PAS-CA: A Cloud Computing Auto-scalability Method for High-demand Web Systems

Marcelo Cerqueira de Abranches
Departamento de Ciência da Computação
Universidade de Brasília
Controladoria-Geral da União (CGU), Brasília, DF
Email: marcelo.abranches@cgu.gov.br

Priscila Solis and Eduardo Alchieri
Departamento de Ciência da Computação
Universidade de Brasília, Brasília-DF
Email: pris@cic.unb.br, alchieri@unb.br

***Abstract*—This work proposes an auto scaling method for high demanding web applications in a cloud computing system. The proposal has the goal to increase efficiency in containers allocation for processing web requests considering a threshold for applications response time. The method was developed using a control algorithm that applies adjustments based on the properties extracted from workload characterization. The method was evaluated in a simulation scenario with different types of workloads and the results show that the proposal achieves good results by allocating containers more efficiently than other solutions already well-known in the cloud computing market.**

I. Introduction

The evolution of cloud computing systems heads towards multi-clouds and the distribution of processing in heterogeneous devices. The IoT (Internet of Things) and its applications incited the emergence of new technologies such as edge computing and fog computing. Lightweight virtualization solutions, nowadays based on containers, appear to be beneficial for applications that need elasticity. In addition, containers offer benefits over traditional virtual machines in the cloud in terms of size and flexibility and are specifically relevant for Platform as a Service (PaaS) [16] applications.

Cloud computing providers aim to offer methods to allocate or deallocate resources on demand to meet the service levels in the contract, or Service Level Agreement (SLA) [12]. However, it is not obvious how a provider can map SLAs such as QoS (Quality of Service) over low-level resources such as memory usage and processing capacity. Also, for provisioned services to meet fast demand fluctuations these provisioning decisions must be made online.

Recently, the use of Docker [15] has been consolidated as a *de facto* standard in the open-source containers technology. Applications can extend from one platform to another, functioning as micro services in Linux servers and PaaS environments. Nowadays, many of the leading cloud providers use Docker [17]. Recently, Google released the GKE (Container Engine) that uses Kubernetes, which automates the process of provisioning, running, and stopping virtual machines and the process of deploying applications in a variable number of containers that can be scaled on demand [14].

One very relevant issue in the cloud market is the cost which is normally proportional to the user's demand. This can be directly influenced by performance requirements. Then, for both sides, clients and providers, it is important to offer the resource efficiently as close as possible in a deterministic mode, facilitating the planning of costs and resources for both sides. Cloud computing systems host large-scale, mass-data-intensive applications. Usually, massive data processing is done via frameworks of MapReduce such a Hadoop or Spark. These frameworks are constructed in a distributed way, and make heavy use of CPU, memory and read/write operations to disk [12]. Then, it is vital for a cloud provider to perform efficient allocation of resources.

Not less important is energy consumption. The correct sizing of systems has a direct impact in energy efficiency of data centers and this relies on resource allocation optimization techniques which also implies in lower energy required for cooling data centers [13].

This work proposes and evaluates an auto-scaling method and an algorithm for elasticity based on containers allocation optimization to comply with performance requests in a high-demand web system using PaaS. The proposal uses a control algorithm, an architecture based on containers and the statistical analysis of different workloads in a web system. A innovative characteristic of the proposal is that it uses the response time perceived by the user as input to the algorithm that calculates the resource needs. This article is structured as follows: section 2 presents the literature review and related works. Section 3 describes the proposal that is later analyzed experimentally in Section 4. Finally, Section 5 presents the conclusions and future work of this research.

II. Literature Review and Related Works

In this section, we define and describe the theoretical concepts and the technological tools that were used to design our proposal.

A. Virtualization and Cloud Managers

Virtualization uses software to emulate hardware functionalities, which allows multiple distinct operating systems and applications to run on the same server [18]. A virtual machine (VM) is a piece of isolated software, with an operating system and its associated applications. In this context, a layer of software called hypervisor is responsible for running virtual machines and managing the use of physical resources, implementing independence between the VMs.

Automation can be achieved through the use of Cloud managers and configuration automation tools. Cloud Managers

normally communicate with *hypervisors*, network and storage devices through APIs (Application Program Interfaces) [19]. Configuration automation tools allow to define and apply the desired configuration for a server farm, considering different parameters [32].

B. Containers

A container is a technology for creating isolated processing instances that allow virtualization in the operating system level. Two containers running on the same system have their own network-layer abstraction, memory, and processes [15]. This isolation is done via kernel *namespaces*. A *namespace* is an abstraction that allows processes within a *namespace* to have their own instance of the global resource. Changes in a *namespace* are seen only by processes that are members of that *namespace* [20]. Linux implements *namespaces* for file system, processes, network and users[16]. The limits and accounting of containers resources are done by the *cgroups*. *Cgroups* allows resource allocation through user-defined process groups on a system[21]. Containers have greater portability than virtual machines when configured in a generic way for any Linux-based operating system.

