Chapter 14
Metrics and Benchmarks for Self-Aware Computing Systems

Nikolas Herbst, Steffen Becker, Samuel Kounev, Heiko Koziolek, Martina Maggio, Aleksandar Milenkoski and Evgenia Smirni

Abstract In this chapter, we propose a list of metrics grouped by the MAPE-K paradigm for quantifying properties of self-aware computing systems. This set of metrics can be seen as a starting point towards benchmarking and comparing self-aware computing systems on a level-playing field. We discuss state-of-the art approaches in the related fields of self-adaptation and self-protection to identify commonalities in metrics for self-aware computing. We illustrate the need for benchmarking self-aware computing systems is with the help of an approach that uncovers real-time characteristics of operating systems. Gained insights of this approach can be seen as a way of enhancing self-awareness by a measurement methodology on an ongoing basis. At the end of the chapter, we address new challenges in reference workload definition for benchmarking self-aware computing systems, namely load intensity patterns and burstiness modeling.

Nikolas Herbst
University of Würzburg, Germany, e-mail: nikolas.herbst@uni-wuerzburg.de

Steffen Becker
Technical University Chemnitz, Germany,
e-mail: steffen.becker@informatik.tu-chemnitz.de

Samuel Kounev
University of Würzburg, Germany e-mail: samuel.kounev@uni-wuerzburg.de

Heiko Koziolek
ABB Ladenburg, Germany e-mail: heiko.koziolek@de.abb.com

Martina Maggio
Lunds Universitet, Sweden e-mail: martina.maggio@control.lth.se

Aleksandar Milenkoski
University of Würzburg, Germany,
e-mail: aleksandar.milenkoski@uni-wuerzburg.de

Evgenia Smirni
College of William and Mary, Williamsburg, VA, USA e-mail: esmirni@cs.wm.edu

14.1 Introduction

Beyond the need for methodologies to assess self-awareness of a computing system, as discussed in Chapter 15, we see an increasing demand for comparisons of self-aware systems on a level-playing field. Established domains in information technology usually come with a set of key-performance indicators or metrics, standard scenarios and measurement rules that taken together form a benchmark. For example, benchmarks of the Standard Performance Evaluation Corporation (SPEC) like for CPUs, virtualization technology, or enterprise applications enable comparisons and decision making. Our expectation is a growing number of self-aware computing systems that may exhibit similarities in their goals, features and application domains. We see this as the major reason, why benchmarking of self-aware computing systems will become more important in the course of the next years not only to design and improve such systems, but also to reliably compare and select them.

A benchmark usually consists of three major building blocks: The first building block is a set of reliable and intuitive metrics that can be combined to a single-valued score. The second building block is the workload definition as an exact definition of the work that is to be performed by the system under test together with a definition of the load profile over time. The third building block is a well-defined measurement methodology (also known as run rules) that assure repeatable measurements.

As a starting point towards the benchmarking of self-aware computing systems, we propose in Section 14.2 an initial set of metrics grouped by the MAPE-K generic control loop. The MAPE-K control loop consists of (i) monitor, (ii) analyze, (iii) plan and (iv) execute phases with a central knowledge repository and can be seen as a special case of the self-aware learning and reasoning loop (as defined in Chapter 1). In Section 14.3, we review the state-of-the-art in two related fields, self-adaptation and self-protection, identifying commonalities with benchmarking applied to self-aware computing system properties. In Section 14.4, we sketch an approach to illustrate how a benchmark can help to improve and better understand a self-aware computing system. In this case, this is achieved by continuously uncovering real-time characteristics of the underlying operating system. In Section 14.5, we explain how two new challenges in defining reference workloads can be addressed to build a self-aware computing system benchmark. We focus on modeling load profiles and realistic burstiness.

14.2 Metrics for Self-Aware Systems

Existing performance metrics, such as response time or throughput, are not sufficient for benchmarking self-aware computing systems as they do not capture all relevant aspects of self-awareness. Existing benchmarks and metrics often focus on a subset of these aspects (designed, e.g., to evaluate self-adaptive, self-protecting aspects) or use domain-specific refinements of a broader set of metrics that could be suited for self-aware computing systems. In order to make a first step towards closing this gap, we went through a process to identify potential candidates of metrics suited to benchmark self-aware computing systems. Those metrics can then be used to quantitatively evaluate, compare, or analyze self-aware computing systems.

The process we went through was threefold. On the one hand, we identified and adopted existing metrics from systems closely related to self-aware computing systems. In particular, we investigated metrics from self-adaptive computing system evaluation and included them, if suited. Second, we used the MAPE-K reference architecture and used it both as a grouping, as well as inspiration for new metrics. Note that this does not mean in any way that the metrics can only be applied to computing systems implemented in the MAPE-K style. The resulting metrics are meant to be general. Finally, we used the definition of a self-aware computing system given in Chapter 1. We went through all aspects of it and tried to come up with metrics designed to quantify these aspects for any given self-aware computing system.

Goal Fulfillment	Monitor	Analyze	Plan	Execute
Proportion of time the is in a goal fulfillment state	Levels of self-awareness	Number of input sources utilized	Proportion of „correct“ decisions made per time unit	Proportion of time the system in an oscillating state
Duration / amount of goal violations per time	Number of monitored internal and external properties	Sophistication of the learning mechanism	Sophistication of the reasoning processes	Duration of an adaptation action
Severity of goal violations	Granularity / precision of sensing the environment	Accuracy of the learned models w.r.t reality	“Precision and recall” of the selected adaptation actions	Correctness of the adaptation actions
Level of goal fulfillment	Size / length of the historically stored properties	Duration of altering the learning process upon changing goals	Extent / granularity of traceability for reasoning	
	Monitoring frequency			
	Monitoring overhead			
	Completeness of the collected and required information			
	Number of required user inputs per time			

Table 14.1: Metric candidates overview

In the following, we illustrate the resulting list of suggestions for metrics addressing different parts of self-aware computing systems. These suggestions are intended as ideas to influence the development of upcoming benchmarks for self-aware computing systems. Table 14.1 provides an overview of all metrics we identified.

14.2.1 Goal Fulfillment

An initial set of metrics aims at quantifying the extent to which a self-aware computing system is able to fulfill its *goals* over time. The following candidates have been identified:

Proportion of time the system is in a goal fulfillment state: This metric indicates the percentage of the time the system fulfills its goals. The closer this metric is to 100%, the better the system adapts to changing conditions.
Duration / amount of goal violations per time: This metrics capture the overall amount of goal violations over the system's runtime and the durations of time intervals the system spent in a state where at least one goal is violated.
Severity of goal violations: This metrics captures for example 60% video quality in an adapting video codec, or 10000 Euros unnecessarily spent for renting servers.
Level of goal fulfillment: This metrics captures, for example, the time in which an autonomous car manages a given track, or the resource efficiency of a cloud infrastructure. This metric captures in addition to pure goal fulfillment cases in which a goal can be fulfilled to different degrees.

