Cloud Adaptation with Control Theory in Industrial Clouds

Cornel Barna, Marios Fokaefs, Marin Litoiu and Mark Shtern
Department of Electrical Engineering and Computer Science
York University
Toronto, Canada
Email: cornel@cse.yorku.ca, fokaefs@yorku.ca, mlitoiu@yorku.ca, mark@cse.yorku.ca

Joe Wigglesworth
IBM Canada Ltd
Toronto Lab
Markham, Canada
Email: wiggles@ca.ibm.com

***Abstract*—The volatility of web software systems, for example due to traffic fluctuations, can be addressed through cloud resource elasticity. Cloud providers offer specific services to automate the process of elasticity, so that application developers can efficiently and effectively manage their cloud resources. Current autoscaling methods mostly employ rule-based or threshold-based techniques. In this work, we discuss a more sophisticated and robust method based on control theory. We present the design for a simple controller and how it can be applied on real cloud environments. We demonstrate the applicability of our controller by deploying it on two cloud environments, one public and one private. Our experiments show that the same controller functions accordingly and maintain the set performance goal in both environments, indicating the potential portability of the controller across clouds.**

I. Introduction

Software systems on the web can be significantly volatile both when it comes to expected changes (e.g. traffic fluctuations during a given period) or unanticipated ones (e.g. hardware or network failures or malicious activity). Thanks to cloud technologies, web application developers are offered the capabilities to manage the virtual infrastructure supporting the software, and reserve and decommission resources at will and on demand through *elasticity*. Cloud Elasticity, as a general concept, refers to the capability of the cloud to provide IT resources and software on demand. Elasticity is implemented using the concept of the autonomic computing MAPE loop, as introduced by IBM [1] consisting of the phases of Monitoring, Analysis, Planning and Execution phases.

Industrial, public, and private cloud providers have increased their efforts and provisions in systematizing and, at certain levels, automating the process of scaling. Current deployed efforts mostly focus on monitoring and execution. In the analysis phase, they employ mainly rule-based, threshold-based or, at most, simple statistical analyses to identify problematic cases in the deployed system's state. These methods are simple enough, quite straightforward, and, although they may not always be optimal [2], they have proved to be functional. These analyses usually result in simple scaling plans, which add or remove VMs in a topology, most often in static steps defined at design time, e.g. one or two at a time.

At the same time, research is pushing towards more sophisticated analyses and a more complex and multi-dimensional planning phase. On one hand, it is becoming evident that not all "bad" situations can be rectified with a single-dimensional strategy. Therefore, autonomic management systems need to properly identify problems, decide and plan for the best adaptive action for each problem. On the other hand, the ability to accurately identify unhealthy situations given large and frequent perturbations is becoming imperative. For this reason, researchers are looking towards more robust and formal methods, including among others discrete optimization algorithms all the way to control theoretic approaches.

Despite their mathematical foundation and their technical robustness, these methods may be too complex for engineers and go beyond their requirements or their capacity. However, there can exist middle-ground solutions with less design complexity, and without compromising exceedingly in robustness. For example, less attention has been paid to techniques of medium complexity, which are fairly robust, but require less expertise to be designed than more complicated techniques like optimization. In this paper, we discuss such a technique, known as Proportional-Integral-Derivative (PID) controllers, and present how it can be applied within an actual cloud environment. These controllers are based on objective functions, which calculate the divergence (error) of performance metrics from a given goal (e.g. to avoid CPU saturation in a VM, or maintain low response time) and issue adaptive actions to minimize the error. Although not an entirely novel method in the research community [30], controllers have not yet gained momentum in industrial applications. The goal of our work is to demonstrate the applicability of controllers and guide practitioners to use them in real environments.

Another dimension when considering the elasticity strategy of a deployment is the cloud environment itself. Although a relatively new technology, cloud computing offers a great variety of architectures and configurations in which one can deploy their web applications; from public to private solutions, from industrial to open-source solutions and so on. Apart from architectural differences and offered services, clouds may differ in terms of their capacity. Variability can greatly influence the system's performance and the effectiveness of certain rule-based policies. This fact indicates that rules, thresholds and the associated policies may not be portable, which hinders the task of elasticity and increases the workload of application

978-1-5090-3684-4/16 $31.00 © 2016 IEEE
DOI 10.1109/IC2EW.2016.13

engineers. In addition to PID, in this work, we also explore the differences between an industrial public cloud, Amazon EC2 [3], and a community-driven research cloud environment, SAVI [4], [5], a custom OpenStack implementation. We also deployed PID controllers in both clouds and examine the existence of any differences and investigate the reasons behind these differences, if any. In the context of our experimentations, we viewed Amazon EC2 as a paid public cloud, and SAVI as a private cloud, given our higher degree of control over the latter.