Virtualization via *hypervisors* consumes more resources than containers since the latter executes on top of operating systems and run in isolated spaces with each other. If a container is not performing any task, it is not consuming resources on the server[15]. In addition, containers can be created and destroyed as needed, since they only have to start or destroy processes in their isolated space.

C. Workload Characterization

The workload characterization is a very important and critical task for any performance evaluation and resource allocation planning. This characterization uses statistical analysis of the time series that represent the load. For the case of web requests, several previous works observed a statistical property known as mono fractal [4], that also appears in other types of Internet traffic. One important property of mono fractal or self-similar processes is the long-term dependence. A process with long-term dependence has a function of autocorrelation defined by $r(k) \sim k^{-\beta}$ with $k \to \infty$ and $0 < \beta < 1$.

In this type of processes, the Hurst parameter (*H*) indicates the degree of self-similarity of a series and is given by $H = 1 - \frac{\beta}{2}$. A self-similar time series with long dependence has $\frac{1}{2} < H < 1$. When $H \to 1$, both the degree of self-similarity and the long-term dependence increase [4]. Then, the calculating the size of the *H* parameter is useful to better understand the workload complexity.

There are several methods for estimating the Hurst (*H*) parameter. One of them is the Kettani-Gubner [5] which calculates *H* using smaller time series than the required by other methods. In this method, the coefficient of autocorrelation is described by Eq.1

$$R(k) = 1/2(|k+1|^{2H}) - 2|k|^{2H} + |k-1|^{2H} \qquad (1)$$

when *k*=1, then $R(1) = 2^{2H-1}$ and *H* may be estimated as $\hat{H} = \frac{1}{2}[1 + \log(1 + \hat{R_n}(1))]$. Under the assumption that the process is ergodic, it is possible to calculate the *H* parameter, according to Eq.2.

$$\hat{H_n} = \frac{1}{2}[1 + log(1 + \hat{R_n}(1))] \qquad (2)$$

In our work, using the above method, we confirmed that the web requests are self-similar. This characterization of the expected workload allowed the generation of different workload profiles that were used to test our proposal.

D. PID Controllers

PID (Proportional-Integral-Derivative) controllers are one of the most widely used control algorithms in the automation industry. In these algorithms, a control variable is read from the *S* system. This value is compared to the desired value, called *setpoint*, and an error term *e(t)* is generated. The error term is combined into the proportional (P), integral (I) and derivative (D) components of the controller and a control signal is generated. This signal feeds an actuator that acts on the system to control the value of the desired variable [6]. Figure 1 shows a block diagram of a PID controller in a feedback loop.

The proportional component depends on the difference between the desired value (*setpoint*) and the current value of the variable. The proportional gain (K_p) determines the output response rate for the error. The derivative component indicates the instantaneous error change rate. The integral component sums the error term over time. The derivative component and the integral component are respectively multiplied by the gains K_d and K_i to determine the output response of these terms.

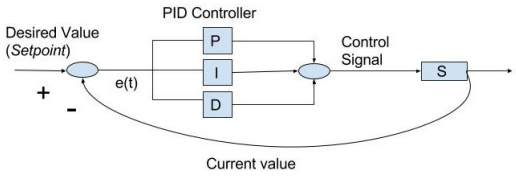

Figure 1: PID controller

Setting the gain parameters K_p, K_i and K_d allows the PID controller to make the necessary adjustments to control the output. There are several methods of adjusting these parameters, such as the manual method, the Ziegler-Nichols method [6] and the Coordinate Ascent(CA) algorithm. In the manual method, the gains of each of the components are adjusted using trial and error. The terms K_i and K_d are set to zero, and the term K_p is increased until the cycle output begins to oscillate. From there, the term K_i is slowly increased to reduce the stationary error. At this point the increment of the term K_d is started, to decrease the oscillations in the cycle output [6].

The Coordinate Ascent (CA) [11] method establishes a loop that optimizes a scoring function. This function defines how close to the goal a variable was in an iteration of the algorithm. Given the first estimate of parameters that affect the value of the scoring function, the algorithm modifies each of these parameters based on a fixed value. Then, it verifies if the

change improves the scoring function. If so, the current value of the parameter is registered, as well as the new value of the scoring function and the step is gradually increased. Otherwise, the step is gradually decreased until the value search interval is less than a pre-set minimum value. The pseudo code in Algorithm 1 describes the CA algorithm. In our proposal, the PID controller is used in a closed loop that controls the application average response time for different workloads. This is achieved by varying the number of containers allocated to process web requests.