14.2.2 Quality of the Information Collection (Monitor Phase)

The following metrics aim at quantifying the information needs of a self-aware computing system, i.e., the amount of *monitoring* required for executing its self-awareness functionality.
Levels of self-awareness: This metric classifies the computing system's self-awareness level (cf. Chapter 1), however, this might be hard to measure empirically. (awareness of self, internal state; awareness of goals; awareness of self-awareness).
Number of monitored internal and external properties: As an example, a result could be that the computing system uses seven performance counters. The more external properties the computing system needs to track the more complex and time-consuming the monitoring would normally be.
Granularity / precision of sensing the environment: The metric captures the precision used to sense the environment. Higher precisions require better sensors but the measurements need more storage space. For example, a system could use a resolution in the range of milliseconds to capture points in time.
Size / length of the historically stored properties: For an example system, the metric could indicate that the collected monitoring data of the last two years is stored and analyzed. The more data is saved, the better the system can reason about its past. However, also the storage requirements increase.
Monitoring frequency: This metric captures the rate at which measurement and monitoring data is collected from the environment (e.g., measurement values per minute).
Monitoring overhead: The metric quantifies the time spent by the system for performing monitoring tasks in contrast to executing other system functions.
Completeness of the collected and required information: The degree to which the data made available to the system about its environment is complete, or can be used to derive all required information. Especially important for systems that obtain data about their environment from unreliable sensors.

Number of required user inputs per time (user-in-the-loop): How often the system asks the user for environment data or advice on next steps. The lower this metric is, the more autonomously the system acts.

14.2.3 Quality the Learning Process (Analyze Phase)

The following set of metrics aim to quantify the system's ability to *analyze and learn* from its observations.
Number of input sources utilized: The metric evaluates how many of the available information sources are actually used in the learning process. It can be defined as number models or views used divided by the total number.
Sophistication of the learning mechanism: This metric captures the effort the system spends on learning. Usually, more effort leads to better behavior, e.g., duration of the evolutionary algorithm (in order to find better solutions, potentially including time-bounds). This metric would to be specified differently depending on the kind of applied learning algorithm.
Accuracy of the learned models w.r.t reality: This captures how representative the learned models are of the real-life entities/phenomena they abstract. For example, the amount of real world states identified by a hidden Markov model or the degree to which a surface explored by a robot matches the real surface. Note that this metric assumes full knowledge of the modeled entities/phenomena.
Duration of altering the learning process upon changing goals: This metric captures the time the learning algorithm takes until it picks up changed system goals and starts to adapt accordingly.

14.2.4 Quality of the Reasoning Process (Plan Phase)

The following metrics aim to capture the *planning* process of a self-aware computing system, i.e., its ability to propose actions that lead to better goal fulfillment.
Proportion of correct decisions made per time unit: The mean number of objectively correct decisions the system makes per per unit of time. This metric assumes that the software designers can specify the correct behavior for all situations that arise during the benchmark.
Sophistication of the reasoning: This metric quantifies how smart the system is able to reason. Possible values can be the number of evaluated alternatives for adaptations or the number of surprising positive findings.
Precision & recall of the selected adaptation actions: How many different adaptation alternatives are considered? How many of them are reasonable ones? The concept of precision & recall is for example employed for benchmarking self-protecting systems as discussed in Sec. 14.3.2.

Extent / granularity of traceability for reasoning: This metric captures the extent to which a self-aware computing system is able to "explain" its decisions. It can either be binary (is able / is not able to present rationale) or could be defined as degree to which one is able to reproduce the decisions made by the system.

14.2.5 Measure the Adaptation Actions (Execute Phase)

The following metrics quantify the performance of the system in *executing* adaptations.
Proportion of time the system in an oscillating / unstable state: The proportion of time the system spends alternating between two or more states while the environment is in a stable state. This metric is related to a jitter metric that has been proposed in the context of elastic cloud systems and describes the number of missing or superfluous adaptations over time (see Sec. 14.3.1).
Duration of adaptation actions: The time an adaptation action takes, e.g., 5 min to start a new server or 5 sec to turn the robot into another direction. This can approximated by measuring the durations in sub-optimal states as done for elastic cloud systems (see Sec. 14.3.1).
Correctness of the adaptation actions: This metric captures how successful adaptations are. It can be defined as the failure rate of adaptations. This metric also relates to a metric proposed earlier in the context of elastic clouds, namely the average amount of over-/under-provisioned resources (see Sec. 14.3.1).

14.2.6 Summary

The presented metrics are a starting point for developing of new metrics and benchmarks for evaluating the level of self-awareness a system exhibits. They do not provide a formal definition of what to measure and several of them are domain-specific, i.e., they need to be refined for a concrete self-aware computing system. Furthermore, they have not yet been extensively validated to make sure that they can be used to adequately characterize self-awareness aspects. In addition, aggregate metrics need to be defined to summarize the different sub-metrics producing an overall self-awareness score. This is required to allow easy comparisons of different self-aware computing systems in standardized benchmarks. It is still an open question, how to define weights for the various sub-metrics when aggregating them to produce a single value. Finally, while some metrics are easy to determine and already pretty clear in their definition, other metrics represent dynamic properties. For those metrics, we need a methodology how to define a representative workload and set up a representative environment in which we execute the system to be evaluated.

14.3 On the State-of-the-Art in Quantifying Self-Adaptation and Self-Protection

The previous section outlined our proposed metric candidates for characterizing self-aware systems. As indicated, several of these metrics are inspired or adapted from the related fields self-adaptation and self-protection. In this section, we describe the state-of-the-art in these two field in detail. In the context of self-adaptation, elasticity of compute clouds is taken as an example. In the context of self-protection, intrusion detection systems (IDSes) are considered.

14.3.1 Quantifying the Quality of Self-Adaptation

The system property of self-adaptation is seen in many different fields of applications like autonomous robots and more. Self-adaptation in the context of elastic resource provisioning in compute cloud resource provisioning is currently a highly discussed topic in academia and industry, and therefore in the following we use it as an example of a self-adaptation property.

When we talk about elasticity in the context of compute clouds, we refer to the definition given in [17] as follows:

Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources *match* the current demand as closely as possible.

