Exploring Alternative Approaches to Implement an Elasticity Policy

Hamoun Ghanbari,
Bradley Simmons, Marin Litoiu
Dept. of Computer Science
York University
hamoun@cse.yorku.ca,
{bsimmons,mlitoiu}@yorku.ca

Gabriel Iszlai
Centre for Advanced Studies
IBM Toronto Lab
giszlai@ca.ibm.com

***Abstract*—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. The elasticity policy can be implemented as a conventional control loop or as a set of heuristic rules. In the control-theoretic approach, complex constructs such as tracking filters, estimators, regulators, and controllers are utilized. In the heuristic, rule-based approach, various alerts (e.g., events) are defined on instance metrics (e.g., CPU utilization), which are then aggregated at a global scale in order to make provisioning decisions for a given application tier. This work provides an overview of our experiences designing and working with both approaches to construct an autoscaler for simple applications. We enumerate different criteria such as design complexity, ease of comprehension, and maintenance upon which we form an informal comparison between the different methods. We conclude with a brief discussion of how these approaches can be used in the governance of resources to better meet a high-level goal over time.**

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. In cloud computing infrastructure (IaaS), platform (PaaS) and software (SaaS) are provided on-demand to end users over the Internet. An important aspect of the *cloud* is its elasticity (i.e., resources can be acquired and released on-demand as required). This elasticity combined with a seemingly infinite supply of resources is a compelling argument in favor of further exploring and/or developing this technology.

At the heart of any autonomic resource provisioning system, there is an implicit or explicit feedback loop which maps the current system measures to the proper action(s) to be taken by an actuator. A policy can be understood to represent "... any type of formal behavioral 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.

Two alternative approaches for implementing the elasticity policy for an application are as follows: (1) through the use of a control theoretic approach; and (2) through specification of a set of heuristic rules. While the first approach is based on mathematical modeling and is designed more with robustness than comprehensibility in mind the second approach is more intuitive, lightweight and easier for a human administrator to interpret,

In control theoretic approaches, a decision for selection of resources to allocate for an application, can be made dynamically based on current application performance and its projection on modified resources. The projection is usually obtained by a performance model, which quantitatively relates application level quality of service (QoS) metrics (e.g., response time of a customer application) with a given resource.

In the heuristic, rule-based approach, Event-Condition-Action rules are utilized instead of robust control formulas (see [8–12]). For example, thresholds are defined on things like CPU utilization of an individual virtual machine (VM) instance hosting an application server for an application. Should this threshold be breached (e.g., *If CPU utilization is less than 20 for 10 minutes*) a request may be sent (action) for an elastic increase to occur (i.e., of VM instances in this tier of the application). Thresholds are also utilized in the application layer on artifacts like the number of HTTP sessions detected over some interval of time.

This paper presents our experience in implementing an elasticity policy for an application using both control-theoretic and heuristic rule-based approaches. The remainder of this paper is structured as follows. Section II describes cloud resources and applications that use these resources. Section III describes a layered approach to automated provisioning. Section IV and V describe the two different common implementations of elasticity policy. Finally Section VI compares the approaches and enumerates benefits and drawbacks of each of them.

II. Applications, Workloads, and Resources

There are two main types of application workloads in the cloud: (1) transactional (e.g., e-commerce websites) and (2) batch (e.g., text mining, video transcoding, and graphical rendering).

The most well-known example of **transactional workloads** are web applications built to serve online HTTP clients. These systems, usually serve well-known content types such as HTML pages, images, and video streams. All of these contents can be statically stored or dynamically rendered by the servers.

978-0-7695-4460-1/11 $26.00 © 2011 IEEE
DOI 10.1109/CLOUD.2011.101

A typical production level e-commerce website will consist of a database back-end (possibly as Master/Slaves cluster), a cluster of application servers (e.g. Java Tomcat application server), a cluster of front-ends (e.g. Apache web server), and a set of load balancers (e.g. HA proxy). A web request has to typically navigate through this layered system to be handled, experiencing certain delay at each layer based of its *resource demand*. Often it is the case that a majority of the demand disproportionately impacts a specific layer of a deployment. That is the reason every layer is deployed as a cluster of servers, so it can be resized based on the demand for that layer.

Batch workloads consist of arbitrary, long running, resource-intensive jobs. Jobs are submitted by clients[1] to worker nodes through a *job queue*[2]. An example which uses batch-style workloads is a video transcoding website. Here the website application servers can obtain video files from end users, place these files on shared storage such as an Amazon S3 [13] bucket, and send the video's meta-data (i.e. path to the file) to a message queue implemented using Amazon SQS [14], where transcoding machines, can read the request messages from the incoming queue, retrieve videos from storage using the meta-data, transcode the video into the target format and put the result into the shared space for the clients to access.