Algorithm 1: COORDINATE ASCENT

```
1  begin
2      While the sum of the steps for each parameter >
         minimum threshold
3        For each parameter
4          increase the parameter by the step value and
         calculate the current score
5        If current score > last score
6          keep the value of the parameter, increase the
         step size and save the score
7        If current score < last score
8          decrease the parameter by the step value and
         calculate the current score
9          If current score > last score
10           keep the value of the parameter, increase the
         step size and save the score
11         If current score < last score
12           decrease the step size
13 end
14 return Optimized Parameters
```

E. Related Work

Several research works addressed auto-scaling in cloud environments. In [7] the authors compare various methods classified into reactive and predictive categories. Regarding CPU load, the PRESS algorithm [9] proposes a method that uses signal processing with fast Fourier transforms to extract dominant frequencies. This is used to generate a time series and multiple time windows. A Pearson correlation index is generated for the various windows compared. If a correlation index greater than 0.85 is found, the average value of resource use in each place of the time series is used to generate a forecast for the next window. In this case, the resources of the virtual machines are adjusted. If a pattern is not identified, the authors propose an approach that uses a Markov chain with a finite number of states to do predictions.

In Haven [8], the authors present an elastic sizing method for cloud environments that is based on monitoring CPU and memory loads for each virtual machine. From previously established thresholds for CPU and memory consumption, the algorithm instantiates new virtual machines and inserts them into a load balancing pool. The implementation of the load balancer uses SDN (Software Defined Network) and has the intelligence to forward request to entities with better processing conditions.

Our proposal, PAS-CA, differs from the PRESS proposal since instead of load prediction, it does resource sizing based on the response time of an application supported by a load balancer. Another difference is that the PRESS algorithm performs vertical scalability, increasing CPU resources to follow workload performance requirements. In our work, we propose horizontal scalability, where new instances are provisioned behind a load balancer to accommodate variations in performance requirements. Horizontal scalability has the advantage that the resources allocated to meet the workload are not limited to the physical resources of a machine and this approach facilitates high availability since the workload is processed by a set of instances in parallel.

Both works, Haven and Press, analyze environments with VM technology in which scalability systems must take into account the VM time for provisioning and configuration. This time may reach minutes, while the provisioning of containers may take only few seconds [1]. Systems that use VMs working reactively in a control loop such as PAS-CA, can experiment constant SLA violations which is why workload predictions and pre-provisioning techniques are commonly used in these cases [7][8] [9]. Then, systems designed to offer elasticity in VM environments may not be suitable for scaling containers. Our proposal differs from Haven's since our method for sizing the system is based on the requests response time. This allows an overall visibility of the performance of the system. Even if there is no excess of processing and memory consumption, the application response times can be reduced due to the increased parallelism. The following paragraphs analyze recent works designed to work with containers and compare our proposal with them.

In [10] the authors propose the Kubernetes' native auto-scaling tool called Horizontal Pod Autoscaler (HPA). HPA scales the environment based on CPU average consumption thresholds of containers. The tool uses the *cadivisor* monitoring tool which must be functional on all Kubernetes worker nodes. The data from *cadvisor* is aggregated by the *heapster* tool. *Heapster* provides the CPU usage of containers, which is used by HPA. PAS-CA differs from HPA because it does not need neither *cadvisor* or *heapster*. PAS-CA works independently from any container monitoring software, since it monitors response times at a single point, as we'll describe in the next section.

In [22], the authors propose a reactive and proactive method to scale containers. The metric is the number of requests per second and the algorithm uses this information to predict the number of containers required to meet the demand. The number of requests that each container can process needs to be known. The authors argue that this value can be set manually, however, the number of requests per second that a container can process depends on its CPU and memory conditions and may have relevant variations in a real environment. The system adds or subtracts a fixed quantity of containers according to the demand. Our proposal, the PAS-CA adds or removes instances according to the PID controller output which defines the number of containers that must be added or subtracted at each iteration of the algorithm, to keep the response time under the defined threshold. In addition, the PAS-CA does not need to know the number of requests per second that a container can handle, since this metric can vary between different applications.

In [23], the authors define the operations per second as

a threshold for scalability. This is done by empirically establishing the number of operations per second that a system can handle. This is complex in a dynamic environment. The works [25] and [26] show how to build and configure a self-scalable application in Amazon's cloud environment. These works use proprietary tools from Amazon. The PAS-CA proposal can be used in any cloud provider because it is based on open source tools and is independent of any third party monitoring tool. Another work [27] describes how to give a scalable environment using Docker containers, but scaling is done manually. PAS-CA scales automatically.

In a previous work [24] we proposed also a method based on control theory for providing and auto scalable environment, using the average response time of an application as the threshold. In that work, the setting of the PID parameters is done manually while PAS-CA uses an algorithm to automatically set the PID parameters for different workloads and shows a better efficiency for allocating resources.

III. The PAS-CA Proposal

The PAS-CA (PID Based Autoscaler-CA) is a method for auto scaling provision. In our proposal, we define a cloud computing environment based on containers and a sizing method that uses the system response time as the parameter to trigger the auto scaling procedure. The PAS-CA uses the architecture described in Figure 2 and works as follows:

1) A threshold (setpoint) for the desired average response time is defined. The request monitor reads the requests response time at the load balancer.
2) The request monitor sends the average response time to the PID which calculates the number of containers required to reach or remain at the set point.
3) The Algorithm 2 is executed and its output is the optimal number of containers for Kubernetes.
4) Kubernetes creates or removes new containers, The system remains with the ideal number of containers until the next round of the algorithm (after 10 seconds). This value was defined in empirical tests as it appeared to be enough for Kubernetes to start new containers and to answer requests in the cluster.