Several metrics for elasticity have been proposed during the last years:
(i) The "scaling latency" metrics in [26] or the "provisioning interval" in [8] capture the time to bring up or drop a resource. This duration is a technical property of an elastic environment independent of the demand changes and the elasticity mechanism itself that decides when to trigger a reconfiguration. Thus, these metrics are insufficient to fully characterize the elasticity of a platform.
(ii) The "reaction time" metric in [24] can only be computed if a unique mapping between resource demand changes and supply changes exists. This assumption does not hold especially for proactive elasticity mechanisms or for mechanisms that have unstable (alternating) states.
(iii) The approaches in [2, 9, 10] characterize elasticity indirectly by analyzing response times for significant workload changes or for SLO compliance. In theory, perfect elasticity would result in constant response times for varying workload intensity. In practice, detailed reasoning about the quality of platform adaptations based on response times alone is hampered due to the lack of relevant information about the platform behavior, e.g., information about the amount of provisioned surplus resources.
(iv) Cost-based metrics are proposed in [11, 21, 41, 42] quantifying the impact of elasticity by comparing the resulting provisioning costs to the costs for a peak-load

static resource assignment or the costs of a hypothetical perfect elastic platform. In both cases, the resulting metrics strongly depend on the underlying cost model, as well as on the assumed penalty for under-provisioning, and thus they do not support fair cross-platform comparisons.

We select a set of metrics for two core aspects of elasticity *accuracy* and *timing*. For these metrics, the optimal value is zero corresponding to a perfectly elastic platform. The following assumptions must hold in order to be able to apply the selected elasticity metrics to compare a set of different platforms: the existence of an autonomic adaptation process, the scaling of the same resource type, e.g., CPU cores or virtual machines (VMs), and that the respective resource type is scalable within the same ranges, e.g., from 1 to 20 resource units.

The set of metrics evaluates the resulting elastic behavior as observed from the outside and are thus designed in a manner independent of distinct descriptions of the underlying hardware, the virtualization technology, the used cloud management software, or the employed elasticity strategy and its configuration. As a consequence, the metrics and the measurement methodology are applicable in situations where not all influencing factors are known. All metrics require two discrete curves as input: the demand curve, which defines how the resource demand varies during the measurement period, and the supply curve, which defines how the actual amount of resources allocated by the platform varies.

In the following, we describe the metrics for quantifying the *accuracy* aspect and a set of metrics for quantifying the *timing* aspect as proposed in [18]

14.3.1.1 Accuracy

The under-provisioning accuracy metric $accuracy_U$, is calculated as the sum of areas between the two curves ($\sum U$) where the resource demand exceeds the supply normalized by the duration of the measurement period T, as visualized in Figure 14.1. Similarly, the over-provisioning accuracy metric $accuracy_O$ is based on the sum of areas ($\sum O$) where the resource supply exceeds the demand.

$$\text{Under-provisioning: } accuracy_U\ [resource\ units] = \frac{\sum U}{T}$$

$$\text{Over-provisioning: } accuracy_O\ [resource\ units] = \frac{\sum O}{T}$$

Thus, $accuracy_U$ and $accuracy_O$ measure the average amount of resources that are under-/over-provisioned during the measurement period T. Since under-provisioning results in violating SLOs, a customer might want to use a platform that does not tend to under-provision at all. Thus, the challenge for providers is to ensure that enough resources are provided at any point in time, but at the same time distinguish themselves from competitors by minimizing the amount of over-

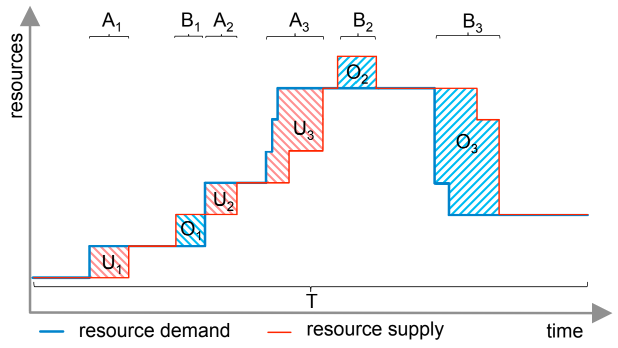


Fig. 14.1: Illustrating example for accuracy (U, O) and timing (A, B) metrics.

provisioned resources. Considering this, the defined separate accuracy metrics for over-provisioning and under-provisioning allow providers to better communicate their elasticity capabilities and customers to select the provider that best matches their needs.

14.3.1.2 Timing

We characterize the *timing* aspect of elasticity from the viewpoint of the pure *provisioning timeshare*, on the one hand, and from the viewpoint of the induced *jitter* accounting for superfluous or missed adaptations, on the other hand.

Provisioning Timeshare

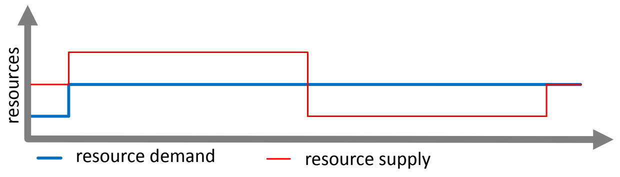
The two accuracy metrics allow no reasoning as to whether the average amount of under-/over-provisioned resources results from a few big deviations between demand and supply or if it is rather caused by a constant small deviation. To address this, the following two metrics are designed to provide insights about the ratio of time in which under- or over-provisioning occurs.

As visualized in Figure 14.1, the following metrics $timeshare_U$ and $timeshare_O$ are computed by summing up the total amount of time spent in an under- $(\sum A)$ or over-provisioned $(\sum B)$ state normalized by the duration of the measurement period. Thus, they measure the overall timeshare spent in under- or over-provisioned states:

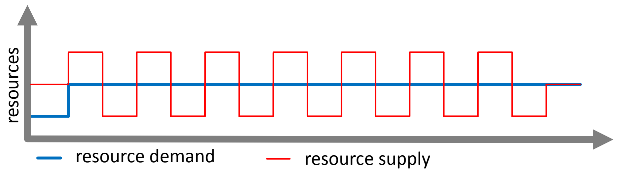
$$\text{Under-provisioning: } timeshare_U = \frac{\sum A}{T}$$
$$\text{Over-provisioning: } timeshare_O = \frac{\sum B}{T}$$

Jitter

Although the *accuracy* and *timeshare* metrics characterize important aspects of elasticity, platforms can still behave very differently while producing the same metric values for *accuracy* and *timeshare*. An example is shown in Figure 14.2.



