Hierarchical Self-Optimization of SaaS Applications in Clouds*

Bradley Simmons[1], Hamoun Ghanbari[1], Sotirios Liaskos[1], Marin Litoiu[1], and Gabriel Iszlai[2]

[1] York University, 4700 Keele Street, Toronto, Canada
{bsimmons || liaskos || mlitoiu}@yorku.ca, hamoun@cse.yorku.ca
[2] IBM, Toronto Software Lab, 8200 Warden Avenue, Markham, Canada
giszlai@ca.ibm.com

Abstract. This chapter 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 provider's elasticity policy for its application server tier. Adaptation is based on strategy-trees, which allow for systematic capture, representation and reasoning about adaptation variability, based on hierarchically organizing different levels of temporal granularity. Thus, 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 objectives. This way, the SaaS provider's attitudes and preferences reflecting their general business needs are incorporated into the adaptation mechanism in an organized and accessible manner. Results from an experiment conducted on a real cloud are presented in support of this approach.

1 Introduction

Cloud computing [2,8,15,23] represents an approach to IT which has emerged in large part due to improvements in virtualization technologies[3] [5] and the construction and commoditization of large data centers from which infrastructure (IaaS), platform (PaaS) and software (SaaS) are provided on-demand to end

* This is an extended version of a short paper published previously: B. Simmons, H. Ghanbari, M. Litoiu and G. Iszlai, "Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree," *Network and Service Management (CNSM), 2011 7th International Conference on*, vol., no., pp.1-5, 24-28 Oct. 2011. "Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than IFIP must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."

[3] http://www.vmware.com/

users over the Internet. In fact, this three-layered cloud computing architecture [20] has assumed the role of a *de-facto* standard.

A *PaaS provider* is an enterprise that is responsible for leasing application environment topologies to SaaS provider clients for various durations of time. A topology is built upon infrastructure that is 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). Additional platform service instances may be purchased from and/or released to the PaaS provider at runtime as needed.

A *SaaS provider* is an enterprise that provides a software offering that is run from within a PaaS topology. There are many possible economic models which can be utilized by a SaaS provider (e.g., free with advertisements, membership cost per period of time, ...).

Regardless of the layer, a service provider has long term business objectives to achieve. Further, as its service is providing some form of IT resources to a dynamic set of clients, how effectively it manages these resources is key to how successful it is in meeting its business objectives. In general, an enterprise will attempt to maximize profit where profit can be understood to represent the difference between revenue and cost.

Consider a SaaS provider running on a cloud. This SaaS provider leases a platform topology from a single PaaS provider and offers one application to a dynamic set of clients. Several aspects of a platform topology (as offered by the PaaS provider) 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 [16]. 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 [13]. One way of specifying an elasticity policy is through a set of policy rules. It has been described previously [30] that a set of policies may be thought of as a strategy. Multiple strategies may be defined to achieve the same set of objectives [31].

It is assumed that the SaaS provider's business objective is to maximize profit. This can involve both the maximization of revenue generation and the minimization of cost. For example, maximization of revenue generation varies directly with the number of clients serviced. Similarly, minimization of cost varies directly with the number of platform service instances that are purchased over time. Different strategies which are defined to achieve the same set of objectives might result in very different outcomes.

Thus, constructing models that allow continuous adaptation of the strategy based on the contextual circumstances (load quantity and quality, prices, service level agreement violation costs) is a challenging problem. One of the problems is the design, comprehension and communication of the adaptation approach

which is overly difficult with continuous mathematical models or parametrized policies and without a more systematic modeling approach.

To address this problem of systematically designing the adaptability aspect of software infrastructures, we have introduced the concept of *strategy-trees* [29]. Strategy-trees constitute hierarchical organizations of adaptation decisions, structured in a way that allows stratification of decisions based on the time horizon in which they apply. Thus, long term decisions/options are refined into shorter term ones until they reach rapid low level alternations of fixed configuration strategies. Strategy-trees have the benefit of offering a better organization of adaptation variability, in a way similar to goal models [18, 22] and feature models [26] used in application software engineering, while allowing designers to reason about the depth and cost of adaptation actions.

In this chapter, we present one step toward the realization of our earlier conceptual work with regards to applying strategy trees to a SaaS layer manager in the context of a business driven cloud optimization architecture [19, 29]. The contributions are as follows. 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 chapter is structured as follows. Section 2 introduces a scenario involving a SaaS provider running on a PaaS topology which is used throughout the remainder of the chapter. Section 3 provides a brief overview of the concept of strategy-trees and describes the design of a simple strategy-tree for the scenario introduced in the previous section. Section 4 introduces the management architecture for managing SaaS applications on top of a PaaS provider's infrastructure. Section 5 presents an experiment demonstrating the effectiveness of this approach in the context of the introduced scenario. Section 6 offers a discussion of the experimental results. Section 7 provides an overview of some related work. Section 8 presents our conclusions and thoughts on future work.

2 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 service 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. The next section will consider the design of three, alternative elasticity policies (policy sets), to achieve the objectives under different sets of expectations and assumptions.

2.1 Elasticity Policy

An elasticity policy governs how and when resources are added to and/or removed from a cloud environment. 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 (i.e., application) 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 sets 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 in resource allocation to all tiers of an application and this may (and would likely) involve the consideration of various application specific QoS metrics as well.

A brief overview of the hierarchical[4], heuristic elasticity policy, utilized in this work, will now be presented (for a more complete overview please refer to [13]). As mentioned already, the policy rules utilized in this work focus on the value of a single performance metric (i.e., `cpu_idle`). When this value is high the implication is that an instance is not being heavily loaded[5]. From a high-level perspective, decisions to grow or shrink the application server tier are made based on a critical number of local observations triggering a global action (i.e., add/remove instances to the application tier). This will now be considered in more detail.