In both cases of batch and transactional workloads the performance of applications is determined by the amount of *allocated resources*. The resources in the cloud are usually configured and quantified in terms of geographical location of data center, number of given physical or virtual machines (i.e. instances), the type of instance bought (i.e. number of CPU cores per instance), speed of each CPU core measured in instructions or CPU cycles per-second, network bandwidth (i.e. inter zone, zone-to-zone, zone-to-Internet), amount of instance RAM, speed of block level storage, and speed of file level storage.

III. Layered Approach to Automated Provisioning

Automated provisioning is decomposed into two layers:

1) the top layer decomposes administrator defined goals into a set of objectives or rules. The goals are often about minimizing cost and satisfying performance constraints [15]. The formal goals described in terms of mathematical optimization are usually broken down using several iterative (e.g. subgradient and interior point methods) or discrete techniques (e.g. heuristic search).
2) the bottom layer tries to satisfy the objectives or rules defined by the top layer. For example, if an objective is defined as a constraint over a performance metric (e.g. *response time = 2.0 seconds*) the bottom layer alters the resource allocation in order to meet that constraint.

[1]Clients could be humans or other software components

[2]A more general construct to job queue is *message queue*. Message queues are used in loosely coupled and fault tolerant systems where multiple components work at different speeds and are subject to failures. They are also used to implement poll based communication.

The focus of this work is only the bottom layer, where the elasticity policy must be enforced on the system over time. In the following sections we explain the way this enforcement is handled using control theoretic and heuristic rule-based approaches.

IV. Control theoretic implementation of elasticity policy

A control-theoretic *autoscaler* implementing an elasticity policy usually has an explicit feedback loop architecture. In this feedback loop a regulator monitors the system's inputs and outputs and controls system output around some value by applying a control signal.

In general, feedback-based control can be used to (1) satisfy a constraint or guarantee an invariant on the outputs of the system under control over time or to (2) optimize an objective function over time. As an example for the first case, one can build a controller to satisfy the invariant that CPU utilization should be less than U_0, or that the response time should be below R_0. An example for optimization based feed-back loops is a controller that aims to minimize CPU power consumption of a web server while keeping the request buffer under a certain size. The general architecture of a controller is depicted in Figure 1.

Control actions can be applied in different layers of the cloud (e.g., hypervisors, virtual machines, containers, or applications within containers). *Reference outputs* (or set-points) or *objective functions* can also be defined based on different metrics on different layers. The most common type of set-points in performance control are defined for hardware *utilization*. Such a loop, for instance, can be used to maintain high server utilization while avoiding overload. For example, in [16, 17] controllers have been used to maintain the utilization of a VM's CPU at certain percentage (usually 60% to 80%) of the total allocated CPU share of the VM (i.e. headroom principal). *Response time* or *throughput* based performance guarantees required by web applications, can also be described in terms of utilization control, and thus, be controlled based on utilization control feedback loops [17–20]. These set-points can also be combined with *power consumption* metrics (e.g. in [21]) to force hardware to operate in a more energy efficient way (e.g. with processor frequency).

Controllers may vary in their architectural complexity. This complexity depends on the existence of a model for the target system, the theory underpinning the model, and the usage (or omission) of an estimator to track hidden variables in the model. The simplest feedback loop can be constructed by directly feeding back the difference between a reference output (i.e., a set-point) and a measured output[3] as an input to the system.

For a system whose dynamics are known (e.g. through system identification techniques), a proper controller can be constructed in form of transfer function at design time using the design techniques such as pole placement, root locus, or

[3]This difference is referred to as a control error.

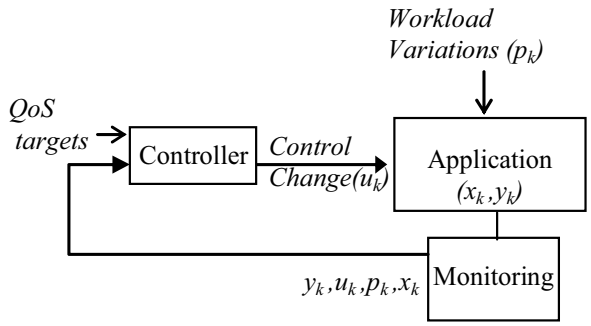


Fig. 1: The general architecture of resource allocation controller.

etc. The model used during system identification can take several forms such as transfer functions, state-space models, difference equations, etc. For example, classical linear continuous proportional-integral (PI) and proportional-integral-derivative (PID) controllers are employed in [22, 23] to processor power management, in [17] to QoS adaptation in web servers, and in [24–26] to load balancing.