(a) Platform A



(b) Platform B

Fig. 14.2: Platforms with different elastic behaviors that produce equal results for *accuracy* and *timeshare* metrics

Both platforms A and B exhibit the same accuracy metrics and spend the same amount of time in under-provisioned and over-provisioned states, respectively. However, the behavior of the two platforms differs significantly. Platform B triggers more unnecessary resource supply adaptations than Platform A.

The *jitter* metric captures this instability and inertia of elasticity mechanisms. Low stability increases adaptation overheads and costs (e.g., in case of instance-hour-based pricing), whereas a high level of inertia results in a decreased SLO compliance.

The *jitter* metric compares the number of adaptations in the supply curve E_S with the number of adaptations in the demand curve E_D. If a platform de-/allocates more than one resource unit at a time, the adaptations are counted individually per resource unit. The difference is normalized by the length of the measurement period T:

$$\text{Jitter metric: } jitter \left[\frac{\#adaptations}{time}\right] = \frac{E_S - E_D}{T}$$

A negative *jitter* metric indicates that the platform adapts rather sluggish to changes in the demand. A positive *jitter* metric means that the platform tends to oscillate like Platforms A (little) and B (heavily) as in Figure 14.2. High absolute values of *jitter* metrics in general indicate that the platform is not able to react on demand changes appropriately. In contrast to the accuracy and timeshare metrics, a *jitter* value of zero is a necessary, but not sufficient condition for a perfect elastic system.

14.3.1.3 An Elasticity Benchmarking Concept

This paragraph shortly sketches an elasticity benchmarking concept as proposed in [17] and its implementation called BUNGEE[1] in [18]. The generic and cloud specific benchmark requirements as formulated by Huppler [19, 20] and Folkerts et al. [11] are considered in this approach. Figure 14.3 shows the four main steps of the benchmarking process explained in the following:

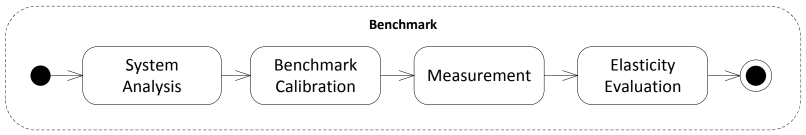


Fig. 14.3: Activity diagram for the benchmark workflow

1. **Platform Analysis:** The benchmark analyzes a system under test (SUT) with respect to the performance of its underlying resources and its scaling behavior.
2. **Benchmark Calibration:** The results of the analysis are used to adjust the load intensity profile injected on the SUT in a way that it induces the same resource demand on all compared platforms.
3. **Measurement:** The load generator exposes the SUT to a varying workload according to the adjusted load profile. The benchmark extracts the actual induced resource demand and monitors resource supply changes on the SUT.
4. **Elasticity Evaluation:** The elasticity metrics are computed and used to compare the resource demand and resource supply curves with respect to different elasticity aspects.

The results of an exemplary benchmark run are plotted in Figure 14.4 and the computed elasticity metrics are shown in Table 14.2.

[1] BUNGEE Cloud Elasticity Benchmark: `http://descartes.tools/bungee`

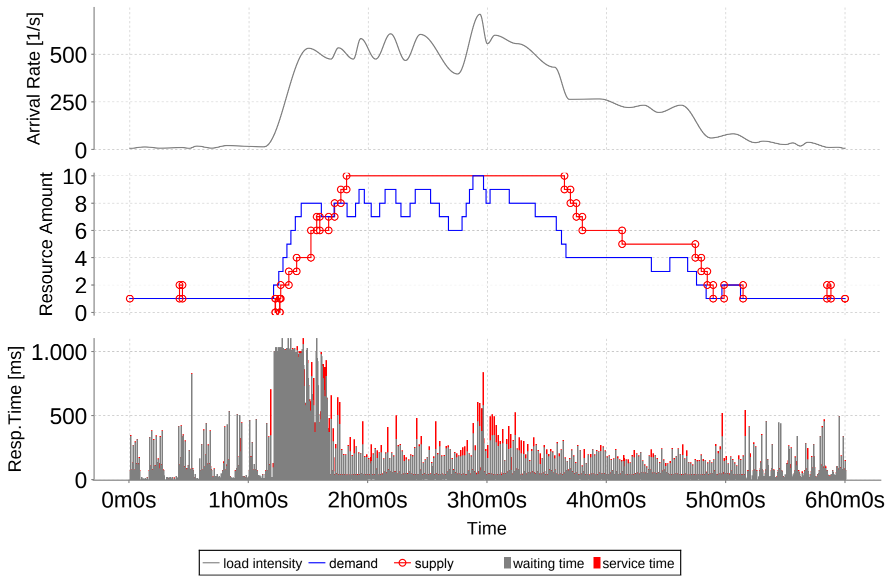


Fig. 14.4: Resource demand and supply curves for an exemplary benchmark run on a public cloud

Table 14.2: Metric results for an exemplary benchmark run

acc_O [#res.]	acc_U [#res.]	ts_O [%]	ts_U [%]	$jitter$ $\left[\frac{\#adap.}{min}\right]$
1.053	0.180	51.9	8.1	-0.033

14.3.2 Quantifying the Quality of Self-Protection

We consider self-protection as a distinct property of self-aware systems. Under self-protection, we understand continuous system protection against malicious activities (e.g., intrusion attempts, resource exhaustions) by performing actions countering these activities in real-time; that is, self-protection may be understood as one of the high-level goals that a self-aware system may have (see Chapter 1). A typical self-aware system performs actions countering malicious activities by utilizing common security mechanisms, such as intrusion detection and prevention systems, access control systems, performance isolation mechanisms for preventing deliberate resource exhaustions, and so on. In this section, we provide an overview of metrics for quantifying properties of intrusion detection systems, which often play a key role when it comes to self-protection. We also discuss the relevance of the considered metrics in the context of self-protection.

14.3.2.1 Quantifying Properties of Intrusion Detection Systems

Intrusion detection is a key enabling technology for self-protection. This is because the timely and accurate detection of intrusion attempts (i.e., security breaches) enables timely reaction in order to stop an on-going attack, or to mitigate the impact of a security breach.

We distinguish between two categories of metrics for quantifying properties of intrusion detection systems (IDSes): *performance-related* and *security-related* metrics. By performance-related metrics, we refer to metrics that quantify the non-functional properties of an IDS under test, such as attack detection delay, capacity, or resource consumption. By security-related metrics, we refer to metrics that quantify the attack detection accuracy of an IDS, such as true and false positive rate.[2] Because of their relevance when it comes to self-protection, we focus on metrics that quantify the attack detection delay as well as on security-related metrics that express the false positive rate.