Rules for Platform Service Instances The policy rules defined for each platform service instance member node of the application server tier of the SaaS provider, introduced for the scenario above, are based on the definition of an acceptable range for the `cpu_idle` metric and an acceptable duration beyond which a violation should be indicated to the management framework. For the remainder of this chapter, we will refer to the upper threshold as `cpu_idle_st` (i.e., the threshold indicating a need to shrink the tier) and `cpu_idle_gt` (i.e., the

[4] It is hierarchical in the sense that there are rules that are specified for individual platform service instance nodes in the application server tier and then there are rules that are specified to govern the application server tier that use the results of these lower level rules in aggregate to guide the addition and/or removal of additional platform service instances.

[5] Alternatively, when this value is low the implication is that an instance is being heavily loaded.

threshold indicating a need to grow the tier) and the durations will be referred to as `shrink_duration` and `grow_duration` respectively. Some details to consider in relation to the action (i.e., add/remove instances to the application server tier) are as follows. The selection of the *range* (i.e., `cput_idle_gt - cpu_idle_st`) will directly impact the addition and removal of server instances to the tier. Consider if the operating range of the system is well outside of this defined range then violations will consistently occur. In contrast, assuming a more accurate prediction of workload and hence a well defined range (i.e., typical operating range is within the threshold values) it will then be the size of the range that will have an impact. Specifically, the size of the range will define how sensitive it is to variation in the workload. Further, the choice of durations will also impact this sensitivity. Specifically, shorter durations will result in more frequent notification to the management system while longer durations will have the opposite effect.

Rules for the Application Server Tier The policy rules defined for the application server tier of the SaaS provider, introduced in the scenario above, involve several configurable parameters as well. First is the definition of the value of a `quorum`. A quorum denotes the percentage of instances that must all be indicating that they are in violation of their local policy rule (all members of the quorum must indicate the same violation e.g., above `cpu_idle_st` or below `cpu_idle_gt`) in order to trigger an auto-scaling action (i.e., grow or shrink the tier). The amount by which the tier is meant to be grown is indicated by the parameter `incr_val`. The amount by which the tier is meant to be shrunk is indicated by the parameter `decr_val`. Whether an auto-scaling action even occurs is controlled by the parameter `refractory_period` which indicates the amount of time that must have elapsed since the last auto-scaling action took place. The way these parameters impact the auto-scaling behaviour of the tier is as follows.

The values `incr_val` (or `decr_val`) affect how aggressively the grow/shrink action will be. Specifically, a larger number indicates a more aggressive action. The value of `quorum` impacts the sensitivity of individual indications of a violation. A larger quorum (i.e. closer to 100%) implies more sensitivity to individual notifications while a smaller quorum implies the opposite. Finally, the larger values of `refractory_period` will result in a more gradual change in tier size while a smaller value will have the opposite affect.

An example of the policy rules defining the auto-scale grow action are provided in Listing 1.1. For the remainder of this document, an elasticity policy is defined by a set of four policy rules (two for growing and two for shrinking). Three different elasticity policies were designed to drive the auto-scaling actions of the application server tier under different circumstances. These policy sets utilized different settings of some of the configurable parameters mentioned above and are presented in Table 1. The first elasticity policy, $P_{Sensitive}$, was designed to be gentle in how it grew/shrunk the tier (i.e., adding/removing only one platform service instance at a time). Similarly, the second elasticity policy, $P_{Tolerant}$, increased and decreased the topology in a gentle fashion as well;

however, the range separating its two thresholds (upper and lower) was three times as large as for $P_{Sensitive}$ making it much less likely to be triggered as often (assuming violations occur inside the defined range). The third elasticity policy, $P_{Aggressive}$, was designed to be much more aggressive in how it grew/shrunk the tier as evidenced by both a small range between its upper and lower thresholds and an increment value (up and down) of two.

```
(a) inst oblig cpu_idle_breach_low {
      subject s = inst_mgr;
      target t = platform_tier_mgr;
      on {e1 ; e2} ! e3
      do emit(t, request_increase)
      when e2.time - e1.time == grow_duration and
        e1.cpu_idle < cpu_idle_gt and
        e2.cpu_idle < cpu_idle_gt and
    }//cpu_idle_breach_low
(b) inst oblig perform_autoscale_grow {
      subject s = paas_mgr;
      target t = platform_tier_mgr;
      on  quorum(platform_tid, action)
      do  t.elastic_grow_action(incr_val)
      when action.equals("grow") and
        !t.refractory_period and
        t.id == platform_tid
    }//perform_autoscale_grow
```

Listing 1.1: Sample policies to auto-scale grow the platform tier specified in a Ponder-like [11] notation. In (a) it is assumed that e3 denotes an event indicating `e3.cpu_idle` $\geq$ `cpu_idle_gt`.

Table 1: Parameter settings defining the three elasticity policies as used in the experiments (i.e., values related to time are scaled by one quarter).

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

3 Strategy-Trees

A *strategy* can be defined as "...a plan of action designed to achieve a long-term or overall aim"[6]. In the context of policy-based management, a set of policies can be understood to implement a strategy. 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 *directive* (i.e., a set of objectives[7]). These alternative strategies often incorporate assumptions, biases and expectations within a given policy set (i.e., the management logic which governs the system's behaviour attempting to achieve the set of objectives). Under different scenarios various assumptions can be more/less correct than others resulting in different degrees of effectiveness for the various strategies. Through monitoring of the progression toward the system's objectives and by utilizing feedback about the effectiveness of the deployed strategy (with regards to achieving the defined objectives) informed decisions can be made allowing an ineffective strategy to be changed to an alternative, to better meet the long term objectives.

The concept of a strategy-tree was introduced to facilitate intelligent switching among defined policy sets (i.e. strategies) at runtime in response to monitored data and in the pursuit of a a directive defined over a long-term horizon of time. 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 orchestrates the 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[8]. For a more comprehensive and formal consideration of strategy-trees and their use in policy management please refer to [28–31]; however, a brief description of the key points follows.