In summary, this paper makes three distinct contributions:

1) **PID controller in real cloud:** The simplicity in designing the PID controller along with its robustness and formality as a mathematical apparatus makes it an attractive solution to address the problem of autonomic elasticity. We demonstrate how PID can be developed and used in real cloud environments. Its applicability and efficiency is evident in our experiments.
2) **PID in multiple clouds:** We have applied the developed PID controller in two real-world clouds and studied its applicability and efficiency under the difference that one is a public cloud, while the other is private.
3) **PID portability:** Our experiments indicate that a PID controller designed for a particular web application can be deployed on any cloud environment. In our experiments, we deployed our test application with its PID controller in both Amazon and SAVI clouds and the controller worked as expected in both environments. The expectation was that the controller will keep the behavior of the system close to the goal, thus guaranteeing a standard level of performance. Although our experiments may not be extensive enough to be perceived as concrete proof, our results are a first indication of PID's portability.

The rest of the paper is organized as follows. Section II provides an overview of the related work. Section III describes the design process and the properties of the PID controller. In Section IV, we discuss the two cloud environments we use in our experiments, Amazon EC2 and SAVI, and their differences. Section V presents our experiments with PID in the two clouds. Finally, Section VI concludes this work.

II. Related Work

Adaptive systems have been proposed to automatically manage web applications and react to change. An adaptive system is a system capable to function properly, within parameters defined by the Service Level Objective (SLO), without human intervention [6]. The system is capable to extract data from the environment where the web application resides (using a series of sensors), analyze it (identify problems that might prevent the application to function optimally or within parameters), create an adaptation plan (if necessary) and implement it. The web application and the resources it uses become the *managed resources*, while the rest of the system are part of the *application manager*. Architecturally, anutonomic systems follow the Monitor-Analyze-Plan-Execute (MAPE-k) loop sugested by IBM [6].

The multi-tenant nature of the cloud, that allows multiple independent applications to share the same hardware, makes cloud-deployed applications more challenging to manage [7]. The simplest type of adaptation strategy that can be designed is one using policies, which are essentially event-condition-action (ECA) rules: *when **event** happens, if **condition** is true, then execute **action***. These rule-based systems have been investigated to some extent [8], [9], [10].

At the heart of an autonomic system is the decision-making process on when adaptation is required, in other words, how to identify the location of the problem and how to optimally determine the type and quantity of resources that need to be added or removed. To analyze the data and create an action plan, some authors turned to models. Zahorjan et al. [11], Eager and Sevcik [12], Lazowska et al. [13], and Reiser and Lavenberg [14] have presented methods to analyze a system from a performance point of view. Balbo and Serazzi [15], and Litoiu et al. [16] have additionally considered how the potential structure of the workload may influence the performance of the deployed system, how bottlenecks shift when the workload mix changes and when the resources become saturated. A method to uncover the worst workload mix and the minimum population required to saturate a system is presented by Barna et al. [17].

Any hardware-software system can be modelled by two layers of queuing networks [18], [19]; one that describes the software resources and the other one for the hardware resources. The queueing theory has been successfully applied to provide elasticity to cloud web applications, even when they are under Denial of Service attacks [20], [21], [22].

Current state-of-the-art cloud environments implement and promote elasticity by offering software services for the automatic scaling of a cloud topology. More specifically, Amazon [23] offers autoscaling capabilities in conjunction with EC2 and CloudWatch, its monitoring service. The developer can set up CloudWatch alarms to go off when particular metrics go beyond specified thresholds (above or below the threshold) and corresponding adaptive actions are triggered to add or remove servers. Alternatively, if the workload pattern is known, the developer can set up *scaling schedules* to adapt the topology even without CloudWatch. OpenStack [24] offers a similar service as a result of the collaboration between its monitoring service, Ceilometer, and its orchestration service, Heat. In both cases, the scaling policies are threshold-based and result in over-simplified adaptive actions (i.e. add/remove servers). Additionally, the services can react only to threshold violations for metrics that the respective monitoring services can measure. Given that the aforementioned monitoring services were primarily designed for billing purposes, they fall short in capturing important metrics, such as response time, service throughput or application level metrics.

Beyond threshold-based and rule-based techniques, there are various proposed techniques ranging from discrete optimization algorithms to control theoretic approaches. For example,

Li et al. [25] propose optimization deployments using bin packing algorithms augmented with integer programming to minimize application response time and infrastructure cost. Other approaches use control theory specific approaches such as model predictive control optimization [26] or other control methods [27], [28], [29], [2]. Depending on the problem and the cost function to be achieved, but also taking into account other criteria, such as maintainability, evolution, cost of development, elasticity designers and implementers can choose an industrial rule based approach or control and optimization based techniques [2].

