Managing a SaaS Application in the Cloud Using PaaS Policy Sets and a Strategy-Tree

Bradley Simmons, Hamoun Ghanbari and Marin Litoiu
York University, Canada
{bsimmons || mlitoiu}@yorku.ca, hamoun@cse.yorku.ca

Gabriel Iszlai
IBM, Toronto Lab, Canada
giszlai@ca.ibm.com

***Abstract*—This paper introduces a framework and a methodology to manage a SaaS application on top of a PaaS infrastructure. This framework utilizes PaaS policy sets to implement the SaaS providers elasticity policy for its application server tier. A strategy-tree is utilized at the SaaS layer to actively guide policy set selection at runtime in order to maintain alignment with the SaaS providers business objectives. Results from an experiment conducted on a real cloud are presented in support of this approach.**

I. Introduction

Cloud computing [1–4] represents an approach to IT which has emerged in large part due to improvements in virtualization technologies [5, 6] and the construction and commoditization of large data centers from which infrastructure (IaaS), platform (PaaS) and software (SaaS) are provided on-demand to end users over the Internet.

A *PaaS provider* is an enterprise that is responsible for leasing application environment topologies to SaaS provider clients for various durations of time. Further, they are responsible for adding and removing certain component elements on-demand to and from the client as necessary. The topologies are built upon infrastructure purchased from various IaaS providers upon which the middleware container instances are run. An application environment topology is composed of a set of system service instances S (e.g., load balancers, LDAP servers, ...), platform service instances P (e.g., web server, application server, database server, ...) and the set of licenses L to support all instances in P (should they be required). The number of instances of any platform service instance may be increased or decreased. A *SaaS provider* is an enterprise that provides a software offering that is run from within a PaaS topology.

Consider a SaaS provider running on a cloud. This SaaS provider leases a platform topology from a PaaS provider and offers one application to a dynamic set of clients. Several aspects of a platform topology may be configured dynamically via policy. A policy can be understood to represent "...any type of formal behavioural guide" that is input to the system [7]. An *elasticity policy* governs how and when resources (e.g., application server instances at the PaaS layer) are added to and/or removed from a cloud environment [8]. One way of specifying an elasticity policy is through a set of policy rules. It has been described previously [9] that a set of policies may be thought of as a strategy. Multiple strategies may be defined to achieve the same set of objectives [10].

In this paper we introduce a framework and a methodology to manage a SaaS application on top of a PaaS provider's infrastructure. This framework utilizes PaaS policy sets to implement the SaaS provider's elasticity policy for its application server tier. A strategy-tree is utilized at the SaaS layer to actively guide policy set selection at runtime in order to maintain alignment with the SaaS provider's business objective, specifically to *maximize profit*. Experimental results are presented that reflect positively on this approach.

The remainder of the paper is structured as follows. Section II reviews the concept of strategy-trees. Section III introduces the management architecture. Section IV introduces a scenario. Section V presents an experiment demonstrating the effectiveness of this approach in the context of the introduced scenario. Section VI presents a brief discussion. Section VII presents our conclusions.

II. Strategy-Trees

The strategy-tree was introduced to address a deficiency in current approaches to distributed system's management. Simply put, there can exist multiple strategies to achieve a set of objectives. These alternative strategies often incorporate assumptions, biases and expectations within a given policy set. Under different contexts various assumptions can be more/less correct than others resulting in different degrees of effectiveness for the various strategies.

In essence, a strategy-tree represents a framework for reasoning about the effectiveness of an active strategy. In this sense it is a tool for meta-policy management [11]. While everything it accomplishes might possibly be done using a set of highly complex and convoluted policies, this abstraction simplifies and organizes the process of evaluating the effectiveness of a deployed policy set and switching among alternative strategies over time in a defined, systematic and hierarchical manner. Further, this approach provides an architecture to facilitate this process of strategic management[1]. For a more comprehensive and formal consideration of strategy-trees and their use in policy management please refer to [9, 10, 12, 13].

[1]Strategy-trees are not meant to handle asynchronous problems. Changes in strategy are gradual and occur on scales of hours, days, weeks, months, years, etc. (i.e., not milliseconds). It is assumed that for gross, pathological errors there are policies defined to handle these situations. There is also overhead associated with deploying policy sets and this should not be ignored.