A strategy-tree, Fig. 1, is composed of three types of nodes: Directive (i.e., circle), AND (i.e., triangle) and OR (i.e., inverted triangle). Associated with each node in the tree is a *quantum attribute value* which denotes when[9] a node's

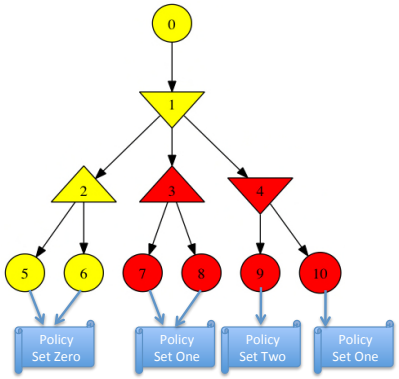
[6] http://oxforddictionaries.com/definition/strategy

[7] An objective represents a constraint on a metric. A metric might be a low level technical metric (e.g. throughput) or a business metric (e.g., profit).

[8] 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. (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.

[9] A quantum attribute value represents a coefficient on some management time unit (MTU).

(that is a member of the active strategy) SAT-element [10] should be executed and in the case of an OR type node its DEC-element as well. There is also a list, *results*, that is associated with each node which enables a child node to pass up the result of its evaluation (i.e., execution of its associated SAT-element) to its parent node (for use in later evaluations). Each leaf node of a strategy-tree is bound to a single policy set[11].



Index	Quantum Attribute Value (hours)
0	168
1	24
2	1
3	1
4	6
5	1
6	1
7	1
8	1
9	1
10	1

Fig. 1: An example strategy-tree with 11 nodes. There are four strategies $S_0 = (0, 1, 2, 5, 6)$, $S_1 = (0, 1, 3, 7, 8)$, $S_2 = (0, 1, 4, 9)$ and $S_3 = (0, 1, 4, 10)$. Currently, S_0 is active as denoted by its yellow coloring (red indicates inactive). The quantum attribute values presented in the table are determined through experimentation, simulation or may even be selected arbitrarily.

At a high level, the algorithm for evaluating a strategy-tree goes as follows[12]. Each time the strategy-tree is evaluated (i.e., each increment) the SAT-elements in the active strategy are processed from leaf to root. A SAT-element is evaluated when the increment value modulo the node's quantum attribute value is equal to zero. Once all the SAT-elements have been evaluated the DEC-elements of the

[10] SAT-elements are used to evaluate whether a set of objectives is satisfied. DEC-elements are used in decision making to determine whether to maintain the current strategy or to switch to an alternative.

[11] While multiple leaf nodes may be bound to the same policy set, leaf nodes of different strategies may only do so under certain constraints. The policy set P_a and P_b that are bound by two leaf nodes that are both direct child nodes of the same OR type node must not be equivalent. Further, should two leaf nodes have a Lowest Common Ancestor (LCA) that is an OR type node and should there be no intervening OR type nodes between either leaf node and this OR type node then the policy sets, P_a and P_b, bound by these two leaf nodes must not be equivalent either [28].

[12] This assumes that the tree has been fully specified, all elements defined, all leaf nodes have been bound to policy sets, and the initial strategy set to active.

active strategy are evaluated from root to leaf[13]. Should any DEC-element decide to switch strategy, the switch is implemented and the algorithm terminates. So, if we assume the strategy-tree in Fig. 1, and further that S_0 is the active strategy then every hour the SAT-elements associated with nodes five and six evaluate and pass up results to node two which aggregates[14] this result and passes it up to node one. At iteration 24, after the twenty-fourth evaluation of these SAT-elements, the SAT-element associated with node one evaluates and passes its result up to node zero. Next, the DEC-element associated with node one is evaluated and a decision, based on the most recent epoch (i.e., 24 hours) of collected data, is used to decide whether to continue using strategy S_0 or whether to switch to one of the three alternatives (i.e., S_1, S_2 or S_3).

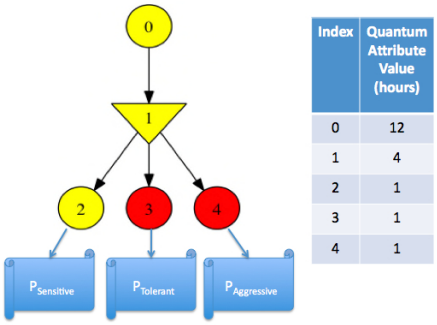
A strategy-tree that has multiple OR type nodes, as in Fig. 1, can be understood to have multiple MAPE loops [17] defined. For example, node four, implements a loop that uses the six most recent results for the evaluation of SAT-elements associated with node nine or ten (depending on whether strategy S_2 or S_3 is active) as well as all monitored data for this six hour period in its decision making process. In contrast, node one, implements a loop which utilizes the previous 24 results for the evaluations of SAT elements associated with nodes five, six and two (when strategy S_0 is active) or for the evaluations of SAT-elements associated with nodes seven, eight and three (when strategy S_1 is active) or the four most recent evaluations of the SAT-element associated with node four (when strategy S_2 or S_3 is active) as well as all monitored data for this 24 hour period in its decision making process. It should be pointed out that in all but the simplest cases, more data than just the previous epoch's[15] will be used in the decision making at a DEC-element.

3.1 Scenario: Designing a Strategy-Tree for the SaaS Provider

This section considers the development of a strategy-tree, Fig. 2, to help guide the system to achieve the objective of the SaaS provider (i.e., maximize profit) introduced in Section 2. Recall from Section 2.1 that three elasticity policies have been defined: $P_{Sensitive}$, $P_{Tolerant}$ and $P_{Aggressive}$. Each of these elasticity policies (policy sets) can be viewed as a particular strategy to achieve the SaaS provider's objectives under a particular set of expectations and assumptions.

The strategy-tree that we will consider consists of only five elements: four directive type nodes and one OR type node. This simple structure was intentionally selected in order to focus on the development of the single DEC-element for the OR type node. To achieve the objective, heuristic trade-offs between maximizing revenue and minimizing costs are utilized. A bias which favours servicing the maximum number of clients while attempting to limit the number of additional platform service instances purchased is applied.

[13] This is a small, yet valuable (in terms of complexity) alteration to the algorithm.
[14] Applies a *boolean* AND to the results.
[15] An epoch is equivalent to the quantum attribute value of the node in question so if a node has a quantum attribute value of 24 hours then the epoch is 24 hours as well.



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

Fig. 2: Strategy-tree used for the experiment.

