Self-managed Computer Systems: Foundations and Examples

Daniel A. Menascé(✉)

George Mason University, Fairfax, VA 22030, USA
menasce@gmu.edu
http://www.cs.gmu.edu/faculty/menasce.html

Abstract. The traditional approach to managing complex computer systems is to use a cadre of skilled IT professionals who use monitoring tools in order to detect when problems arise. They are then able to use their skills and experience to determine what actions should be taken to solve the problems. This approach is no longer viable for highly complex, networked computer information systems that have numerous configuration knobs, and operate in environments that vary with time at a very high rate. In this case, one cannot expect that design-time configurations will make the system operate optimally at run-time. For that reason, complex systems need to manage themselves using controllers that make the systems self-configuring, self-optimizing, self-healing, and self-protecting. This paper provides a formalism to describe self-managed systems and discusses concrete examples that illustrate how these properties are enforced by controllers in a variety of domains including cloud computing, fog/cloud computing, Internet datacenters, distributed software systems, and secure database systems.

Keywords: Autonomic computing · Self-managed systems · Utility functions

1 Introduction

The traditional approach to managing complex computer systems is to use a cadre of skilled IT professionals who use monitoring tools in order to detect when problems arise. They are then able to use their experience to determine what actions should be taken to solve the problems. This approach is no longer viable for networked computer systems composed of a very large number of interconnected servers, have many software layers that may include services developed by many different vendors, are composed of hundreds of thousands of lines of code, and are user-facing. The complexity described above is compounded by the fact that the workload intensity of these complex systems varies in rapid and hard-to-predict ways.

For the above reasons, it is virtually impossible for human beings to change the configuration settings of a complex computer system in near real-time in

J. Filipe et al. (Eds.): ICEIS 2019, LNBIP 378, pp. 17–36, 2020.
https://doi.org/10.1007/978-3-030-40783-4_2

order to steer the system to an optimal or near optimal operating point that meets user-established Quality of Service (QoS) goals. Recognizing this, IBM introduced the concept of *autonomic computing*, as a sub-discipline of computer science that deals with systems that are self-configuring, self-optimizing, self-healing, and self-protecting [15]. Autonomic computing systems are also referred to as *self-managed* systems.

Self-managed systems have stringent QoS requirements in terms of response time, throughput, availability, energy consumption, and security. The values of the metrics above depend on the current settings of the configuration knobs. Additionally, there are *tradeoffs* between these metrics. For example, the throughput of a database server is a function of its maximum number of database connections. However, contention for processing and I/O resources increases with the number of database connections. As a result, the average response time increases with resource contention. As another example, a system's security increases as stronger encryption algorithms are used. However, these stronger algorithms imply in added CPU processing time and increased response time. As yet another example, current microprocessors allow for the CPU clock frequency to be adjusted by software. Lower clock frequencies reduce energy consumption but increase response time.

This paper is an extended version of the conference paper [21]. The rest of this paper is organized as follows. Section 2 describes the fundamentals of self-managed systems and provides a concrete example based on automatically allocating CPU shares to virtual machines. Section 3 discusses how an autonomic controller can be used to provide elasticity to cloud providers allowing them to cope with workload surges by dynamically varying the number of servers offered to users. Section 4 provides an example of how an autonomic controller can deal with tradeoffs between security and response time by dynamically varying the security policies of an Intrusion Detection and Prevention Systems (IDPS). The next section discusses how an autonomic controller can dynamically control the voltage and frequency of a CPU in order to meet performance requirements with the least possible energy consumption. Section 6 provides a list of other examples of self-managed systems. Finally, Sect. 7 discusses some concluding remarks.

2 Fundamentals of Self-managed Systems

This section discusses the basics of self-managed systems aka *autonomic computing* systems, a term coined by IBM [15] more than a decade ago. Additionally, this section provides a simple example to illustrate the notation and formalism presented here.

2.1 Self-managed Systems

The term autonomic computing was inspired by the central autonomic nervous system, which unconsciously regulates bodily functions such as the heart and respiratory rate, digestion, and others, based on high-level goals. For example,

if you arrive at an airport late for your flight, you will run to the gate, more adrenaline will be secreted into your bloodstream, your heart rate will accelerate, and your lungs will breath at a higher rate; all of this without you being conscious. But, you have a high-level goal that is driving all of it: catch your flight.

Figure 1 illustrates the basic components of a self-managed system. The system to be controlled is subject to a *workload* that consists of the sets of all inputs to the system (e.g., requests, transactions, web requests, and service requests). The *output metrics* of the system are associated with the QoS delivered by the system when processing the inputs.

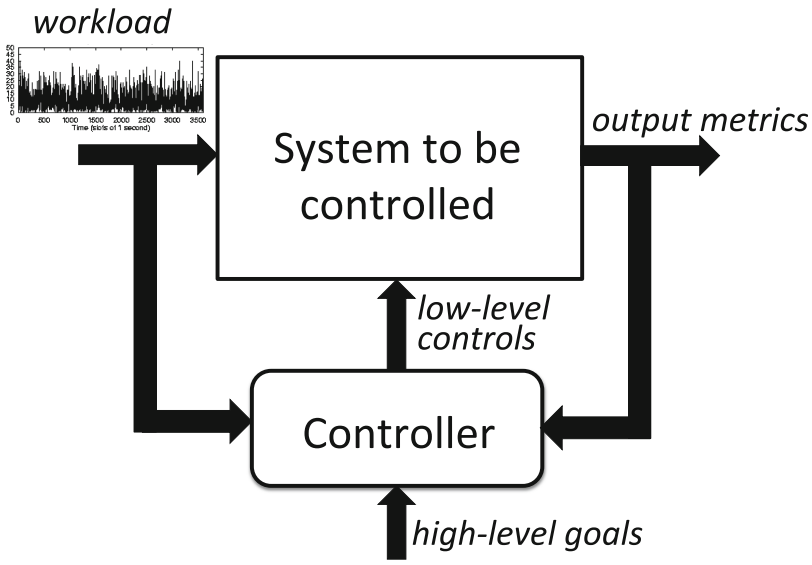

Fig. 1. Basic components of a self-managed system (From [21]).

Figure 1 also depicts a controller that *monitors* the system input, i.e., the workload, its output metrics, and compares the measured output metrics with high-level goals established by the system stakeholders. Examples of high-level goals are: (a) 95% of web requests have a response time less than or equal to 0.8 s; (b) the average search engine throughput is at least 4,600 queries/sec; (c) the availability of the e-mail portal is greater than or equal to 99.98%; and (d) the percentage of phishing e-mails filtered by the e-mail portal is greater than or equal to 90%. The controller reacts to deviations from the high-level goals established by the stakeholders and automatically derives a *plan* to change the system's configuration by acting on low-level controls in a way that improves the system's QoS and makes it compliant, if the system resources permit, with the high-level goals.