Attack detection delay. The attack detection delay (also known as 'attack detection and reporting speed') is typically evaluated in the context of IDSes coupled with attack-response mechanisms (see, e.g., the work of Sen et al. [39]). Given that attack-response mechanisms perform actions countering attacks detected by IDSes, the fast detection and reporting of attacks by an IDS is crucial for the timely prevention of attacks, which, in turn, is crucial for effective self-protection.

The attack detection delay can be quantified as the time needed for an IDS to issue an alert after an attack has occurred. However, the way in which attack detection delay is quantified may depend on the type of employed IDS. For instance, the attack detection delay is often considered in the context of distributed IDSes. A distributed IDS is a compound IDS consisting of multiple intrusion detection sub-systems (i.e., nodes) possibly deployed at different sites that communicate to exchange intrusion detection-relevant data, for example, attack alerts. Each node of a distributed IDS typically reports an on-going attack to the rest of the nodes when it detects the attack. The immediate detection and reporting of attacks by each IDS node is crucial for the timely detection of coordinated attacks (i.e., attacks targeting multiple sites in a given time order). Therefore, the attack detection speed in the context of distributed IDSes is typically evaluated by measuring the time needed for an IDS to converge to a state in which all its nodes are notified of an on-going attack, as done by Hassanzadeh et al. [16] and Sen et al. [39].

Attack detection accuracy. The benefits of evaluating IDS attack detection accuracy are manifold. For instance, one may compare multiple IDSes in terms of their attack detection accuracy in order to deploy an IDS that operates optimally in a given environment, thus reducing the risks of a security breach. The security research community has developed multiple metrics for quantifying the attack detection accuracy, such as the *basic* metrics false and true positive rate. The false positive rate $\alpha = P(A|\neg I)$ quantifies the probability that an alert generated by an IDS is not an intrusion, but a regular benign activity; the true positive rate

[2] We refer the reader to Chapter 22 for in-depth discussions on these metrics.

$1-\beta = 1-P(\neg A|I) = P(A|I)$ quantifies the probability that an alert generated by an IDS is really an intrusion.[3] There are also *composite* metrics, that is, metrics that combine the basic metrics in order to enable the analysis of relationships between them, such as a ROC (Receiver Operating Characteristic) curve [28], and the metrics developed by Gaffney and Ulvila [12] and Gu et al. [13].

The measure of false positive rate is important when it comes to evaluating the attack detection accuracy of an IDS employed as part of a system featuring self-protection — such an IDS may exhibit high false positive rate and cause the triggering of many unnecessary actions countering attacks (e.g., shutting down targeted network services or sub-systems), which may have negative effects. For example, the system may incur high performance costs or become unavailable to users mistakenly labeled as users performing malicious activities. We now discuss the composite 'expected cost' (C_{exp}) metric developed by Gaffney and Ulvila [12]. This metric is a representative *cost-based* metric. Under cost-based metrics, we understand metrics designed to quantify costs, such as performance or financial costs, incurred by a system performing actions countering activities labeled as malicious by an IDS. Cost-based metrics characterize the impact of the false positive rate and are therefore relevant in the context of this work.

Gaffney and Ulvila combine ROC curve analysis with cost estimation by associating an estimated cost with each IDS operating point (i.e., an IDS configuration that yields given values of the true and false positive rates). A ROC curve is a two-dimensional depiction of the accuracy of a detector as it plots true positive rate against the corresponding false positive rate. Gaffney and Ulvila introduce a cost ratio $C = C_\alpha / C_\beta$, where C_α is the cost of an IDS alert when an intrusion has not occurred, and C_β is the cost of not detecting an intrusion when it has occurred. To calculate the cost ratio, one would need a cost-analysis model that can estimate C_α and C_β.

C_{exp} for a given IDS operating point can be calculated as $C_{exp} = Min(C\beta B, (1-\alpha)(1-\beta)) + Min(C(1-\beta)B, \alpha(1-B))$.[4] This formula can be obtained by analyzing (i.e., "rolling back") a decision tree whose leaves are costs that may be incurred by an IDS (i.e., C_α and C_β). For more details on the analytical formula of the 'expected cost' metric, we refer the reader to [12].

Using C_{exp} one can identify an optimal IDS operating point (i.e., an IDS configuration that yields optimal values of both the true and false positive detection rate) in a straightforward manner. The identification of an optimal IDS operating point is a common goal of IDS evaluation studies. A given operating point of an IDS is considered optimal if it has the lowest C_{exp} associated with it compared to the other operating points. In Figure 14.5, we depict a ROC curve annotated with the minimal C_{exp} for an IDS such that $1-\beta$ is related to α with a power function (i.e., $1-\beta = \alpha^k$). In Figure 14.5, we depict values of $1-\beta$ such that $\alpha = 0.005, 0.010, 0.015$, and $k = 0.002182$. We obtain the values of α and k from the work of Gaffney and Ul-

[3] A denotes an alert event (i.e., an IDS generates an attack alert); I denotes an intrusion event (i.e., an attack is performed).

[4] $1-\alpha = 1-P(A|\neg I) = P(\neg A|\neg I)$; $\beta = 1-P(A|I) = P(\neg A|I)$; B is the base rate (i.e., prior probability that an intrusion event occurs — $P(I)$).

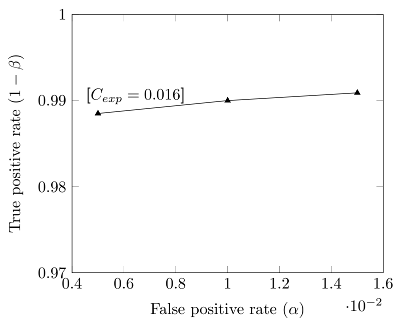


Fig. 14.5: Quantifying costs using the expected cost metric

vila [12]. Further, we assume a base rate $B = 0.1$ and a cost ratio $C = 10$ (i.e., the cost of not responding to an attack is 10 times higher than the cost of responding to a false alert).