Characterizing the Elasticity Policies The various strategies need to be understood in order to evaluate their effectiveness and reason about switching among possible alternatives. The SaaS provider was able to characterize each of the three (see Section 2.1 and Table 1) 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 service instances purchased 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 of these regressions into four distinct categories (indicating different degrees of increase/decrease in numbers of sessions).

Design of SAT-Elements The three leaf nodes (e.g., nodes two, three and four) each have quantum attribute values of one hour. This implies that each hour, they evaluate their SAT-element and pass the result up to node one. Node one evaluates its SAT-element every four hours and passes its result up to node zero. Each of these elements is evaluating the following objective:

– *The number of additional platform service instances purchased, divided by the epoch, should not exceed the hourly mean for that particular strategy.*

Design of the DEC-Element The design of the DEC-element for node one was much more involved than for the SAT-elements in the tree. This is normal as deciding among alternative approaches can be difficult at the best of times.

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 (or lack of detection) of a trend in the number of current sessions observed over the previous epoch. Specifically, the slope 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). Should no strong trend be detected, data from the MDB as indicated by the values from the *results* list (about the additional purchased platform service instances) for the previous four hour epoch would then be utilized.

The logic underpinning the DEC-element works as follows. Every increment of the MTU[16], prior to evaluation of the strategy-tree, data about the application is collected and stored in the MDB. Specifically, it includes the previous 15 current session readings and the 15 additional purchased platform service instance readings as well. A regression is performed on the set of current session readings and this is then stored for later use.

When the DEC-element at node one is executed and the method `decider`[17] is invoked, the following steps occur. First, the four most recent regressions are collected from the MDB. Next, the previous 60 additional platform service instance purchases are collected and summed. This data in combination with the current active strategy denotes the *context*. Regardless of whether the *active* strategy is $S_0 = (0, 1, 2), S_1 = (0, 1, 3)$ or $S_2 = (0, 1, 4)$, Fig. 2, a call is made to the method, `map_degrees_to_range`. This method accepts an array of the four most recent hourly regressions $R = [r_1, r_2, r_3, r_4]$. From each regression r_i the slope m_i is extracted and an integer value returned denoting membership in one of the four defined categories:

(i.) $0° < m \leq 85° \mapsto 1$
(ii.) $m > 85° \mapsto 3$
(iii.) $0° > m \geq -85° \mapsto -1$
(iv.) $m < -85° \mapsto -3$

This resultant array of integer values $I = [i_1, i_2, i_3, i_4]$ is multiplied by an array of weights $W = [w_1, w_2, w_3, w_4]$ where $w_1 = 1$, $w_2 = 2$, $w_3 = 4$ and $w_4 = 8$. Notice that the weight values in this array are increasing which ensures that emphasis is placed on the more recent regression's slope. The summation of the multiplication of $W * I$ is returned: $\Sigma_{i=0}^{n} W[i]I[i]$.

Two trends were identified as being of interest: steeply increasing (SI) and steeply decreasing (SD). Specifically, any sum ≥ 27 is considered to be SI while any sum $\leq$ -27 is considered to be SD. All other values are considered to be less indicative of a trend (either increasing or decreasing) and this is when questions about the number of purchased platform service instances are considered.

If the current active strategy is S_0 the following reasoning is used. If the value of *sum* as returned by the call to `map_degrees_to_range` indicates *SD* or *SI* then the variable *next_strategy* is set to two. The rationale for this choice is that if a steep increase or decrease in the number of sessions is observed, it is important (according to the preferences of the SaaS provider) to respond to this by using the aggressive policy set $P_{Aggressive}$ (i.e., S_2) in order to ensure

[16] MTU refers to the management time unit which in the case of this scenario is 60 minutes.
[17] This is the name of the method which evaluates the DEC-element's decision problem.

that available resources are allocated/de-allocated to handle the sharp change in demand. Otherwise (i.e., it is not SD or SI), if the mean number of platform service instances purchased (over the previous four hours) is greater than the mean for S_0 (recall that this is equivalent to $P_{Sensitive}$) then set *next_strategy* to two (indicating S_2) otherwise, set it to one (indicating S_1).

If the current active strategy is S_1 the following reasoning is used. For the cases where *sum* indicates either *SD* or *SI* the identical reasoning as for S_0 is used. However, as it is now $P_{Tolerant}$ being employed if no apparent trend is perceived and if the mean number of platform service instances purchased (for the four hour epoch) is greater than the mean for this strategy then *next_strategy* is set to zero otherwise it is set to one. A simple rationale is used in deciding the switch to S_0. It had been determined during characterization of the three elasticity policies (see Section 2.1) that the hourly mean number of additional platform service instances purchased is lower under S_0 than S_1. Since no trend has been observed (i.e., not SD or SI), and the mean number of platform service instances purchased over the previous epoch has exceeded the mean for S_1 by switching to S_0 there will be fewer additional platform service instances purchased in the next epoch. It is hoped that this will apply a downward pressure on the overall number of additional platform service instances purchased (i.e., decrease the cost). Similarly, if the current active strategy is S_2 the same reasoning as for S_1 is used, except, the threshold for S_2 is substituted in place of the threshold for S_1.

4 Architecture

The following section will provide an overview of our proposed management architecture, Fig. 3. Since the scenario we are considering involves a SaaS provider running an application on top of a PaaS topology we focus only on these two layers.

4.1 PaaS Layer

At the PaaS layer, we assume the traditional policy-based management architecture (PBM) consisting of Policy Repository, Policy Decision Points (PDP) and Policy Enforcement Points (PEP) [21]. While the SaaS provider may also utilize PBM internally, we leave this undefined for now as we are only considering PaaS layer policy sets in this work and their management by a strategy-tree in order to achieve its root directive.

The PaaS provider has access, via a monitoring subsystem, to numerous performance metrics (e.g., cpu utilization, throughput, etc). It also has access to various OS (e.g., ps count, ps cputime, ...) and middleware level metrics (e.g., request queue length, transmitted bytes, session count, ...) 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.

4.2 SaaS Layer

A strategy-tree is used at the SaaS layer[18] to dynamically alter policy set deployment at runtime. We assume that it has access to various performance metrics that allow it to determine whether the currently deployed strategy is effective or not. For example, it is aware of how many platform service instances it has purchased over time and also how many sessions it has serviced. The management database (MDB) is where this data is stored. All elements of the strategy-tree have access to it.