Self-managed systems work along the following dimensions: (a) *Self-configuring*: The system automatically decides how to best configure itself when new components or services become available or when existing ones are decommissioned. (b) *Self-optimizing*: The system attempts to optimize the value of its QoS metrics (e.g., minimizing response time, maximizing throughput and availability). (c) *Self-healing*: The system has to automatically recover from failures. This requires that the root causes of failures be determined and that recovery plans be devised to restore the system to an adequate operational state. In addition, the system has to predict the occurrence of failures and prevent their manifestation. (d) *Self-protecting*: The system has to be able to detect and prevent security attacks, even zero-day attacks, i.e., attacks that target publicly known but still unpatched vulnerabilities.

Optimizing a system for the four dimensions above may be challenging because there are tradeoffs among them. For example, it may be necessary to add several cryptographic-based defenses to improve a system's security. However, these defenses have a computational cost and increase the response time and decrease the throughput [17]. As another example, one may increase the reliability of a system, and therefore improve its self-healing capabilities, by using redundant services with diverse implementations. However, this approach tends to increase response time.

In addition, there usually are constraints in terms of cost and/or energy consumption associated with this optimization problem, which has to be solved in near real-time to cope with the rapid variations of the workload. This problem is a multi-objective optimization problem [24]. In order to deal with the tradeoffs, it is common to use *utility* functions for each metric of interest and then combine them into a *global utility* function to be optimized.

2.2 Utility Functions

A utility function indicates how useful a system is with respect to a given metric. Utility functions are normalized (in our case in the [0, 1] range) with 1 indicating the highest level of usefulness and 0 the lowest. A utility is a dimensionless quantity. For example, if the metric is response time, the utility function of the response time decreases as the response time increases, and approaches 1 as the response time decreases and approaches zero. As another example, a utility function of availability increases as the availability increases.

We assume here that all utility functions are *consistent*, i.e., they increase or decrease in the right direction according to the metric. So, a utility function that increases as the response increases is not consistent. Figure 2 shows two examples of utility functions in the shape of sigmoid functions. The top part of the figure shows three different utility functions of execution time with different shape factors (α) but with the same service level goal ($\beta = 65.0$), which is the inflection point of the curve. The bottom part of Fig. 2 shows three different availability utility functions. The inflection point is the same for all of them, i.e., 0.99.

The controller of Fig. 1 typically awakes at regular time intervals, called *controller intervals* of duration denoted as Δ. Then, the controller (a) verifies all the *monitoring* data collected during the past controller interval(s), (b) *analyzes* how

the measured output metrics compare with the high-level goals, (c) generates, if necessary, a *plan* to change the configuration controls to bring the system in line with the high-level goals, and (d) *executes* the plan by sending commands to the system. The plan is generated based on *knowledge* of *models* of the system behavior, which will guide the generation of new configuration parameters as explained in what follows. The paradigm described above is called MAPE-K, which stands for **M**onitor, **A**nalyze, **P**lan, and **E**xecute based on **K**nowledge [15]. Figure 3 shows the details of the elements of an autonomic controller and the MAPE-K loop.

2.3 Formal Definition of an Autonomic Controller

We formalize here the operation of an autonomic controller (just controller heretofore). To that end we define the following notation.

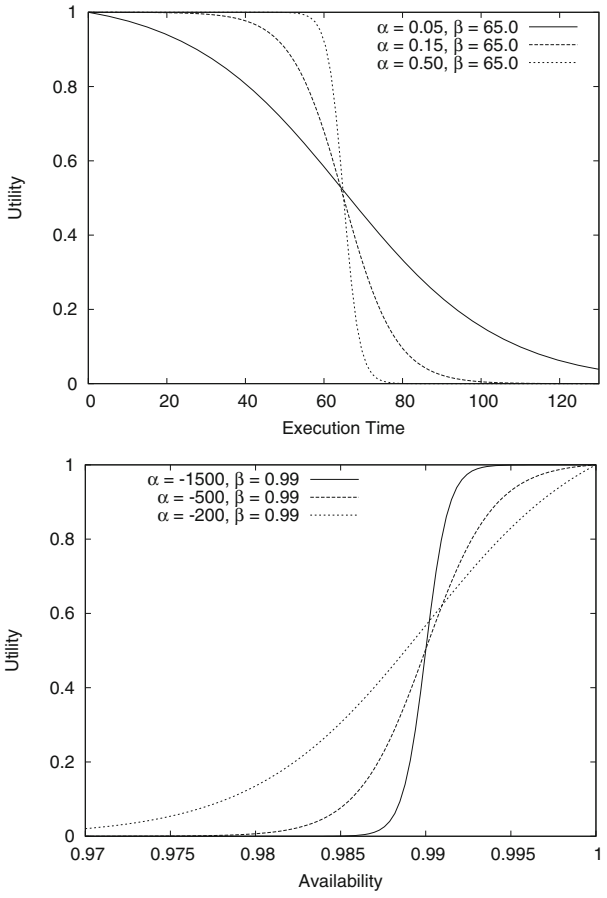

Fig. 2. Top: examples of utility functions for execution time. Bottom: examples of utility functions for availability. All examples are sigmoid functions. (From [21]).

- K: number of configuration knobs (low level controls in Fig. 1) the controller is able to change.
- $\mathbf{C}(t) = (C_1(t), \cdots, C_K(t))$: vector of values of the K configuration knobs at time t.
- $\mathcal{C}$: set of all possible vectors $\mathbf{C}(t)$.
- $\mathbf{W}(t)$: workload intensity at time t. This is usually the workload intensity in the last controller interval(s) but could also be the predicted workload for the next controller interval.
- $S(t) = (\mathbf{C}(t), \mathbf{W}(t))$: system state at time t, which consists of the system configuration and the workload at time t.
- m: number of output metrics monitored by the controller.
- $\mathcal{D}_i$: domain of metric i ($i = 1, \cdots, m$).
- $x_i(t) \in \mathcal{D}_i$: value of metric i ($i = 1, \cdots, m$) at time t.
- $g_i(S(t))$: function used to compute (i.e., estimate) the value of metric i when the system is at state $S(t)$. So, $x_i(t) = g_i(S(t)) = g_i((\mathbf{C}(t), \mathbf{W}(t))$. The function $g_i()$ represents a *model* of the system being controlled. This function can be obtained by solving an analytic model or can be learned from previous observations. In virtually all cases of interest, the functions $g_i()$ are non-linear.
- $U_i(x_i) \in [0, 1]$: utility function for metric i. This is a function of the values of metric i.
- $U_g(x_1, \cdots, x_m) = f(U_1(x_1), \cdots, U_m(x_m))$: global utility function, which is a function of all individual utility functions.

