An Algorithm Based on Response Time and Traffic Demands to Scale Containers on a Cloud Computing System

Marcelo Cerqueira de Abranches
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Distrito Federal (61) 3107-6737/3658
Controladoria-Geral da União
Federal Government of Brazil
Brasília, Distrito Federal (61) 2020-6964
Email: marcelo.abranches@cgu.gov.br

Priscila Solis
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Distrito Federal (5561) 3107-6737/3658
Email: pris@cic.unb.br

***Abstract*—This paper proposes a cloud computing architecture based in containers and on an algorithm that intends to achieve an efficient allocation of processing resources to comply with response time requirements in a Web system. The algorithm is based on the characterization of web requests and on a PID (Proportional - Integral- Derivative) controller. The proposal was evaluated with a real time series obtained from an operational massive web system in a controlled infrastructure. The results show that the proposal achieves the expected response times allocating a lower number of containers than other related proposals.**

I. Introduction

Some of the most challenging and interesting topics on cloud computing environments are auto elasticity algorithms [13] and load balancing procedures. Several recent works address elasticity in cloud computing environments [13], [11], [5] [7]. Elasticity is a key feature in the cloud computing area and is the main characteristic that distinguishes this computing paradigm from the other ones such as grid computing or cluster computing.

The scalability describes the systems ability to reach a certain scale. Is the ability of the system to be enlarged as necessary, mainly to accommodate future growth adding more resources. Elasticity is a dynamic property that allows the system to scale on-demand in an operational system. Elasticity is the ability for clients to quickly request, receive, and release, many resources as needed. The elasticity implies in fluctuations, i.e., the number of resources used by a client may change over time.

The policy to implement elasticity can be manual or automatic. A manual policy means that the user is responsible for monitoring his virtual environment and applications and for performing all elasticity actions. Normally, the cloud provider provides interfaces with the system with this purpose. In automatic policy, the control is done by the cloud system, in accordance with user requirements, normally specified in the Service Level Agreement (SLA). Then, auto elastic means to automatically adapt the environment, and even optimize resources according to the user demands.

This work proposes an algorithm and an architecture to promote auto elasticity on a cloud computing environment based on the efficient allocation of resources. Our work is focused on processing power elasticity.

This paper makes the following contributions: first, we propose a cloud computing architecture, integrating several technologies to promote auto elasticity. Secondly, we analyse and characterize the web requests of a massive web system and propose an algorithm that using this typical workload allows to allocate processing resources to comply with QoS requirements, in our case, the response time. And finally, this paper evaluates the proposal in an experimental environment using several scenarios.

This work is organized as follows: section 2 presents the related work. Section 3 contains the literature review and the theoretical concepts used in our proposal. Section 4 describes the tools and technologies used in the proposed architecture. Also this section details the proposed auto scaling algorithm. Section 5 presents the experimental results. Finally, section 6 presents the conclusions and future work of this research.

II. Related Work

The work [11], compares different methods to obtain auto elasticity on a cloud computing environment. These methods can be classified into 2 categories: reactives and predictives. Those techniques are based on machine learning, queueing theory, control theory, temporal series analysis, among others.

The work [5] proposes an algorithm called PRESS to predict CPU loads by extracting consumption patterns and adjust resource allocation. Their approach uses two methods to perform online predictions: the first is based on the use of signal processing (Fast Fourier Transforms -FFT) to extract dominant frequencies. This frequencies are used to generate a time series and different time windows are compared. The Pearson correlation index is generated for various windows. If

978-1-5090-3216-7/16/$31.00 ©2016 IEEE

it is obtained a Pearson correlation index greater than 0.85, the average value of the resources in each position of the time series is used to generate a forecast to the next window and the virtual machine's resources are adjusted. If a pattern is not identified, they propose an approach that uses a Markov chain with finite number of states to perform the forecast.

Another work that proposes auto elasticity is Haven [13]. This proposal is based on monitoring CPU and memory loads for each virtual machine in a load balancing pool. From CPU and memory thresholds previously established, Haven loads new virtual machines and insert them in the load balancing pool. In addition, the load balancer which is implemented with SDN (Software Defined Network), directs each request to the least loaded member of the pool.

The work [7] proposes and provides a tool called HPA (Horizontal Pod Autoscaler). This tool works by scaling the environment (number of containers) from CPU average thresholds of containers that serve a given application.

Our proposal is different from PRESS, as it does not perform load prediction, but rather, performs resource allocation or deallocation, based on the response time observed from an application behind a load balancer, prior to a workload characterization. Our proposal is reactive since it uses the variation of the average response time of an application to decide using control theory the allocation or deallocation of resources. Also, our proposal is different from PRESS that performs vertical scaling. Our proposal performs horizontal scalability, that is, new instances are allocated behind a load balancer. As our proposal, Haven also performs horizontal scalability but with virtual machines and our proposal uses containers [2].