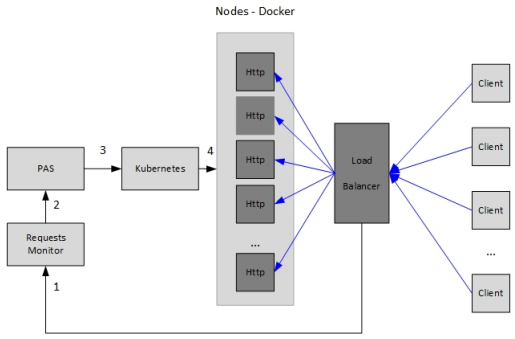

Figure 2: PAS-CA Architecture

To enable the algorithm operation which works with dynamic real-time data, we used the processing flow shown in Figure 3 which describes the tools used in the PAS-CA architecture. The flow in Figure 3 begins with the HAproxy

Algorithm 2: PAS

Input: Average response time of the cluster, Current number of *containers*
Output: Desired number of *containers*

begin
- Read the desired threshold of average response time of the requests: $t\_ms\_desired$
- Read the current number of *containers*: $n\_containers\_current$
- Read the average response of the cluster in ms: $t\_ms\_current$
- Calculate the error: $e(t)$ = $t\_ms\_desired$ - $t\_ms\_current$
- Calculate the PID output:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{d}{dt} e(t) \quad (3)$$

end
return *n_desired_containers*

[3], which is a load balancer used by Reddit, Stack Overflow, Server Fault, Instagram and Red Hat OpenShift. The Haproxy balances the HTTP requests in a round robin mode for the Docker and Kubernetes servers that host the containers. When the Docker and Kubernetes node receives the requests, the Kubernetes proxy service balances the load between the containers and the Kubernetes nodes. Therefore, there are two levels for load balancing: one between the Docker and Kubernetes nodes where the balancing agent is the Haproxy, and another inside the nodes. Their access logs are sent to Flume.

To make data available, Flume sends in real time the load balancer access logs to a memory channel that loads the data from the source into memory. Then Spark Streaming consumes this data stream to process the requests response time. The response time information is saved by Spark Streaming in a time series format on the Redis server. The integration with Redis is possible through the Kairos library, which creates the structure for storing the time series. With this data available on Redis, PAS-CA runs and determines the ideal number of containers. Kubernetes receives this number and uses the kubectl tool to adjust the allocation within a replication controller.

The Coordinate Ascent algorithm optimizes the PID parameters by minimizing a scoring function. First, the algorithm defines the initial values for the parameters *Kp*, *Ki* and *Kd* and for the parameter update step size. An initial value is also established for the mean squared error which is the scoring function. In our case, the scoring function is the square of the sum of the differences of average response times to the set point within the time window. The window was set to 2000 seconds. Experimentally we verified that this time would be sufficient to adjust the PID parameters.

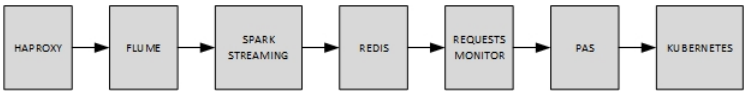

Figure 3: Processing flow for the PAS-CA architecture

IV. Simulation Scenarios and Experimental Results

A. Simulation Environment Configuration

The simulation environment was configured using a Kubernetes v1.4.1 cluster on a *CoreOS* operating system (899.6.0 (2016-02-02)), virtualized on *VMWare ESXi* 5.5.0. The cluster has the following components: 1 master node (4 vCPUs, 6 GB of RAM), 1 etcd node (4 vCPUs, 6 GB of RAM) and 3 *worker* nodes (4 vCPUs, 6 GB RAM). Two additional Virtual machines (VMWare ESXi 5.5.0) were installed on the Ubuntu 14.04.3 LTS operating system, with the following settings and tools: 1 Haproxy 1.5.4 node (4 vCPUs, 4G GB RAM) , 1 Spark 1.5.2 node plus Redis 2.8.4 (2 vCPU, 10 GB RAM) and 1 Flume 1.7.0 node plus the PAS (2 vCPU, 4 GB of RAM).

The evaluation scenarios used the Rubis [2] environment, which is modeled as an ebay clone (www.ebay.com). RUBiS implements the basic e-commerce features like product registration, sales, auction bids, product navigation by region and categories. The installed version of RUBiS was 1.4.3 obtained at [29]. The publication of the service in Kubernetes was done through the creation of a *Replication Controller* and the configuration of a *NodePort* type service.

Haproxy was set to balance requests between the *Docker/Kubernetes* nodes using their IP addresses and the ports published by the *Kubernetes NodePort* service. Each container had its processing and memory resources limited to 300 millicores and 300 MB of RAM. We established these limits after verifying that these resources were enough to support the initial conditions of the performance tests in several scales of intensity.