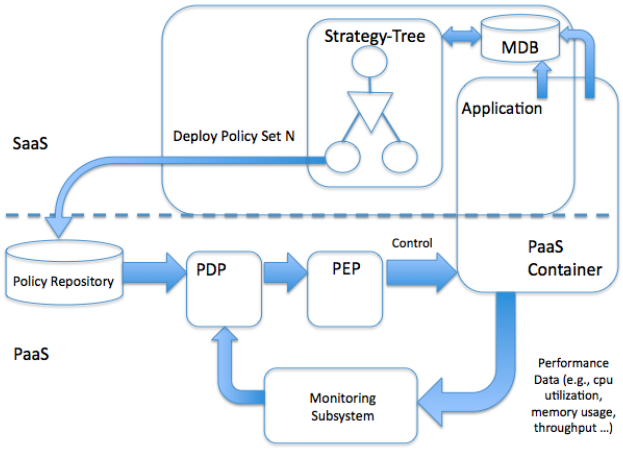


Fig. 3: Proposed management architecture

5 Experiment

The following experiment is based on the scenario of the SaaS provider introduced in Section 2 and is composed of two parts. First, the three elasticity policies (i.e., Table 1) are characterized against a workload as described in Section 3.1. Then the strategy-tree (i.e., Fig. 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 service instances purchased and mean response time as measured at the client.

5.1 Experimental Setup

For this experiment, Amazon (i.e., EC2, EBS) was used as the IaaS provider. All topology instances were built atop virtual machine instances (VMI)s running

[18] It is the SaaS layer manager at this point.

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 strategy-tree was implemented in Ruby and the initial encoding is from within an XML file[19]

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 [1] (Figs.4a and 5a).

Experiment time was scaled by four The monitoring system at the SaaS provider takes a reading every minute (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 chapter, this will represent a session.

5.2 Characterizing the Elasticity Policies

The first step when using a strategy-tree requires that a characterization of the various strategies be performed. The results of running a single workload (FIFA '98, Day 41, partial excerpt) against the SaaS application utilizing each of the three elasticity policies is presented in Fig. 4. Notice that the differences in both current sessions (Fig. 4b) and additional platform service instances utilized over time (Fig. 4c) varies substantially among the three alternative strategies.

Following each run, the hourly mean number of platform service instances purchased by the SaaS provider when operating under that elasticity policy was computed. Also, hourly regressions were computed on the *number of sessions* [20] serviced by the SaaS offering for each complete run under each elasticity policy. The slopes of these regressions were then partitioned into four categories (i.e.,

[19] This is an updated version from previous work where it was implemented in Java.
[20] There were 15 readings per experiment hour.