The controller component can also include an estimator which dynamically adjusts the controller gain using the state estimates of the model. For example Linear-Quadratic-Gaussian (LQG) control (see Figure 2(a)) dynamically adjusts LQ-optimal gain based on a *state estimate* which is used to form a control signal. The estimator is usually a simple or extended Kalman filter [27] able to maintain a good estimate of model unknowns by calibrating itself to the measurements during runtime. As an example, [16] tries find the proper CPU share a_k for a VM that satisfies the headroom principal. In this case the controller tries to maximize the likelihood that $u_k = 60\%a_k + w_k$ (where u_k is CPU utilization and w_k is the random perturbation) by adjusting a_k based on the estimation error $e_k = u_k - 60\%a_k$. That is the controller is itself a Kalman filter with state equation $a_k = a_k + v'_k$ in which the allocation signal a_k is a tracked state variable.

Modeling the system to well-known queuing theory and estimating the parameters of such model is also a common practice. These performance models can take several different forms such as simple steady state [28–30], layered for multi-tier applications [31–35] or transient. Once the system is characterized in terms these types of theories, better reasoning can be done in terms of concepts of the theory. In advanced control formulations for QoS management these models are used within a controller to rapidly explore multiple allocation decisions and find near-optimal service provisioning during each control step [36] (see Figure 2(b)). In limited lookahead controllers (LLCs) [37, 38] a search space, formed by different choices of control actions over the predicted model [39], is explored for optimal solution. This makes it possible to address limitations of classical control techniques on non-linear systems and complex goals [37, 38].

V. Heuristic Rule Based Resource Provisioning

An architecture for a rule-based autoscaler is presented in Figure 3. Using rule bases as a substitute for robust control theory, was first introduced in fuzzy systems; and since then, there has been a debate over applicability of these rule based control systems to the practical control theoretic problems. Solutions to many classic control problems (e.g. inverted pendulum experiment) and real world ones (e.g. camera auto-focus and energy-efficient motors) have been implemented using these fuzzy rule based control techniques (e.g. [40, 41]).

As far as we are aware, the only publicly available rule-based provisioning solutions implemented for the cloud are using crisp rule engines (e.g. RightScale platform [9]); however, according to [42] one can emulate more complex probabilistic or fuzzy rules using a large collection of crisp rules. Here we will focus on how this current crisp rule engine implementation works in the context of provisioning.

Rules are specified as follows: 'if *condition* then *action*' where the *condition* represents a term defined recursively based on the value(s) of monitored metric(s)[4]. An *action* might involve the realization of a global management decision (e.g. adding or removing servers), a local management decision (e.g. increasing a process's memory), or emitting an event (e.g., an *alert*). Further, an implicit target exists for the types of management rules being considered in this paper. For example, in a transactional application the target might be an individual server while in a batch oriented application it might be the message queue.

In a rule-based autoscaler the process of deriving global provisioning decisions using the locally fired rules of individual servers is as follows: if a rule specification's *condition* holds for longer than the defined *decision_duration* in the rule's

[4]A condition can be a conjunction or disjunction of other conditions

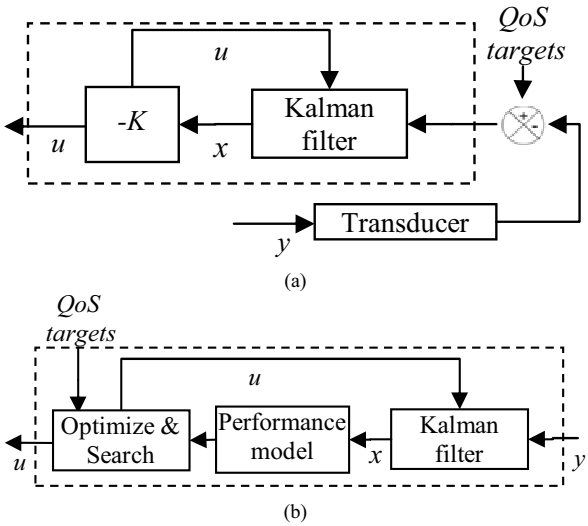


Fig. 2: Structure of (a) an LQG regulator and (b) an optimization based controller.

Alert Target	Metrics
Hardware	CPU utilization, disk access, network interface access, memory usage
General OS Process	cpu-time, pagefaults, real-memory (resident set) size
Load balancer	request queue length, session rate, number of current sessions, transmitted bytes, num of denied requests, num of errors
Web server	transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting, ...)
Application server	total threads count, active threads count, used memory, session count
Database server	number of active threads, number of transactions in (write, commit, roll-back, ...) state
Message Queue	average number of jobs in the queue, average job's queuing time

TABLE I: List of the metrics used in defining autoscaling alerts.

target, an alert is triggered from that target. If the associated target is eligible, it will then vote for performing the specified action. Once the number of targets voting to grow exceeds a specified *decision threshold* (e.g. 51%) a scaling action will be performed. If, after resizing, the number of instances falls within the interval defined between *min_instances* and *max_instances* a scaling action will launch/terminate a number of instances configured by a *resize number*. Once a scaling action occurs, the system is not allowed to perform another scaling action until a *refractory period* has passed. This is because, it takes some time to launch a new server (i.e. copying the server image to a physical host and booting a VM from it) before it becomes operational and starts offloading some of the work from the overworked instances. If these additional servers make enough impact, the triggered alerts in some targets disappear, the number of votes decreases below the decision threshold and autoscaling will be stopped. Otherwise, after the refractory period, a new autoscaling action will be triggered again.