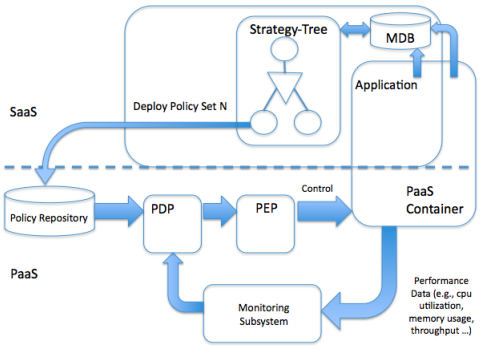


Fig. 1: Proposed management architecture

III. Architecture

The following section will provide an overview of our proposed management architecture, Figure 1.

A. PaaS Layer

At the PaaS layer, we assume the traditional policy-based management architecture (PBM) [14]. The PaaS provider has access, via a monitoring subsystem, to numerous performance metrics (e.g., cpu utilization, throughput). It also has access to various OS and middleware level metrics as well. We assume that the PaaS provider exposes these metrics to its SaaS clients so that they may define policy rules with which to implement their elasticity policies. Policy rules are specified in the traditional *On-event-If-condition-Then-action* syntax.

B. SaaS Layer

A strategy-tree is used at the SaaS layer to dynamically alter policy set deployment at run time based on monitored data.. The management database (MDB) is where this data is stored (e.g., how many platform instances have been purchased, how many sessions have been serviced). All elements of the strategy-tree have access to the MDB.

IV. Scenario

Consider a SaaS provider offering a standard multi-tiered application to a dynamically growing and shrinking set of clients. Revenue is proportional to the number of users (sessions) that utilize the service (as each user is statistically linked to some amount of advertising dollars). Cost is impacted by the (i) cost of purchasing the topology and (ii) additional platform instances purchase over time. There is also a (subjective) cost associated with the loss of future business which is a more speculative (and varies with client response time).

The objective of this SaaS provider is to maximize profit by both maximizing revenue and by minimizing cost. It should be noted that maximizing revenue can have an adverse effect on minimizing cost and vice versa. This interrelatedness greatly complicates the achievement of the main objective. Trade-offs must be made in the pursuit of the overall objective.

Parameter	$P_{Sensitive}$	$P_{Tolerant}$	$P_{Aggressive}$
`incr_val`	1	1	2
`decr_val`	1	1	2
`quorum`	51%	51%	51%
`cpu_idle_gt`	45	40	50
`grow_duration`	7 min	7 min	8 min
`cpu_idle_st`	50	55	55
`shrink_duration`	7 min	7 min	8 min
`refractory_period`	8 min	8 min	6 min

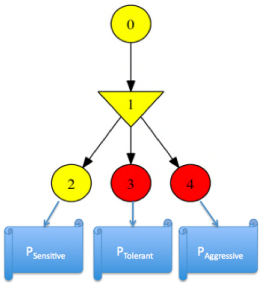
TABLE I: Parameter settings defining the three elasticity policies as used in the experiments (i.e., values related to time are scaled by one quarter).

A. Elasticity Policy

In a production setting, the elasticity policy might be highly complex in order to handle the numerous eventualities and situations that are likely to arise. However, in this illustrative scenario, several simplifying assumptions have been made in order to streamline and focus the discussion. It is assumed that the SaaS offering is tightly cpu-bound. This assumption allows us to focus on the single metric, `cpu_idle`, which is considered exclusively in the design of the policy set for this scenario. Further, the policy rules defining the elasticity policy focus only on the application server tier of the SaaS offering. In reality, an elasticity policy is meant to govern changes is resource allocation to all tiers of an application. A brief overview of the elasticity policies, utilized in this work, will now be presented (for a more complete overview please refer to [8]).

A lower threshold (`cpu_idle_gt`) and an upper threshold (`cpu_idle_st`) are defined on the value of `cpu_idle` for a platform instance. Should the value of *cpu_idle* be less than `cpu_idle_gt` for longer than a configured period of time (i.e., referred to as `grow_duration`) then a request to grow is made. Similarly, if the value of `cpu_idle` is greater than `cpu_idle_st` for longer than the `shrink_duration` then a request to shrink is made. At the second level of the hierarchy, if a configured percentage (referred to as a `quorum` of platform instances have indicated a request to grow (or to shrink) and is has been at least `refractory_period` of time since the last elastic action was taken, then `incr_val` platform instances are added to the platform (`decr_val` platform instances are removed).

Three different elasticity policies (i.e., $P_{Sensitive}$, $P_{Tolerant}$ and $P_{Aggressive}$) were designed, based on various heuristics, to drive the auto-scaling actions of the application server tier. These policy sets utilized different settings of some of the configurable parameters mentioned above and are presented in Table I. The SaaS provider was able to characterize the various elasticity policies against a standard workload (i.e., trace data that they had access to). For each policy set, the mean hourly number of additional platform instances was computed. They were also able to monitor the current number of sessions at four minute intervals. On this data they performed hourly regressions and partitioned the slopes into four distict categories (indicating different degrees of increase/decrease in number of sessions).



