HillClimb@Cloud
Cloud Computing and Virtualization
MEIC-IST

83531 -Miguel Belém - miguelbelem@tecnico.ulisboa.pt
83567 - Tiago Gonçalves - tiago.miguel.c.g@tecnico.ulisboa.pt
83576 - Vítor Nunes - vitor.sobrinho.nunes@tecnico.ulisboa.pt

Abstract

This report explains the second phase of the Cloud Computing and Virtualization course project. This project makes use of code instrumentation to obtain metrics from a running worker so we can estimate the cost of a request on a Load Balancer and Auto-Scaler.

1. Introduction

This project consists on developing a Load Balancer and Auto-Scaler for a program running on Amazon Web Services that searches for the maximum value of a map using different search strategies (A*, BFS, DFS). For that, we instrumented the code so that we could produce metrics about the running instances and adapt the system to get the maximum value of the instances running in AWS at a minimum cost.

Although this project uses a specific program to load the instances its objective is to be able to generalize the procedure in a way that the Load balancer and Auto-Scaler would work even if the given program is completely different given that the code running on the instances would be correctly instrumented.

2. Architecture

We have two types of Instances. The Worker Instance, which runs the Web Server that solves the requests made by the users to the Load Balancer. The Load Balancer (Auto-Scaler and Metric Storage System) instance, which has the Load Balancer that distributes the requests from the clients to the running Worker Instances. It also implements the Auto Scaling functions, such as launching (scale-up) and terminating instances (scale-down). This instance is also runs the Web Server for the MetricStorageServer which is the point of communication with our table on DynamoDB.

2.1. Types of Instances

2.1.1 Worker Instance

We created a Linux Image based on the Linux AMI AWS - t2.micro image, which is the one eligible for free tier usage on AWS. It has a single core (up to 3.3 Ghz) and 1 GB of RAM. The Linux distribution was updated and extra packages needed for running the Web Server and BIT were installed. We load the Web Server that takes accepts requests on port 8000 and altered on-boot scripts to start up the Web Server every time that the system is booted.

We created an AMI so that we could replicate the instance and use it on a Load Balancer.

2.1.2 Manager Instance

The Manager Image has the same properties as the Workers' Image, but instead of running the Worker Web Server on boot, it executes the Load Balancer (from now on called LB) Web Server and Metric Storage Server (from now on called MSS). The LB receives its requests on port 8001 and the MSS on the port 8002. All communication between Web Servers is made using HTTP requests using different contexts to execute distinct operations.

2.2. Worker

The Worker's Web Server has 3 contexts to receive requests:

- /climb - receives the parameters for the search algorithm and runs it.
- /ping - answers to a ping request
- /progress - returns the progress of all the requests running on the machine

2.3. Manager

2.3.1 Load Balancer

2.3.2 Auto Scaler

We created an Auto Scaling Group on AWS that increases the number of instances by 1 if the average CPU utilization is over 60 % in a period of 5 minutes and decreases the number of instances by 1 (to a minimum of 1 instance) if the average CPU utilization of all instances is under 40% for a period of 5 minutes. This allows to scale up in a situation of continuous heavy load while not scaling down instantly if we get a slight pause of incoming requests.

We want to be careful shuting down instances as they take some time starting up and if we get a temporary decrease and terminate them and then we might get back to a state where we can't deal with all requests that turns to a higher response time to those requests. To avoid this we wait to see if the decrease of requests (lower CPU usage) is not temporary (hence the 5 minutes under 40% CPU load). Also if for some reason an instance becomes unhealthy (unresponsive or the Web Server crashed for some reason) then the auto-scaler will terminate that instance and start another.

2.3.3 Metric Storage Manager

3. Instrumentation

We used BIT (Bytecode Instrumenting Tool) presented in the laboratories to to alter the Java Class Byte Code of the program so we could measure the metrics that we wanted in a way that would help us decide the cost of replying to a certain request.

This metrics need to be heavily weighted as they might give a big overhead to the the original program. To store this metrics we created a class called Metrics, which stores all of the counter for each type of metric. We guarantee that each thread only counts its own metrics by using a ConcurrentHashMap to store all the metrics, where the key is the Thread ID and the Value is the Metrics Object (containing all the metrics). At the end of each request we store permanently the results of the instrumentation and remove the Metrics file from the HashMap, this allows to reset the counters for that specific thread.

With the metrics it's also stored the parameters given to program so we can associate those with the load it creates and in the future compare them with new requests.

3.1. Metrics Used

We decided to try a few simple metrics. Instructions run, Basic Blocks, Methods Called, Branches Queries, Branches Taken and Branches not taken. We ran the instrumented Web Server on an instance of AWS and made some requests (one at a time) while tracking the time it took to reply to each request. We obtained a table with all the values and concluded that some metrics grew linearly with the time to reply a request. Those metrics were Instructions Run, Basic Blocks, Branches and Branch not Taken.

Methods Called and Branch Taken were inconsistent because they depended on the search algorithm being used and do not necessarily mean a higher cost to reply to a request as different methods could have different number of instructions which could mislead the cost of the request.

As the other metrics scaled linearly with the time to reply to the request we can associate a higher number on this metrics to a more CPU intensive task. We decided to stick with Basic Blocks and Branches not Taken due to its linear grow matched with response time to the request and also because they provided the lower overhead to the running code.

3.2. Storing the metrics

3.2.1 Real Cost Calculation

3.2.2 Cache

We noticed some latency when performing multiple requests to dynamoDB, in order to avoid some of this latency, a local cache was created in MSS. This cache as N entries (for testing proposes is set for N=20, in a real system the value must be reviewed), it implements last recently used politic, so when request 21 arrives, request 1 is deleted from cache. With this cache if a request arrives to MSS and it is in cache, we do not need to consult dynamoDB and the cost is instantly returned to the Load Balancer.

3.2.3 Cost Estimation

When a new request arrives and we already have some information in dynamoDB, we will try to estimate the cost of the new one, based on the ones previously stored. To this we fetch from dynamo the closest ones. The following filters are applied to classify if a request in dynamo is close:

- Same search algorithm;
- Same image;
- The initial point must be the same with a margin of + or - 10;
- search area must be the same with a margin of + or - 2000. With all the requests matched we perform an average of their costs and now that is the estimated cost of the new request.

3.2.4 Retrieving Cost