Another difference of our proposal regarding to HPA is that while HPA performs CPU consumption measurements inside containers, our proposal performs measurements outside of containers, i. e., in the load balancer. This approach has the advantage to watch system performance, regardless of the level of resource utilization of containers. In Section 5 a comparison is made between the HPA and the solution proposed in this paper .

III. Theoretical Concepts and Literature Review

A. Containers

Container is a technology for creation of isolated processing instances and enables virtualization at the operating system level to provide protected processing portions and when running on the same system, they are not aware that are sharing resources as each one has its own network abstraction layer, memory and processes[2].

Containers have a great portability, because they can run on any operating system based on Linux. Virtualization depends on a hypervisor to achieve similar portability. Virtualization via hypervisors consumes more resources than containers. If a container is not performing any task, it is not consuming resources on the server [2]. Besides that, containers are very dynamic to be created and destroyed, as they just have to start or destroy processes in its isolated space.

B. Web Servers and Load Balancers

In our proposal, the infrastructure hosting the web system must be prepared to meet the demand with high performance, scalability and high availability. However there are many challenges to be addressed so that a cluster of distributed servers can function efficiently as if it were a single server. These challenges range from the routing of requests to the members of the cluster, methods for choosing the member to receive the workload and methods to maintain the connection status.[4].

In load balancing at the transport layer, the requests are distributed among the members of the cluster based on informations like IP addresses and ports. The load balancer distributes client connections, which must know the IP address cluster, among the various servers that effectively respond the requests. In this case, as the load balancing process is based on layer 4, the server is selected regardless of the content or the type of request. [1]. When load balancers are in layer 5, the distribution of workloads is based on the contents of the requests.

C. PID Controllers

PID controllers (Proportional - Integral- Derivative) are control algorithms that are widely used in industry. Examples of its applications are temperature control environments, and drone control. PID controllers have three coefficients : proportional , integral and derivative. These coefficients are varied in order to obtain the desired optimum control response for a given process.

A PID controller works in a closed loop system where it is possible to read the current state of a particular variable that is being controlled, and according to its value, an action is performed so that the variable of interest converges and remains on a desired level (even considering external disturbances), for the next iterations of time. [8]

Thus, the PID controller should read the current state of the variable and calculate the output response, by calculating the proportional, integral and derivative components and then adding the three for calculating the control output. The proportional component depends on the difference between the desired value (*setpoint*) and the current value of the variable. This difference is referred to as an error. The integral component adds the error term over time. The derivative response is proportional to the rate of change of the process variable.

In order to produce the necessary adjustments to the system, the PID controllers use gain parameters K_p K_i and K_d that should be adjusted. There are several methods for adjusting these parameters, such as the manual method (guess and check)) and the Ziegler -Nichols method [8].

In the manual method, the gains of each component are adjusted using trial and error. For this, the effects that each parameter causes the controller output must be known. In this method , the terms K_i and K_d are set to zero, and the term K_p is increased until the cycle output starts to oscillate. From there, the term K_i should be slowly raised to reduce the steady error. At this point the term K_d is incremented, in order to decrease the oscillations at the cycle output. The discussion on other methods of parameter adjustment is beyond the scope of this paper, since we used the manual adjustment method.

IV. The Proposed Solution

Our proposal is based on a cloud computing environment based on containers and a method of auto elasticity to comply with a required system response time. For this purpose, we use a closed loop system with a PID controller, which responds to changes to system response time by increasing or decreasing the number of containers in a load balancing cluster that process web requests.

In the next subsections, we describe the tools that were integrated and the implementation of the algorithm to provide the features that enable our proposal.

A. Docker

Docker started as a project of the PaaS company (Platform as a Service) dotCloud in 2013 [12] proposing to be an integrator and facilitator for adoption of containers in production environments and in large scale. Docker uses kernel features to isolate the containers from the server, creating isolated processes, network and privileges. The limitation and accounting of resources (CPU, memory, disk space and I/O) is made through the use of cgroups. Also the use of the file system is done efficiently because it is based on copy-on-write, which allows changes to a container to be simply a differential update of the previous image.

One of the greatest advantages of Docker is the ability to find, download and start images of containers that were created by other developers very quickly and conveniently.

B. Kubernetes

Kubernetes is a system developed by Google [6], and made available to the community, which aims to manage the life cycle of containers in the nodes of a cluster. Thus, Kubernetes is an orchestrator of containers, being able to schedule the launch of containers between the nodes of a cluster, to do admission control of containers, resource balancing and provides scalability to the environment. Kubernetes also provides features such as service discovery between containers, service publication for access from outside the cluster and load balancing between containers [6].