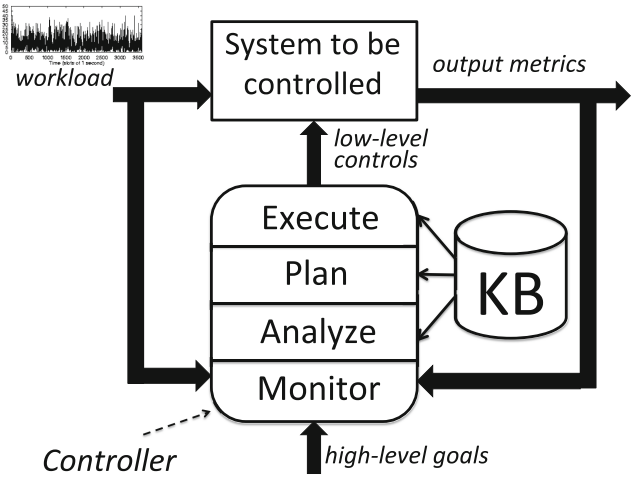

Fig. 3. An autonomic controller and the MAPE-K loop. KB = Knowledge Base.

The functions $U_i()$, $i = 1, \cdots, m$ and $U_g()$ are the high-level goals and are determined by the stakeholders.

At any time instant t at which the controller wakes up, it selects values for the configuration parameters that will be in place from time t to time $t + \Delta$; when the controller wakes up again at time $t + \Delta$ it makes another selection of parameters.

Because the global utility function is a function of the values of the metrics (i.e., $U_g(x_1, \cdots, x_m)$) and because each value x_i is a function $g_i(S(t)) = g_i((\mathbf{C}(t), \mathbf{W}(t))$ of the system parameters, the controller needs to find a configuration vector $\mathbf{C}^*(t)$ that maximizes the global utility function. More precisely,

$$\mathbf{C}^*(t) = \text{argmax}_{\forall\ \mathbf{C}(t) \in \mathcal{C}}\{f(U_1(g_1((\mathbf{C}(t), \mathbf{W}(t)))), \cdots, U_m(g_m((\mathbf{C}(t), \mathbf{W}(t)))))\}. \quad (1)$$

In many cases, we may want to add constraints such as a cost constraint: $Cost(\mathbf{C}(t)) \leq \text{CostMax}$.

It should be noted that complex computer systems have a large number of configuration knobs and the number of possible values of each is usually large. Therefore, we have a combinatorial explosion in the cardinality of $\mathcal{C}$, which is of the order of $\prod_{k=1}^{K} \mid C_k(t) \mid$, where $\mid C_k(t) \mid$ is the number of possible values of configuration knob k.

Additionally, the solution of the optimization problem stated above has to be obtained in near-real time. For this reason, we often resort to the use of combinatorial search techniques such as hill-climbing, beam-search, simulated annealing, and evolutionary computation to find a near-optimal solution in near real-time [11].

Most designers of autonomic controllers use global utility functions as the function to be optimized. Another approach, presented in [12] uses the optimal multi-dimensional utility vectors on a Pareto front identifying the scalarization weights that makes each utility vector better than all other optimal utility vectors. Exact solvers may be used if they are able to solve the optimization problem in a timely manner so that it can be used for control purposes.

2.4 VM CPU Shares Mapping Example

In order to illustrate the formalism above consider the following scenario. A physical machine runs K virtual machines (VM). Each virtual machine k ($k = 1, \cdots .K$) is allocated a share $C_k(t)$ of the physical CPU at time t, where $\sum_{k=1}^{K} C_k(t) = 1$. An example of CPU shares allocation can be found in VMware's vSphere 4.1 Resource Allocation Shares capability. So, the vector of values of the K configuration knobs at time t is $\mathbf{C}(t)$. The workload intensity at time t is $\mathbf{W}(t) = \{\mathcal{W}_1(t), \cdots, \mathcal{W}_K(t)\}$ where $\mathcal{W}_k(t), k = 1, \cdots, K$ is the average arrival rate of requests to VM k at time t. Thus, the system state at time t is $S(t) = (\mathbf{C}(t), \mathbf{W}(t))$. The example in this section illustrates a controller that dynamically changes at each controller interval the CPU shares allocated to each VM in order to maximize a global utility function defined as a function of the utility function of each of the K VMs (see below).

Let the average CPU time of requests at VM k be denoted as D_k when VM k is allocated 100% of the CPU. Thus, the average CPU time of requests at VM k is $D_k/C_k(t)$ when VM k is allocated a share $C_k(t)$ of the CPU. Assuming for simplicity that the workload is CPU bound, the average response time $R_k(t)$ of requests submitted to VM k is given by Eq. (2) using well-known queuing theory results [22].

$$g_k(S(t)) = R_k(t) = \frac{D_k/C_k(t)}{1 - \mathcal{W}_k(t)D_k/C_k(t)}. \tag{2}$$

Note that $D_k/C_k(t)$ does not include any contention for the virtual CPU at VM k while $R_k(t)$ is the sum of $D_k/C_k(t)$ with the contention for the use of the virtual CPU at VM k. Assume that the utility function $U_k(R_k(t))$ assigned by the stakeholders to VM k is the sigmoid function given by Eq. (3).

$$U_k(R_k(t)) = \frac{1 + e^{\alpha_k \cdot \beta_k}}{e^{\alpha_k \cdot \beta_k}} \frac{e^{\alpha_k(\beta_k - R_k(t))}}{1 + e^{\alpha_k(\beta_k - R_k(t))}} \tag{3}$$

where $R_k(t)$ is given by Eq. (2). Note that $U_k(R_k(t)) = 1$ when $R_k(t) = 0$ and $\lim_{R_k(t)\to\infty} U_k(R_k(t)) = 0$.

Equation (4) shows an example of a global utility function as a weighted average of the utility functions of all VMs. The weights w_k are such that $\sum_{k=1}^{K} w_k = 1$.

$$U_g(\mathbf{C}(t), \mathbf{W}(t)) = \sum_{k=1}^{K} w_k U_k(R_k(t)). \tag{4}$$

The autonomic controller will then wake up at every Δ seconds and compute an allocation of CPU shares to the K VMs that achieves an optimal (i.e., maximum) or near-optimal value of the global utility $U_g(\mathbf{C}(t), \mathbf{W}(t))$.

Figure 4 shows four consecutive time instants at which the controller wakes up. There are two workloads in this case (solid blue and dashed red in the figure) indicated by the average value of the arrival rate of request in each controller interval of duration Δ. The bottom part of the figure shows how the allocation of CPU shares to the two VMs varies due to the variation of the workload in the previous interval. For example, at time $t+\Delta$ the controller assigns 70% of the CPU to VM1 and 30% to VM2. However, during the next controller interval, the workload submitted to VM2 surpasses that of VM1 and the allocation of CPU shares changes to 25% to VM1 and 75% to VM2. During the subsequent controller interval, the workload submitted to VM1 exceeds that of VM2 and the controller allocates 80% of the CPU to VM1 and the remaining 20% to VM2.