B. Workload Characterization and Types

We collected a set of real web accesses, from 05/25/2015 to 06/25/2015, from a high demand web site, the Brazil's transparency portal (www.transparencia.gov.br) on 1 second scale. The time series were analyzed using the Kettani-Gubner method in which we calculated the self similarity parameter. The H= 0.87 was observed at different scales : 1, 100 and 600 seconds and this confirms the fractal nature of the web traffic. These series were used to generate at a 1 second time scale simultaneous request directed to the IP address of the load balancer, which distributes the requests between the Kubernetes cluster nodes. We generated different workloads by multiplying the original workload by several indexes preserving its statistical properties: load_2 in which the time series is multiplied by 2, load_3 (time series multiplied by 3) and for a variable load that works as follows: the system is submitted to load_1 (series is multiplied by 1) per 1000 seconds, then a load_2 (series multiplied by 2) for 1000 seconds, then load_3 for a further 1000 seconds, followed again by the load_2 and load_1, for another 1000 seconds each. This workload is defined as load_1_2_3_2_1.

The different workloads were generated using the *ab (apache bench)* tool [28]. The time series with the number of requests at each second from the access logs of the Brazil's Transparency Portal [30] were used to define the number of concurrent requests that were generated at every second of the test.

We distributed the concurrent accesses in each second to the Rubis pages as follows: 10% of home page hits, 10% of product queries with random class and region, 40% of random product queries, and 40% of queries to random user profiles. We defined this percentage distribution in the accesses considering the behavior of a user that accesses the system home page, makes a query for a product, in a certain geographic region, and from there spends most of the time browsing between products, and verifying profiles of other users, who carry out the sales.

C. PID Parameters Tuning

To compare the results with previous works, first we defined the PID parameters manually, as follows: Kp = 0.016, Ki = 0.000012 and Kd = 0.096. These values were find after several tests with the workload [24].

The resulting PID parameters optimized by the *Coordinate Ascent* algorithm, for the thresholds can be seen in Table I. To calculate the optimized parameters for 80, 100 and 120 ms threshold of the PAS-CA, we ran the algorithm as follows: (1) the parameters Kp, Ki and Kd were set to 0.01; (2) the initial step was set to 0.001; (3) the convergence threshold of the sum of the steps was set to 0.000001; (4) the system was exposed to the load_3 and the Coordinate Ascent was run until its convergence for each configuration.

D. Experimental Results

This section presents the results of the experimental tests and its comparison with PAS (with manual PID parameter tuning) and with HPA, a commercial auto scaling tool developed by Google. For each test, we repeated the experiment 10 times. Each test lasted 1200 seconds. We set the confidence interval at 95%, with 9 degrees of freedom. We set the resource allocation limits for these tests to 300 millimeters of processing and 300 MB of RAM. These settings were established to allow Rubis to always start in good conditions regarding CPU and memory consumption. The smallest number of containers in all tests was 3 to have at least one container running on each node at the starting point.

The HPA, PAS, and PAS-CA settings as well as the identifiers used for each of the configurations can be found in Table I. For HPA, the maximum average processing limits were set at 20% and 50%. These values, 20% and 50% were set because the HPA scaling with 20% processing allows good response times, keeping the containers in a safe processing level. The HPA scaling with 50% processing allows smaller allocation of

containers, consequently worsening the application response times, but still with a secure level. Thresholds above 50% cause high levels of processing, which may cause containers to freeze, reset connections and timeouts.

For PAS 80, PAS 100 and PAS 120, the PID parameters were found manually. For PAS 80 CA, PAS 100 CA and PAS 120 CA, we used the parameters automatically optimized by the *Coordinate Ascent*. We configured the thresholds for the PAS and PAS-CA algorithms aiming to bring response times compatible with those obtained with the settings used in the HPA algorithm.

The metrics defined for the tests are the following: the average waiting time at the client application layer, the average waiting time at the load balancer, the percentage of failed requests, the average number of containers allocated during the tests and the average efficiency in containers allocation. We calculated the average number of containers by saving the number of allocated containers at each second, and at the end dividing the sum of the total numbers of containers by the total time. The average efficiency in the containers allocation is defined in Eq. 4, where Nc is the average number of containers during the experiments, and T is the average response time in milliseconds at the application layer.

$$E = 1/(Nc * T) \tag{4}$$

The identification of columns in tables III and IV, are the following: (1) confidence interval for the average response time in milliseconds at the application layer; (2) confidence interval for the average allocation of containers; (3) confidence interval for the percentage of failed requests; (4) confidence interval for the efficiency of the algorithm; (5) confidence interval for the average response time in milliseconds measured at the load balancer.