The conditions which result in an alert being issued are defined based on various metrics. In a typical transactional system one can use hardware level metrics of each host, process level metrics of each operating system process, and service specific metrics for different types of servers (e.g. load balancer, web server, application server, and database server). In a batch oriented system one can usually define alerts based on (i) the average number of jobs in the queue or (ii) average job's queuing time (as opposed to hardware level measurements). For example, in the former, an autoscaler tries to maintain a certain *jobs-per-worker* ratio. So, if the jobs-per-worker ratio is say 10, and there are currently 50 jobs in the queue, it will adjust to 5 worker instances. Table I summarizes the metrics that can be used to issue an alert.

A template of an alert specification for a transactional system is presented in Table II. Using the alerts of Table II, one can configure an autoscaler to launch an additional instance when a majority of the servers are being overworked and then to remove an instance when servers are underutilized due to a decline in traffic. In this example, adding multiple alerts defined based on multiple metrics is also possible . For example, for an application which is both memory and CPU intensive at times, one can set up alert specifications based on both CPU utilization and available memory.

There are various ways that one can configure an autoscaler. For example, for a transactional application deployment with l hosts, m available metrics, and n predicates for defining conditions, each host might choose a subset of [predicates $\times$ metrics] (i.e. 2^{n*m} choices) to define its alerts. These alternatives, together with global configuration parameters *decision threshold*, *resize number*, *min_instances*, and *max_instances* form a huge configuration space and this makes it hard to investigate each configuration parameter's impact on the behavior of the autoscaler.

We consider the autoscaler as a transfer function which takes the workload and maps it to a function representing the number of active instances over time (i.e., a *node curve*). Next, we qualitatively categorized the effect of each configuration parameter on this transfer function. To further simplify it, we assumed there is only one subsystem under autoscaling (e.g. application server tier) with homogeneous instance types and CPU utilization is the only metric used in defining alert types.

In our experiments all the alerts in all targets were identical and based on the template introduced in Table II (e.g., two alerts where defined, one for scale-up action is case of over-utilization and one for scale-down in case of under-utilization). Over-utilization is considered when CPU idle time

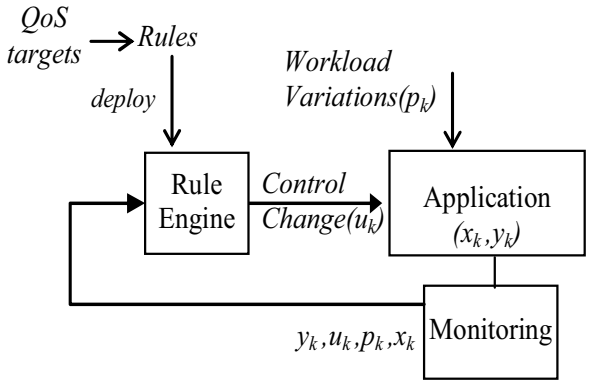


Fig. 3: Architecture of a rule-based autoscaler.

Condition	Duration	Action
If `cpu-0/cpu-idle` is < `a%`	`x` minutes	vote to `grow`
If `cpu-0/cpu-idle` is > `b%`	`y` minutes	vote to `shrink`

TABLE II: Template of alert specifications.

is less than the lower bound *cpu-idle-lb* and underutilization is defined as when CPU idle time is more than the upper bound *cpu-idle-ub*. The interval $[$*cpu-idle-lb*, *cpu-idle-ub*$]$ formed by these two bounds, together with *decision threshold*, *refractory period*, *decision durations*, *resize numbers*, and $[$*min_instances*, *max_instances*$]$ interval form a simpler configuration space that we investigate in this work. We summarize the effect of different autoscaler configurations on the scaling behavior as follows:

Operating interval formed by $[$*cpu-idle-ub*, *cpu-idle-ub*$]$ is the main tool in configuring the long term size of the cluster. It basically forces the autoscaler to act until the cluster reaches the desired average operating interval (here in terms of CPU idle time, e.g. $[20\%, 30\%]$ idle).