The infrastructure of a Kubernetes cluster is composed of master nodes that control worker nodes, which run the containers. All settings of the cluster are stored in a distributed configuration repository, called Etcd. PODs are the basic unit within Kubernetes. Containers are grouped in PODs and these generally represent an application. These are created using Replication Controllers which are used to define PODs that can be scaled horizontally. Replication Controllers are also responsible for maintaining the desired number of PODs active in a cluster.

C. Apache Spark, Flume, HAproxy and Redis

Apache Spark is a distributed processing tool, ideal for processing large databases. It was developed by AMPLab (UC Berkeley) and performs data processing in memory by default.

Its basic structure of abstraction are the RDDS (Resilient Distributed DataSets), which are collections of elements that can undergo operations in parallel, making it possible to generate new RDDS from transformations such as map, reduce, filter and join on RDDS [16]. This tool was chosen to integrate the solution because it allows the solution to perform processing of large databases in text format, in a scalable way.

Apache Spark offers an API called Spark Streaming, which allows real-time data processing, through the creation of structures called DStreams (discretized streams), which are sequences of RDDs. The creation of DStreams is made by the StreamingContext class where you can configure the duration of each window of DStreams [16]. In our proposal, the duration of each window was set to 5 seconds and the motivations for this are further detailed in the next section.

In our proposal, the Spark Streaming is used to process the load balancer logs, collecting the response time of each of the requests that the system serves, in real time. This response time information is stored in a time series format on a Redis server, so that it can be used by the auto scaling algorithm proposed in this article.

Spark receives the log entries of the load balancer in text format, using the Flume tool. Flume is a service that aggregates, collects and moves large volumes of data flow. For its operation it creates a source that receives the data of interest. This source is connected to a channel, where the data will travel toward a sink [3].

Therefore, the function of this tool in the solution, is to send, in real time, the load balancer access logs to Spark, through the creation of a source of type Syslog, which receivers the Load Balancer logs, and a memory channel that carries the source data in memory. From there, Spark consumes this data stream through an Avro Sink, which travels over the network.

The load balancer used in the solution is HAproxy, which can act as a layer 4 or 7 load balancer, SSL terminator, reverse proxy and other [17]. Currently, this is the load balancer used by web sites as Reddit, Stack Overflow, Server Fault, Instagram among others, and has been chosen as the cloud load balancer of Red Hat's OpenShift. HAProxy has the following log format:

May 18 06:24:25 10.125.7.229 haproxy[1078]: 10.125.8.252:43839 [18/May/2016:06:24:24.988] cherrypy cherrypy/10.125.7.227 0/0/2/26/28 200 169 - - —- 1/1/1/0/0 0/0 "GET /generate HTTP/1.0"

In the above line, there are informations such as the waiting time in queue at the application server, *http* method and response code, among others. The part with *0/0/2/26/28* contains the information: *Tq '/' Tw '/' Tc '/' Tr '/' Tt*, where *Tr* is the time in milliseconds that the load balancer waits until it receives a complete response of a web request to the server [17]. So this represents the total time of the request processing by the container.

In our proposal, the SparkStreaming processes the log entries and separates the field *Tr* and stores it in the Redis database, in a time series format. SparkStreaming is also responsible for converting the format of the date of each line to the number of seconds from 0 hour of every day, to support the creation of time series.

Redis is used to store the time series, because it can provide low latency both for writing and reading, as it keeps the data as memory structures [14]. The integration of the solution with Redis is made through the use of the Kairos library, which creates a structure for storing time series in databases such as Redis, Mongo, SQL or Cassandra [9]. This library provides features, such as setting the number of entries to keep in the database and the minimum time unit of interest of a series. In the case of this work we keep stored in the series data of the last 600 seconds, which is sufficient for the algorithm operation.

Kairos also allows the calculation of statistical parameters of the series, using configurable time windows. This allows, for example, to compute average response time of an application in the last two minutes. The minimum unit of time set in this solution is 1 second. This allows good flexibility for configuring the time windows for statistical calculations and enables the generation of graphics with good time resolution for monitoring and evaluation of the solution.

D. The Cloud Architecture

The proposed architecture is shown in Figure 1 and described in the PAS pseudocode in this section. The PID controller is used to maintain the average response time of a particular application within a certain threshold. Our proposed architecture, hereby called as PAS (PID based Autoscaler) operates based on the following sequence:

1) Establishment of an average time threshold (setpoint) desired to the system to answer requests. The monitor receives the response time of the requests arriving at the load balancer;
2) The monitor sends the average response time (the average of the last 200 seconds) so that the PAS algorithm calculates the number of containers needed to reach the setpoint. The average of the last 200 seconds was defined because it was found experimentally that this value is adequate since avoids that outlier values influence on the operation of the system.
3) PAS runs algorithm 1 and inform the desired number of containers to Kubernetes. The current number of containers is obtained using a Kubernetes tool called kubectl, and the average response time of the cluster is determined using the time series present in the Redis database.
4) Kubernetes creates, mantains, or removes new containers, and ensures that the environment will remain with the desired number of containers until the next round of the algorithm (in this case, after 10 seconds). This value of 10 seconds was chosen because it was found that it is sufficient for Kubernetes to load new containers.

