CloudPrime Report

Miguel Pires
miguel.pires@tecnico.ulisboa.pt

May 2016

Abstract

CloudPrime is a prime number factorization service that employs a cluster of web servers that perform the computation. The objective of this system is to use metrics extracted through instrumentation at the servers in order to balance load evenly across all instances. Furthermore, the system also tries to scale the size of the cluster in a way that balances the objectives of minimizing latency and minimizing operating costs. To this end, the system employs intelligent load balancing and auto scaling techniques that aim to make it as efficient as possible. Additionally, the system also uses the existence of multiple machines to provide some resilience to faults.

1 Introduction

This report aims to describe the development of *CloudPrime*, an elastic cluster of web servers that provide an integer factorization service. *CloudPrime* employs web servers to perform the computation of prime components, as well as other components that perform support roles such as making the system scale and balancing the load across nodes.

This CPU-intensive operation is of particular interest since several cryptographic protocols and algorithms function on top of the assumption that a semi-prime number, that is easy to construct, is exponentially hard to deconstruct. This is particularly useful to implement trapdoor functions [5]. *CloudPrime*'s service focuses on one type of trapdoor function and, in particular, in its inverse operation, prime integer factorization. Since this inverse operation is computationally very hard, it makes sense to employ a cluster of machines to perform calculations in parallel. For this reason, the cluster must balance load in such a way that optimizes both operational costs and factoring time. As such, the two main goals of *CloudPrime* are: to scale the cluster to dimensions that allow it to perform efficient computations while striking a balance between timeliness of computation and cost of running machines; when contention is inevitable, to balance the requests across the cluster in a way that is conscious of the operation's cost and minimizes the average waiting period.

2 Architecture

CloudPrime's architecture is divided into four components:

- Web Server
- Load Balancer
- Auto-Scaler
- Metrics Storage System (MSS)

The three first components were implemented as part of the project and are included in the source code. For the MSS, Amazon's Simple Storage System (S3) was used. The Auto-Scaler and the Load Balancer were joined in one single, multi-threaded Java application that performs the responsibilities of both. These two components were implemented as one program to facilitate and optimize the management of replicas. For instance, in the case where the cluster should be scaled down, the Auto-Scaler tries

to remove a replica that currently has no requests pending. Since this replica has no requests pending and, therefore, has no load, it's highly likely that the next request the Load Balancer receives will be assigned to this instance if the replica isn't removed from the cluster in the meantime. Removing a working instance would either result in a failed request, degrading the system's availability, or in the request being forwarded to another instance, increasing the system's latency. In practice, the system would redirect the requests to another instance. To cope with the concurrent operation of the Auto-Scaler and the Load Balancer, these two entities are joined in one single application that is able to coordinate between the two logical entities without complex protocols that would be needed if the same coordination were to be done by message-passing between distributed entities. In the specific case of scaling down the cluster, if a replica with no load is found, a lock is placed on it and no requests are allowed to be assigned to it. By doing this we ensure that requests don't fail and the cluster scales down as safely as possible. This is only possible to do with such efficiency and simplicity because the coordination is local.

Since both of these entities interact frequently with the set of running instances, the management and direct interaction with the Elastic Compute Cloud (EC2) instances is mediated by an additional entity, the Instance Manager. The Instance Manager's role is to perform any operation that might affect the cluster's size or status. In the previous example, the Auto-Scaler asks the Instance Manager to remove a replica and the Load Balancer asks the same Instance Manager to assign a request to the replica with the lowest load. It is in the Instance Manager that the coordination is implemented.

3 Auto-Scaler

The Auto-Scaler's purpose is to scale the cluster's size accordingly to its current load, striking a balance between minimizing contention and, therefore, minimizing request latency, and minimizing server running time which in its turn would minimize operational costs.

The Auto-Scaler constantly monitors the cluster's status in order to react quickly to events that could jeopardize the availability or quality of the provided service. With this in mind, the Auto-Scaler's operation consists in monitoring the number of available or pending instances, scaling the cluster up or down based on the minimum and maximum number of instances compared to the actual cluster size, updating the instances' state and acting in accordance to those changes (e.g., a pending instance may now be running, a running instance may have stopped or been rebooted by the Health Check, etc), printing status information about the cluster, requesting CloudWatch monitoring data, ordering and parsing that data to produce a historic of individual instance load and calculating the cluster's load. The cluster's load is an average of the individual CPU load of each instance, itself averaged over a sliding time window of customizable length.