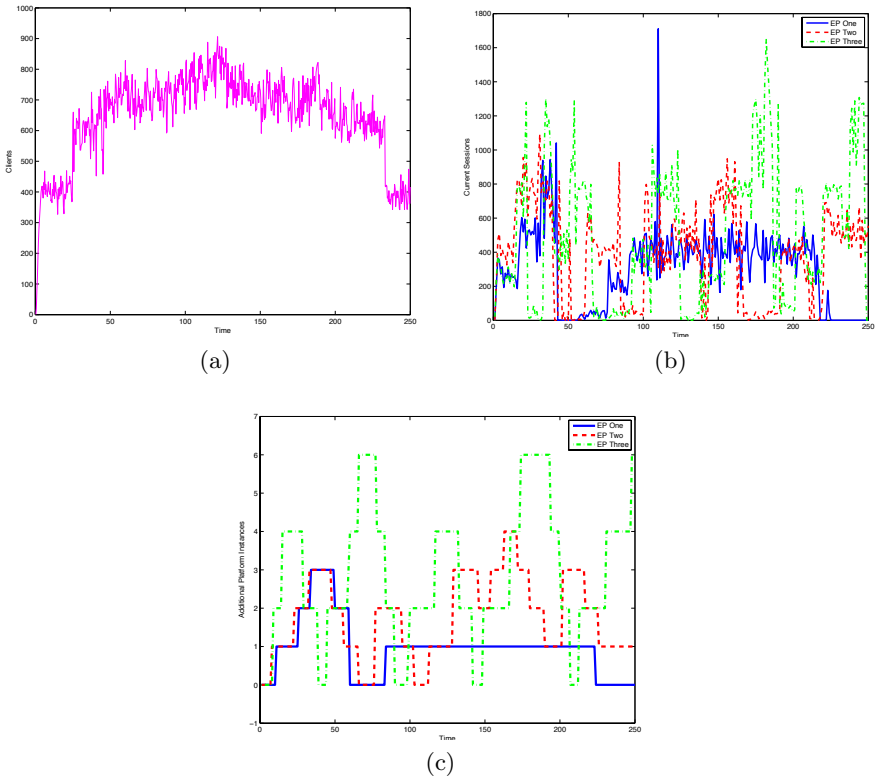


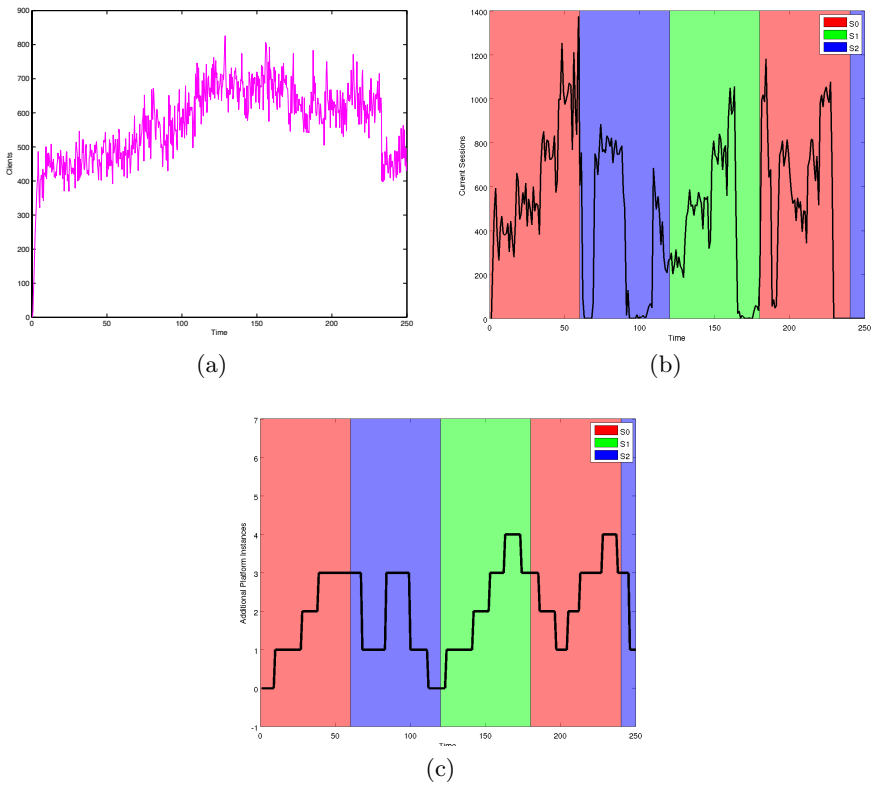
Fig. 4: (a) FIFA '98, Day 41, partial excerpt. (b) Number of sessions processed by the application in response to the workload using three alternative elasticity policies (EP)s. (c) Additional platform service instances being added and released in response to the workload under the three alternative EPs during characterization phase.

ranges) of roughly equal occurrence as described in Section 3.1. These characterizations were used in the design of elements for the strategy-tree as previously described.

5.3 Experimental Results

An alternative workload (Fig. 5a) was selected (FIFA '98, Day 43, partial excerpt). The workload was pre-processed so as to stretch the y-coordinates by a factor of 1.4 (to increase the number of clients)[21]. We ran three repetitions for each strategy, Table 1, and for the strategy-tree, Fig. 2. Plots from one of the

[21] 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)



(a) (b) (c)

Fig. 5: (a) FIFA '98, Day 43, partial excerpt (stretched by 1.40). (b) Total number of sessions processed versus time: strategy-tree. (c) Platform service instance usage versus time: strategy-tree.

three runs using a strategy-tree are presented in Figs. 5b and 5c. In this run, all three strategies were used at various points in time as indicated by the alternative background colourings. An overview of the results for the individual trials is presented in Table 2. This table assigns an integer ranking (i.e., one denotes best while 12 denotes worst) as follows: a strategy is more successful when it services a greater number of total sessions, purchases fewer additional platform service instances and provides a lower mean response time to its clients.

Figure 6a presents the mean total number of sessions serviced for each set of three runs for each approach (i.e., $P_{Sensitive}$, $P_{Tolerant}$, $P_{Aggressive}$ and the strategy-tree). Figure 6b presents the mean number of platform service instances purchased for each set of three runs for each approach. Figure 6c presents the mean of the mean response time at the client for each set of three runs for each approach.

Table 2: 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.

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

6 Discussion

The experiment presented in Section 5 was preliminary in both its scope and complexity; however, it demonstrates that a strategy-tree can achieve its root's directive (i.e., maximize profit). Recall from Section 2 that the objective of the SaaS provider is to maximize profit by both maximizing revenue and by minimizing cost. It attempts to achieve this through the strategy-tree which employs a bias (at its DEC-element) that favours servicing the maximum number of clients while attempting to limit the number of additional platform service instances purchased. Finally, recall that their is also an additional speculative cost associated with loss of future business due to sub-par response time.

In terms of best individual results (see Table 2) the strategy-tree approach finished first both in total number of sessions serviced and mean response time at the client. It also finished second for additional platform service instances purchased. The best individual results for $P_{Sensitive}$ were eighth for total number of sessions, fourth for additional platform service instances purchased and second for mean response time at the client. The best individual results for $P_{Tolerant}$ were third for total number of sessions, first for additional number of platform service instances purchased and sixth for mean response time at the client. The best individual results for $P_{Aggressive}$ were fifth for total number of sessions, ninth for additional number of platform service instances purchased and fourth for mean response time at the client. Further, it should be pointed out that the strategy-tree approach never obtained the worst result for any of the metrics while all individual strategies did. Also, its two worst individual results were in terms of the number of additional platform service instances purchased and this can be understood due to the bias in favour of servicing client sessions. These results argue in favour of the effectiveness of the strategy-tree at facilitating trade-offs while maintaining alignment with the root directive.

In terms of aggregate results, the mean value over three trials for the total number of sessions (see Fig. 6a) and the mean value over three trials for mean response time at the client (see Fig. 6c) were better for the approach utilizing a strategy-tree than for any of the individual strategies. Further, the mean value over three trials for number of platform service instances purchased (see Fig. 6b) was much less for the approach utilizing a strategy-tree than for

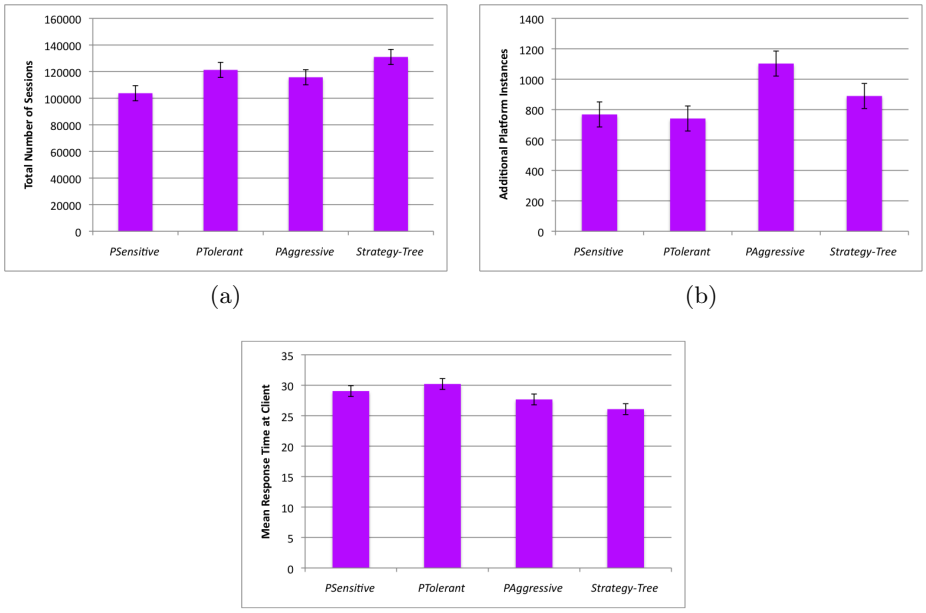


Fig. 6: Mean of three trials for each elasticity policy and for the strategy-tree (plus and minus one standard error) for (a) Total sessions. (b) Total number of platform service instances. (c) Mean response time as measured at the client.