On the first experiment, we used load_2 for 1200 seconds. The results obtained are summarized in Table II. In this test, load_2 was not enough for the average processing of the containers to reach 50%. Then, the HPA 50 did not allocate more containers. Also, the same happened for PAS 120 and PAS 120 CA. In this test, the load intensity did not raise the application average response time above 120 ms. The results in table II show that the intervals for container allocation (column 2) remained in "3.0 , 3.0" for those settings. Columns 1 and 4 in Table II show equal average response time for HPA 20, PAS 80 and PAS 80 CA with HPA 20 algorithm achieving the best efficiency. However, the efficiency of these algorithms for this load is very similar. For load_2, the algorithms that scaled resources and stand out in the efficiency metric are PAS 100 and PAS 100 CA. This is because, for load_2, these algorithms allocated less containers and were able to show better average response time for the clients.

Column 3 in Table II shows the percentage of failed requests for the configurations. It is possible to see that these values stay close to zero in most of the tests which demonstrates the robustness of the algorithms to create and to destroy containers, without impact in the performance requests. Column 5 shows that PAS 80 and PAS 80 CA achieved average response times close to the desired threshold (80 ms). For PAS 100, PAS 100 CA, PAS 120 and PAS 120 CA, the load was not enough to stress the PAS algorithm so the mean value remained below 100 ms.

The results of the experiments with load_3 and 1200 seconds are shown in Table III. Also, the waiting time of the clients for PAS 80 CA was close to HPA 20, but PAS 80 CA shows a narrower confidence interval, which demonstrates a more predictable behavior of the response time in this test. The HPA 50 allocated less containers, but had a longer response time. In these tests, the creation and destruction of containers performed by the auto scaling algorithms did not represent critical problems since the confidence intervals for the percentage of failed requests show small numbers. Also, the PAS CA tended to lose fewer requests than the PAS, that uses manually optimization. The results also show that the settings of the PAS versions were able to keep up the average response time on the balancer close to the required thresholds. The HPA algorithm tended to have a wider confidence intervals than those of the PAS-CA, which demonstrates a better predictability in meeting response time requirements for this algorithm. The results also show that PAS-CA obtained better response time than PAS and similar efficiency. For a response time equal to HPA 20, PAS 80 CA obtained better efficiency.

To verify the difference between PAS and HPA with the variable load (load_1_2_3 _2_1), we performed tests for PAS 80 CA and HPA 20 configurations. The results are summarized in Table IV.The results show that the average response time at the application layer and the average allocation of containers during the tests were both better for PAS 80 CA. Also, both algorithms obtained similar average response time values, within the confidence interval, however, the confidence interval is smaller for PAS 80 CA. PAS 80 CA also allocated a smaller number of containers.

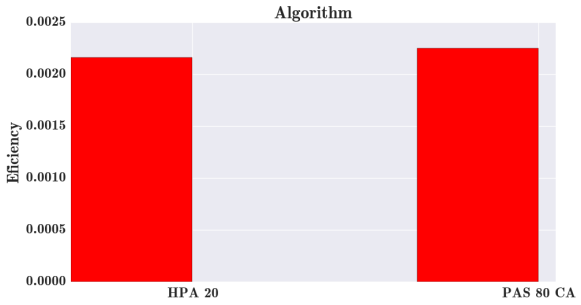

Figure 4: Efficiency for variable load

The results in Table IV (column 4), and in Figure 4 show that PAS 80 CA has a better efficiency than HPA 20. This results is consistent with the previous tests, where PAS 80 CA had response times close to HPA 20, but allocated fewer containers. The number of failed requests was low for both PAS 80 CA and HPA 20, which indicated a degree of robustness for the different loads, even under abrupt changes. The PAS 80 CA version had average response times at the load balancer smaller than the HPA 20, set at 80 ms, but the results show that the value was kept close to 70 ms. This can be explained by the fact that in these tests with variable load, the system is submitted to load_1 by 2000 seconds, and it was observed in earlier tests that for this workload the response time was below 80 ms, even without the system being scaled.

Table I: Setup of the Algorithms

Algorithm	Free CPU	Response Time Threshold	Kp	Ki	Kd
HPA 20	20%	-	-	-	-
HPA 50	50%	-	-	-	-
PAS 80	-	80 ms	0.01	0.000012	0.096
PAS 80 CA	-	80 ms	0.00121	0.0000505	0.00112
PAS 100	-	100 ms	0.01	0.000012	0.096
PAS 100 CA	-	100 ms	0.00131	0.0000515	0.00101
PAS 120	-	120 ms	0.01	0.000012	0.096
PAS 120 CA	-	120 ms	0.00135	0.000041	0.00115