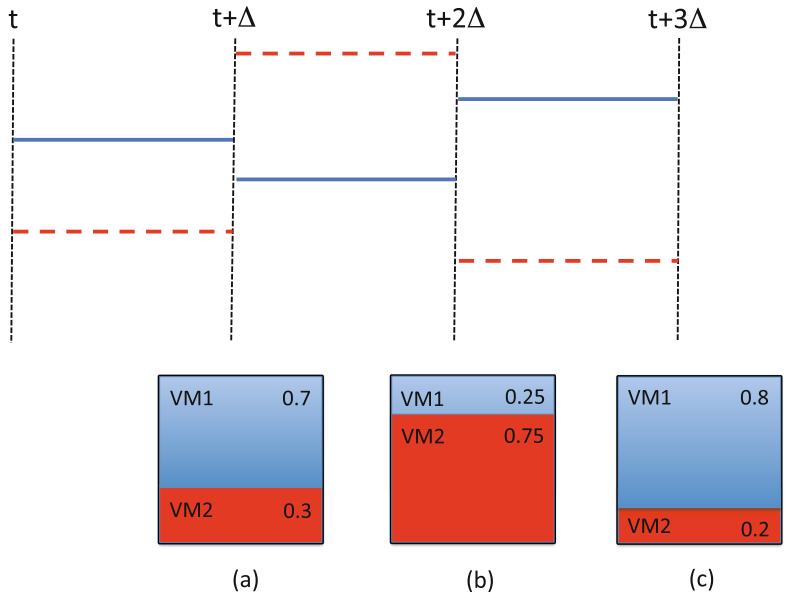

Fig. 4. Example of VM CPU allocation variation every Δ time units. There are two different workloads (solid blue and dashed red). (Color figure online)

3 Taming Workload Surges

Most user-facing systems such as Web sites, social network sites, and cloud providers suffer from the phenomenon of *workload surges* (aka flash crowds), i.e., periods of relatively short duration during which the arrival rate (measured in arriving requests per second) exceeds the system's capacity (measured in the maximum number of requests per second that can be processed). The ratio between the average arrival rate of requests and the system's capacity is called *traffic intensity* and is typically denoted by ρ in the queuing literature [22]. A queuing system is in steady-state when $\rho < 1$.

The top of Fig. 5 illustrates an example of a workload intensity surge from traces publicly made available by Google. As the figure illustrates, the surge occurs in the interval between 600 s and 1,500 s, during which time the workload intensity increased by a 4.5 factor: from an average of 0.2 req/sec to 0.9 req/sec. The peak of the surge occurred at time equal to 1,200 s. The middle curve of Fig. 5 shows that the response time increased from its pre-surge value of 10 s to a peak value of 375 s, i.e., a 37.5-fold increase. Additionally, the peak response time caused by the surge occurred at 1,600 s, i.e., 300 s after the peak of the surge occurred.

The bottom part of Fig. 5 shows various curves obtained by using an elasticity controller that employs an analytic model to predict the response time of a multi-server queue under surge conditions (i.e., when $\rho > 1$) [31]. This model establishes a relationship between the maximum desirable response time, the traffic intensity, and parameters that determine the geometry of the surge (the red curve in the

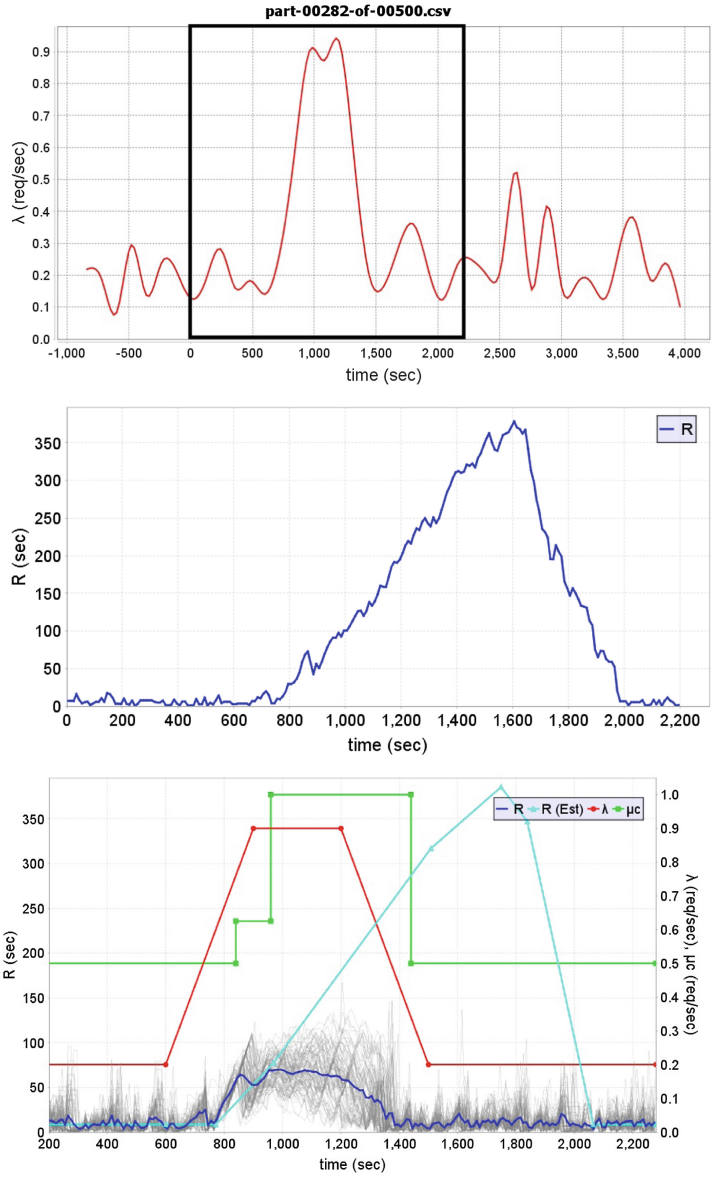

Fig. 5. (a) Top – Example of a trapezoidal workload surge from Google's cluster-usage trace file, part-00282-of-00500.csv; workload surge period: 600–1,500 s; average arrival rate before and after surge: 0.2 req/sec; maximum arrival rate during surge: 0.9 req/sec; **(b) Middle** – System's response time for the duration corresponding to the black highlighted box from the top figure; **(c) Bottom** – Red curve: approximated trapezoidal workload; Green curve: total server capacity; Cyan curve: Estimated response time curve based on the red curve; Blue curve: Response time with the controller averaged over 100 independent runs using the Google trace workload in part (a) above. See [31]. (Color figure online)

bottom figure is a trapezoidal approximation of the surge in the top figure). The cyan curve is a predicted response time curve based on the trapezoidal approximation and is obtained from the analytic model.