Because of the negative effects that IDSes exhibiting high false positive rates may have, the research and industrial communities have designed IDS false alert filters. A typical false alert filter detects false alerts issued by an IDS and blocks their delivery. For instance, the de-facto standard network-based IDS Snort [37] features suppression of alerts.[5] Besides technical benefits, false alert filters bring usability-related benefits; that is, they reduce the cognitive overhead incurred on IT security officers who deal with IDSes on a daily basis, an issue acknowledged by many researchers (e.g., Komlodi et al. [23]). Meng et al. [29] have proposed a cost-based metric called 'relative expected cost' (C_{rec}), which enables the quantification of costs incurred by IDSes that use false alert filters. C_{rec} is based on the 'expected cost' metric (C_{exp}) [12], however, in contrast to C_{exp}, C_{rec} quantifies costs associated with the accuracy of an IDS's false alert filter at classifying alerts as true or false. Values of C_{rec} can be associated with each IDS operating point on a ROC curve. We refer the reader to the work of Meng et al. [29] for more details on the 'relative expected cost' metric.

Cost-based metrics require cost-analysis models for estimating costs, such as C_α and C_β. However, such models can be difficult to construct in practice since they assume the availability of parameters that might not be easy to measure (e.g, man-hours). Further, cost-based metrics are not objective — they quantify of the attack detection accuracy of an IDS based on a subjective measure (i.e., cost). However, cost-based metrics may be of significant value when the relationships between the different attack detection costs (e.g., cost of missing an attack, cost of a false alert) can be estimated and when such estimations would be considered as sufficiently accurate by the IDS evaluator for a particular IDS evaluation study. For instance, given a statement such as *"a false alarm is three times as costly as a missed attack"*, a cost-based metric would be crucial to identify an optimal IDS operating point.

[5] See, for example, http://manual.snort.org/node19.html#SECTION00343000000000000000.

Open challenge. Many modern environments exhibit self-adaptivity, a trait of self-awareness (see Chapter 1), in terms of on-demand provisioning of resources to computing nodes with respect to changes in the workload intensity. An example is a virtualized Cloud environment that has *elastic* properties; that is, the hypervisor governing the environment may provision on-demand (i.e., hotplug) CPU and memory resources to virtual machines (VMs). With the increasing adoption of virtualization technology, the practice has emerged to deploy IDSes in virtualized environments. For instance, a network-based IDS, such as Snort [37], deployed in a designated, secured VM may tap into the physical network interface card used by all VMs. Therefore, it can monitor the network activities of all guest VMs at the same time while being isolated from, and transparent to, potential malicious VM users.

Existing metrics for quantifying IDS attack detection accuracy are defined with respect to a *fixed* set of hardware resources available to the IDS. Their values express the properties of the IDS for a specific hardware environment in which the IDS is expected to reside during its lifetime [15]. However, many modern virtualized environments (e.g., Cloud environments) have elastic properties; that is, resources can be provisioned and used by VMs, on-demand during operation. Elasticity may have an impact on the measured attack detection accuracy of an IDS deployed in a virtualized environment. For example, the measured attack detection accuracy of a network-based IDS under test may depend on the number of dropped packets by the IDS in the time intervals when attacks have been performed. Large amounts of dropped packets in such intervals due to lack of resources may manifest themselves as low IDS attack detection accuracy.

Based on the above, we believe that the use of conventional metrics may lead to inaccurate measurements in cases where the on-demand provisioning of resources to the VM where an IDS under test resides has significant impact on the IDS's attack detection accuracy.[6] This, in turn, may result in the deployment of misconfigured or ill-performing IDSes in production environments, increasing the risk of security breaches. We argue that novel metrics and measurement methodologies for measuring the attack detection accuracy of IDSes deployed in virtualized environments are needed. Such metrics and methodologies should take into account the behavior of a given IDS under test as its operational environment changes. As a result, they would allow to quantify the ability of the IDS to scale its attack detection efficiency as resources are allocated and deallocated during operation. Metrics for quantifying IDS attack detection accuracy that take elasticity of virtualized environments into account are discussed in detail in Chapter 22.

14.4 Enhancing self-awareness by benchmarking

In some context, self-awareness may be used mainly to dimension the system. The knowledge of some relevant properties (like the latency of a job for real-time sys-

[6] Under conventional metrics, we understand existing IDS attack detection accuracy metrics, which do not take elasticity of virtualized environments into account.

tems) guides the choice of the hardware to be used for a specific task. This is especially crucial where, for space limitations, the amount of computing capacity that can be inserted into the physical space is limited.

In the automotive domain, for example, processors take care of multiple jobs at the same time. Some of these jobs are high priority ones, like the cruise control. Some other jobs have lower priority, like the entertainment system. Obtaining the awareness of how much computing capacity is needed for each of the required tasks can simplify the resource allocation and the entire design process. Applications can be packed in the same physical cores, to save space and minimize the amount of necessary hardware. While the system is running, the same awareness can guide scheduling and resource allocation decisions, minimizing energy and performance loss.

14.4.1 Unveiling the Real-Time Properties of Schedulers

The design principles that are followed when developing resource management techniques in operating systems include *simplicity*, *low overhead/memory footprint* and *efficiency*. Operating systems schedulers are usually developed following these principles and guidelines. Once a scheduler is developed, its performance is tested with a set of benchmarks. However, these benchmarks usually check the functionality of the schedulers and the correctness of their behavior from a functional point of view, while the ability to provide real-time guarantees is rarely properly quantified.

The real-time community has put a great effort in developing efficient scheduling algorithms and, more in general, resource management policies to respond to a broad variety of application models, types of execution platforms, etc. Unfortunately, a large share of these results remains confined to the theory and is not implemented into operating systems. In fact, operating system developers are resistant to the adoption of real-time methods and algorithms, mostly because general purpose operating systems must function in a more complex scenario than the one abstracted in real-time models, while real-time scheduling algorithms tend to respond to specific problems and needs.

In an attempt to close the gap between these two worlds (the implementation side and the theoretical algorithm side), one can basically follow two strategies: (1) simplifying the development of schedulers and providing tools that reduces the implementation burden; (2) abstracting relevant quantities that can describe the behavior of schedulers based on the execution of real tasks. This latter solution unveils the real-time properties of tasks when they are executed on top of the real implementation of scheduling policies. In turn, this introduces *self-awareness* in the system. The developer now has more information about the execution of tasks and is provided with relevant data that can be used to design the system (for example for hardware dimensioning).

`rt-muse`[7] in an application-independent tool that takes as input a model of a multi-threaded application, where each thread is constructed by basic elements, called *phases* and produces as output an analysis of what happens when the threads are executed on a real Linux platform.

