

Taming Complexity with Self-managed Systems

Daniel A. Menascé^a

Department of Computer Science, George Mason University, Fairfax, VA, U.S.A.

Keywords: Autonomic computing, self-managed Systems, Utility Functions.

Abstract: Modern computer information systems are highly complex, networked, have numerous configuration knobs, and operate in environments that are highly dynamic and evolving. Therefore, one cannot expect that configurations established at design-time will meet QoS and other non-functional goals at run-time. For that reason, the design of complex systems needs to incorporate controllers for adapting the system at run time. This paper describes the four properties of self-managed systems: self-configuring, self-optimizing, self-healing, and self-protecting. It also describes through concrete examples how these properties are enforced by controllers I designed for a variety of domains including cloud computing, fog/cloud computing, Internet datacenters, distributed software systems, and secure database systems.

1 INTRODUCTION

Modern computer systems are networked, are 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. Additionally, these systems have stringent Quality of Service (QoS) requirements in terms of response time, throughput, availability, energy consumption, and security. These systems have a very large number of configuration settings that significantly impact their QoS behavior.

This complexity is compounded by the fact that the workload intensity of these complex systems varies in rapid and hard-to-predict ways.

For these reasons, it is virtually impossible for human beings to change the configuration settings of a complex system in near real-time in order to steer the system to an optimal operating point that meets user-established 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 (Kephart and Chess, 2003). Autonomic computing systems are also referred to as *self-managed* systems.


The rest of this paper is organized as follows. Section 2 describes the basics of self-managed systems. 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, Section 7 provides some concluding remarks.

2 BASICS OF SELF-MANAGED SYSTEMS

This section discusses the basics of self-managed systems aka *autonomic computing* systems, a term coined by IBM (Kephart and Chess, 2003) more than a decade ago. 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.

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*

^a  <https://orcid.org/0000-0002-4085-6212>

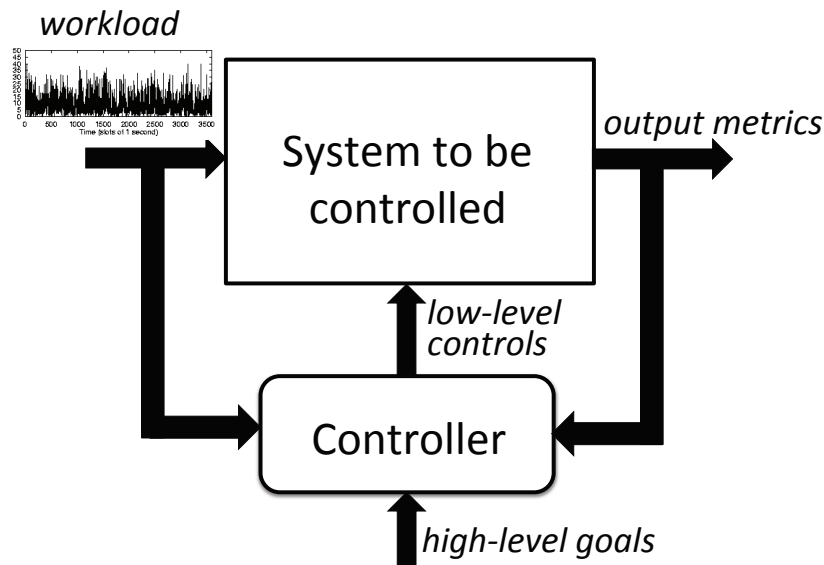


Figure 1: Basic components of a self-managed system.

of the system are associated with the QoS delivered by the system when processing the inputs. Examples of output metrics are: (a) 95% of web requests have a response time less than or equal to 0.8 sec; (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 99.978%; and (d) the percentage of phishing e-mails filtered by the e-mail portal is greater than or equal to 90%.

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. The controller reacts to deviations from the desired QoS levels established by the stakeholders by automatically deriving a *plan* to change the system's configuration by changing 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 occur-

rence 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 (Menascé, 2003). 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 (Miettinen, 1999). 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.

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. 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. 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 examples of utility functions. The lefthand side of the figure shows three different functions with different shape factor (α) but with the same service level goal ($\beta = 65.0$), which is the inflection point of the curve. The righthand side 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* and 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 (Kephart and Chess, 2003).