The autonomic controller monitors the traffic intensity ρ at regular intervals and detects when it exceeds 1. At this point it uses the analytic model to compute the minimum number of servers needed to bring down the response time. Every time the controller wakes up and notices that $\rho > 1$ it adjusts the number of needed servers. The green step curve in the bottom of Fig. 5 shows that the system capacity increased twice during the surge and that the response time (blue curve at the bottom of Fig. 5) reached at most 50 s instead of 375 s without the controller.

4 Autonomic Intrusion Detection Prevention Systems

As indicated in Sect. 2, the properties of self-managed systems include self-optimizing and *self-protecting*. In this section, we present an example of a work [3] that discusses the design, implementation, and use of an autonomic controller to dynamically adjust the security policies of an Intrusion Detection Prevention System (IDPS).

There are two types of IDPSs: data-centric and syntax-centric. The former type inspects the data coming from a backend database to a client and determines if the security policies of the IDPS allow the requesting user to receive the data. The latter, inspects the syntax of SQL requests and determines if the security policies of the IDPS allow the requesting user to submit that request to the backend database. Because no single IDPS is able to cover all types of attacks, many systems use several data-centric and several syntax-centric IDPSs.

So, an incoming request will have to be processed by several syntax-centric IDPSs of different types and an outgoing response will have to be handled by several different data-centric IDPSs. While this process increases the security of a system, it may severely degrade its performance.

For example, when a system is under a high workload, it might be acceptable to modify the security policies to relax some of the security requirements temporarily to meet increasing demands. Additionally, since in most situations, different system stakeholders view priorities differently, the relaxation in security requirements should ideally be based on predefined stakeholder preferences and risks.

We designed an autonomic controller that dynamically changes the system security policies in a way that maximizes a utility function that is the combination of two utility functions: one for performance and another for security [3]. The former is a function of the predicted response time and the latter is a function of the detection rate and false positive rate. Users are classified into *roles* and security policies are associated with the different roles. A security policy for a *role* r is defined as a vector $\boldsymbol{\rho}_r = (\epsilon_{r,1}, \cdots, \epsilon_{r,i}, \cdots, \epsilon_{r,M})$ where $\epsilon_{r,i} = 0$ if IDPS i ($i = 1, \cdots, M$) is not used for requests of role r and equal to 1 otherwise.

Figure 6 illustrates the results of experiments conducted with the controller in a TPC-W e-commerce site [19]. The x-axis for all graphs is time measured in controller intervals (i.e., the time during which the controller sleeps).

The graph in Fig. 6(a) illustrates the variation of the workload intensity measured in number of requests received by the system over time. As it can be seen, the workload is very bursty and varies widely (between 50 req/sec and 140 req/sec). The high workload peaks cause response time spikes that violate the Service Level Agreements (SLA) of 1 s for access to the home page and 3 s for search requests as illustrated in Fig. 6(b). Figure 6(c) shows three global utility curves. The top curve is obtained when the controller is enabled and shows that the utility is kept at around 0.8 despite the variations in the workload. The middle curve is obtained when the controller is disabled and the security policy is pre-configured and does not change dynamically; in this case the global utility is about 0.6. Finally, the bottom curve is obtained when a full security policy (i.e., one in which all IDPSs are enabled for all roles) is used. In this case, a very low global utility of around 0.48 is observed.

Thus, as Fig. 6 shows, the autonomic controller is able to maintain the global utility at a level 67% higher than when all IDPSs are enabled by reducing the security policies when the workload goes through periods of high intensity.

5 Autonomic Energy-Performance Control

Power consumption at modern data centers is now a significant component of the total cost of ownership. Exact numbers are difficult to obtain because companies such as Google, Microsoft, and Amazon do not reveal exactly how much energy their data centers consume. However, some estimates reveal that Google uses enough energy to continously power 200,000 homes [20].

Most modern CPUs provide Dynamic Voltage and Frequency Scaling (DVFS), which allows the processor to operate at different levels of voltage and clock frequency values. Because a processor's dynamic power is proportional to the product of the square of its voltage by its clock rate, it is possible to control the power consumed by a processor by dynamically varying the clock frequency. However, lower clock frequencies imply in worse performance and higher clock rates improve the processor's performance at the cost of higher power consumption. Therefore, it would be ideal to dynamically vary a processor's clock rate so that as the workload intensity increases, the clock rate is increased to meet response time SLAs. And, as the workload intensity decreases the clock frequency should be decreased to the lowest value that would maintain the desired SLA so as to conserve energy.

Many microprocessors allow for states in which a different voltage-frequency pair is allowed. For example, the Intel Pentium M processor supports the following six voltage-frequency pairs: (1.484 V, 1.6 GHz), (1.420 V, 1.4 GHz), (1.276 V, 1.2 GHz), (1.164 V, 1.0 GHz), (1.036 V, 800 MHz), and (0.956 V, 600 MHz) [14]. As indicated above, microprocessors with DVFS offer a discrete set of voltage-frequency pairs.

We designed and experimented with an autonomic DVFS controller that dynamically adjusts the voltage-frequency pair of the CPU to the lowest value that meets a user-defined response time SLA [20].