Proportional-Integral-Derivative (PID) controllers were first introduced as autonomic management components for web applications by Gergin et al. [30], under the assumption that the application was multi-tier and that each tier was implemented in a cluster (i.e. multiple virtual machines performing the same task). Also, it was assumed that the service's demands for resources (e.g. CPU, memory etc.) of each tier was known and constant. In this paper, we deal with more realistic assumptions: not all tiers are clustered and we only have access to the response time of the application and to CPU utilization of the clustered tier, which we can monitor. In practice, we do not assume a model of the application or a model of the cloud as it happens with complex optimization algorithms.

Concerning the deployment models for clouds, the National Institute of Standards and Technology (NIST) [31] differentiates between them based on usage and boundaries. According to NIST, the private cloud is *"provisioned for exclusive use by a single organization comprising multiple consumers"*, while the public cloud is *"provisioned for open use by the general public"*. Dillon et al. [32] cite the motivation behind using a private cloud including taking advantage of in-house resources, higher security, and lower data transfer cost from the organization to the public cloud. Armbrust et al. [33] expand the motivation for a private cloud if workload stays constantly high and sufficient utilization of resources is maintained, while when demand fluctuates it is more cost-efficient to use the public cloud with a pay-per-use pricing model. Finally, Sotomayor et al. [34] comment on the additional challenges in the private cloud of having to offer proper and uniform management services for the virtual infrastructure besides also maintaining and managing the physical infrastructure.

III. Proportional-Integral-Derivative Controller (PID)

The Proportional-Integral-Derivative (PID) is the most used controller in industry, reaching 90% of all instalments in some industries [35]. The power of PID comes from its simplicity but also from its effectiveness. Conceptually, the error, $e(t)$ (cf. Fig 1) is the difference between the observed (y) value of the metric and the set goal (y^{goal}) for it. The error is processed by the PID controller and fed back to the controlled system as input u. The input in the process (software system) will then consist of three components: a proportional, an integral and a derivative of the error. The individual components to calculate the input u at time t are aggregated in one term as follows:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de(t)}{dt} \quad (1)$$

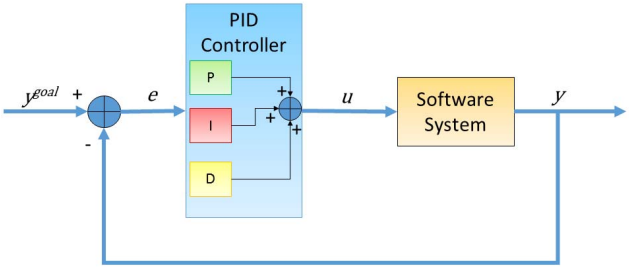


Fig. 1. A PID feedback control loop

The first term, *proportional control* (P), adjusts the command u in direct proportion to the input (the error). The adjustable parameter, K_p is the *proportional gain coefficient*. It signifies the importance of the current errors and the resulting output is positively and highly correlated with the error (e.g. if the error is large and positive, the output will also be large and positive). The second term, *the integral control*, takes into account the error history and integrates it. That is, it cumulates the error based on historical observations. The adjustable parameter, K_i, is called the *integral gain coefficient*. It signifies the ability of the controller to reduce the cumulative effect of the error. For example, if the error accumulates over time, this parameter will dictate the controller to apply even stronger outputs. The third term, *the derivative control*, anticipates where the process is heading by looking at the rate of error. The derivative gain coefficient, K_d, is tunable. This parameter contributes to proactive output from the controller in an effort to correct anticipated errors.

The PID coefficients, K_p, K_i and K_d values can be manually or automatically tuned [35] in such a way that the quality of control indices (overshoot, steady error, rising time) meet some predefined trajectory. It is largely expected that the controller design will either have the expertise or the deep knowledge of application domain to experimentally and manually find and refine the coefficients. Alternatively, the designer could have possession of models that describe the application, from which the coefficients could be automatically or systematically extracted.

The main advantage of PID controllers is that they can be implemented without detailed and extensive knowledge of the controlled system. Compared with optimization control schemes, PID does not require a model of the application or cloud. The only design issue is the tuning of the parameters K_i, K_d, K_p and this can be done experimentally, manually or automatically. PID controllers have also been proved robust enough to make them applicable to a large number of application domains.