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

- K : number of configuration knobs (low level controls in Fig. 1) the controller is able to change.
- $\vec{C}(t) = (C_1, \dots, C_K)$: vector of values of the K configuration knobs at time t .
- \mathcal{C} : set of all possible vectors $\vec{C}(t)$.
- $\mathcal{W}(t)$: workload intensity at time t . This is usually the workload intensity in the last controller interval but could also be a predicted workload for the next controller interval.
- $S(t) = (\vec{C}(t), \mathcal{W}(t))$: 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 .
- $x_i(t) \in \mathcal{D}_i$: value of metric i ($i = 1, \dots, m$) at time t .

- $g_i(S(t))$: function used to compute (i.e., estimate) the value of metric i at time t . So, $x_i(t) = g_i(S(t)) = g_i(\vec{C}(t), \mathcal{W}(t))$. The function $g_i(\cdot)$ represents a model of the system being controlled. In virtually all cases of interest, the functions $g_i(\cdot)$ 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 .
- $U_g(x_1, \dots, x_m) = f(U_1(x_1), \dots, U_m(x_m))$: global utility function, which is a function of all individual utility functions.

The functions $g_1(\cdot), \dots, g_m(\cdot)$ are typically analytic models used to estimate the values of each of the m metrics as a function of the current or future system state $S(t)$. The functions $U_i(\cdot)$, $i = 1, \dots, m$ and $U_g(\cdot)$ 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 will wake up again and possibly make another selection of parameters.

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

$$\vec{C}^*(t) = \underset{\vec{C}(t) \in \mathcal{C}}{\operatorname{argmax}} \{f(U_1(g_1(\vec{C}(t), \mathcal{W}(t))), \dots, U_m(g_m(\vec{C}(t), \mathcal{W}(t))))\}$$

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

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} .

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 (Ewing and Menascé, 2014).

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),

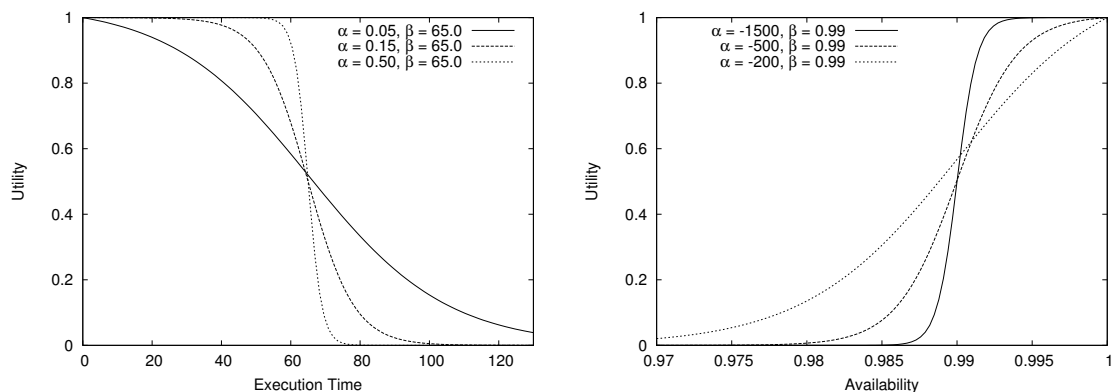


Figure 2: Left side: examples of utility functions for execution time. Right side: examples of utility functions for availability. All examples are sigmoid functions.

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. A queuing system is in steady-state when $\rho < 1$.

The top of Fig. 3 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 sec and 1,500 sec, during which time the workload intensity increased by a 4.5 factor: from an average of 0.2 requests/sec to 0.9 requests/sec. The peak of the surge occurred at time equal to 1,200 sec. The middle curve of Fig. 3 shows that the response time increased from its pre-surge value of 10 sec to a peak value of 375 sec, i.e., a 37.5-fold increase. Additionally, the peak response time caused by the surge occurred at 1,600 sec, i.e., 300 sec after the peak of the surge occurred.

The bottom part of Fig. 3 shows various curves obtained by using an elasticity controller that uses an analytic model used to predict the response time of a multi-server queue under surge conditions (i.e., when $\rho > 1$) (Tadakamalla and Menascé, 2018). 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 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 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. 3 shows that the system capacity increased twice during the surge and that the response time (see blue curve at the bottom of Fig. 3) reached at most 50 sec instead of 375 sec without the controller.

