CloudPrime
Cloud Computing and Virtualization
Checkpoint 1

Gonçalo Grazina, Isabel Costa, Samuel Gomes
Instituto Superior Técnico, MEIC/METI

1 Introduction

THE objective of this project is to develop a web server elastic cluster that performs CPU-intensive calculations. The proposed solution is implemented using Amazon Web Services in order to create a scalable service that has a minimal response time when a user queries the server for the factorization of a number.

2 System Architecture

As for now, CloudPrime is divided in three main components: *(i)* Web Server, *(ii)* Load Balancer, *(iii)* Auto-Scaler. As we can observe in figure 1, the entry point for user requests is the load balancer and it is responsible for redirecting the client request to an active web server. In the current phase of the implementation the redirection choice is the default but later it will be based on the performance metrics stored in the Metrics Storage System, which include data such as the method calls during requests and other performance data to help the Load Balancer choose the appropriate web server. The web server performs the calculations based on the user request, and replies with the result. Finally, the Auto-Scaler is in charge of retrieving the performance data and adjusting the number of active web servers. For now the Auto-Scaling group is configured using the AWS Interface, where alarms are set to react to certain events that happen in the group machines, such as CPU utilization reaching a defined maximum value.

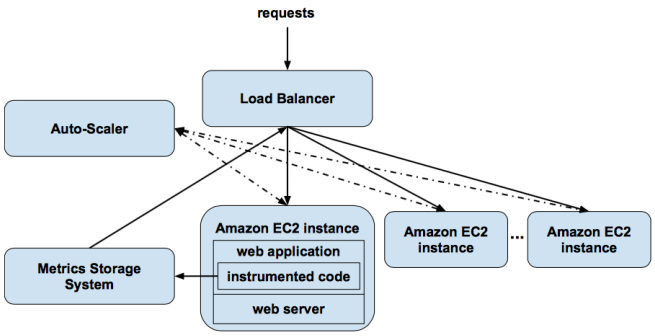

Fig 1. Architecture of CloudPrime

2.1 Solution Description

In this section we describe the Web Server implementation and the configuration of the Load Balancer and Auto Scaler.

2.1.1 Web Servers

The Web Server was developed using the Java 1.7 version and has multi-thread capability in order to receive multiple requests from clients. For each client request, the

- *Gonçalo Grazina, nr. 65970,*
  E-mail: goncalo.n.grazina@tecnico.ulisboa.pt,
- *Isabel Costa, nr. 76394,*
  E-mail: isabel.costa@tecnico.ulisboa.pt,
- *Samuel Gomes, nr. 76415,*
  E-mail: samuel.gomes@tecnico.ulisboa.pt,
  Instituto Superior Técnico, Universidade de Lisboa.

Manuscript received Month Day, 2015.

Web Server executes the Factorization program with the corresponding client query. There are two possible implementations of the Factorization class: our own version of the algorithm (*Factorize.java/FactorizeMain.java*) and the one based in the source code provided in the course webpage (*IntFactorization.java/IntFactorizationMain.java*). The Web Server can be configured to use either one of the Factorization programs. Both *Factorize* classes have two methods, which execute the main loop to add the found prime numbers and recursively find the next prime number to add to the list of prime numbers. For both algorithms the *BigInteger* number type is used since the clients can request the factorization of very large numbers that the primitive types *int* or *long* can not support.

The *WebServer.java* (based on the lab's class), is multi-threaded where each thread is responsible for creating Instances and calling the factorization methods. The queried number is extracted from the URL, where it is represented by the string */fact&x*. This string represents the Web Server context (*/fact*) and the number to be factorize, which comes after the & character.

2.1.2 Load Balancer

The Load Balancer was configured to use our Virtual Private Cloud (VPC) default subnet *(172.30.0.0/16)*, with three availability zones to reroute the client's requests to the various instances running the web servers (subnet *(172.30.0.0/24)*). The security group was configured according to the information presented in the lab class. It consists in two inbound traffic rules, one for HTTP requests on port 8000 and another for SSH in port 22.

For the listener configuration:

- **Load Balancer Protocol:** HTTP;
- **Load Balancer Port:** 80;
- **Instance Protocol:** HTTP;
- **Instance Port:** 8000;

The Load Balancer is listening to HTTP traffic on port 80 and each request is rerouted to port 8000, where the Web Servers are listening for client requests.