Table II: Test results for load_2 with 95% of confidence

	1	2	3	4	5
HPA 20	128.474, 147.355	3.678, 3.779	0.0, 0.0	0.001, 0.001	73.016, 89.781
HPA 50	140.829, 175.214	3.0, 3.0	0.0, 0.0	0.002, 0.002	90.348, 103.327
PAS 80	139.632, 148.248	3.633, 3.644	0.000003, 0.000003	0.001, 0.001	81.110, 81.979
PAS 80 CA	135.526, 139.204	4.036, 4.048	0.0, 0.0	0.001, 0.001	75.965, 76.782
PAS 100	140.908, 152.892	3.015, 3.015	0.0, 0.0	0.002, 0.002	91.022, 93.987
PAS 100 CA	139.546, 154.080	3.020, 3.023	0.0, 0.0	0.002, 0.002	91.131, 94.915
PAS 120	145.905, 164.929	3.0, 3.0	0.0, 0.0	0.002, 0.002	90.601, 99.916
PAS 120 CA	143.285, 165.867	3.0, 3.0	0.0, 0.0	0.002, 0.002	91.797, 98.768

Table III: Test results for load_3 with 95% of confidence

	1	2	3	4	5
HPA 20	164.797, 204.326	5.560, 5.684	0.00001, 0.000001	0.0009, 0.0009	61.696, 87.191
HPA 50	199.988, 271.039	3.0, 3.0	0.0, 0.0	0.001, 0.001	134.060, 166.074
PAS 80	168.619, 225.594	5.009, 5.111	0.00007, 0.00007	0.00100, 0.00100	84.202, 95.339
PAS 80 CA	181.819, 185.631	5.270, 5.283	0.00004, 0.00004	0.00103, 0.00103	84.544, 85.648
PAS 100	193.845, 221.755	4.254, 4.277	0.00009, 0.00009	0.0011, 0.0011	103.742, 110.421
PAS 100 CA	180.234, 207.569	4.370, 4.388	0.00001, 0.00001	0.0011, 0.0011	98.089, 99.315
PAS 120	207.609, 215.419	3.492, 3.502	0.00009, 0.00009	0.0013, 0.0013	118.899, 120.642
PAS 120 CA	203.998, 208.970	3.556, 3.561	0.000009, 0.000009	0.0013, 0.0013	113.701, 117.323

Table IV: Test results for variable load with 95% of confidence

	1	2	3	4	5
HPA 20	108.786, 115.244	4.137, 4.143	0.000003, 0.000003	0.0021, 0.0021	73.477, 75.618
PAS 80 CA	108.523, 110.753	4.052, 4.078	0.000003, 0.000003	0.0022, 0.0022	70.861, 71.536

E. Zero mean tests

To compare the results, we performed a set of zero-mean tests to statistically verify the results of the different systems within a confidence interval [31]. We set the confidence interval to 95%, using the *t*-distribution with 9 degrees of freedom. In the context of this work, each system represents one of the evaluated algorithms.

We chose HPA 20 and PAS 80 CA because they have configurations with the highest performance requirements among the evaluated scenarios. We compared the response times, container allocation, and efficiency for load_2, load_3 and for the variable load. In the following tables the lines represent "system 1" and the columns represent "system 2".

Table V shows that for load_2 the zero-mean interval for the response time is almost symmetric, indicating that there was no statistical difference among the algorithms for this test. Table VI shows a negative interval indicating that HPA 20 allocated less containers than PAS 80 CA. The table VII shows a totally positive interval indicating that for load_2 HPA 20 had a better efficiency. For load_3 the results for the response time show an interval almost symmetric between negative and positive values, which indicates the equivalence of the two systems. For variable load the interval shows equivalent response times, however, the HPA 20 shows a tendency for longer response times. Also the results in Table VI show that HPA 20 allocated more containers for load_3 and for the variable load. Finally, in Table VII the results show that for load_3 and for the variable load, PAS 80 CA was more efficient. All results in this section are consistent with the results presented in the previous section, which validates the accuracy of the evaluation method used in this work.

Table V: Zero mean tests for response time with 95% of confidence

	load_2	**load_3**	**variable load**
System 1/System 2	**PAS CA 80**	**PAS CA 80**	**PAS CA 80**
HPA 20	(-12.045, 13.145)	(-20.77, 22.44)	(-0.511, 5.264)

Table VI: Zero mean tests for the average number of containers with 95% of confidence

	load_2	**load_3**	**variable load**
System 1/System 2	**PAS CA 80**	**PAS CA 80**	**PAS CA 80**
HPA 20	(-0.385, -0.241)	(0.278, 0.414)	(0.059, 0.098)

Table VII: Zero mean tests for efficiency with 95% of confidence

	load_2	load_3	variable load
System 1/System 2	**PAS CA 80**	**PAS CA 80**	**PAS CA 80**
HPA 20	(1.54e-04,1.55e-4)	(-6.53e-5,-6.5e-5)	(-8.95e-5,-8.9e-5)

V. Conclusions and Future Work

This paper presented PAS-CA, an auto scaling method for cloud computing systems based on containers. For the provision of auto scaling, we used as metric the response time of web applications, a metric that directly impacts the user's perception. The proposed method uses an architecture that integrates Big Data tools, containers orchestrators, control and optimization techniques. The experimental scenarios were tested with different workload types configured from real traffic in a high demanding web system. The results show that in most of the cases, our proposal PAS-CA has a better efficiency in containers allocation while keeping response time under the defined threshold.