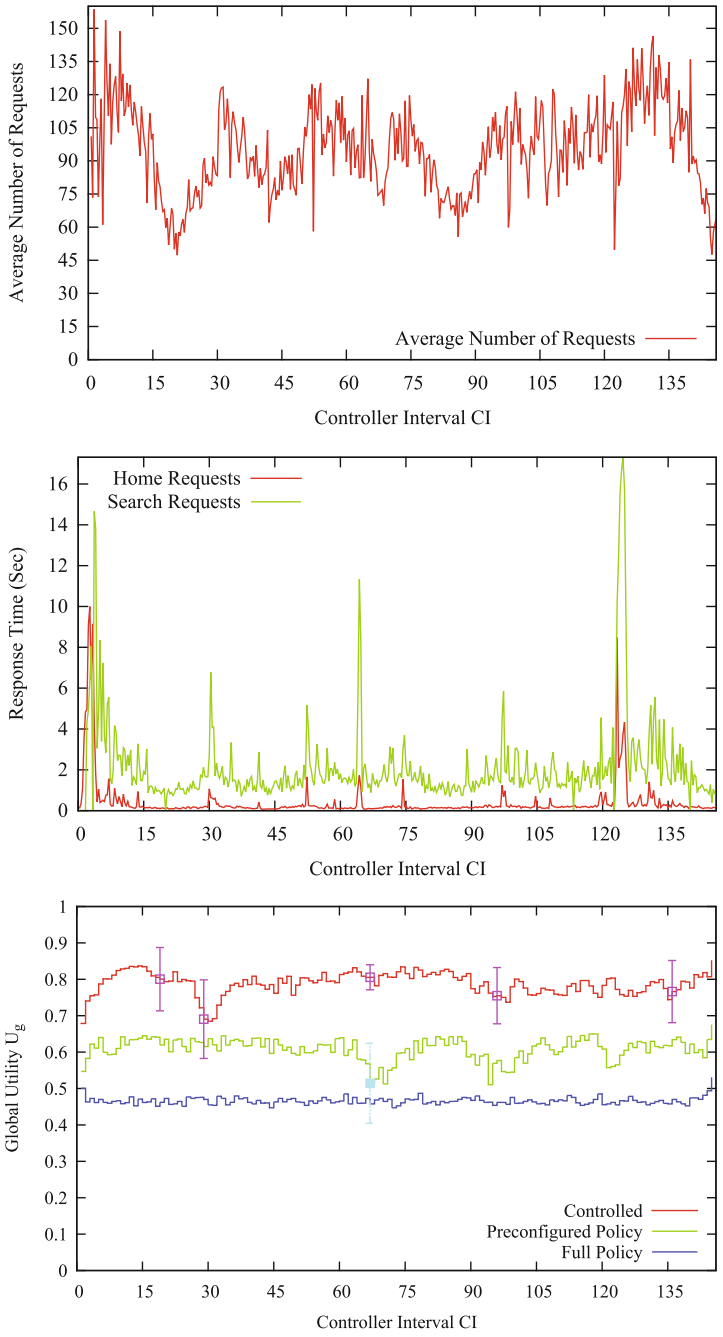

Fig. 6. Experiment results (see [3]): (a) **Top:** Workload variation, (b) **Middle:** Response time for Home and Search page requests without the controller, (c) **Bottom:** Three global utility values: with the controller, for a fixed pre-configured policy, and for a full security policy.

Figure 7 illustrates an example of the variation of the average arrival rate (λ) of requests over time. As it can be seen, the workload intensity varies widely between 0.01 tps and 0.61 tps.

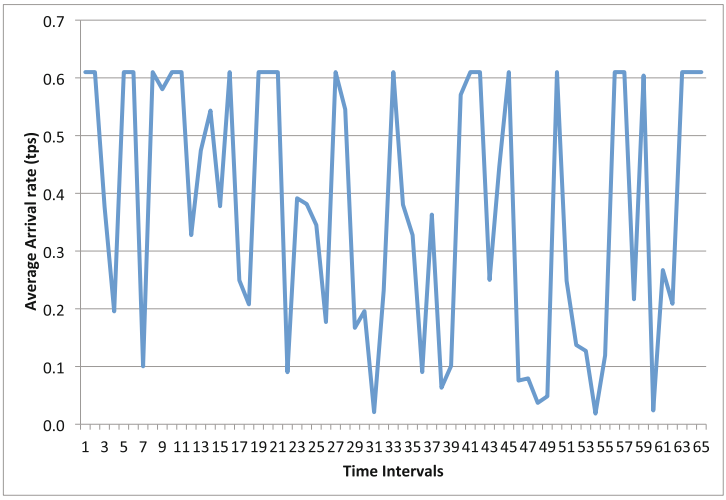

Fig. 7. Average transaction arrival rate (in tps) vs. time intervals. (see [20]).

The DVFS autonomic controller is able to react to these variations as shown in Fig. 8 that shows three different curves. The x-axis follows the same time intervals as in Fig. 7 but the scale on that axis is labelled with the values of λ over the interval. The solid blue curve shows the variation of the *relative power consumption* that results from the variation of the voltage and CPU clock frequencies. We define relative power consumption as the ratio between the power consumed by the processor for a given pair of voltage and frequency values and the lowest power consumed by the processor, which happens when the lowest voltage and frequencies are used.

As can be seen, the shape of the relative power curve follows closely the variation of the workload intensity. Higher workload intensities require higher CPU clock frequencies and voltage levels and therefore higher relative power consumption. The dashed curve of Fig. 8 shows the variation of the average response time over time. The first observation is that the average response time never exceeds its SLA of 4 s. The response time, given that the I/O service demand is fixed throughout the experiment, is a function of the arrival rate λ and the CPU clock frequency during the time interval. This curve and the dotted line (i.e., the CPU residence time) in the same figure clearly show how the autonomic DVFS controller does its job.

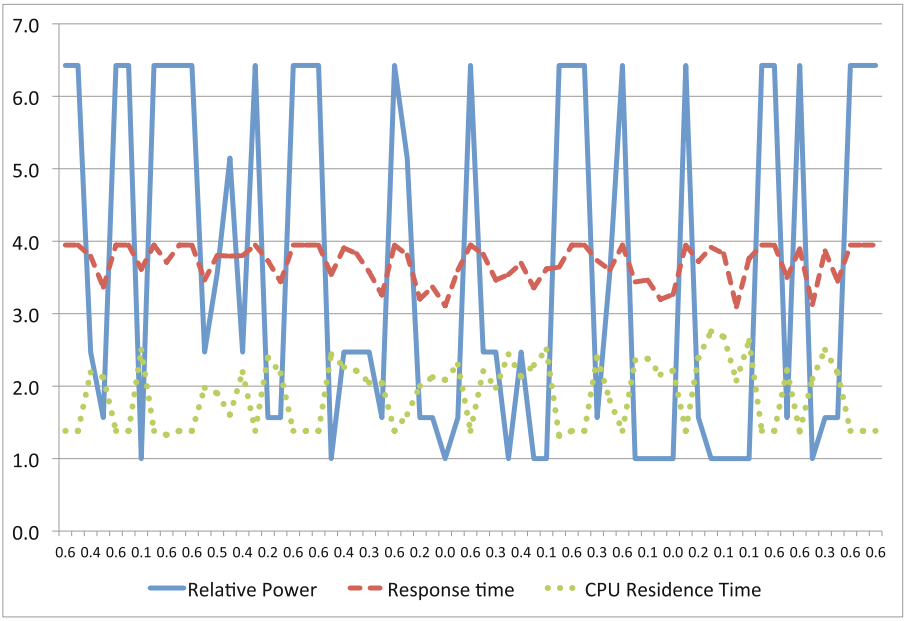

Fig. 8. Solid blue line: Relative power vs. time intervals; **Dashed red line:** Average response time (in sec) vs. time intervals; **Dotted green line:** CPU residence time (in sec) vs. time intervals; Time intervals are labelled with their arrival rate values (in tps). See [20]. (Color figure online)

Other work on autonomic power-performance control can be found in [4,13, 16,26,33,34].