E. Solution Operation

In order to allow the operation of the algorithm which works with dynamic data received in real time, we use the processing flow shown in Figure 2. Haproxy acts as the load balancer of the solution. Its logs are sent to Flume that puts the data in a memory channel and sends it to Spark Streaming, thus providing the information necessary for operation of the algorithm.

Algoritmo 1: PAS

```
Input: Average response time of the cluster, Current
       number of containers
Output: Desired number of containers
1  begin
2      Read the desired threshold of average response time
         of the requests: $t\_ms\_desired$
3      Read the current number of containers:
         $n\_containers\_current$
4      Read the average response of the cluster in ms:
         $t\_ms\_current$
5      Calculate the error: $e(t)$ = $t\_ms\_desired$ -
         $t\_ms\_current$
6      Calculate the PID output:
7
         $u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t)$   (1)
8      $n\_desired\_containers$=$n\_containers\_currrent$ +
         $u(t)$
9  end
10 return $n\_desired\_containers$
```

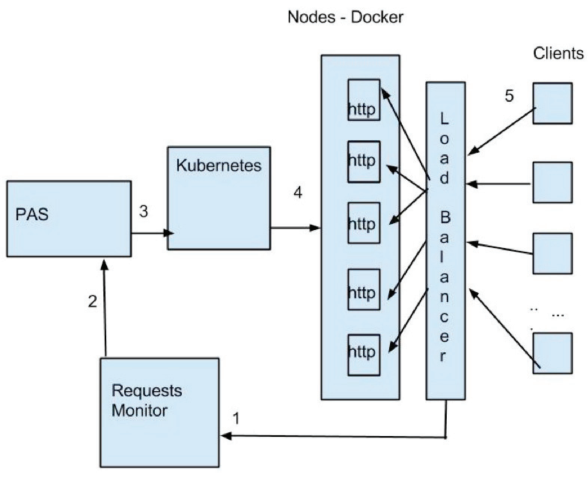

Figure 1. PAS Architecture

Haproxy balances the requests in round robin mode between the Docker/Kubernetes nodes hosting the containers. When the Docker/Kubernetes node receives the request, the Kubernetes proxy performs load balancing between the containers of each server.

So, two levels of load balancing are performed, one between the Docker/Kubernetes nodes, where Haproxy performs the load balancing, and other internally within the Docker/Kubernetes nodes where the service Kubernetes proxy performs the load balancing.

We have developed in this research a set of specific codes to customize the interaction between the tools, for example, to generate the time series with system response times and for the creation and destruction of containers, among others.

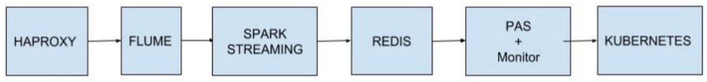

Figure 2. Processing Flow Between the Set of Tools

V. Experimental Results

A. Environment and Evaluation Scenarios

To evaluate the solution we configured a Kubernetes v1.1.2 cluster on the top of Coreos (899.6.0 (2016-02-02)) operating system, virtualised with VMWare ESXi 5.5.0. This environment was configured for solution validation. A production environment could benefit more if the system Coreos was installed directly on physical machines because the virtualization layer would be eliminated.

The cluster was built with the following components: 1 master node (4 vCPUs, 6 GB of RAM) , 1 Etcd node (4 vCPUs, 6 GB of RAM) and 4 worker nodes (4 vCPUs, 6 GB of RAM). Ubuntu 14.04.3 LTS Virtual machines (VMWare ESXi 5.5.0), with the following settings and tools: 1 haproxy 1.5.4 node (4 vCPUs , 4G GB of RAM) , 1 Spark 1.5.2 + Redis 2.8.4 node (2 vCPU 10 GB of RAM) and 1 Flume 1.7.0 + PAS node (2 vCPU , 4 GB RAM)

We defined 4 evaluation scenarios. For the scenarios 1, 2 and 3, we generated an image of a container that runs the web server Cherrypy 5.1.0. We configured a link on this server. The link generates in each request an array of random size between 1,000 and 10,000 elements.

The service publication in Kubernetes was made through the creation of a Replicaction Controller and the configuration of a service of the type NodePort. The HAProxy load balancer was configured to balance requests between the Docker/Kubernetes nodes using the IP addresses of the nodes and ports published by the service of the type NodePort. Each container had its processing and memory resources limited to 18 MB of RAM and 24 millicores of CPU.