Index	Quantum Attribute Value (hours)
0	12
1	4
2	1
3	1
4	1

Fig. 2: Strategy-tree used for the experiment. Circles denote SAT-elements. A SAT-element is where the satisfaction of a set of objectives is periodically evaluated. This period is defined by the node's *quantum_attribute_value* which is equivalent to an *epoch*. Inverted triangles denote DEC-elements. A DEC-element is where decisions about whether to maintain the current strategy or to switch to an alternative are made. Yellow indicates the *active* strategy.

B. Design of the Strategy-Tree

This section considers the development of a strategy-tree, Figure 2, to help guide the system to achieve the objective of the SaaS provider (i.e., maximize profit). To achieve the objective, heuristic trade-offs between maximizing revenue and minimizing cost are utilized. A bias which favors servicing the maximum number of clients while attempting to limit the number of additional platform instances purchased is applied.

1) Design of SAT-Elements: The three leaf nodes (i.e., nodes two, three and four) and their parent node (i.e., node one) share a similar objective: *The number of additional platform instances purchased divided by the epoch should not exceed the hourly mean for that particular strategy.*

2) Design of the DEC-Element: In order to guide performance toward achieving the objective (i.e., maximize profit) it was decided that a two step approach would be utilized when deciding whether to continue using a particular strategy or whether to switch to an alternative. This decision would be based first upon the detection of a trend in the number of current sessions observed over the previous epoch. Specifically, the slopes of the hourly regressions (for the previous four hour epoch) constructed from the readings (i.e., current number of sessions) taken every four minutes would provide a simple heuristic for detecting a rapid increase or decrease in the client demand on the system (and hence guide the decision making process to use the more aggressive strategy i.e., $S_2 = P_{Aggressive}$). Should no strong trend be detected, data from the MDB, as indicated by the values of the *results* list, would then be utilized in the decision making process.

V. Experiment

The following experiment is based on the scenario presented in the previous section (i.e., Section IV) and is composed of two parts. First, the three elasticity policies (i.e., Table I) are characterized against a workload as described in Section IV-A. Then the strategy-tree (i.e., Figure 2) is deployed and each policy set and the strategy-tree are run against a novel workload and compared in terms of total sessions, number of additional platform instances purchased and mean response time as measured at the client.

A. Experimental Setup

For this experiment, Amazon (i.e., EC2, EBS) was used as the IaaS provider. All platform instances were built atop virtual machine instances (VMI)s running either CentOS 5.4 i386 (i.e., front end servers and application server instances) or Ubuntu 8.04 i386 (i.e., database) and configured as m1.small instances (i.e., 1.7 GB memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB instance storage, 32-bit platform and I/O Performance: Moderate).

RightScale was used as a PaaS management framework. A standard, multi-tiered application topology was selected from their catalog (with various modifications to suit our needs). This platform topology consisted of two front-end servers running Apache and HAProxy an array of Tomcat instances and a back end database running MySQL 5.0. The concepts of elastic scaling of a server array using alerts (based on voting tags) employed by Rightscale allowed us to specify our elasticity policies. We wrote lightweight Policy and PolicySet classes that were implemented in Ruby. Once fully specified, a PolicySet could be deployed (utilizing Rightscale's restful API) at which point it would result in the configuration of the platform topology with the correct elasticity policy as previously described.

The client is run on a separate EC2 instance and simulates the correct number of clients as defined by the workload for the duration of the experiment. The workloads used both to characterize the three elasticity policies and for the actual experiment were excerpts from the FIFA '98 workload [15] (Figures 3a and 3d).

Experiment time was scaled by four. The monitoring system at the SaaS provider takes a reading every minutes (i.e., four minutes of experiment time). At the SaaS provider layer, a simple Java-based web application was deployed on the described PaaS topology. A client connects to the front-end, is directed to an application server, a loop executes some pre-defined number of times (i.e., for this work we focused on the CPU) communication with the database tier occurs and a response is issued. For the remainder of the paper, this will represent a session.

B. Experimental Results

Initially, the three elasticity policies were characterized versus a workload, Figure 3a. As described previously various details were computed for use in the strategy-tree elements. Notice how different the system behaves under the alternative elasticity policies, Figures 3b and 3c.