If the desired operating interval is located at high CPU idle values (e.g. $[40\%, 50\%]$), there will be a lot of instances launched. Autoscaling will eventually stop due to the increase in the number of idle instances, resulting in less alerts being signaled (as the triggering threshold is no longer reached). Notice that the autoscaling action is only performed if the intensity of the workload can get cluster nodes to the desired operating point. Thus, if the operating point is configured when nodes are fully utilized (e.g. $[0\%, 10\%]$ CPU idle time), the cluster might become unresponsive for low intensity workloads.

If this interval is configured to be narrow and it is placed in an operating area reachable by the workload, it will make the autoscaler sensitive to changes. Such an autoscaler would do some thrashing just to keep the operating point at this narrow interval. If the interval is wide, the cluster will be less responsive, and the cluster utilization will have more space for variations.

Decision duration contributes to the responsiveness of the cluster. A shorter decision duration for each node results in a quicker response and, thus, it is better at detecting highly variable workloads (e.g. a workload that changes direction every 15 minutes). In case of steady or monotonically increasing/decreasing workloads the cluster is going to stabilize on a specific size no matter what decision duration is.

Resize numbers represents the amount of servers to add or remove should a decision threshold (i.e., quorum) be reached (e.g., 51%). This has a short term effect on the size of the cluster by making it grow or shrink faster. This might be useful for heavily increasing or decreasing workloads. However, in a long run for a lightly changing workload this might only have effect on smoothness of cluster size over time.

Decision threshold is global parameter closely related to decision duration, which can affect the responsiveness of the cluster specially in a heterogeneous clusters. In a heterogeneous cluster, nodes with less processing power might reach the saturation level faster. In this case a lower decision threshold might result in a better autoscaling decision. The same applied to cluster composed of two different types of nodes (e.g. frontend and application server) where the load on one type of node is higher than the other (e.g. a web application with lots of demand on static resources would result into more utilization at the frontend).

Instance bounds formed by $[$*min_instances*, *max_instances*$]$ acts as constraint on the size of the cluster.

Refractory period is the interval of time which follows an autoscaling action during which no autoscaling actions may occur. With small refractory periods, the cluster makes resizing decisions more frequently resulting in more responsive behavior.

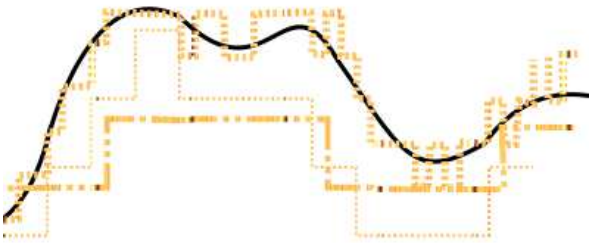


Fig. 4: Effect of different autoscaler configurations on the scaling behavior.

Figure 4 schematically represents the behavior of three different autoscaler configurations on the scaling behavior. The workload is represented by a solid line, while the number of instances (an a representative of autoscaler behavior) in three different configurations is represented by step-like functions.

A. Experiment

We tested a commercial implementation of a rule-based management system for managing cloud based services called RightScale[9]. We used Amazon Web Services (AWS) as the IaaS provider. All platform instances were built atop VM instances running Ubuntu 9.10 i386 or CentOS 5.4 i386 (i.e., front end servers, application server instances, and databases) and configured as *m1.small* instances (i.e., 1.7 GB memory, 1 EC2 Compute Unit, 160 GB instance storage, 32-bit platform and moderate I/O Performance).

RightScale was utilized as the PaaS provider. A standard, multi-tiered application topology was selected from their catalog. The platform topology consisted of two front-end servers running Apache, HAProxy and Tomcat, a backend database running MySQL 5.0 and a server array of Tomcat instances.

A simple web application was deployed on the topology. When an automated client connected to this web application, load was generated on its virtual CPU, interactions with the database tier occurred and then a response was returned to the client.

Three different elasticity policies were designed to drive the autoscaling actions of the application server tier. These rule sets utilized different settings of the configurable parameters mentioned above and are presented in Table III. The FIFA '98 workload [43] was used as a representative workload against which the application was run.

A plot of the platform instances purchased over time and the average utilization of a server in the cluster is presented

Parameter	EP_1	EP_2	EP_3
resize numbers	1	1	2
decision threshold	51%	51%	51%
operating interval	[50,45]	[55,40]	[55,50]
decision duration	7 min	7 min	8 min
refractory_period	8 min	8 min	6 min
instance bounds	[2,20]	[2,20]	[2,20]

TABLE III: Parameter settings defining the three elasticity policies (EP).

in Figure 5(b) and (c) for each set of rules. It is noticeable that different rule sets result in different elastic behavior for the same workload.