Scenario 4 was evaluated with a more elaborate Web system than the arrays generator of scenarios 1, 2 and 3. The evaluation was made using the workload Rubis [15], which is modeled to be a clone of eBay (www.ebay.com). Rubis implements the basic features of ebay: product registration, sale, bidding, browsing products by region (United States) and categories. The installed version of Rubis 1.4.3 was obtained in https://github.com/sguazt/RUBiS.

In the tests we used the PHP version of Rubis and a MySQL 5.5 database. MySQL was installed in a virtual machine with Ubuntu 14.04.1 (16 vCPU and 4 GB of RAM). MySQL has been configured to allow caching of Rubis tables. These high settings of CPU and RAM of the MySQL virtual machine were carried out to ensure that there would be no bottlenecks in access to the application database, since the purpose of the tests is to test Web service auto scaling. The database was populated from the dump obtained in http://download.forge.ow2.org/rubis/rubis_dump.sql.gz.

As in the configuration described for scenarios 1, 2 and 3, the service publication in Kubernetes was made through the creation of a Replicaction Controller and the configuration of a NodePort service. The HAproxy load balancer was configured to balance requests between the Docker/Kubernetes nodes using the IP addresses of the nodes and ports published by the NodePort service. Each container had its processing and memory resources limited to 500 MB of RAM and 160 millicores of CPU.

The PID in the PAS algorithm utilized the following parameters: $Kp = 0.016$, $Ki = 0.000012$ and $Kd = 0.096$, which was set after several tests with the workloads, adjusting the parameters using the manual method, or *guess and check* , as described in section III-C.

B. Workload

The workload generation was made using "ab" tool (apache bench). In order to generate a load with a realistic profile we collected a set of accesses of the Portal da Transparência (www.transparencia.gov.br), between May/2016 and June/2016. The captured time series in the range of 1 second is the number of accesses on that time interval. The series was characterized with Kettani-Gubner method [10]. The self-similarity and long dependence of the series was confirmed with the Hurst parameter $H = 0.87$ in the scales of 1 second, 10 seconds and 100 seconds.

A sample of the obtained series can be seen in Figure 3. This series is used to generate in every second, simultaneous requests directed to the IP address of the load balancer which distributes requests to the nodes of the Kubernetes cluster. The workload intensity was set at 3 levels by multiplying the time series by 1, 1.5 and 2, and preserving the same self-similarity index. These loads are referred in the experiments as load_1 , load_1.5 and load_2 .

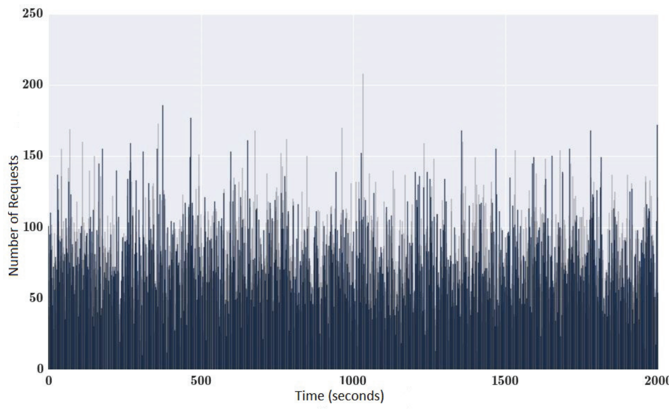

Figure 3. Workload Sample, H=0.87

1) Scenario 1: In this scenario the response time threshold at the load balancer (*setpoint*) was set at 50 ms and we applied load_1 and load_1.5. Figure 4 shows the system response time while under load_1. The graph shows the adjustment caused by the allocation of containers performed by the PAS algorithm and stability achieved close the setpoint of 50 ms.

Figure 5 shows the allocation of containers. The number of containers at the beginning of the experiment was equal to 2. The system allocated the required number of containers so that the average response time shown in Figure 4 reached the setpoint. At the end of the run there were 26 allocated containers .