In addition to the Auto-Scaler's responsibilities mentioned above, each instance is assigned a Health Check that periodically asks the web server to factor the number 9 and expects "3, 3." as a response. This feature allows the system to deal with instance-level failures in a way that doesn't affect the whole cluster. If the Health Check detects an application-level failure such as a failure to respond, indicating that the application could have crashed, or an incorrect response, indicating Byzantine behaviour, it reboots the instance. As is explained in Section 4, this doesn't mean that the requests running in the instance will fail.

The main parameters (and their default values) of the Auto-Scaler are:

- Minimum instances - 1 instance
- Maximum instances - 5 instances
- Maximum cluster load - 80
- Minimum cluster load - 30
- Auto-Scaler's check period - how often the Auto-Scaler should poll CloudWatch to obtain metrics and update the system's status - 4 seconds

- Cooldown period - how soon after a scaling activity occurs, can another scaling activity take place - 180 seconds
- Analysis time window - how many minutes into take past should be taken into account when calculating an instance's average CPU load - 1 minutes
- Grace period - how much time should be allowed for an instance to boot - 125 seconds
- Healthy threshold - how many times the Health Check can fail before the instance is rebooted - 2
- Health Check period - how often the Health Check pings the instance - 5 seconds

Every parameter is customizable at compile-time. In the case of the *analysis time window*, if x minutes are specified as the desired time window length and only a smaller amount of data points are available, then the system will use as much as possible.

It is also worth mentioning, that every operation is performed in a way that minimizes latency and tries to keep quality of service high. For instance, when the Auto-Scaler tries to remove multiple instances, the Instance Manager will try to remove only instances that are idle and only if that is impossible will the system remove instances with running requests. Once more, this doesn't mean those requests will fail.

4 Fault Tolerance

Another way that *CloudPrime's* cluster improves availability and quality of service is through fault tolerance. The system implements mechanisms that ensure as much availability as possible. For instance, if the application crashes, the system is able to detect that crash and resend the requests that were pending in that instance to other servers while, at the same time, rebooting the faulty instance. This ensures that, even in the presence of a fault, the client will receive a correct response. Another way that *CloudPrime* strives to improve availability is, as was already mentioned, when scaling down the cluster. When the Auto-Scaler scales the cluster down it first tries to remove only instances that have no pending requests and only if that is impossible the system will remove working instances. However, even if the system removes working instances it will also reroute the pending requests of those instances to other servers, therefore ensuring that system is available and every client receives a response to his requests.

5 Load Balancer

The expected behaviour for the Load Balancer is to deal with concurrent requests in such a way that on average every server experiences the same load and every request receives a fair amount of computation time from the cluster. If one server has too many requests executing concurrently then those requests are going to take very long to complete, degrading the quality of service for a particular client.

The Load Balancer serves as the entry point for the system. Clients issue requests to the Load Balancer's address who is responsible for finding the best server to which the request should be forwarded and then forwarding the response back to the client. To do this, the Load Balancer interacts with the Instance Manager to obtain the best match for a request and store the request's input number. The input number of the factorization must be stored to allow subsequent executions of the Load Balancer to make estimations and predictions based on the requests that are currently being computed at an instance. In order to balance the system's load effectively, the Load Balancer must be able to tell which servers are more overloaded and which ones are not. Several algorithms were implemented to this effect and those are discussed in Section 6.

6 Algorithm

6.1 Load Balancing Algorithms

The Load Balancer makes use of an algorithm that chooses the least overloaded server to fulfill a certain request. Various algorithms were implemented during the development of *CloudPrime*:

- Round-robin
- Cost based
- Time based
- Cost based and time discounted

The first algorithm was implemented for the first checkpoint and simply returns the instance that is stored after the previously returned one. This algorithm doesn't estimate cost or apply any sort of logic or insight about the problem and, therefore, is the least interesting.

The second algorithm considers every instance and estimates the cost of the requests that are currently being executed in it. The cost of a request is estimated by taking the input number and using the cost model to predict it's cost. This is done for every request running on an instance and the instance's load is estimated to be the sum of every individual cost. The algorithm returns the server with the smallest cost.

The third algorithm implements the same logic as the second one but uses a time based model instead of cost based. Essentially it assigns requests to the server whose requests sum up to the lowest amount of time.

The fourth algorithm combines the cost model and the time model to leverage as much information as possible about the requests. This algorithm is based on the second one but uses predictive information about much estimated time a request has left to determine if it should be ignore or not. This could prove useful if one server is currently running requests with a high cost but that are close to completion. In this case, it would be interesting to ignore the cost of that request since it's nearing completion.