6 Other Examples of Self-managed Systems

Self-managed systems have been used in a wide variety of systems in addition to the examples discussed above. This section provides several additional examples.

6.1 Autonomic Fog Computing

In [28,29], the authors discuss a controller for fog/cloud computing environments that dynamically determines the portion of a transaction that should be processed at a fog server vs. at a cloud server. The controller deals with tradeoffs between local processing (less wide area network time but higher local congestion) and remote processing (more wide area network traffic but use of more powerful servers and therefore less remote congestion). The controller was validated with data obtained from several IoT traces [30].

6.2 Autonomic Resource Allocation in Cloud Computing

The authors in [2], discuss the design and evaluation of an autonomic controller that dynamically allocates and re-allocates communicating virtual machines (VM) in a hierarchical cloud datacenter. Communication latency varies if VMs are colocated in the same server, same rack, same cluster, or same datacenter. The controller employs user-specified information about communication strength among requested VMs in order to determine a near-optimal allocation. Another approach was described in [32] in which the authors present a novel, autonomic, Adaptive Bin Packing (ABP) algorithm for autonomic resource allocation in the cloud.

6.3 Autonomic Checkpointing

The authors in [5] show how one can dynamically control the checkpointing frequency of processes in a distributed system so as to balance execution time and availability tradeoffs. The more often a checkpoint is taken the less time a process is available for useful computation. On the other hand, less frequent checkpoints incur in more work to be redone in case of failures, which extends process execution time.

6.4 Autonomic Moving Target Defenses

The work in [8] presents analytic models of Moving Target Defense (MTD) systems with reconfiguration limits. MTDs are security mechanisms that periodically reconfigure a system's resources to reduce the time an attacker has to learn about a system's characteristics. When the reconfiguration rate is high, the system security is improved at the expense of reduced performance and lower availability [37]. To control availability and performance, one can vary the maximum number of resources that can be in the process of being reconfigured simultaneously. The authors of [8] developed a controller that dynamically varies the maximum number of resources being reconfigured and the reconfiguration rate in order to maximize a utility function of performance, availability, and security.

6.5 Autonomic Distributed Software Systems

The **D**istributed **A**daptation and **RE**covery (DARE) framework designed at Mason [1] uses a distributed MAPE-K loop to dynamically adapt large decentralized software systems in the presence of failures. The **S**elf-**A**rchitecting **S**ervice-Oriented **S**oftware S**Y**stem (SASSY) project [18], also developed at Mason, allows for the architecture of an SOA system to be automatically derived from a visual-activity based specification of the application. The resulting architecture maximizes a user-specified utility function of execution time, availability, and security. Additionally, run-time re-architecting takes place automatically when services fail or the performance of existing services degrades. Other examples of autonomic software adaptation can be found in [9,25,35,36,38].

6.6 Autonomic Smart Manufacturing

In [23] the authors describe how autonomic computing can be used to dynamically control the throughput and energy consumption of smart manufacturing processes. They use a queuing network model of the manufacturing system to predict the throughput and energy consumption of a car production system.

6.7 Autonomic Multi-tiered Web Sites

The authors in [10] presented the detailed design of an autonomic load balancer (LB) for multi-tiered Web sites. They assumed that customers can be categorized into distinct classes (gold, silver, and bronze) according to their business value to the site. The autonomic LB is able to dynamically change its request redirection policy as well as its resource allocation policy, which determines the allocation of servers to server clusters, in a way that maximizes a business-oriented utility function.

6.8 Autonomic Datacenters

In [6,7], the authors presented a self-managed method to assign applications to servers of a data center. As the workload intensity of the applications varies over time, the number of servers allocated to them is dynamically changed by an autonomic controller in order to maximize a utility function of the application's response time and throughput.

7 Concluding Remarks

Most modern information systems are very complex due to their scale and resource heterogeneity, consist of layered software architectures, are subject to variable and hard-to-predict workloads, and use services that may fail and have their performance degraded at run-time. Thus, complex information systems typically operate in ways not foreseen at design time.

Additionally, these software systems have a large number of configuration parameters. A few examples of parameters include: web server (e.g., HTTP keep alive, connection timeout, logging location, resource indexing, maximum size of the thread pool), application server (e.g., accept count, minimum and maximum number of threads), database server (e.g., fill factor, maximum number of worker threads, minimum amount of memory per query, working set size, number of user connections), TCP (e.g., timeout, maximum receiver window size, maximum segment size).

Some parameters have a discrete set of values (e.g., maximum number of worker threads, number of user connections) and others can have any real value within a given interval (e.g., TCP timeout, DB page fill factor). In the latter case, the parameter values in a continuous range have to be discretized in order for combinatorial search methods to be used. The authors in [27] discussed a method for

evaluating the impact of software configuration parameters on a system's performance.

As discussed in this paper, it is next to impossible for human beings to continuously track the changes in the environment in which a system operates in order to make a timely determination of the best set of configuration parameters necessary to move the system to an operating point that meets user expectations. For that reason, complex systems have to be self-managed.

A useful framework to reason about self-managed systems is the **M**onitor **A**nalyze **P**lan and **E**xecute based on **K**nowledge (MAPE-K) loop described by IBM [15]. This loop is continuously executed by an autonomic controller that (1) monitors a managed system's inputs and outputs, (2) analyzes if the outputs have violated user-defined Service Level Objectives usually specified in the form of utility function, (3) plans adaptation actions that optimize the utility function, and (4) executes the plan. All four steps are based on a knowledge repository of models that can be used to predict future system states and corresponding values of the utility function based on a set of actions to be taken by the controller.

Many adaptive systems determine how they should evolve based on their recent past history. However, as indicated in [39], one can take a proactive adaptation approach that uses adaptive forecasting methods in order to anticipate future states of a system and determine the best actions based on predicted future states.

Acknowledgements. I would like to thank my former Ph.D. students whose work is referenced here: Arwa Aldhalaan, Firas Alomari, Noor Bajunaid, Mohamed Bennani, Warren Connell, John Ewing, Mohan Krishnamoorthy, Uma Tadakamlla, and Venkat Tadakamalla.