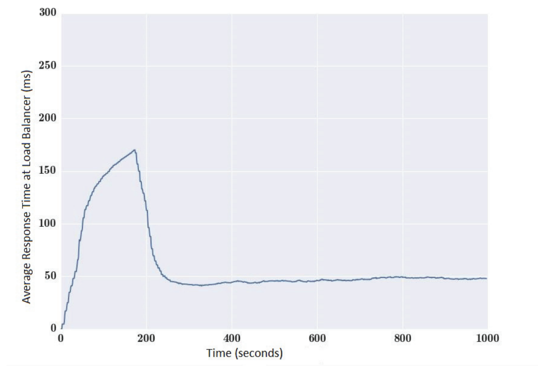

Figure 4. Response Time (ms) x Time (s), load_1, *setpoint* 50 ms

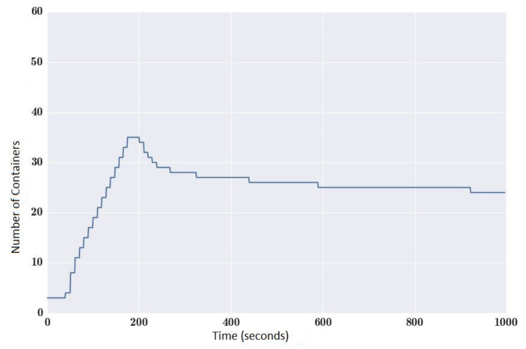

Figure 5. Number of Containers x Time (s), load_1, *setpoint* 50 ms

Figure 6 shows the average response time of the system while under load_1.5 and the setpoint kept at 50 ms. Figure 7 shows the containers allocation during this test. It is worth noting that the container allocation was adjusted to keep the average response time of the system near the setpoint. At the end of the run, 52 containers were allocated. In this case, the system under a greater load, allocated more containers to keep the response time within the defined threshold.

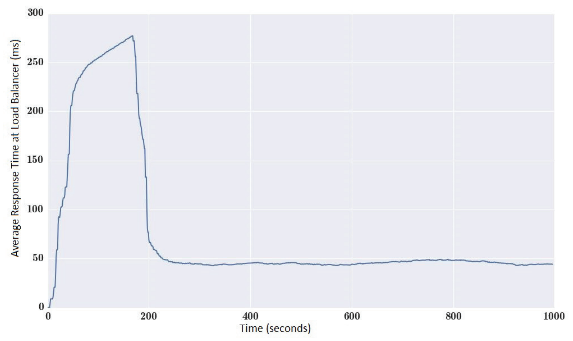

Figure 6. Response Time (ms) x Time (s), load_1.5, *setpoint* 50 ms

2) Scenario 2: In this scenario, the response time threshold in the load balancer (*setpoint*) was set at 50 ms and we applied load_1 for 1000 seconds, then load_1.5 for another 1000 seconds (starting at second 1001), and then we applied load_1 for more 1000 seconds (starting at second 2001). The

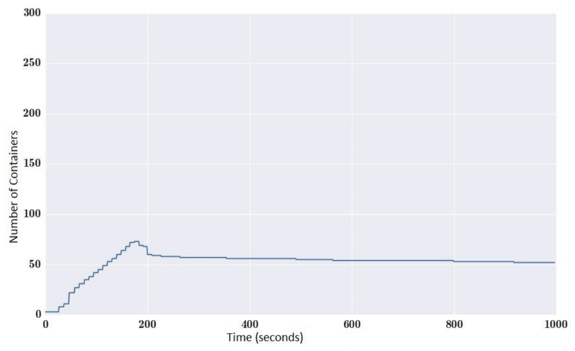

Figure 7. Number of Containers x Time (ms), load_1.5, *setpoint* 50 ms

purpose of this test was to evaluate the behavior of the system during sudden intensity changes.

As can be seen in figure 8 and 13, the system is capable to adjust itself increasing the number of containers when the intensity increases and decreases. It is observed that after exposing the system to load_1.5 at second 1000, the response time remains slightly over 50 ms, and after applying load_1 at second 2000 the response time remains slightly below 50 ms. Better results could be obtained for this case, readjusting the parameters of the PID controller.

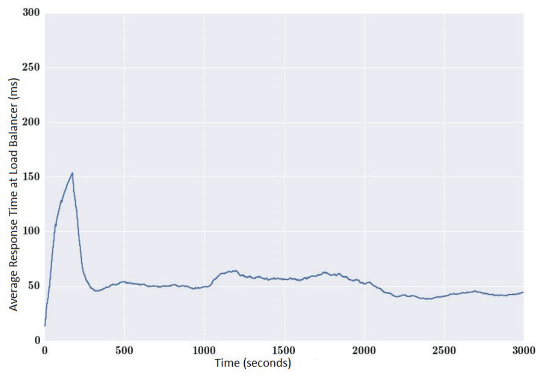

Figure 8. Response Time (ms) x Time (s), Variable load, *setpoint* 50 ms

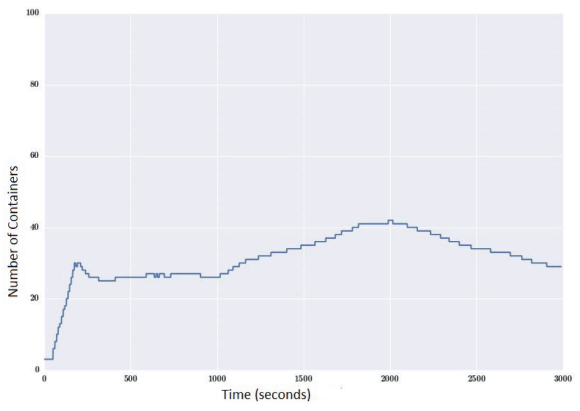

Figure 9. Response Time (ms) x Time (s), Variable load, *setpoint* 50 ms

3) Scenario 3: Scenario 3 compares the proposed algorithm in this paper with the HPA [7].