VI. Discussion

In our past experiences we explored the idea of model based control, by developing more realistic and accurate models of computing systems [44, 45]. We have also performed a through investigation of estimator component of model based control loops using different variations of Kalman filter [46] and adopted low-cost estimation mechanisms for large scale web applications [34]. We utilized these models in an autonomic controller aimed at maintaining QoS and Service Level Agreements (SLAs) of a software system in [36]. In that work we used provisioning, tuning and load balancing as control mechanisms. Further, one of the authors has experience working in the domain of policy-based management [47, 48] and has previously introduced a high level feedback based management structure for governing the active deployment of policy sets, called a *strategy-tree* [49].

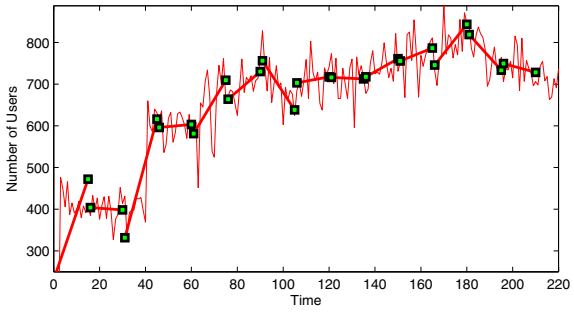
In this paper we have considered two alternative approaches to implementing an elasticity policy for an application. While we feel that these approaches are complementary in a general sense, we shall enumerate some observations we made in the following sections.

A. Design Complexity

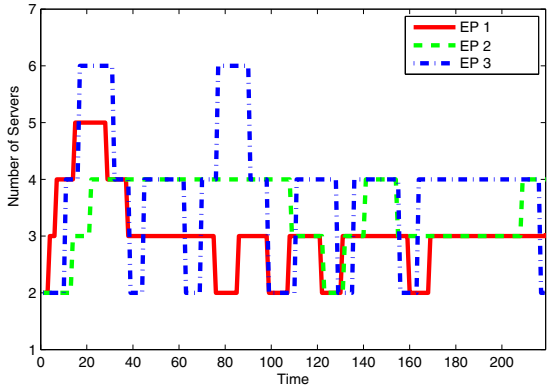
Designing a formal model (control theory) can be a complicated process. It has taken years to build existing mathematical performance models. It might be also very time consuming to tailor an existing model towards a specific need, for example adding predictor variables to the model. As an example, in case of computer performance models, it might be hard to tailor a queuing model, to take into account factors such as multi-programming level or threading, synchronous calls, thread pools, effect of RAM, and swapping.

Mathematical models incorporate assumptions that may not always hold at runtime. For example a mean value analysis solution to a queuing network problem is based on the assumption that the workload is an stationary process with a fixed mean over time and Poisson distribution. Such model can only address the time-invariant/static behavior of the network. This, for example, makes it impossible to use it for temporary bursts of traffic which cause short term saturation.

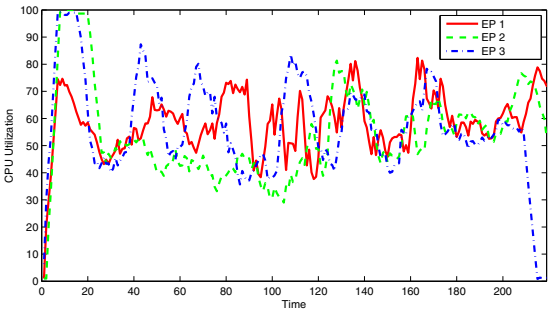
In contrast to compact mathematical formulas, advanced machine learning techniques can be used to quickly build new



(a)



(b)



(c)

Fig. 5: The (a) workload versus (b) the number of fired servers and (c) the average utilization of a server in the cluster for three elasticity policies (EP_1, EP_2 and EP_3) .

models for new situations. Such techniques, however, need a training phase when applied in a new environment. For example a performance model, which is a result of training a neural net with performance data obtained from an Apache HTTP Server-MySQL bundle should be retrained if the bundle uses memory object caching system such as Memcached.

Rules can also be complex. Multiple rules might conflict with each other in terms of outcome. Different conflict resolution or aggregation policies might be used and result in different behaviors. Rules might have multiple variables. With

large numbers of rules, their conditions might overlap in configuration space or they might miss achieving full coverage. To make sure that whole configuration space is covered, default rules can be used. But they might not be as efficient. Finally, designing optimal rules in an n-dimensional variable space, using some mathematical methods like Markovian models can be as hard as model building [50].

B. Ease of comprehension

Models require a deep understanding of mathematics and are not easily interpreted by non-specialists. Some models such as those developed for performance evaluation of closed queuing networks can only be solved using iterative methods Other models built using neural nets are also hard to understand. Also compactness of these models, although helpful, would result in some hidden behavior that looks implicit when used in control or optimization.

Rules, on the other hand, are sometimes easier to understand. Designer's concerns can be mapped to rules in a more straightforward way. For example in the provisioner, it is easy to spot the reason a certain scaleup action has occurred based on past events, and, to see what rules have been contributed to the decision. There is also less hidden behavior compared to mathematical models.