Regarding PID and clouds, in a related work of ours [30], we introduced a PID controller to control the response time

of an application deployed in cloud. It is based on the fact that response time of an application depends on the degree of CPU utilization (and consequently saturation) of its servers; the more saturated the servers become the higher the response time. While the saturation of other resources (memory, disk etc.) may also affect the response time, we narrow our scope to computationally intensive application that will most likely saturate CPU utilization over other resources. This is done without loss of generality, as the PID is defined independently from the resource, which it controls. Formally, in the context of CPU heavy applications, the mean response time R_i of a server and of a cluster is given by [36]:

$$R_i = \frac{D_i}{1 - U_i} \tag{2}$$

where U_i is the mean overall utilization of the cluster and D_i is the mean demand of the application for CPU.

From the equation above, one can compute the objective utilization U_i^{goal} of tier i that will keep the response time at the desired level R_i^{goal}:

$$U_i^{goal} = \frac{R_i^{goal} - D_i}{R_i^{goal}} \tag{3}$$

In an ideal situation, by implementing a PID controller for each tier of the application, and assuming no interference between controllers, we can keep the response time constant by keeping U_i^{goal} constant. In reality, D_i changes over time and the response time is influenced by other factors besides the mean utilization. In addition, not all the tiers are clustered and their utilization can be controlled dynamically. For example, most web applications are still deployed with a large database running on a single powerful machine and the application logic tier is deployed on a cluster that can be scaled up and down.

In the context of this particular paper, we investigate and evaluate PID controllers under the following conditions:

1) We can auto-scale only one tier of the application.
2) The goal of the controller is to maintain utilization of the tier, which can be monitored, to certain levels. By extension (Eq. 2), we also control response time of this tier.
3) The output of the PID controller is the number of servers in the controlled cluster.

The purpose is to evaluate the PID's ability to scale a cluster (i.e. a tier). In this sense, we want to see how the PID performs on a single cluster, without the potential interference from other controlled clusters (through PID or other controllers) or without any performance bottlenecks from the other tiers (for example, saturation of the data cluster may cause saturation to the web cluster due to waiting time). Other factors that may affect even the performance of a single PID on a single cluster may also include:

1) large and fast variations of the workload might not be easy to track by the controller;
2) the influence of other tiers on the overall throughput and therefore on the the controlled utilization might render the control unstable;
3) the variations of the response time will be too high because we control the response time at one tier (cf Eq. 2) and even there the control is not perfect.

Factor 2 may be minimized by assigning large enough VMs to those tiers to ensure that they will have enough capacity without additional delay. In addition, we assume that the workloads are limited in intensity and they do not cause the saturation of the uncontrolled tiers. Eventually, the goal is to create evidence that the PID controller can replace the auto-scaling rules [2].

IV. Amazon EC2 and SAVI Clouds

In this work, we focus our study and experiments on Amazon EC2 and SAVI clouds. Amazon EC2 is an industrial public cloud built for elasticity. It offers a variety of high capacity resources, including computation, network and storage resources and there is high availability guaranteed. Its services allow for efficient management of resources by offering comprehensive APIs to change the topologies, both horizontally (add or remove resources) and vertically (changing size or type of resources).

SAVI is a research cloud environment built on top of OpenStack. The novelty of SAVI, especially compared to Amazon EC2, is its tiered architecture; SAVI has a *core* node acting as a high capacity datacenter and lighter *edge* nodes distributed geographically acting as local computation nodes or as a fault-tolerance mechanism. The nodes communicate with each other through a high bandwidth private network, making intra-communication fast and more secure. In another additional novelty, SAVI is using a single orchestration service, called Heat, in order to deploy and manage resources of a single topology not only within one node, but across multiple nodes as well.

In our experiments, we study these two cloud environments based on them being public or private. We examine the implication of this property on the PID controller's design and functionality.

A. Public vs Private Cloud

In the context of our work, we consider Amazon EC2 as a public cloud, as it is intended, and SAVI as a private cloud. As a public cloud, Amazon EC2 offers and guarantees high availability of its resources[1]. Given that high availability is part of an agreement, usually contractual, the public cloud provider must guarantee the promised levels of quality of service. On the other hand, the lack of such an explicit guarantee reduces the probability for high availability in a private cloud, although it would be irresponsible and damaging to have low resource availability. SAVI is also a research platform, which may pose additional challenges towards availability, either because of hardware or network failures, or because of low capacity and

[1] https://aws.amazon.com/ec2/sla/

high resource demands. Nevertheless, such challenges may be present even for industrial private clouds, either because of limited resources or lack of expertise or dedication in cloud computing. Eventually, availability is the direct consequence of cloud capacity in terms of hardware resources. In this sense, public clouds can offer high availability due to high cloud capacity as they expect a potentially large number of end-users.