Phases can be selected from a library that includes pure computation, resource locking and memory usage. The tool is also extensible, the effort to create new phases (possibly needed to capture application-specific behaviors) is minimal. The results of the experiments are reproducible, as the tool relies only on tools that are integrated in the Linux kernel. `rt-muse` records execution traces of the benchmark application. The recorded traces are analyzed, providing both per-thread metrics and aggregated features such as the bandwidth and the delay of the computing capacity given to the application. The analysis is based on plugins, each one providing some desired real-time feature. The usage of plugins enables both the configuration of the analysis procedure and the development of additional analysis methods. Currently, `rt-muse` supports three types of analysis. The runmap analysis is aimed at providing migration-relevant information and execution maps, unveiling how the computing resources are utilized. The statistical plugin approximates the empirical data about the execution times of the threads with probability distributions. The supply analysis produces abstractions of the computing capacity based on the concept of a supply function [3, 25].

`rt-muse` executes the multi-threaded program directly on the hardware and transfers the data via a UDP network connection, to avoid generating logging overhead on the machine that is executing the threads.

The tool was used to discover interesting facts about the behavior of Linux scheduling classes. In Linux terms, `SCHED_OTHER` is the standard algorithm in Linux, called Completely Fair Scheduler (CFS). The main advantage of using this algorithm is to enforce fairness among the running threads. Threads that have the same characteristics should receive an approximately equal amount of CPU. A set $\mathscr{T} = \{\tau_1, \dots, \tau_6\}$ of 6 threads was run using `rt-muse`. All the τ_i threads have the same characteristics. Their job is composed by one single phase $\phi_{i,1}$ which simply executes some mathematical instructions. In other words, every thread executes a certain number of mathematical operations for each job and jobs are run one after the other without stopping. The scheduling parameters of all the threads belonging to $\mathscr{T}$ are the same: the threads affinity mask contains three CPUs, $[1,2,3]$, avoiding the execution on CPU #0. The experiment lasted 100 seconds and `rt-muse` was recording the start time of each job. Ideally, one would expect that six threads with the same characteristics, running on three cores, would receive similar budgets, possibly obtaining each half of a CPU.

Table 14.3 reports the amount of CPU α'_i that is used by each thread as estimated by the tool and the delay in executing the thread δ'_i. The last three columns correspond to the share of consumed CPU.

The allocated computing capacities α'_i are almost all equal to each other, except α'_6 which is noticeably larger than the others. A possible explanation could be in the

[7] https://github.com/martinamaggio/rt-muse/

	CPUUSAGE		CPUSHARE		
τ_i	α'_i	δ'_i	#1	#2	#3
τ_1	0.412	0.591	0.062	0.614	0.324
τ_2	0.410	1.314	0.191	0.332	0.476
τ_3	0.409	1.398	0.009	0.440	0.551
τ_4	0.413	1.839	0.159	0.302	0.539
τ_5	0.413	0.684	0.109	0.545	0.346
τ_6	0.567	3.985	0.834	0.062	0.104

Table 14.3: Threads scheduled by SCHED_OTHER.

longer time τ_6 executed over the same CPU. Although other interfering operating system threads could be easily accommodated on CPU #0, which is not used by the set $\mathscr{T}$, still the overall delivered bandwidth is $\alpha'_* = 2.641862$, quite less than the full 3 CPUs dedicated to $\mathscr{T}$.

From this experiment a developer would learn that the Completely Fair Scheduler in Linux is in fact not that fair among threads and that to enforce real-time guarantees on the computation capacity offered to the threads it is better to use real-time scheduling policies like SCHED_RR and SCHED_FIFO. In subsequent experiments with the tool, however, it was discovered that despite both SCHED_RR and SCHED_FIFO, timing properties, they do not enforce any type of fairness among the threads and the push/pull migration system that they use has quite many defects [27].

14.5 Addressing the Challenges in Defining Reference Workloads for Benchmarking Self-Aware Computing System Properties

Benchmarking is a critical step for effective capacity planning and resource provisioning, and consequently may guide the design of self-aware systems. An effective benchmark should evaluate the system responsiveness under a wide range of client demands from low to high, but most benchmarks are designed to assess the system responsiveness under a *steady* client demand. If systems are provisioned for steady *peak* loads to avoid the deleterious effects of sudden workload surges, they consistency suffer from low resource utilizations [4], which results in ecosystems that are energy inefficient and wasteful. In addition, system behavior under high yet steady client demand may actually be very different than under varying or bursty conditions [30,32]. Because of its tremendous performance implications, variability and burstiness must be accounted in the design of self-aware systems and must be incorporated into benchmarking.

Burstiness in workloads can be broadly defined as workload surges that occur aperiodically, with various frequencies, and (usually) short duration. In cloud computing, for example, virtual machine (VM) workloads have been observed to be highly bursty [5, 31, 44]. For a typical web server, burstiness can be an outcome of the flash crowd effect, where a web page linked by a popular blog or media site suddenly experiences a huge increase of the number of hits. Consider also the case of an auction site (e.g., eBay) where users compete to buy an object that is going to be soon assigned to the customer with the best offer, but also in e-business sites as a result of special offers and marketing campaigns. Generally, if variability or even burstiness in the workload is observed, it can be catastrophic for performance, leading to dramatic server overloading, uncontrolled increase of response times and, in the worst case, service unavailability [30–32].

Standard benchmarks lack the ability to produce a representatively varying load profile with burstiness because user arrivals are defined by a Poisson process, i.e., they are always assumed to be independent of their past activity and independent of each other. Exponential inter-arrival times are incompatible with the notion of burstiness for several reasons:

Temporal locality: intuitively, under conditions of burstiness, arrivals from different sources cannot happen at random instants of time, but they are instead condensed in short periods across time. Therefore, the probability of sending a request inside this period is much larger than outside it. This behavior is inconsistent with classic distributions considered in performance engineering of web applications, such as Poisson, hyper-exponential, Zipf, and Pareto, which all miss the ability of describing temporal locality within a process.

Variability of different time scales: Variability within a traffic surge is a relevant characteristic for testing peak performance degradation. Therefore, a benchmarking model for burstiness should not only create surges of variable intensity and duration, but also create fluctuations within a surge. This implies a hierarchy of variability levels that cannot be described by a simple exponential distribution and instead requires a more structured arrival process.

Lack of aggregation: In standard client-server benchmarks each thread on the client machines uses a dedicated stream of random numbers, thus inter-arrival times of different users are always independent. This is indeed representative of normal traffic, but fails in capturing the essential property of traffic surges: users act in an aggregated fashion which is mostly incompatible with independence assumptions.

In the following subsections, we shortly outline an approach to model variable load profiles and present a methodology that addresses the above points and provides a seamless way to incorporate burstiness into benchmarks.