4 AUTONOMIC INTRUSION DETECTION PREVENTION SYSTEMS

As indicated in Section 2, the properties of self-managed systems include self-optimizing and *self-protecting*. In this section, we present an example of a work (Alomari and Menascé, 2013) 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. Because no IDPS is able to cover all types of possible 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

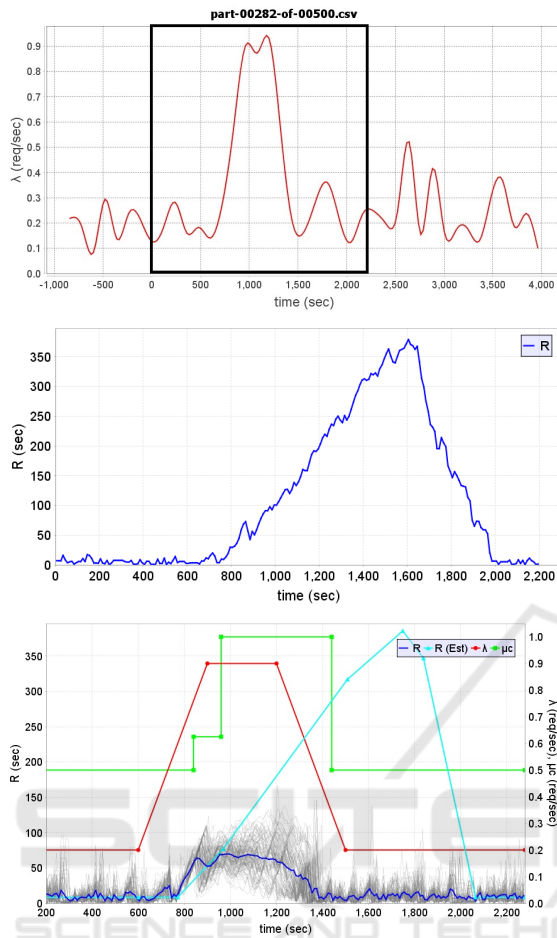


Figure 3: **(a) Top** – Example of a trapezoidal workload surge from Google’s cluster-usage trace file, part-00282-of-00500.csv; workload surge period: 600-1,500 sec; average arrival rate before and after surge: 0.2 requests/sec; maximum arrival rate during surge: 0.9 requests/sec; **(b) Middle** – System’s 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 (Tadakamalla and Menascé, 2018).

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 pre-defined 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 (Alomari and Menascé, 2013).

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 $\vec{p}_r = (\epsilon_{r,1}, \dots, \epsilon_{r,M})$ where $\epsilon_{r,i} = 0$ if IDPS i is not used for requests of role r and equal to 1 otherwise.

Figure 4 illustrates the results of experiments conducted with the controller in a TPC-W e-commerce site. 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. 4 (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 second for access to the home page and 3 seconds for search requests as illustrated in Fig. 4 (b). Figure 4 (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. 4 shows, the autonomic controller is able to maintain the global utility at a level 67% higher than when the 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 continuously power 200,000 homes (Menascé, 2015).

Most modern CPUs provide Dynamic Voltage and Frequency Scaling (DVFS), which allows the pro-