The most significant difference between public and private cloud is cost. In public clouds, the cost is straightforward as virtual machines are charged per hour of operation[2]. In private clouds, the cost for resources is usually considered as capital cost, charged once when they are purchased and then they are amortized in time. Variable costs that depend on the allocation and execution of virtual machines include mostly power expenses. In terms of variable costs, a private solution may be more cost-efficient than outsourcing to a public cloud. However, unprecedented costs, including hardware failures, may increase the total costs of running a private cloud making the decision to invest in private or public infrastructure an optimization problem under uncertainty.

The biggest advantage of private clouds against public ones is the level of control over the infrastructure. The cloud engineer can make changes to both hardware and software at will to fit individual needs. In SAVI cloud, we had this advantage, where we could require more hardware resources, release virtual resources in mass and modify OpenStack services. In addition, we have access to experimental services and concepts, products of research. The additional control also enabled us to more effectively addressed problems; access to specific logs gave us deeper insight to debug issues which are normally not visible in detail to end-users.

V. Experiments

To evaluate the applicability of the PID controller in maintaining a desired behaviour for a web application, and also to study the implications of different cloud environments to the controller's functionality, we conducted two experiments; one on Amazon EC2, as a public cloud, and one on SAVI, as a private cloud.

The controller was supposed to regulate the behavior of a three-tier web application on the topology shown in Figure 2. The goal of the controller was to maintain an average of 55% CPU utilization (y^{goal}) in the web cluster by adjusting the number of web servers in the cluster (u), as a response to fluctuation in the workload, which result in fluctuation in the CPU utilization (e). By assigning an average CPU utilization as the goal implies that the controller will maintain utilization *around* this point within a range ($\pm$ 15%). In practice, utilization is kept between 70% and 40%. Our experience and previous experiments have shown that these are regular thresholds since above the upper bound, a VM becomes saturated affecting the response time and below the lower bound, the VM is underutilized causing unnecessary costs.

[2]https://aws.amazon.com/ec2/pricing/

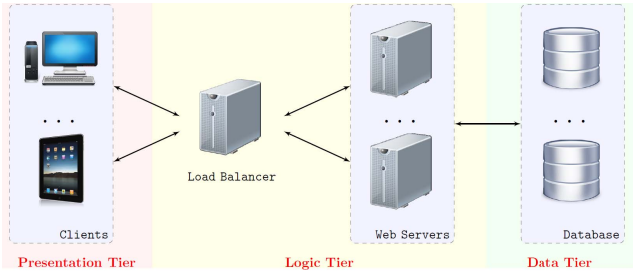


Fig. 2. Three-tier topology of the web application.

A. Experimental setup

To deploy our web application, we first had to decide on the sizes of VMs for each tier. For the worker servers of the web cluster, we chose small sized machines; 1VCPU, 2GB RAM and 20GB storage for SAVI; 1VCPU, 1.7GB RAM and 160GB storage for Amazon EC2. In order to avoid bottlenecks on the load-balancer and database servers, we chose large instances for both; 4VCPU, 8GB RAM and 80GB storage for SAVI; 2VCPU, 7.5GB RAM and 2x420GB storage for Amazon EC2.

We developed our own workload generator to simulate the same number of users that would behave in the exact same ways for both experiments. A user would make a request to the web application, wait for the reply (this waiting time is the response time), and then wait 500 milliseconds (this waiting time is considered to be the thinking time, or the time required for a user to read and process the reply from the server). According to this behavior, an increase in response time would result in a decrease in the number of requests made over a period of time (i.e. the arrival rate). The experiments start with a constant workload, and then we add variations.

After a short period of experimentation and manual training, the coefficients for the controller were set to:

- proportional coefficient $K_p = 3$
- integral coefficient $K_i = 0.6$
- derivative coefficient $K_d = 0$

In the context of our experiments, the coefficients imply that, first, we put higher emphasis on the present error. This means that we want the PID to produce output to immediately correct any errors that may arise. Second, we put a lower emphasis, but not zero, to the cumulative error. This means that we want the controller to fix the cumulative error only when it becomes high. This makes sense since we want the utilization to stay within a range and not close to a single number. Finally, we give no emphasis to future error. Given that the workload can be very volatile with the possibility that no discernible patterns may arise to enable predictions, we don't want future errors to affect the output of the controller negatively.

To handle the automation in topology deployment, monitoring it in a continuous way, creating and implementing the action plan as identified by the controller, we have used the Hogna framework [37]. Hogna is a modular and pluggable architecture that clearly separates the four components of

the MAPE loop. In this regard, we can easily replace any analysis component with the PID controller. Additionally, Hogna provides all the metrics either through the monitoring service of the host cloud or by registering to additional agents, which go beyond the cloud's monitoring capabilities. For example, we have an agent deployed to monitor response time on the application level.