In Subsection 6.2, the theory behind the usage of the prediction model is presented and, in Section 7 the results are presented as well as the optimizations that derived from their analysis.

6.2 Predictive Model

As previously mentioned, the system implements a model-based prediction algorithm that accumulates data points and is able to estimate the cost of a request based on it's input number. The model is implemented by a custom-made linear regression library built specifically to support the Big Integer Java class. The available methods of this library include methods to add data points, extract predictions for an input value, calculate errors and calculate the determination coefficient [2], R^2, where R is the correlation coefficient (also known as Pearson's R). R^2 is measured from 0 to 1 and is an indication of how much of the dependent variable's (Y) variance can be explained by the independent variable (X).

The cost that is being modeled is obtained through metrics that are extracted at the instances by instrumenting the web server's code and are stored in the MSS. The Metrics Updater entity at the Auto-Scaler/Load Balancer polls the MSS for new files, parses the files and feeds the data to the models. Every file is deleted so as not to be inserted in the model multiple times. The extracted metrics are:

- Maximum Stack Depth
- Byte Code Executed
- Function Calls

The three algorithms that were based on the predictive model use the same custom-made linear regression library. Their only difference is the dependent variable that is being predicted. The *maximum stack depth* comes from the intuition that it will be as large as there are iterations of the factorization algorithm since it is recursive. The *byte code executed* follows the intuition that it is more or less a direct metric of the time spent executing code. The fact that some instructions are more costly than other is what motivated the next metric. The *functions calls* come from the fact that most of the work is being performed by the computationally expensive BigInteger class methods[1] and, therefore, those functions calls will contribute more to the cost of the factorization.

7 Evaluation

For the evaluation of *CloudPrime*, each web server was executed on a t2.micro AWS instance running

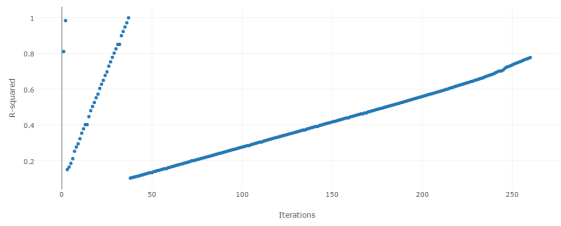

Figure 1: The evolution of R^2 through time

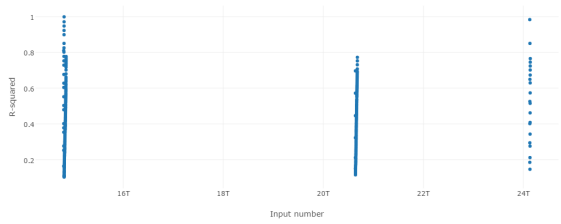

Figure 2: R^2 grouped by input number

Amazon Linux with 1 virtual CPU and 1 GB of RAM. Both the Load Balancer and the clients were executed locally in a machine running Windows 10 with a 2.2 GHz Intel Core i7 processor and 8 GB of RAM.

7.1 Model Evaluation

The results of a preliminary experiment with multiple clients can be viewed in Figure 1[1]. These results may look encouraging at first, since there is a clear increase of R^2 over time and, therefore, a strong upwards trend in the correlation of the two variables. This is easy to explain since the model incorporates new data as time passes and it's predictions become more accurate. However, by examining Figure 2 we can see that R^2 grows locally and so the model is learn in a localized way. For instance, after seeing inputs numbered 2, 4, 7 and 10, the model's prediction for the number would be very precise but for the number 20 the model would be inaccurate. This means that the algorithm becomes very accurate in predicting similar inputs but can't extrapolate when values start to vary considerably. More precisely, the model can accurately predict the cost of a request that is similar to requests it has seen before. However, it fails to extrapolate to a different order of magnitude or even to different requests of the same order of magnitude when number become very high. The variability of the inputted numbers is clearly a problem because the groups of numbers are so distinct that the model can't relate them to one another. Each group of numbers is the result of a single client even though the clients generate semi-primes of similar magnitude and, as such, don't differ significantly in factoring time. Bear in mind that these numbers are semi-primes generated from two prime numbers whose bit length begins at 20 and 25, respectively. For this reason, the length and, therefore, the order of magnitude of these groups of numbers is the same and grows at the same. Upon close inspection of Figure 2, you can see that the second group of numbers is close to 22 trillion and the third group is closer to 24 trillion.[2] In fact, these numbers are of the same magnitude yet the model can't cope with the difference in values.