Our proposal was compared with a Google auto scaling tool called HPA and the results show that PAS-CA has potential to become an auto scaling alternative for cloud systems based on containers. In the scenarios evaluated, PAS-CA obtained better efficiency for containers allocation than HPA. The future work of this research intends to integrate other metrics of interest for the provision of auto scaling, such as CPU, memory and tail latency thresholds.

References

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

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

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

[4] Crovella, Mark E and Bestavros, Azer. Self-similarity in World Wide Web traffic: evidence and possible causes. Networking, IEEE/ACM Transactions on 1997.

[5] Kettani, Houssain and Gubner, John and others. A novel approach to the estimation of the Hurst parameter in self-similar traffic. Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on 2002.

[6] National Instruments. Explicando a Teoria PID. http://www.ni.com/white-paper/3782/pt/, 2015.

[7] Lorido-Botrán, Tania and Miguel-Alonso, José and Lozano, Jose Antonio. 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, 2012.

[8] Poddar, Rishabh and Vishnoi, Anilkumar and Mann, Vijay. HAVEN: Holistic load balancing and auto scaling in the cloud. 2015 7th International Conference on Communication Systems and Networks (COMSNETS). 2015.

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

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

[11] Thrun, Sebastian and Montemerlo, Mike and Dahlkamp, Hendrik and Stavens, David and Aron, Andrei and Diebel, James and Fong, Philip and Gale, John and Halpenny, Morgan and Hoffmann, Gabriel and others. Stanley: The robot that won the DARPA Grand Challenge. Journal of field Robotics 2006.

[12] Zhang, Qi and Cheng, Lu and Boutaba, Raouf Cloud computing: state-of-the-art and research challengesChallenge. Journal of internet services and applications 2010.

[13] Mastelic, Toni and Oleksiak, Ariel and Claussen, Holger and Brandic, Ivona and Pierson, Jean-Marc and Vasilakos, Athanasios V. Cloud computing: Survey on energy efficiency. ACM Computing Surveys (CSUR). 2015.

[14] Abel Avran. Google Announces Cloud Container Engine Using Kubernetes. https://www.infoq.com/news/2014/11/google-cloud-container-engine, 2016.

[15] Docker. The Definitive Guide to Docker Containers, year = 2016. https://www.Docker.com/sites/default/files/WP-\%20Definitive\%20Guide\%20To\%20Containers.pdf, 2016.

[16] Felter, Wes and Ferreira, Alexandre and Rajamony, Ram and Rubio, Juan. Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. Journal of field Robotics 2015.

[17] Jeffrey Schwartz. Are Containers the Beginning of the End of Virtual Machines? https://virtualizationreview.com/articles/2014/10/29/containers-virtual-machines-and-docker.aspx 2016.

[18] VMware. What is Virtualization? https://http://www.vmware.com/solutions/virtualization.html 2016.

[19] Sefraoui, Omar and Aissaoui, Mohammed and Eleuldj, Mohsine. OpenStack: toward an open-source solution for cloud computing International Journal of Computer Applications. 2016.

[20] Linux. Man-page Namespaces(7) http://man7.org/linux/man-pages/man7/namespaces.7.html 2016.

[21] RedHat. Resource Management Guide https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html 2016.

[22] Kan, Chuanqi. DoCloud: An elastic cloud platform for Web applications based on Docker Advanced Communication Technology (ICACT), 2016 18th International Conference on. 2016.

[23] Müller, Carlos and Truong, Hong-Linh and Fernandez, Pablo and Copil, Georgiana and Ruiz-Cortés, Antonio and Dustdar, Schahram. An Elasticity-aware Governance Platform for Cloud Service Delivery. Services Computing (SCC), 2016 IEEE International Conference on. 2016.

[24] Abranches, Marcelo; Solis, Priscila. An Algorithm Based on Response Time and Traffic Demands to Scale Containers on a Cloud Computing System 15th IEEE International Symposium on Network Computing and Applications (NCA 2016). 2016.

[25] Amazon. AWS, Tutorial: Scaling Container Instances with CloudWatch Alarms. http://docs.aws.amazon.com/AmazonECS/latest/developerguide/cloudwatch_alarm_autoscaling.html 2017.

[26] Amazon. Deploying Elastic Beanstalk Applications from Docker Containers. http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_docker.html 2017.

[27] Hanzel Jesheen. Auto Scaling with Docker. https://botleg.com/stories/auto-scaling-with-docker/ 2017.

[28] Apache. Apache HTTP Server benchmarking tool http://httpd.apache.org/docs/2.4/programs/ab.html 2017.

[29] Squazt. Implementation of the Rice University Bidding System (RUBiS). https://github.com/sguazt/RUBiS 2015.

[30] CGU. portal_transparencia Portal da Transparência http://transparencia.gov.br/ 2016.

[31] Jain, Raj. The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling John Wiley & Sons. 1990.

[32] Walberg, Sean. Automate system administration tasks with puppet Linux Journal. 2008.