The comparison follows this procedure: the system configured with the HPA_80 (configured to scale when the average consumption of the containers of a given Replication Controller is above 80 %) is exposed to load_1, load_1.5 and load_2 during 1000 seconds. At the end of the test the average waiting time of the requests at the application layer (customer perspective) is observed.

From these data, it was defined a *setpoint* to use with the algorithm (PAS) for comparing with the response time close to the HPA reference. The results will be compared by checking the average amount of containers allocated during the experiments and the average response times achieved in the customer application layer.

As can be seen in Figure 10 the average response time in the application layer with the HPA algorithm for each load were: 66.18 ms for load_1, 110.12 ms for load_1.5 and 144.01 ms to load_2 .

For comparative purposes, the PAS *setpoints* was configured to deliver an average response time close to those obtained with the HPA. For this, we configured *setpoints* slightly below those observed in the HPA, as the *setpoint* is controlled in the load balancer, so the time measured in the customer application layer should be slightly higher. The values set for *setpoints* are: 50 ms (PAS_50) for the load_1 , 80 ms (PAS_80) for load_1.5 and 100 ms (PAS_100) for load_2 .

Figures 10 and 11 show that for response times near the value obtained by HPA, PAS system allocated less *containers* than HPA. The comparative of the *container* allocation shows that: PAS_50 allocated 44.02 % of which was allocated by the HPA_80, PAS_80 allocated 36.07 % of which was allocated by the HPA_80 to load_1 .5 and PAS_100 allocated 12.72% of which was allocated by the HPA_80 to load_2 .

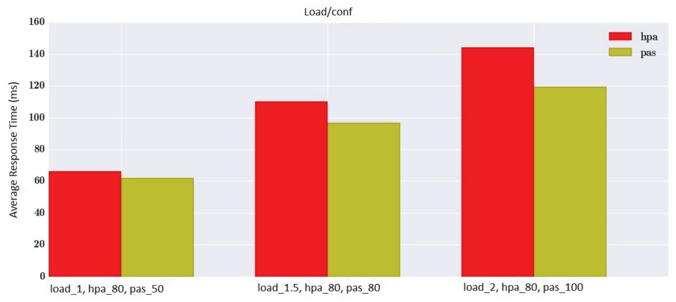

Figure 10. Comparison between HPA and PAS (Average Response Time of the System)

4) Scenario 4: In scenario 4 we evaluated the behavior of the PAS algorithm in the Rubis environment. We used loads_1.5. To generate request variability at every second the accesses to the links is divided as follows: 10 percent home page access, 10 percent of queries to the list of products with random category and random region, 40 percent of visits to random products and 40 percent of queries to random user profiles.

In this test the *setpoint* is set to 50 ms and it is applied load_1.5. Figure 13 shows the container allocation during the test, needed to control the application response time (*setpoint* = 50 ms) for load_1.5, and figure 12, shows the response

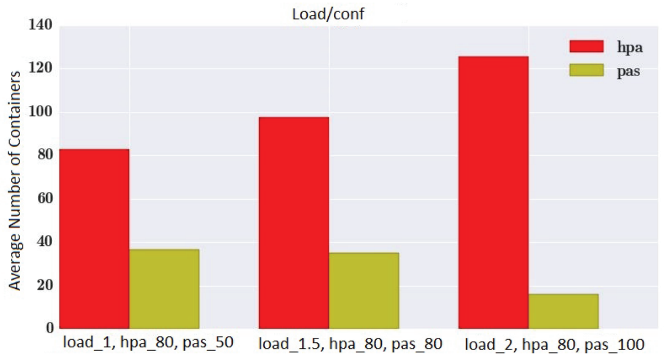

Figure 11. Comparison between HPA and PAS (Average number of containers)

time controlled near the *setpoint*. As can be seen, even with a much more varied workload than the array generator, PAS can control the average response time of the application close to the threshold .