$P_{Aggressive}$. More precisely, the strategy-tree approach serviced approximately 26% more sessions, while using approximately 16% more platform service instances and achieving an improvement in mean response time at the client of approximately 10% when compared to $P_{Sensitive}$. The strategy-tree serviced approximately 8% more sessions[22], while using approximately 20% more platform service instances and achieving an improvement in mean response time at the client of approximately 14% when compared to $P_{Tolerant}$. The strategy-tree serviced approximately 13% more client sessions, while using approximately 19% fewer platform service instances and achieving an improvement in mean response time at the client of approximately 6% when compared to $P_{Aggressive}$.

Various factors may have adversely impacted the results presented here. For example, the set of experiments (both characterizations of the three elasticity policies and the actual experimental runs) were performed on the Internet and not on a private test-bed; further, the experiments were run without regard for time of day and this might have had an impact.

The policy sets that were used were heuristic in nature with no formal methodology utilized in their design. We intend to investigate designing techniques to better determine thresholds for our policy rules, using formal modelling

[22] This is actually quite substantial (i.e., approximately 10,000 sessions).

techniques and building on work done in [6]. It should be emphasized that the intent of this chapter 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 chapter were developed in an *ad-hoc* fashion in contrast to the formal refinement approaches such as [4, 10, 25].

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 [14] utilized signal processing techniques to detect patterns in workloads to assist in prediction. These forms of techniques would be interesting to apply inside DEC-elements of a strategy-tree.

7 Related Work

The strategy-tree was introduced to allow for changes in strategy (i.e., policy set) to be made in response to changes in experienced workloads and/or failures of the implicit assumptions underpinning policy set construction. Regardless, it is predicated on achieving a long-term directive over some horizon of time based on a pre-defined set of policy sets. The work presented by [3] describes a mechanism for applying reinforcement learning in the context of policy management. In this work, the *active* (i.e., deployed) policy set is utilized to dynamically construct a model of the system which guides the autonomic manager resulting in improved performance. As changes in the deployed policy set occur Bahati et al. are able to demonstrate that in a majority of cases transformations can be performed on the state-transition graph thus retaining much of the previously learned information; however, they also demonstrate that in certain circumstance (i.e. model transformation $\Psi_5[G_n^P]$), the initial model must be discarded and a new one initialized. We see a DEC-element as potentially representing a transformation of this type. Further, the idea of dynamically evolving a policy set is one that we have currently left unexplored but see value in. Presently, a change in policy set implies a change in strategy (at any particular DEC-element). However, allowing for policy sets to be learned may result in a less constrained approach.

The work in [7] introduces the SYMIAN decision support tool which is used to determine effective incident management strategies. Each DEC-element in a strategy-tree denotes a unique locus of control for selecting among a set of strategies. However, unlike SYMIAN, there exists a hierarchy of objectives to achieve in which the effectiveness of employed strategies to achieve lower layer objective sets directly contributes to the perceived achievement of those objective sets higher in the tree structure. Further, unlike SYMIAN, which facilitates strategy design, the decision making at a DEC-element in a strategy-tree is done automatically at runtime over a pre-determined set of alternative strategies. While the DEC-element described in this chapter was designed based on a

simple heuristic, a need for more involved reasoning among potential strategies especially as temporal granularities grow and out-degree of DEC-elements (i.e., OR type nodes) increases is clear. Each DEC-element represents a multi-criteria decision problem in which the current context (i.e., monitored data, the current active strategy and the set of available strategies at the particular node) must be considered while trying to achieve a local set of objectives. The decision to maintain or switch current strategies can become a highly complex and challenging problem requiring well thought out models, and approaches. In effect, a strategy-tree with multiple DEC-elements (as in Fig. 1) represents a hierarchical (over time) set of decision problems to be solved.

The work in [24] manages the Quality of Service (QoS) provisioning of Diff-Serv over MPLS networks in alignment with business objectives. A model of business utility is introduced relating business indicators, SLA indicators, objectives and policies. The business indicators are assigned weights and a set of mapping functions are derived to facilitate the generation of effective policy parameters. This approach is an extension to the policy refinement work in [25]. The policies that are considered are simple rules and the weights (while justified) appear somewhat arbitrarily selected. Regardless, the method demonstrated through simulation seems promising. It should be emphasized that once a strategy is selected, that is it. There is no mechanism to change strategy automatically. In contrast, a strategy-tree utilizes feedback at (typically) multiple temporal granularities to allow for changes in strategy to be made over time. One possible avenue would be to determine various weight settings based on sets of assumptions (as presented informally in the paper) and then to construct a strategy-tree to alternate among these alternative strategies over time while maintaining alignment with the long term business objectives.