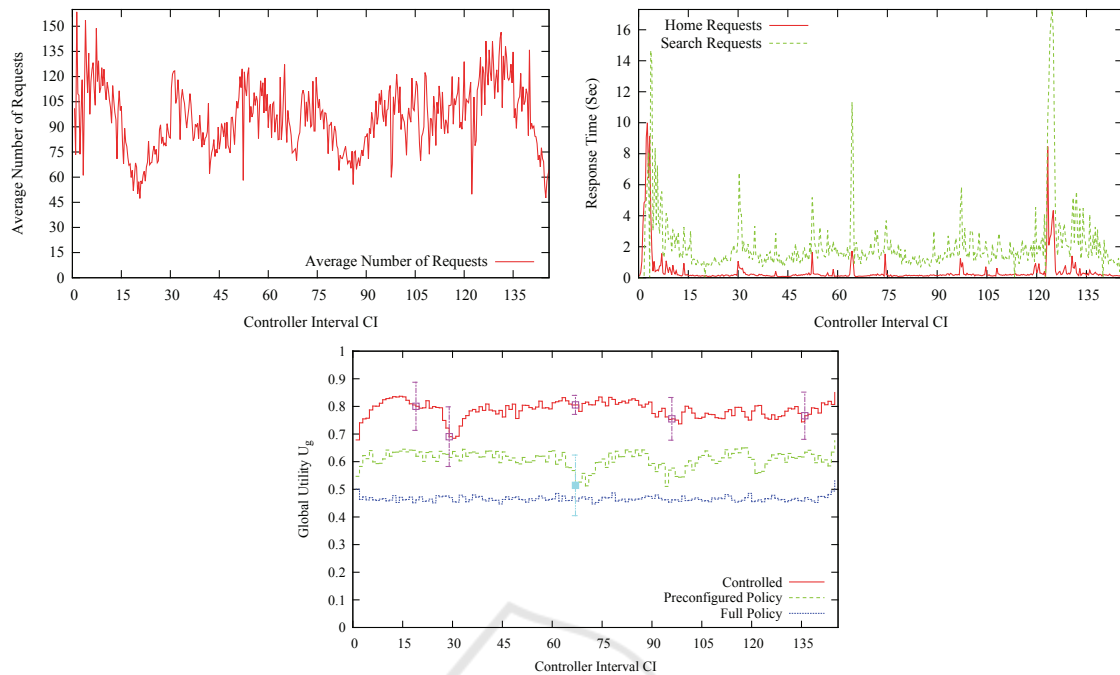


Figure 4: Experiment results (see (Alomari and Menascé, 2013)): (a) (top-left) Workload variation, (b) (top right) Response time for Home and Search page requests without the controller, (c) (bottom) Three global utility values: with the controller, for a fixed pre-configured policy, and for a full security policy.

cessor 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. 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) (Intel, 2004). 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 (Menascé,

2015).

Figure 5 illustrates an example of the variation of the average arrival rate (λ) over time. As it can be seen, the workload intensity varies widely between 0.01 tps and 0.61 tps.

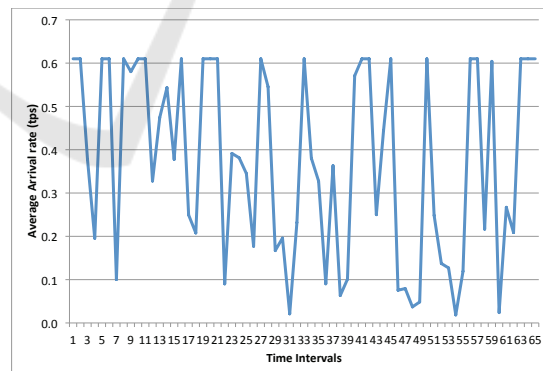


Figure 5: Average Transaction Arrival Rate (in tps) vs. Time Intervals.

The DVFS autonomic controller is able to react to these variations as shown in Fig. 6 that shows three different curves. The x-axis follows the same time intervals as in Fig. 5 but the scale on that axis is labelled with the values of λ over the interval. The solid curve shows the variation of the *relative power consumption* that results from the variation of the voltage and CPU clock frequencies. We define the relative power con-

sumption 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. 6 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 sec. 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.

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. In (Tadakamalla and Menascé, 2019), the authors discuss an autonomic controller 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 authors in (Bajunaid and Menascé, 2018) 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 work in (Connell et al., 2018) 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. 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 (Connell et al., 2018) 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.

The **D**istributed **A**daptation and **R**Ecovery (DARE) framework designed at Mason (Albassam et al., 2017) 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-**O**riented **S**oftware **S**ystem (SASSY) project (Menascé et al., 2011), 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.

In (Menascé et al., 2015) the authors describe how autonomic computing can be used to dynamically control the throughput and energy consumption of smart manufacturing processes.

The authors in (Aldhalaan and Menascé, 2014), 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.

The authors in (Ewing and Menascé, 2009) 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.

In (Bennani and Menasce, 2005), 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.