There were two obvious improvements after seeing these results. The first is simply making the web server ignore the metrics derived from the Health Check's requests. These requests factor the number 9 and, as can be seen in the left column of Figure 2, make up the majority of data points that are fed to the model and are very distant from the requests we are trying to predict. Ignoring these requests will potentially ignore a lot bias in the model. The second improvement was to generate data points that are distant in terms of order of magnitude to allow the model to extrapolate new cost values from previously observed ones. In order to do this, the MSS was expanded to contain two additional repositories where training data would be accumulated and read at the beginning of the Load Balancer's execution. This would provide the algorithm with a basis for it's predictions and hopefully lead to better results.

[1]The results were derived used the cost based algorithm since the mixed algorithm was only developed later

[2]In the American short scale of orders of magnitude

One of the two additional repositories in the MSS stores persistent metrics about the cost of requests and the other one stores persistent data about the time of execution of requests.

Figure 3 shows the result of running clients whose requests vary considerably in orders of magnitude and quickly become very expensive to compute. This experiment should tell us whether the predictive model is able to extrapolate from it's known data points. As can be seem in Figure 3, the results begin at a much higher rate and stay at high levels even when the order of magnitude causes the execution time to grow. However, there is a point where the determination coefficient starts to fall and indicates a lack of quality in the adjustment of the model to the data. This can be explained by the fact that the requests that provoke the sharp drop in R^2 are computationally expensive and take more than 10 minutes to complete. Since the execution time for these requests is so high, it's difficult to train the model for them and the predictions undershoot reality. Without data points for higher orders of magnitude, the model tries to predict a linear growth and for very large orders of magnitude the growth in execution time isn't linear but exponential. It's worth mentioning that further testing, done after the system had accumulate over 300 data points, showed a significant improvement in large numbers (ranging from values of 0.87 to values as high as 0.97). These results are not included due to lack of space.

The cost based and time discounted algorithm was designed after these tests and the training data set for the time model is less complete than the data set for the cost model. However, since the time model and the time predictions are used to complement the cost estimate, the algorithm is expected to have a comparable performance. It is possible that, since the data points regarding the latency of requests started being collected at a later date, the insufficiency of data might cause problems for the accuracy of time predictions.

7.2 Overall Evaluation

In this subsection, the quality of the solution will be evaluated without looking at specifics of the model and the predictive algorithm. We will look at system traces to examine the instances' individual load over time and to determine if the load is being effectively balanced across the instances of the cluster. For each of the presented figures, the actual system traces are included as an appendix.

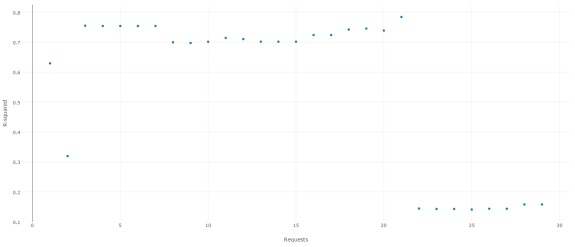

Figure 3: The evolution of R^2 through time

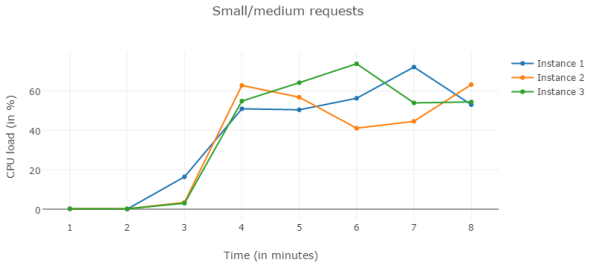

Figure 4: System load due to small/medium requests

Figure 4 represents a system trace where multiple clients placed requests for factorizations in parallel and each requests took from 2 seconds up to 6 seconds. As the figure clearly demonstrates, the system load increases similarly across all instances and even though some instances may have a larger load than others at a given point in time, that is quickly compensated. In the majority of data points extracted, the difference between the CPU load of the three replicas is about 10 - 15%.

Figure 5 illustrates a system trace where multiple clients place requests for factorizations in parallel. In this case, the running time of each request ranged from the tens of seconds to several minutes. It's in-