Rules can get complex as their numbers grow. If the rule description language is not rich enough in terms of syntactic constructs, multiple rules might be used to address a single concern (e.g. *when CPU utilization is high, RAM should be increased for the web server only if there are a lot of busy threads*). One-to-many mapping between concerns and rules might result in a scattering of concerns among multiple rules making it difficult to track and make changes.

C. Maintenance of structure

In order to keep the configured autoscaler up-to-date and usable in a changed environment it is necessary to perform modifications (i.e. maintenance). Depending on the way the autoscaler is implemented, the cost of this modification might be different. Re-tuning the model can be performed by performing offline regression on a new dataset. Changing the structure of the model might be harder in some cases such as mathematical models, because the laws used to derive the relationship between variables must be investigated again. In heuristic models however, adding or removing variable is not difficult. In supervised learning methods, such as neural networks, one can just re-calculate new weights using the new data.

In the case of rules, one can easily customize the system for a certain environment without having to totally redesign it. Rules may be added or removed incrementally and thus, it is easy to see what rules have contributed to a decision.

VII. Conclusion and future work

This paper has presented an overview of our experiences implementing an elasticity policy. Recall that the elasticity policy was derived from a high-level goal. A goal may have multiple decompositions [51]. Each decomposition incorporates various assumptions and expectations about the system. At run-time, should these assumptions underpinning the elasticity policy prove incorrect, the system may behave poorly (or even pathologically). It is very important to constantly evaluate the performance of the system in relation to expectations. Further, this should be done on multiple time scales as well.

Therefore, as was mentioned previously (see Section III), a second layer of reasoning among the numerous possible elasticity policies is required. Strategy-trees offer one such approach. In a strategy-tree, each DEC-element (OR node) represents a multi-criteria decision problem [52]. Further, the strategy-tree facilitates complete MAPE loops at multiple granularities of time. Managing elasticity policy deployment using strategy-trees to better achieve economic objectives (i.e., maximize revenue) will be considered in our future work.

VIII. Acknowledgments

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

References

[1] B. Hayes, "Cloud computing," *Communications of the 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," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, February 2009, http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[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, no. 4, pp. 535–545, 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 Computer Systems*, 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 *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*. New York, NY, USA: ACM, 2003, pp. 164–177.

[6] "VMware," http://www.vmware.com/ [online January 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*. IEEE Computer Society, June 2004, pp. 3–12.

[8] M. Desertot, C. Escoffier, and D. Donsez, "Autonomic management of J2EE edge servers," in *Proceedings of the 3rd international workshop on Middleware for grid computing*. Grenoble, France: ACM, 2005, p. 16.

[9] "Rightscale autoscaling using voting tags," http://support.rightscale.com/12-Guides/Lifecycle_Management/03_-_Understanding_Key_Concepts/RightScale_Alert_System/Alerts_based_on_Voting_Tags/Understanding_the_Voting_Process [online January 2011].

[10] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Elastic management of cluster-based services in the cloud," in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ser. ACDC '09. New York, NY, USA: ACM, 2009, pp. 19–24.

[11] T. Chieu, A. Mohindra, A. Karve, and A. Segal, "Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment," in *Proceedings of the 2009 IEEE International Conference on e-Business Engineering*. IEEE Computer Society, 2009, pp. 281–286.

[12] Y. Zhang, G. Huang, X. Liu, and H. Mei, "Integrating Resource Consumption and Allocation for Infrastructure Resources on-Demand," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 75–82.

[13] "Amazon Simple Storage Service (Amazon S3)," http://aws.amazon.com/s3 [online January 2011].

[14] "Amazon Simple Queue Service (Amazon SQS)," http://aws.amazon.com/sqs [online January 2011].

[15] J. Sauvé, F. Marques, A. Moura, M. C. Samaio, J. Jornada, and E. Radziuk, "SLA design from a business perspective," *Proceedings of the 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, pp. 72–83, October 2005.

[16] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6th international conference on Autonomic computing (ICAC '09)*. Barcelona, Spain: ACM, Jun. 2009, pp. 117–126.

[17] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, "Performance guarantees for web server end-systems: A control-theoretical approach," *IEEE Transactions on Parallel and Distributed Systems*, pp. 80–96, 2002.

[18] T. Abdelzaher and C. Lu, "Modeling and performance control of internet servers," in *Proceedings of the 39th IEEE Conference on Decision and Control*, vol. 3. IEEE, 2000, pp. 2234–2239.

[19] 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 *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium.*, 2003, pp. 208–217.

[20] T. F. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu, "Feedback performance control in software services," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 74–90, Jun. 2003.

