intensive
- queries have a varying level of complexity
- there can be many simultaneous queries

Load Balancer

The load balancer is the entry point for the whole system, it receives queries from clients and forwards them to the most suitable server to handle that request. The load balancer is able to:

- get metrics data from the MSS
- estimate a query's complexity and duration
- know what queries each worker is computing and the details of each of those queries

With all of these functionalities, the load balancer can compute what is the best worker to forward a request to, in a specific moment in time.

Auto Scaler

The auto scaler has the ability of creating and destroying worker instances. It is constantly receiving data from the workers and if it detects that a certain metric stays above or below a certain threshhold the it will scale the size of the system accordingly.

Metrics Storage System

The metrics storage system we are using is Amazon's DynamoDB. We write the metrics to a single table, where each query corresponds to a table entry.

4 Instrumentation

Regarding our approach to instrumentation, we are counting the number of basic blocks and we apply the instrumentation to every class in the solver package. The instrumentation class we use is located in the code at instrumentation/bit-samples/**BasicBlocks.java**

We support concurrent queries by using a map where each thread has its own entry. At the start of each basic block, we increment the basic block count.

The main solver class is specially instrumented because at the end of the main solving function, we add code to write the number of basic blocks to the server. Specifically, we write the data to the LocalDatabase class, located at webserver/pt/ulisboa/tecnico/cnv/data/ **LocalDatabase.java**. The server reads the data from a map in this class afterwards.

4.1 Experimental data

Please see Table 1

The metric that correlates the most to execution time is basic blocks, with a correlation of 0.999748567031058. (Table with correlation data will be available next delivery)

All execution times across different instrumentation methods are very similar. Every test was done individually, using the t2.micro instance cpu at 100 percent usage. All execution times were considerably slower with instrumentation but different instrumentation methods only varied very slightly between themselves, even when only counting the number of methods. Therefore we will use basic blocks as a metric for server load, but we must be careful so the counter doesn't overflow.

5 Data Structures

Web Servers and Instrumentation

The webservers only store a reference an instance of the dynamoDB client (AWSDynamoDBClient). Then they access the LocalDatabase class to get instrumentation metrics. The LocalDatabase class uses a map where the key is a threadID and the value is the number of basic blocks. This is essentially the same map as the one stored in the instrumentation class BasicBlocks.

Load Balancer

The load balancer has a list of servers. For each server, we store the load, url and a map of queries.

The key to the query map is is an HttpExchange hashCode and the value consists of an object with the query itself, an estimate for that query and its approximate start time. The estimate object contains an estimate for both load and duration of a query.

Auto Scaler

The auto scaler has a list of worker nodes. For each worker, we know its current cpu and memory usage.

Metrics Storage System

What we store in DynamoDB:

- threadID
- start time in milliseconds
- elapsed time in milliseconds
- start time in a readable format
- number of basic blocks

- solving strategy
- max unassigned entries
- puzzle lines
- puzzle columns
- puzzle name

There is a java DTO (data transfer object), used to communicate with DynamoDB: DynamoMetricsItem.

6 Algorithms

Auto Scaler

```
main_func:
    for every server:
        while true:
            get cpu_usage
            get memory_usage
            wait(t)

    if (cpu_usage > threshhold):

        if (number_of_times_excedeed_cpu_threshhod > NUMBER_OF_TIMES_NEEDED)
            scaleUp()
            number_of_times_excedeed_cpu_threshhod = 0
        else
            number_of_times_excedeed_cpu_threshhod += 1

    else if (cpu_usage < threshhold and number_of_times_excedeed_threshhod > NUMBER_OF_TIMES_NEEDED)

        if (number_of_times_excedeed_memory_threshhod > NUMBER_OF_TIMES)
            scaleUp()
            number_of_times_excedeed_memory_threshhod = 0
        else
            number_of_times_excedeed_memory_threshhod += 1
```

Load Balancer

```
Estimate getEstimateFromMetrics(metrics_list):

    load = average load from metrics_list
    duration = average duration from metrics_list
    estimate = new Estimate(load, duration)
    return estimate

Estimate estimateCost( Query query ):

    metrics_list = MSS.get(query)

    if (metrics_list is empty):
        return DEFAULT_ESTIMATE

    return getEstimateFromMetrics(metrics_list)

Server getServerWithLowestLoad( Request request ):
    min_load_server = server_list.getFirst
    min_load = MAX_SERVER_LOAD

    for each server in servers_list:
        load = 0

        for each server_request in server:

            // check if request is finished, according
            start_time = server_request.start_time
            current_time = get_current_time()
            duration = server_request.estimate.duratic
            time_left = start_time + duration - curren

            if ( time_left > 0):
                load += server_request.estimate.load

        if (load < min_load_server):
            min_load = load
            min_load_server = server

    return min_load_server

void receiveRequest( Request client_request ):

    // get query
    query = client_request.getQuery()

    estimate = estimateCost(query)

    min_load_server = getServerWithLowestLoad()
```

```
// increase estimate if there already are queries running on the server
// given n = number of already running queries, the penalty increases by a factor of n squared
num_running_queries = min_load_server.requests.size()
estimate += num_running_queries * ESTIMATE_MULTI_QUERIES_PENALTY

// save request data in the load balancer
new_server_request = new ServerRequest(query, estimate, get_current_time())
min_load_server.requests.put(client_request.id, new_server_request)

// forward query to a server
response = server.url.send(query)

min_load_server.requests.delete(client_request.id)

return response
```

		Icount		StatisticsTool	-load_store	
params	time (ms)	instructions	basic blocks	methods	field load	field s
[s=BFS, un=40, n1=9, n2=9, i=SUDOKU_PUZZLE_9x9_101]	13942.664968	158517712	67931256	538	1562	7
[s=DLX, un=40, n1=9, n2=9, i=SUDOKU_PUZZLE_9x9_101]	46233.257012	532691229	226033469	3365	319702	10363
[s=CP, un=40, n1=9, n2=9, i=SUDOKU_PUZZLE_9x9_101]	10661.968626	121463643	52050945	455	1933	7
9x9 – 81						
[s=BFS, un=81, n1=9, n2=9, i=SUDOKU_PUZZLE_9x9_101]	24612.430407	279979715	119982415	750	2725	7
[s=DLX, un=81, n1=9, n2=9, i=SUDOKU_PUZZLE_9x9_101]	48034.77761	552845411	234808699	3387	319974	10504
[s=CP, un=81, n1=9, n2=9, i=SUDOKU_PUZZLE_9x9_101]	21349.97692	242918885	104099873	573	3171	7
16x16						
[s=BFS, un=256, n1=16, n2=16, i=SUDOKU_PUZZLE_16x16_01]	96864.427757	1087036028	465827028	2430	15173	7
[s=DLX, un=256, n1=16, n2=16, i=SUDOKU_PUZZLE_16x16_01]	148817.661942	1822662087	743240409	12387	4912688	34293
[s=CP, un=256, n1=16, n2=16, i=SUDOKU_PUZZLE_16x16_01]	115752.591019	1317551065	564613803	2192	22155	7

Table 1: Instrumentation data obtained from various instrumentation metrics