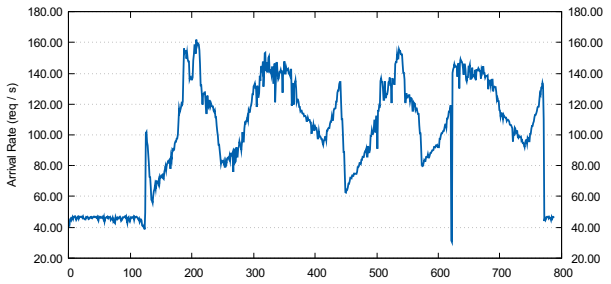
Besides the controller, we have also implemented "proxies" to deploy Hogna in clouds along with the application. These proxies essentially invoke the cloud's APIs either to pull monitoring data or to execute the adaptive plan. One difference between the proxies for the two clouds, Amazon EC2 and SAVI, is that the former is using a flat VM deployment while the latter is using an orchestrated deployment. A flat deployment implies that the proxy directly invokes the service responsible for creating and starting instances and then the proxy is responsible for installing software to each VM and creating the dependencies between them to define the topology. In the orchestrated method, the deployment is specified in an abstract way through a *template* that already defines the components of the topology, the software in each component and their relationship. When the template is implemented concrete parameters are passed as input to create the actual topology. The two models have differences on the level of managing the topology, since flat topology may give the engineer more control, while orchestration promotes reusability and maintenance. Nevertheless, we found no variation in the performance of the two clouds or of PID in them, which we could attribute to this difference.

B. Results

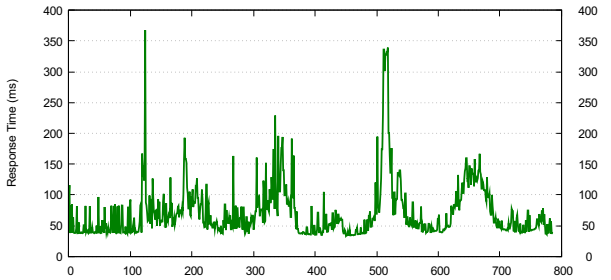
Figures 3 and 4 show the results for the SAVI and Amazon EC2 experiments respectively. One first observation is that the controller was able to maintain utilization around the set goal in both cases. In particular, for Amazon EC2, PID kept the utilization closer to 55%. However, according to the motivation behind setting the goal, PID was equally successful in the SAVI experiment in keeping CPU utilization between 70% and 40%.

The differences in the SAVI experiment can be explained by instabilities in that cloud caused by hardware issues, as it can be more probable in private clouds. For example, at the end of the SAVI experiment we generated the same constant workload used at the beginning. All the metrics (response time, arrival rate and CPU utilization) being similar with the beginning of the experiment, we investigated further why 3 VMs were required compared to the 2 used in the beginning. We found that because of a hardware error, one VM was unresponsive and only two were actually handling the workload.

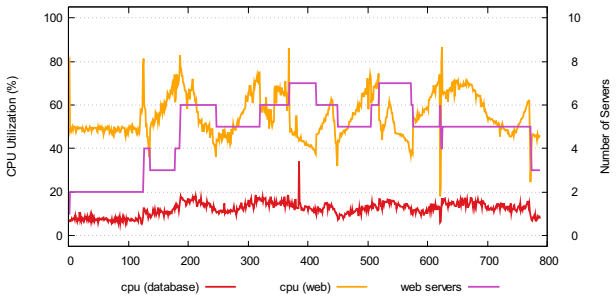
The experiments also revealed that the public and private instances have different processing power, although their nominal specifications are similar (same number of VCPUs and similar RAM size). Indicatively, in the private cloud we used maximum 6 instances, but in Amazon EC2 we used 9. Additionally, the constant workload was handled by only 2 instances in SAVI, but needed 3 instances on the public cloud.



(a) Arrival rate



(b) Response time

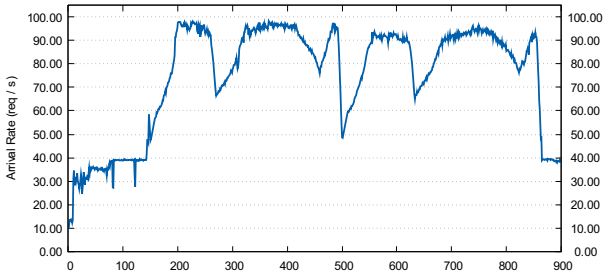


(c) CPU utilization and number of web servers.