The health check rule is configured as follow:

- **Ping Protocol:** HTTP;
- **Ping Port:** 8000;
- **Ping Path:** "/f.html?n=49";
- **Response Timeout:** 2 seconds;
- **Health Check Interval:** 5 seconds;
- **Unhealthy Threshold:** 3;
- **Healthy Threshold:** 5;

We chose the factorization of the number 49 to the health check because it is a simple number to factorize and it is a semiprime, so the results are more predictable. Since the time for the factorization of the number 49 is fast, for the reasons mentioned above, we determine that even with a heavy workload on the instances, the factorization of this number should not take longer then 5 seconds.

2.1.3 Security group configuration

Type	Protocol	Port Range	Source
HTTP	TCP	80	0.0.0.0/0
Cust. TCP	TCP	8000	0.0.0.0/0
SSH	TCP	22	0.0.0.0/0

Table 1. Security group inbounds ports

The port 80 (HTTP) was added to the system security group inbound ports to be able to cope with the interactions from the exterior to the load balancer.

2.1.4 Auto Scaler

The auto-scaling group was configured in order to manage the numbers of Web Servers running. In the instances that run Web Servers, the CloudWatch detailed monitoring was enabled so the observed metrics (such as CPU utilization) can be analysed. To have an appropriate set of observations we ran a series of factorization requests with primes and semi-primes. Table 1 and Table 2 show the CPU utilization observed by factorizing each semi-prime number presented, using our implementation of the factorization algorithm (Table 1) and using the algorithm provided in the course webpage (Table 2). As we can observe, the factorization of larger semi-prime numbers requires higher computational power therefore the CPU utilization grows significantly.

Based on the previous observations we defined the following alarms for the auto-scaling group to take action:

prime 1	prime 2	semiprime	Aver. CPU Util.
1224545639	1961623	2402096890012097	15,34%
1224545639	17548579	21489035885096981	29,86%
1869115181	61300387	114577483942875047	63,17%
1869115181	65055721	139810538886375047	79,00%
1869115181	101000993	188782489312374733	96,17%
1969115251	151001023	297338417305901773	93,39%

Table 2. CPU Utilization with our version of the factorization algorithm

prime 1	prime 2	semiprime	Aver. CPU Util.
1224545639	1961623	2402096890012097	13,50%
1224545639	17548579	21489035885096981	46,00%
1869115181	61300387	114577483942875047	64,11%
1869115181	65055721	139810538886375047	89,17%
1869115181	101000993	188782489312374733	81,00%
1969115251	151001023	297338417305901773	100,00%

Table 3. CPU utilization with the version of the factorization algorithm provided in the course webpage

- If CPU Utilization <= 30 for 60 seconds the auto scaler removes 1 instance;
- If CPU Utilization >= 62 for 60 seconds the auto scaler adds 1 instance;

Since the growth of CPU Utilization seems to be exponential for larger number in only one request, we determined that 30% and 62% were the most appropriate values for the minimum and maximum load, respectively, for the auto scaler to remove or add instances.

3 WebServer Instrumentation

As mentioned in Section 2, in the second phase of this project the Load Balancer will redirect the client's requests to an available web server based on the MSS. These metrics are gathered during run time and have the objective of providing information to help the Load Balancer optimize the system resources by choosing the most appropriate web servers. In order to gather these metrics, we developed an instrumentation tool called FactIntr. The tool features include:

- Associates the requester thread id with the metrics gathered;
- Records the method calls from the call stack;
- Registers the number of methods, instructions and basic blocks.

The results are stored in the file *InstrumentationResults.log* stored in the instrumented-Output/log subfolder of the web server code folder, which will be accessed by the Load Balancer in the second phase of the project.

The log file includes the metrics gathered from the instrumentation tool and the number to factorize, requested by the user. An example of the log file is as follow:

————Number To be factored: 200 ————
Thread id: 13
invocation stack info:
found method in stack: recCalcFactors, ocurred 18 times.
found method in stack: calcFactors, ocurred 1 times.
found method in stack: main, ocurred 1 times.
found method in stack: ¡init¿, ocurred 1 times.
general results: 481 instructions in 195 basic blocks were executed in 21 methods.

————Number To be factored: 1050 ————
Thread id: 15
invocation stack info:
(...)

As an example, we can see from the log file that the factorization of larger numbers is more complex and the recursive function is called more often. From this information we can determine which incoming factorization requests are more CPU intensive and adjust the redirection to the most appropriate web server.

4 Conclusion

We found very interesting building this solution until now because we were able to learn aspects about the AWS that are very usefull. In the next checkpoint the most important aspect will be the coding of an improved version of the load balancer based on the Metric Storage System (that will be based on the metrics recorded to files).