The Stitch language [9] was introduced to specify adaptation strategies in the context of the Rainbow Framework [12]. Stitch is based on three main concepts: *operators*, *tactics* and *strategies*. An operator maps to a system provided command (i.e., an effector), a tactic represents a conditional evaluation of a set of actions (i.e., calls to operators) and an expected set of effect(s) and a strategy which is a tree of conditional tactic delay nodes[23]. At runtime a strategy is selected from the set of possible strategies based on its overall utility across a set of quality dimensions, d, in a particular context. The sum of utility values is constructed based on weights, w_i, which are arbitrarily defined (e.g., $U = \sum w_d u_d$) as are the utility functions for each dimension. Comparatively, a strategy, in terms of a strategy-tree, denotes the entire set of management policies which are deployed at any given point in time. With strategy-trees we are not trying to manage a single adaptation, rather, we are attempting to guide the system, in a more scalable fashion to achieve the long-term objectives of the administrator. In fact, Stitch and the strategy-tree approach are complementary, as a module of Stitch strategies could be viewed as the deployed policy set while alternative

[23] Associated with each node in a strategy is a probability that its condition will evaluate to true and a delay specifying the horizon over which the effect of the tactic's execution will be observable.

modules of strategies (i.e., ones with different probabilities, weights and utility functions) could be thought of as alternative policy sets and a strategy-tree could then be constructed to switch among these modules at runtime to maintain alignment with the long term management objectives.

8 Conclusions

The work presented in this chapter 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 provider's business objectives. An experiment on a real cloud was presented that demonstrates the promise of this approach and the usefulness of dynamically switching among active strategies at runtime.

In future work we would like to use a more realistic application in which multiple classes of clients are defind and various admission control policies and tuning policies can be used to augment the complexity of the application's elasticity policy. We also intend to continue developing the concept of the strategy-tree. While initially, a strategy-tree was designed to capture implicit objectives underpinning the various policy sets, we think there may also be an interesting research space connecting it to explicit objectives. Specifically, we feel that there may exist a link between the concepts of goal-models, awareness requirements [27] and strategy-trees that might allow us to automate the generation of the tree structure as well as the various SAT-elements so that the strategy-tree is more directly connected to the objectives of the service (e.g., SaaS) provider and easier to build and use. We are also beginning to suspect that perhaps simulation is a better avenue for demonstrating longer-term strategic management than through experimental work.

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. Arlitt, M., Jin, T.: A workload characterization study of the 1998 world cup web site. IEEE Network 14(3), 30–37 (May/Jun 2000)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (February 2009)

3. Bahati, R.M., Bauer, M.A.: Towards adaptive policy-based management. In: NOMS. pp. 511–518 (2010)
4. Bandara, A., Lupu, E., Moffett, J., Russo, A.: 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)
5. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles. pp. 164–177. ACM, New York, NY, USA (2003)
6. Barna, C., Litoiu, M., Ghanbari, H.: Autonomic load-testing framework. In: Autonomic Computing, 2011. International Conference on. ACM, New York, NY, USA (June 2011)
7. Bartolini, C., Stefanelli, C., Tortonesi, M.: Symian: Analysis and performance improvement of the it incident management process. IEEE Transactions on Network and Service Management 7(3), 132–144 (2010)
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Comp. Syst. 25(6), 599–616 (2009)
9. Cheng, S.W., Garlan, D.: Stitch: A language for architecture-based self-adaptation (2012), submitted for Publication
10. Craven, R., Lobo, J., Lupu, E., Russo, A., Sloman, M.: Decomposition techniques for policy refinement. In: Network and Service Management (CNSM), 2010 International Conference on. pp. 72 –79 (Oct 2010)
11. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: Proceedings of the International Workshop on Policies for Distributed Systems and Networks. pp. 18–38. POLICY '01, Springer-Verlag, London, UK (2001)
12. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
13. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Exploring alternative approaches to implement an elasticity policy. In: Cloud Computing (CLOUD), 2011 IEEE International Conference on. pp. 716 –723 (July 2011)
14. Gong, Z., Gu, X., Wilkes, J.: Press: Predictive elastic resource scaling for cloud systems. In: Network and Service Management (CNSM), 2010 International Conference on. pp. 9 –16 (2010)
15. Hayes, B.: Cloud computing. Commun. ACM 51(7), 9–11 (2008)
16. Kephart, J.O., Walsh, W.E.: 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)
17. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
18. Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., Mylopoulos, J.: On goal-based variability acquisition and analysis. In: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06). IEEE Computer Society, Minneapolis, Minnesota (September 2006)
19. Litoiu, M., Woodside, M., Wong, J., Ng, J., Iszlai, G.: A business driven cloud optimization architecture. In: 25th Symposium on Applied Computing. ACM (March 2010)

20. Mell, P., Grance, T.: The nist definition of cloud computing (2009)
21. Moore, B.: Policy core information model (pcim) extensions, rfc 3460 (Jan 2003)
22. Mylopoulos, J., Chung, L., Liao, S., Wang, H., Yu, E.: Exploring alternatives during requirements analysis. IEEE Software 18(1), 92–96 (2001)
23. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., Ben-Yehuda, M., Emmerich, W., Galan, F.: The reservoir model and architecture for open federated cloud computing. IBM Journal of Research and Development 53(4), 4:1 –4:11 (July 2009)
24. Rubio-Loyola, J., Charalambides, M., Aib, I., Serrat, J., Pavlou, G., Boutaba, R.: Business-driven management of differentiated services. In: Network Operations and Management Symposium (NOMS), 2010 IEEE. pp. 240 –247 (2010)
25. Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G.: A methodological approach toward the refinement problem in policy-based management systems. IEEE Communications Magazine 44(10), 60–68 (October 2006)
26. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. Requirements Engineering, IEEE International Conference on 0, 139–148 (2006)
27. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 60–69. SEAMS '11, ACM, New York, NY, USA (2011)
28. Simmons, B.: Strategy-Trees: A Novel Approach To Policy-Based Management. Ph.D. thesis, The University of Western Ontario (2010)
29. Simmons, B., Litoiu, M., Ionescu, D., Iszlai, G.: 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)
30. Simmons, B., Lutfiyya, H.: Strategy-trees: A feedback based approach to policy management. In: Proceedings of the 3rd IEEE international workshop on Modelling Autonomic Communications Environments. pp. 26–37. MACE '08, Springer-Verlag, Berlin, Heidelberg (2008)
31. Simmons, B., Lutfiyya, H.: Achieving high-level directives using strategy-trees. In: Proceedings of the 4th IEEE International Workshop on Modelling Autonomic Communications Environments. pp. 44–57. MACE '09, Springer-Verlag, Berlin, Heidelberg (2009)