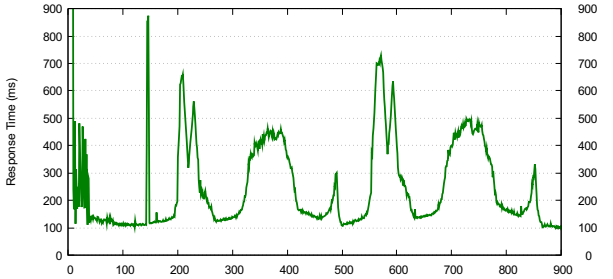
Fig. 3. PID controller on SAVI cloud.

This difference is another explanation for the greater variations in SAVI. Given that we can only add or remove servers in integer numbers (e.g. 1 at a time or 2 at a time), in SAVI with greater processing power, adding or removing one server has significant impact on CPU utilization. For example, notice how significantly utilization fluctuates when we add or remove one server around samples 250 and 300. This impact is less significant in Amazon EC2. In theory, the PID controller can issue decimal inputs u (which, in practice we round up), so in principal the CPU utilization would be kept around the goal.

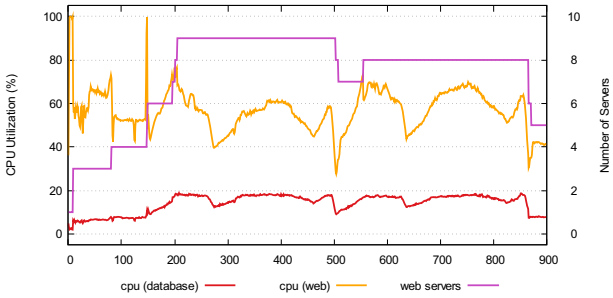
We have also noticed that there is a difference in shape of the response time and arrival rate between the two experiments (see figure 3b vs. figure 4b and figure 3a vs. figure 4a). In the SAVI cloud the overall response time is closer to the minimum possible (aprox. 50 ms) while in the public cloud there is a more significant deviation that follows the arrival



(a) Arrival rate



(b) Response time



(c) CPU utilization and number of web servers.

Fig. 4. PID controller on Amazon EC2 cloud.

rate (in the public cloud the minimum possible response time was around 100 ms). The arrival rate in the public cloud seems to flaten when more users are accessing the system, and doesn't have the variations observable in SAVI (altough, in both cases we simulated the same number of users that behaved in the same way). This difference cannot be attributed to saturated instances (the CPU utilization shows that there is still capacity to handle more workload, see figure 4c). After further investigation, we found that this bahaviour appears because the network became saturated. In SAVI, all the traffic was sent over a private network, with large bandwidth. For the Amazon EC2 experiment, the traffic was sent over the public network, with limited capacity that became saturated.

Another observation is the ability of the PID to alternate between adding or removing 1 or 2 servers at the time. This indicates the dynamic nature of adaptation provided by the PID controller, which is an advantage over traditional rule-based or threshold-based techniques as it can allow the system to react faster to large changes.

In spite of the differences between the private and the public clouds, the same PID controller was able to achieve the goal in both environments. This is an indication that the PID controller is portable between cloud environments. Admittedly, our experiment sample (two clouds, one application) is statistically small to make any claim about a concrete proof towards the controller's portability. However, this indication motivates us to further explore this possibility with additional, more extensive experiments. Nevertheless, having attributes the differences in trace between the two experiments to properties specific to each cloud, we can deduce that the same PID controller for the web application works in both cases. This is a reasonable deduction given that the controller is designed for a given metric and on how the controller's designed ranks the importance of absolute divergences from a set goal (K_p), the cumulative effect of these divergence (K_i) and the change rate of the divergences (K_d). How the metric is measured in each cloud is irrelevant for the controller.

VI. Conclusion

We have presented a control-theoretic approach to achieve automatic scaling of web applications deployed on cloud environments. We designed a Proportional-Integral-Derivative (PID) controller for a web application and deployed it on two real clouds, Amazon EC2, perceived public, and SAVI, perceived private. Our experiments showed that the PID controller maintained the predefined performance goal effectively in both clouds. This provided us with a first indication of the controller's potential portability.

The experiments also indicated important differences between private and public clouds. These differences mainly concentrated around availability, cloud capacity and resource capacity, and cloud variability. Despite these differences, reflected in the measured performance of the deployed application, the PID controller was able to maintain the goal.

This work has demonstrated partially the potential of control-based autoscaling methods. There is still significant margin to further investigate the full capabilities of such methods. The first step for our future work is to considerably extend our experiments, expanding to more cloud environments, more applications and more complicated settings to establish and possibly prove the ability of the PID controller to be portable across cloud environments. Our second future goal is examine the capabilities for a network of PID controllers, known as *cascaded PID architectures*. In such architectures, each PID can be responsible for a single metric producing one kind of adaptive actions. Combining and orchestrating such PID controller will enable multidimensional analyses and planning in autonomic cloud management systems.