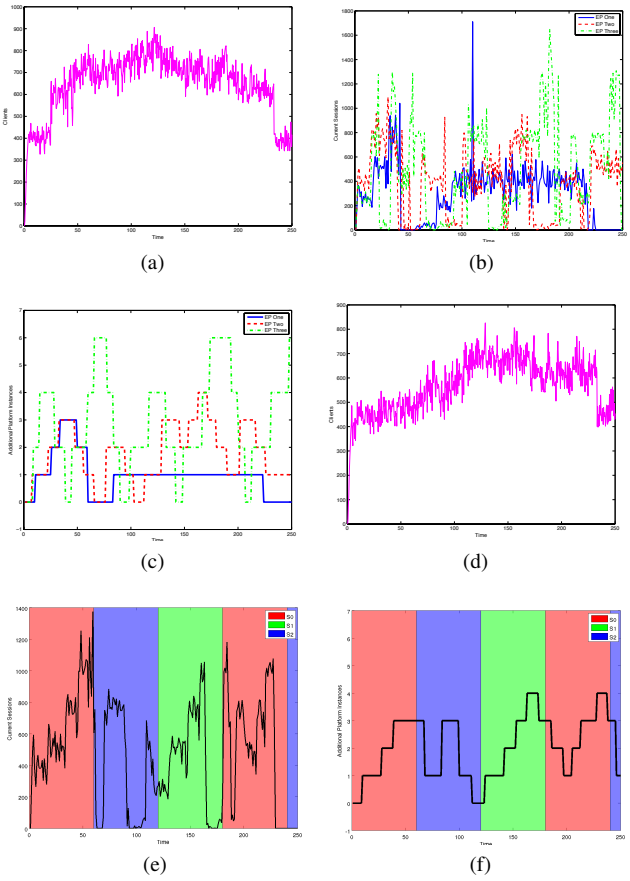
Next, an alternative workload, Figure 3d, was selected. The workload was pre-processed so as to stretch the y-coordinates by a factor of 1.4 (to increase the number of clients) [2]. Three repetitions were run for each policy set, Table I, and for the

[2]This stretch was applied as the workload did not look very interesting initially (i.e., its maxima were much less than the day 41 partial excerpt data we had initially worked with)

Metric	$P_{Sensitive}$	$P_{Tolerant}$	$P_{Aggressive}$	ST
Tot. Ses.	8,11,12	3,4,6	5,9,10	1,2,7
Add. Inst.	4,5,6	1,3,7	9,11,12	2,8,10
MRT	2,10,11	6,8,12	4,5,9	1,3,7

TABLE II: Placement for various approaches. Total Sessions (Tot. Ses.), Additional Instances (Add. Inst.), Mean Response Time at the client (MRT), and Strategy-tree (ST). There are twelve trials. For each row, 1 denotes the best result for that metric and 12 denotes the worst.

strategy-tree, Figure 2. An overview of the results for the individual trials is presented in Table II. Figures 4a, 4b and 4c present the mean total number of sessions serviced, the mean number of platform instances purchased and the mean of the mean response time at the client for each set of three runs for each approach respectively. Results from one of the runs using the strategy-tree are presented in Figures 3e and 3f.

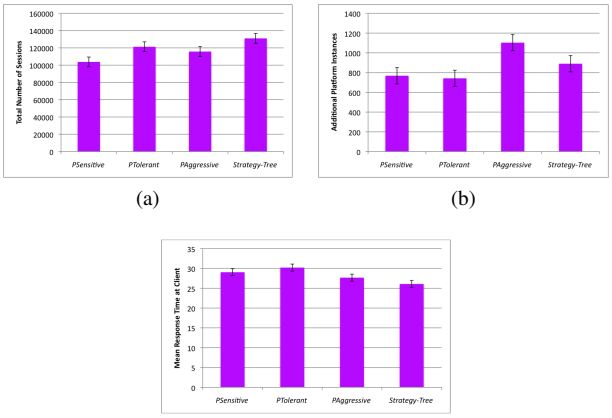


(a) (b) (c) (d) (e) (f)

Fig. 3: (a) FIFA '98, Day 41, partial excerpt. (b) Number of sessions processed by the application in response to the workload using three alternative elasticity policies (EP)s. (c) Additional platform instances being added and released in response to the workload under the three alternative EPs during characterization phase. (d) FIFA '98, Day 43, partial excerpt (stretched by 1.40). (e) Total number of sessions processed versus time: strategy-tree. (f) Platform instance usage versus time: strategy-tree.