[21] N. Kandasamy, S. Abdelwahed, and J. P. Hayes, "Self-optimization in computer systems via online control: Application to power management," in *First International Conference on Autonomic Computing (ICAC'04)*. New York, New York: IEEE Computer Society, May 2007, pp. 54–61.

[22] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron, "Control-theoretic dynamic frequency and voltage scaling for multimedia workloads," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM, 2002, pp. 156–163.

[23] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu, "Power-aware QoS management in Web servers," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. IEEE, pp. 63–72.

[24] J. Hellerstein and I. ebrary, *Feedback control of computing systems*. Wiley Online Library, 2004.

[25] C. Lu, G. Alvarez, and J. Wilkes, "Aqueduct: online data migration with performance guarantees," in *Proceedings of the Conference on File and Storage Technologies*. USENIX Association, 2002, pp. 219–230.

[26] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using control theory to achieve service level objectives in performance management," *Real-Time Systems*, vol. 23, no. 1, pp. 127–141, 2002.

[27] G. Welch and G. Bishop, "An introduction to the Kalman filter," *University of North Carolina at Chapel Hill, Chapel Hill, NC*, 1995. [Online]. Available: http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html

[28] D. C. Petriu, "Approximate mean value analysis of client-server systems with multi-class requests," in *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. ACM New York, NY, USA, 1994, pp. 77–86.

[29] D. C. Petriu and C. M. Woodside, "Approximate mean value analysis based on markov chain aggregation by composition," *Linear Algebra and its Applications*, vol. 386, pp. 335–358, 2004.

[30] E. Badidi, L. Esmahi, and M. Serhani, "A queuing model for service selection of multi-classes qos-aware web services," in *Third IEEE European Conference on Web Services (ECOWS)*, 2005, p. 9.

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

[32] S. Ramesh and H. G. Perros, "A multi-layer client-server queueing network model with synchronous and asynchronous messages," in *Proceedings of the 1st international workshop on Software and performance*. ACM New York, NY, USA, 1998, pp. 107–119.

[33] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, "Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 2, 2006.

[34] H. Ghanbari, M. Litoiu, M. Woodside, T. Zheng, J. Wong, and G. Iszlai, "Tuning tracking of performance model parameters using dynamic job classes," in *Proceedings of the second ACM/SPEC International Conference on Performance Engineering (ICPE 2011), to appear*. ACM, 2011.

[35] T. K. Liu, S. Kumaran, and Z. Luo, "Layered queuing models for enterprise javabean applications," in *Proc. 5th International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 4–7.

[36] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, May 2005.

[37] S. Abdelwahed, N. Kandasamy, and S. Neema, "A control-based framework for self-managing distributed computing systems," in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM, 2004, pp. 3–7.

[38] N. Kandasamy, S. Abdelwahed, and J. Hayes, "Self-optimization in computer systems via on-line control: application to power management," in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, pp. 54–61.

[39] V. Bhat, M. Parashar *et al.*, "Enabling self-managing applications using model-based online control strategies," in *2006 IEEE International Conference on Autonomic Computing*. IEEE, 2006, pp. 15–24.

[40] V. Bevilacqua, E. Grasso, G. Mastronardi, and L. Riccardi, "A soft computing approach to the intelligent control," in *4th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 2007, pp. 1312–1317.

[41] "Introduction to FuzzyJess," http://marek.piasecki.staff.iiar.pwr.wroc.pl/dydaktyka/isa/2010_l/fuzzyJDocs/FuzzyJess.html.

[42] D. Dubois, E. Hellermeier, and H. Prade, "On the representation of fuzzy rules in terms of crisp rules," *Information Sciences*, vol. 151, pp. 301 – 326, 2003.

[43] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *Network, IEEE*, vol. 14, no. 3, pp. 30 –37, may. 2000.

[44] M. Litoiu, "APERA (Application performance evaluator and resource allocation tool)." [Online]. Available: www.alphaworks.ibm.com/tech/apera

[45] M. Litoiu, "A performance analysis method for autonomic computing systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 2, no. 1, p. 3, 2007.

[46] 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.

[47] B. Simmons, H. Lutfiyya, M. Avram, and P. Chen, "A Policy-Based Framework for Managing Data Centers," in *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*. IEEE, 2006, pp. 1–4.

[48] B. Simmons and H. Lutfiyya, "Policies, grids and autonomic computing," in *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*. ACM, 2005, pp. 1–5.

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

[50] K. G. Shin, C. Krishna, and Y. hang Lee, "Optimal dynamic control of resources in a distributed system," *IEEE Transactions on Software Engineering*, vol. 15, no. 10, pp. 1188 –1198, Oct. 1989.

[51] 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.

[52] M. Litoiu, "A performance engineering method for web applications," in *Web Systems Evolution (WSE), 2010 12th IEEE International Symposium on*, 2010, pp. 101 –109.