Acknowledgment

This research was supported by IBM Centres for Advanced Studies (CAS) and the Natural Sciences and Engineering

Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network.

REFERENCES

[1] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, "A design space for self-adaptive systems," in *Software Engineering for Self-adaptive Systems II*. Springer, 2013, pp. 33–50.

[2] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 716–723.

[3] Amazon, "Amazon EC2," https://aws.amazon.com/ec2/.

[4] J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia, "Savi testbed: Control and management of converged virtual ict resources," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE, 2013, pp. 664–667.

[5] SAVI, "Smart Applications on Virtual Infrastructure," http://www.savinetwork.ca/.

[6] "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2005.

[7] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010. [Online]. Available: http://dx.doi.org/10.1007/s13174-010-0007-6

[8] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended kalman filters," in *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005, pp. 334–345.

[9] T. Söderström and P. Stoica, *System identification*. Prentice-Hall, Inc., 1988.

[10] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster computing*, vol. 12, no. 1, pp. 1–15, 2009.

[11] J. Zahorjan, K. C. Sevcik, D. L. Eager, and B. I. Galler, "Balanced job bound analysis of queuing networks," in *SIGMETRICS '81: Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1981.

[12] D. L. Eager and K. C. Sevcik, "Performance bound hierarchies for queuing networks," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 99–115, 1983.

[13] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queuing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[14] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queuing networks," *J. ACM*, vol. 27, no. 2, pp. 313–322, 1980.

[15] G. Balbo and G. Serazzi, "Asymptotic analysis of multiclass closed queuing networks: multiple bottlenecks," *Performance Evaluation*, vol. 30, no. 3, pp. 115–152, 1997.

[16] M. Litoiu, J. Rolia, and G. Serazzi, "Designing process replication and activation: A quantitative approach," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1168–1178, 2000.

[17] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in *Autonomic Computing, 2011. International Conference on*, ser. ICAC '11. New York, NY, USA: ACM, June 2011, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998598

[18] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995.

[19] D. A. Menascé, "Simple analytic modeling of software contention," *SIGMETRICS Performance Evaluation Review*, vol. 29, no. 4, pp. 24–30, 2002.

[20] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu, "Model-based adaptive dos attack mitigation," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, ser. SEAMS 2012. New York, NY, USA: ACM, 2012, pp. 119–128. [Online]. Available: http://dx.doi.org/10.1109/SEAMS.2012.6224398

[21] ——, "Mitigating dos attacks using performance model-driven adaptive algorithms," *ACM Trans. Auton. Adapt. Syst.*, vol. 9, no. 1, pp. 3:1–3:26, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2567926

[22] C. Barna, M. Shtern, M. Smit, H. Ghanbari, and M. Litoiu, "Model-driven elasticity and dos attack mitigation in cloud environments," in *11^{th} International Conference on Autonomic Computing (ICAC 2014)*. Philadelphia, PA: USENIX Association, Jun 2014, pp. 13–24.

[23] Amazon, "Autoscaling," https://aws.amazon.com/autoscaling/.

[24] Openstack, "Heat: Openstack Orchestration," https://wiki.openstack.org/wiki/Heat.

[25] J. Z. Li, M. Woodside, J. Chinneck, and M. Litoiu, "Cloudopt: multi-goal optimization of application deployments across a cloud," in *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011, pp. 162–170.

[26] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna, "Replica placement in cloud through simple stochastic model predictive control," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 80–87.

[27] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems*. John Wiley & Sons, 2004.

[28] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 299–310.

[29] Y. Lu, T. Abdelzaher, C. Lu, L. Sha, and X. Liu, "Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*. IEEE, 2003, pp. 208–217.

[30] I. Gergin, B. Simmons, and M. Litoiu, "A decentralized autonomic architecture for performance control in the cloud," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 574–579.

[31] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.

[32] T. Dillon, C. Wu, and E. Chang, "Cloud computing: issues and challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Ieee, 2010, pp. 27–33.

[33] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[34] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *Internet computing, IEEE*, vol. 13, no. 5, pp. 14–22, 2009.

[35] K. J. Aström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[36] A. Zhang, P. Santos, D. Beyer, and H. Tang, "Optimal server resource allocation using an open queueing network model of response time," *HP laboratories Technical Report, HPL2002301*, 2002.

[37] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern, "Hogna: A Platform for Self-Adaptive Applications in Cloud Environments," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10^{th} International Symposium on*, May 2015, pp. 83–87. [Online]. Available: http://dx.doi.org/10.1109/SEAMS.2015.26