VI. Discussion

The policy sets that were used were heuristic in nature with no formal methodology utilized in their design. We intend to investigate techniques to better determine thresholds for our policy rules, using formal modeling techniques and building on work done in [16]. It should be emphasized that the intent of this paper was to explore reasoning about the performance of the policy sets via a strategy-tree rather than focusing on the optimal design of a specific elasticity policy. In fact, the elasticity policies used in this paper were developed in an *ad-hoc* fashion in contrast to the formal refinement approaches such as [17–19].



(a) (b) (c)

Fig. 4: Mean of three trials for each elasticity policy and for the strategy-tree (plus and minus one standard error) for (a) Total sessions. (b) Total number of platform instances. (c) Mean response time as measured at the client.

One limitation of the strategy-tree presented here is that it is only reactive in nature. Specifically, it only considers the previous epoch's history. This falls into the problem of local minima/maxima (i.e., hill climbing problem) . One possible approach to improve this limitation would be to utilize the growing history over time. However, to truly implement good decision making in DEC-elements prediction is required. Work by [20] utilized signal processing techniques to detect patterns in workloads to assist in prediction. These forms of techniques would be interesting to apply inside the DEC-elements of a strategy-tree.

VII. Conclusions

The work presented in this paper is an initial step toward the realization of our business driven cloud optimization architecture. We introduced an architecture and methodology for managing a SaaS application on top of a PaaS provider's infrastructure. This framework utilizes PaaS policy sets to implement the SaaS provider's elasticity policy for its application server tier. A strategy-tree is utilized at the SaaS layer to actively guide policy set selection at runtime in order to maintain alignment with the SaaS providers business objectives. An experiment was presented that demonstrates the promise of this approach and the usefulness of dynamically switching among active strategies at runtime.

ACKNOWLEDGMENT

This research was supported by the IBM Centre for Advanced Studies (CAS), the Natural Sciences and Engineering Research Council of Canada (NSERC), Ontario Centre of Excellence (OCE), Amazon Web Services (AWS) and Rightscale.

REFERENCES

[1] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, 2008.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009.

[3] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, pp. 4:1 –4:11, july 2009.

[4] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Comp. Syst.*, vol. 25, no. 6, pp. 599–616, 2009.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 164–177, ACM, 2003.

[6] VMWare. Available at: http://www.vmware.com. [online April 2011].

[7] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *POLICY '04: Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pp. 3–12, IEEE Computer Society, June 2004.

[8] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implementing an elasticity policy in the cloud," in *In Proceedings of the IEEE Fourth International Conference on Cloud Computing (CLOUD 2011)*, pp. 716–723, July 2011.

[9] B. Simmons and H. Lutfiyya, "Strategy-trees: A feedback based approach to policy management," in *Proceedings of the 3rd IEEE international workshop on Modelling Autonomic Communications Environments*, MACE '08, (Berlin, Heidelberg), pp. 26–37, Springer-Verlag, 2008.

[10] B. Simmons and H. Lutfiyya, "Achieving high-level directives using strategy-trees," in *Proceedings of the 4th IEEE International Workshop on Modelling Autonomic Communications Environments*, MACE '09, (Berlin, Heidelberg), pp. 44–57, Springer-Verlag, 2009.

[11] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, (London, UK), pp. 18–38, Springer-Verlag, 2001.

[12] B. Simmons, *Strategy-Trees: A Novel Approach To Policy-Based Management*. PhD thesis, The University of Western Ontario, 2010.

[13] B. Simmons, M. Litoiu, D. Ionescu, and G. Iszlai, "Towards a cloud optimization architecture using strategy-trees," in *I2TS 2010: 9th International Information and Telecommunication Technologies Symposium, Rio de Janeiro, Brazil, December 13-15, 2010. Proceedings*, 2010.

[14] B. Moore, "Policy core information model (pcim) extensions, rfc 3460," Jan. 2003.

[15] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE Network*, vol. 14, pp. 30–37, May/Jun 2000.

[16] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in *Autonomic Computing, 2011. International Conference on*, (New York, NY, USA), ACM, June 2011.

[17] A. Bandara, E. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pp. 229 – 239, 2004.

[18] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A methodological approach toward the refinement problem in policy-based management systems," *Communications Magazine, IEEE*, vol. 44, no. 10, pp. 60 –68, 2006.

[19] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Decomposition techniques for policy refinement," in *Network and Service Management (CNSM), 2010 International Conference on*, pp. 72 –79, Oct 2010.

[20] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *Network and Service Management (CNSM), 2010 International Conference on*, pp. 9 –16, 2010.