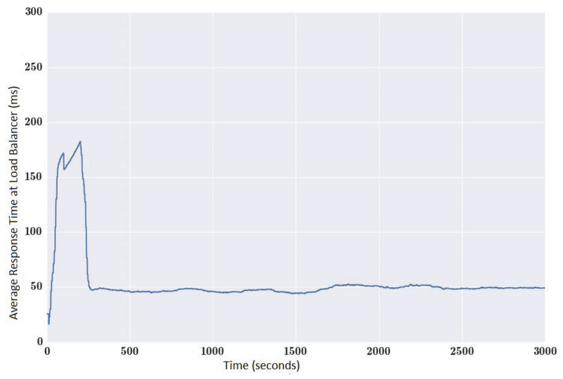

Figure 12. Rubis Response Time (ms) x Time (s), load_1.5, *setpoint* 50 ms

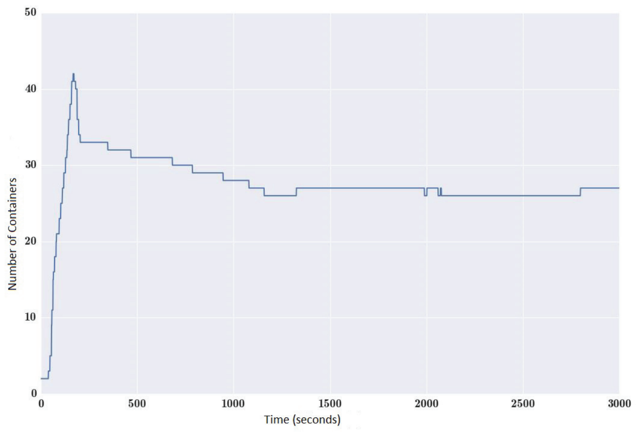

Figure 13. Rubis Number of Containers x Time (s), load_1.5, *setpoint* 50 ms

C. Analysis of Results

The experimental results show that for all the scenarios our proposal is efficient for resource allocation. In the scenario (V-B3), the PAS algorithm optimizes the allocation of the number of containers to hold the values of *setpoints* within a given threshold. The HPA allocates a greater number of containers to achieve equivalent threshold response times for requests in the application layer. This result shows that the allocation of a larger number of containers can increase the complexity of load balancing time and not necessarily produce better response times.

The obtained results show that the PAS algorithm proposed in this work has the potential to promote an optimization of the number of allocated containers in a cloud computing environment.

The tested scenarios show that the PAS algorithm is a viable alternative to promote auto elasticity to comply with a required response time. Furthermore, performing measurements outside the container, allows PAS to be a generic tool for providing auto elasticity in cloud systems.

VI. Conclusion and Future Works

This paper presented an algorithm based on response time to scale containers on a Cloud Computing system. The proposal defines a cloud computing architecture based on containers and uses a PAS algorithm (PID based Autoscaler) to optimize resource allocation.

The proposal was evaluated in 4 scenarios using different workloads characterized from real world applications. The results shows that our proposal has the potential application to provide auto elasticity in cloud computing systems based on containers. The comparison with the native tool of Kubernetes, the HPA shows a higher efficiency for the PAS proposal.

In future work we intend to improve the PAS algorithm with sophisticated methods for setting the PID parameters. Furthermore the algorithm will be tested with other container orchestrators, such as Mesos and Docker Swarm, to verify that PAS can be a generic tool to provide auto elasticity in cloud computing environments based on containers.

References

[1] Mitchell Anicas. Mitchel Anicas an introduction to haproxy and load balancing concepts. https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts, 2014.

[2] Docker. Docker the definitive guide to docker containers. https://www.Docker.com/sites/default/files/WP-%20Definitive%20Guide%20To%20Containers.pdf, 2016.

[3] Flume. Flume flume user guide. https://flume.apache.org/FlumeUserGuide.html, 2016.

[4] Katja Gilly, Carlos Juiz, and Ramon Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, 2011.

[5] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[6] Google. Google container cluster manager from google. https://github.com/kubernetes/kubernetes, 2016.

[7] Google. Google horizontal pod autoscaler. https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/design/horizontal-pod-autoscaler.md, 2016.

[8] National Instruments. National Instruments explicando a teoria pid. http://www.ni.com/white-paper/3782/pt/, 2015.

[9] Kairos. Kairos time series data storage in redis, mongo, sql and cassandra. https://pypi.python.org/pypi/kairos, 2015.

[10] Houssain Kettani, John Gubner, et al. A novel approach to the estimation of the hurst parameter in self-similar traffic. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 160–165. IEEE, 2002.

[11] Tania Lorido-Botrán, José Miguel-Alonso, and Jose Antonio Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, 12:2012, 2012.

[12] Nick Martin. Nick Martin a brief history of docker containers' overnight success. http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success, 2015.

[13] Rishabh Poddar, Anilkumar Vishnoi, and Vijay Mann. Haven: Holistic load balancing and auto scaling in the cloud. 2015.

[14] Redis. Redis redis documentation. http://redis.io/documentation, 2015.

[15] Rubis. Rubis rubis: Rice university bidding system. http://rubis.ow2.org/, 2009.

[16] Spark. Spark spark programming guide. https://spark.apache.org/docs/1.5.2/programming-guide.html, 2015.

[17] Willy Tarreau. HAProxy haproxy configuration manual. http://www.haproxy.org/download/1.5/doc/configuration.txt, 2015.