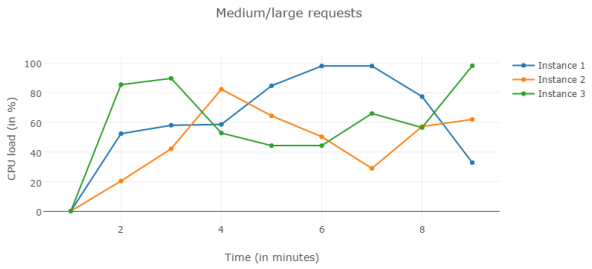

Figure 5: System load due to medium/large requests

teresting to see how the system reacts to more complicated requests because, since these requests run for longer they are also less granular. If the clients issue 100 requests that each take 1 second to execute, this is clearly easier to distribute than 2 requests that take 50 seconds each to execute. In this example, two replicas would carry the entire burden of the computation while any other replica would be idle. In fact, by examining Figure 5 we notice a much bigger variance when compared to Figure 4. However, despite the bigger variance, no instance is consistently more overloaded than the other. Even when the first instance carries a higher load for a while, after a few minutes some other instance takes its place. This can be easily explained by the existence of long running requests that raise the CPU load to around 90 - 100%. These longer requests raise an instance's CPU load to higher levels than multiple, smaller requests because more requests also imply more time spent on I/O operations and, therefore, less CPU computation. Therefore, we can safely conclude that in terms of end-user quality of service and cluster load, the load of computation is effectively balanced across all instances of the cluster.

8 Conclusion

Regarding the quality of the algorithm and its results, the results were positive since the model achieved a positive and usually high correlation. Effectively estimating the cost of prime factorizations proved to be a difficult challenge since this calculation is subject to significant change even when provoked by small changes in the input. Possibly, consistently excellent results could have been achieved if the algorithm were allowed to run for a longer time and if the models had more data points to work with. Unfortunately, the request limit for S3 in the free tier was reached and further training would have meant incurring in financial debt with Amazon. Another limitation that hampered the effectiveness of the model was that larger requests took very long to complete (the two biggest requests for factorization were around 10 minutes and 1 hour) and, therefore, data points associated with larger input numbers were scarce. Other techniques such as neural networks[4] or Markov Chain Monte Carlo (MCMC)[3] could be used to try to approximate the data points to a mathematical function but, not only would that require a very large amount of data points, the chances of actually finding a better prediction model for this problem could be still be slim.

Independently of the model's functioning and possible limitations, the fact that the CPU loads at each server were so similar, even when under contention and using long running requests, showed that the load was balanced as was intended and no server was unjustly overloaded with requests.

References

[1] BigInteger - divide. `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/math/MutableBigInteger.java#MutableBigInteger.divide%28java.math.MutableBigInteger%2Cjava.math.MutableBigInteger%2Cjava.math.MutableBigInteger%29`. [Online; accessed 09-May-2016].

[2] Determination Coefficient. `https://www.wikiwand.com/en/Coefficient_of_determination`. [Online; accessed 07-May-2016].

[3] Markov Chain Monte Carlo. `https://www.wikiwand.com/en/Markov_chain_Monte_Carlo`. [Online; accessed 07-May-2016].

[4] Neural Networks. https://www.wikiwand.com/en/Artificial_neural_network. [Online; accessed 07-May-2016].

[5] Trapdoor function. https://www.wikiwand.com/en/Trapdoor_function. [Online; accessed 07-May-2016].

Appendix

System Trace - Small/Medium requests

```
###     STATUS   ###
#       Ids:
#       i-13fc8c99
#       i-14fc8c9e
#       i-15fc8c9f
#       Available replicas: 3
#       Pending replicas: 0

#       Points:  0.16% 0.0% 16.39% 50.85% 50.33% 56.17% 72.0% 52.95%
#       Average CPU load (last 1 minutes): 52.95%
#       Points:  0.17% 0.17% 3.44% 62.71% 56.72% 41.02% 44.5% 63.11%
#       Average CPU load (last 1 minutes): 63.11%
#       Points:  0.17% 0.17% 3.0% 54.75% 64.07% 73.67% 53.83% 54.33%
#       Average CPU load (last 1 minutes): 54.33%
#       Cluster load: 56.796665%
########
```

System Trace - Medium/Large requests

```
###     STATUS   ###
#       Ids:
#       i-c8c4b442
#       i-c9c4b443
#       i-c6c4b44c
#       Available replicas: 3
#       Pending replicas: 0

#       Points:  0.16% 52.37% 58.03% 58.64% 84.67% 98.0% 98.0% 77.38% 32.88%
#       Average CPU load (last 1 minutes): 32.88%
#       Points:  0.16% 20.49% 42.2% 82.33% 64.43% 50.33% 28.98% 57.21% 62.0%
#       Average CPU load (last 1 minutes): 62.0%
#       Points:  0.17% 85.42% 89.67% 52.83% 44.33% 44.33% 66.0% 56.5% 98.17%
#       Average CPU load (last 1 minutes): 98.17%
#       Cluster load: 64.35%
########
```