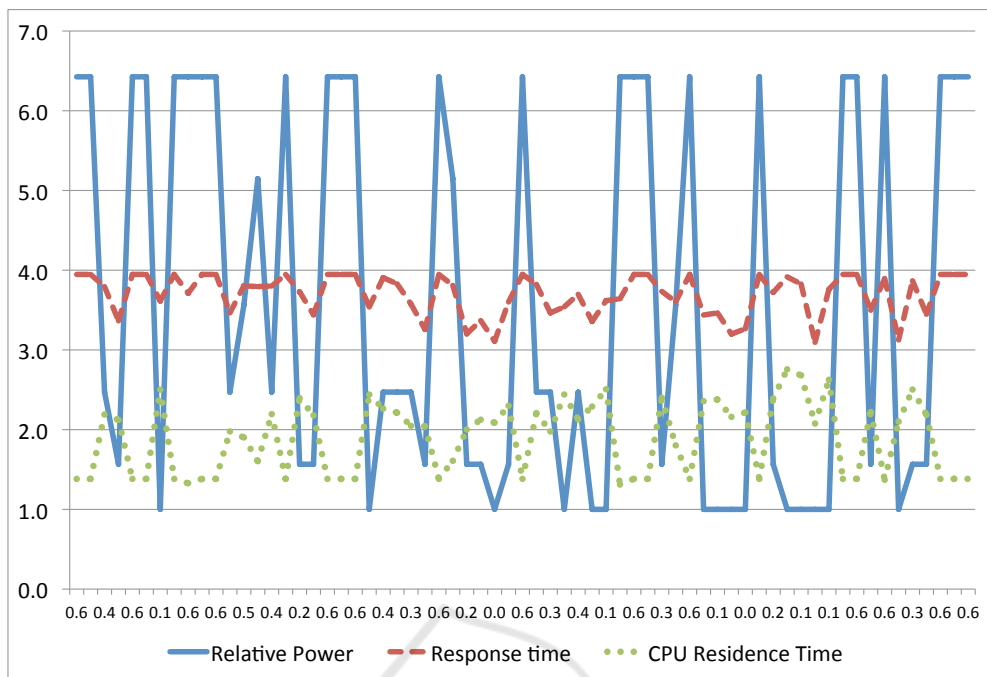


Figure 6: Solid line: Relative Power vs. Time Intervals; Dashed Line: Average Response Time (in sec) vs. Time Intervals; Dotted Line: CPU Residence Time (in sec) vs. Time Intervals; Time Intervals are labelled with their arrival rates (in tps).

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). The authors in (Sopitkamol and Menascé, 2005) 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.

REFERENCES

- Albassam, E., Porter, J., Goma, H., and Menascé, D. A. (2017). DARE: A distributed adaptation and failure recovery framework for software systems. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 203–208.
- Aldhalaan, A. and Menascé, D. A. (2014). Autonomic allocation of communicating virtual machines in hierarchical cloud data centers. In *2014 Intl. Conf. Cloud and Autonomic Computing*, pages 161–171.
- Alomari, F. B. and Menascé, D. A. (2013). Self-protecting and self-optimizing database systems: Implementation and experimental evaluation. In *Proc. 2013 ACM Cloud and Autonomic Computing Conference, CAC '13*, pages 18:1–18:10, New York, NY, USA. ACM.

- Bajunaid, N. and Menascé, D. A. (2018). Efficient modeling and optimizing of checkpointing in concurrent component-based software systems. *Journal of Systems and Software*, 139:1 – 13.
- Bennani, M. and Menasce, D. (2005). Resource allocation for autonomic data centers using analytic performance models. In *Proc. Intl. Conf. Automatic Computing, ICAC '05*, pages 229–240, Washington, DC, USA. IEEE Computer Society.
- Connell, W., Menascé, D. A., and Albanese, M. (2018). Performance modeling of moving target defenses with re-configuration limits. *IEEE Tr. Dependable and Secure Computing*, page 14.
- Ewing, J. and Menascé, D. A. (2009). Business-oriented autonomic load balancing for multitiered web sites. In *Proc. Intl. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MAS-COTS*. IEEE.
- Ewing, J. M. and Menascé, D. A. (2014). A meta-controller method for improving run-time self-architecting in SOA systems. In *Proc. 5th ACM/SPEC Intl. Conf. Performance Engineering, ICPE '14*, pages 173–184, New York, NY, USA. ACM.
- Intel (2004). Enhanced Intel speedstep technology for the Intel Pentium M processor.
- Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, 36(1):41–50.
- Menascé, D. (2003). Security performance. *IEEE Internet Computing*, 7(3):84–87.
- Menascé, D., Goma, H., Malek, S., and Sousa, J. (2011). SASSY: A framework for self-architecting service-oriented systems. *IEEE Software*, 28:78–85.
- Menascé, D. A. (2015). Modeling the tradeoffs between system performance and CPU power consumption. In *Proc. Intl. Conf. Computer Measurement Group. CMG*.
- Menascé, D. A., Krishnamoorthy, M., and Brodsky, A. (2015). Autonomic smart manufacturing. *J. Decision Systems*, 24(2):206–224.
- Miettinen, K. (1999). *Nonlinear Multiobjective Optimization*. Springer.
- Sopitkamol, M. and Menascé, D. A. (2005). 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 '05*, pages 53–64, New York, NY, USA. ACM.
- Tadakamalla, U. and Menascé, D. A. (2019). Autonomic resource management using analytic models for fog/cloud computing. In *Proc. IEEE Intl. Conf. Fog Computing*. IEEE.
- Tadakamalla, V. and Menascé, D. A. (2018). Model-driven elasticity control for multi-server queues under traffic surges in cloud environments. In *2018 Intl. Conf. Autonomic Computing (ICAC)*, pages 157–162. IEEE.