References

1. Albassam, E., Porter, J., Gomaa, H., Menascé, D.A.: DARE: a distributed adaptation and failure recovery framework for software systems. In: 2017 IEEE International Conference on Autonomic Computing (ICAC), pp. 203–208 (2017)
2. Aldhalaan, A., Menascé, D.A.: Autonomic allocation of communicating virtual machines in hierarchical cloud data centers. In: 2014 International Conference on Cloud and Autonomic Computing, pp. 161–171 (2014)
3. Alomari, F.B., Menascé, D.A.: Self-protecting and self-optimizing database systems: implementation and experimental evaluation. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference, CAC 2013, pp. 18:1–18:10, New York, NY, USA. ACM (2013)
4. Arnaboldi, M., Brondolin, R., Santambrogio, M.D.: Hyppo: hybrid performance-aware power-capping orchestrator. In: 2018 IEEE International Conference on Autonomic Computing (ICAC), pp. 71–80 (2018)
5. Bajunaid, N., Menascé, D.A.: Efficient modeling and optimizing of checkpointing in concurrent component-based software systems. J. Syst. Softw. **139**, 1–13 (2018)
6. Bennani, M., Menascé, D.: Resource allocation for autonomic data centers using analytic performance models. In: Proceedings of International Conference on Automatic Computing, ICAC 2005, pp. 229–240, Washington, DC, USA. IEEE Computer Society (2005)

7. Bennani, M.N., Menasce, D.A.: Assessing the robustness of self-managing computer systems under highly variable workloads. In: International Conference on Autonomic Computing, 2004, Proceedings, pp. 62–69 (2004)
8. Connell, W., Menasce, D.A., Albanese, M.: Performance modeling of moving target defenses with reconfiguration limits. IEEE Trans. Dependable Secure Comput. (2018). https://doi.org/10.1109/TDSC.2018.2882825
9. Esfahani, N., Yuan, E., Canavera, K.R., Malek, S.: Inferring software component interaction dependencies for adaptation support. ACM Trans. Auton. Adapt. Syst. **10**(4), 26:1–26:32 (2016)
10. Ewing, J., Menascé, D.A.: Business-oriented autonomic load balancing for multitiered web sites. In: Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS. IEEE (2009)
11. Ewing, J.M., Menascé, D.A.: A meta-controller method for improving run-time self-architecting in SOA systems. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE 2014, pp. 173–184. ACM, New York (2014)
12. Horn, G., Rozanska, M.: Affine scalarization of two-dimensional utility using the Pareto front. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 147–156 (2019)
13. Imes, C., Zhang, H., Zhao, K., Hoffmann, H.: Copper: soft real-time application performance using hardware power capping. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 31–41 (2019)
14. Intel. Enhanced Intel speedstep technology for the Intel Pentium M processor (2004)
15. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
16. Krzywda, J., Ali-Eldin, A., Wadbro, E., Ostberg, P., Elmroth, E.: Alpaca: aaplication performance aware server power capping. In: 2018 IEEE International Conference on Autonomic Computing (ICAC), pp. 41–50 (2018)
17. Menascé, D.: Security performance. IEEE Internet Comput. **7**(3), 84–87 (2003)
18. Menascé, D., Gomaa, H., Malek, S., Sousa, J.: SASSY: a framework for self-architecting service-oriented systems. IEEE Softw. **28**, 78–85 (2011)
19. Menasce, D.A.: TPC-W: a benchmark for e-commerce. IEEE Internet Comput. **6**(3), 83–87 (2002)
20. Menascé, D.A.: Modeling the tradeoffs between system performance and CPU power consumption. In: Proceedings of the International Conference on Computer Measurement Group, CMG (2015)
21. Menascé, D.A.: Taming complexity with self-managed systems. In: 21st International Conference on Enterprise Information Systems (ICEIS), vol. 1, pp. 5–13 (2019)
22. Menascé, D.A., Almeida, V.A.F., Dowdy, L.W.: Performance by Design: Computer Capacity Planning by Example. Prentice Hall, Upper Saddle River (2004)
23. Menascé, D.A., Krishnamoorthy, M., Brodsky, A.: Autonomic smart manufacturing. J. Decis. Syst. **24**(2), 206–224 (2015)
24. Miettinen, K.: Nonlinear Multiobjective Optimization. Springer, New York (1999). https://doi.org/10.1007/978-1-4615-5563-6
25. Pfannemueller, M., Krupitzer, C., Weckesser, M., Becker, C.: A dynamic software product line approach for adaptation planning in autonomic computing systems. In: 2017 IEEE International Conference on Autonomic Computing (ICAC), pp. 247–254 (2017)

26. Schmitt, N., Iffländer, L., Bauer, A., Kounev, S.: Online power consumption estimation for functions in cloud applications. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 63–72 (2019)
27. Sopitkamol, M., Menascé, D.A.: A method for evaluating the impact of software configuration parameters on e-commerce sites. In: Proceedings of the 5th International Workshop on Software and Performance, WOSP 2005, pp. 53–64. ACM, New York (2005)
28. Tadakamalla, U., Menascé, D.: FogQN: an analytic model for fog/cloud computing. In: Proceedings of the 1st Workshop on Managed Fog-to-Cloud (mF2C), pp. 307–313 (2018)
29. Tadakamalla, U., Menascé, D.: Autonomic resource management using analytic models for fog/cloud computing. In: IEEE International Conference on Fog Computing (ICFC 2019), pp. 69–79 (2019a)
30. Tadakamalla, U., Menascé, D.: Characterization of IoT Workloads, pp. 1–15 (2019b)
31. Tadakamalla, V., Menascé, D.A.: Model-driven elasticity control for multi-server queues under traffic surges in cloud environments. In: 2018 International Conference on Autonomic Computing (ICAC), pp. 157–162. IEEE (2018)
32. Tantawi, A.N., Steinder, M.: Autonomic cloud placement of mixed workload: an adaptive bin packing algorithm. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 187–193 (2019)
33. Tesfatsion, S.K., Wadbro, E., Tordsson, J.: Perfgreen: performance and energy aware resource provisioning for heterogeneous clouds. In: 2018 IEEE International Conference on. Autonomic Computing (ICAC), pp. 81–90 (2018)
34. von Kistowski, J., Deffner, M., Kounev, S.: Run-time prediction of power consumption for component deployments. In: 2018 IEEE International Conference on Autonomic Computing (ICAC), pp. 151–156 (2018)
35. Weyns, D., Malek, S., Andersson, J.: Forms: unifying reference model for formal specification of distributed self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **7**(1), 8:1–8:61 (2012)
36. Yuan, E., Esfahani, N., Malek, S.: A systematic survey of self-protecting software systems. ACM Trans. Auton. Adapt. Syst. **8**(4), 17:1–17:41 (2014)
37. Zangeneh, V., Shajari, M.: A cost-sensitive move selection strategy for moving target defense. Comput. Secur. **75**, 72–91 (2018)
38. Zoghi, P., Shtern, M., Litoiu, M., Ghanbari, H.: Designing adaptive applications deployed on cloud environments. ACM Trans. Auton. Adapt. Syst. **10**(4), 25:1–25:26 (2016)
39. Zuefle, M., Bauer, A., Lesch, V., Krupitzer, C., Herbst, N., Kounev, S., Curtef, V.: Autonomic forecasting method selection: examination and ways ahead. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 167–176 (2019)