14.5.1 Modeling of Load Intensity Patterns

Many modeling approaches include concepts to define workload intensity by attaching this information to modeled workload scenarios. Examples can be found in the

SPE approach by [40] and in UML-SPT [34]. UML-SPT supports scenarios with open and closed workloads that are parameterized by specifications of arrivals (occurrence patterns), or a population with a think times distribution (external delay). The allowed attribute values include the definition of probability distributions and different arrival patterns—namely bursty, bounded, and periodic. Other approaches to be mentioned in this context are the CSM [36] and MARTE [35].

We conclude from reviewing the above-mentioned approaches, that a load intensity profile definition is a crucial element to complete a workload characterization. The observed or estimated arrival process of transactions (on the level of users, sessions or requests/jobs arrivals) needs to be specified. As basis to specify time-dependent arrival rates or inter-arrival times, the extraction of a usage model should provide a classification of transaction types, that are statistically indistinguishable in terms of their resource demanding characteristics.

A *Load Intensity Profile* is an instance of an arrival process. A workload that consists of several types of transactions is then characterized by a set of load intensity profile instances.

Load intensity profiles are directly applicable in the context of any open workload scenario with a theoretically unlimited number of users, but are not limited to those [38]. In a closed or partially closed workload scenario, with a limited number of active transactions, the arrival process can be specified within the given upper limits and zero. Any load intensity profile can be transformed into a time series containing arrival rates per sampling interval.

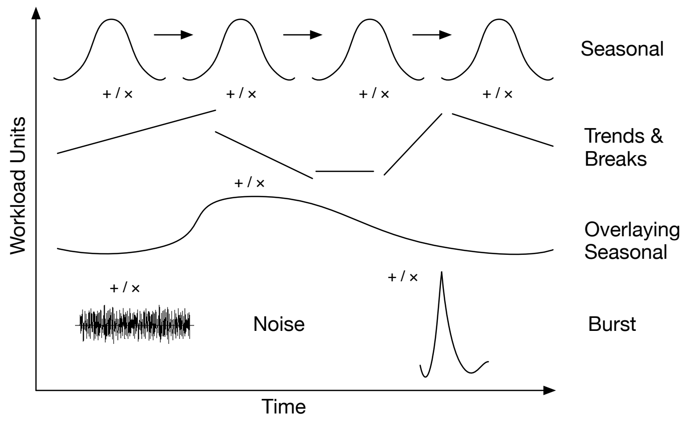


Fig. 14.6: Elements of Load Intensity Profiles as proposed in [22]

For a load profile to be representative for a given application domain it has to be a mixture of *(i)* one or more (overlaying) seasonal patterns, *(ii)* long term trends including trend breaks, *(iii)* characteristic bursts and *(iv)* a certain degree of noise.

These components can be combined in an additive or multiplicative manner over time as visualized in Figure 14.6.

At early development stages, load intensity profiles can be estimated by domain experts by defining synthetic profiles using statistical distributions or mathematical functions. At a higher abstraction level, the Descartes Load Intensity Model (DLIM) allows to descriptively define the seasonal, trend, burst, and noise elements in a wizard-like manner [22]. DLIM is supported by a tool-chain named LIMBO [1].

A good starting point for a load intensity profile definition at the development stage is to analyze the load intensity of comparable systems within the same domain.

We identify the following open challenges in the field of load profile description and their automatic extraction:

- Seasonal patterns may overlay each other (e.g., weekly and daily patterns) and change in their shape over time. Current load profile extraction approaches do not fully support these scenarios.
- Based on meta-knowledge or clustering techniques, seasonal patterns can be by classified and extracted into separate models, e.g., a model for ordinary working days, for public holidays, and for weekends, which would result in more accurate load profile extraction.

14.5.2 Markov Arrival Processes for Modeling Burstiness

A Markov arrival process (MAP) can be seen as a simple mathematical model of a time series, such as a sequence of interarrival times, for which we can accurately shape distribution and correlations between successive values. Correlations among consecutive think times are instrumental to capture periods of the time series where think times are consecutively small and thus a surge occurs, as well as to determine the duration of the surge.

We use a class of MAPs with two states only, one responsible for the generation of “short” inter-arrival times implying that users arrive in closely spaced arrivals, possibly resulting in surges, while the other is responsible for the generation of “long” interarrival times associated to periods of normal traffic. In the “short” state, interarrival times are generated with mean rate λ_{short}, similarly they have mean rate $\lambda_{long} < \lambda_{short}$ in the “long” state. In order to create correlation between different events, after the generation of a new interarrival time sample, our model has a probability p_{short} that two consecutive times are short and a different probability p_{long} of two consecutive think times being both long. The values of p_{short} and p_{long} shape the correlations between consecutive interarrival times and are instrumental to determine the duration of the traffic surge. The two probabilities p_{short} and p_{long} are independent of each other.

In order to gain intuition on the way this model works, we provide the following pseudo code to generate a sample of n_t interarrival time values $Z_1, Z_2, \ldots, Z_n, \ldots, Z_{n_t}$ from a MAP parameterized by the tuple $(\lambda_{long}, \lambda_{short}, p_{long}, p_{short})$:

```
function: MAP_sample($\lambda_{long}$, $\lambda_{short}$, $p_{long}$, $p_{short}$, $n_t$)
/* initialization in normal traffic state */
active_state = “long”;
for $n = 1, 2, \ldots, n_t$
/* generate sample in current state */
    $Z_n$ = sample from exponential distribution
        with rate $\lambda_{active\_state}$;
/* update MAP state */
    $r$ = random number in $[0, 1]$;
    if active_state =“long” and $r \geq p_{long}$
        active_state = “short”;
    else if active_state =“short” and $r \geq p_{short}$
        active_state = “long”;
    end
end
```

Figure 14.7 summarizes the traffic surge model described above. Note from the pseudo code that the problem of variability of different time scales is solved effectively in MAPs by the fact that, if the MAP is in a state i, then the samples are generated by an exponential distribution with rate λ_i associated with state i. This creates fluctuations within the traffic surge. The probability of arrivals inside the traffic surge is larger than outside it, thanks to the state change mechanism that alters the rate of arrival from λ_{long} to λ_{short}.

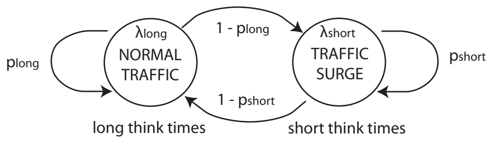


Fig. 14.7: Model of traffic surges based on regulation of interarrival times

Summarizing, we propose to generate interarrival times in such a way to periodically enter into a state “short” that facilitates the formation of burstiness because here users have smaller interarrival times and thus submit requests more often. The parameterization of the MAP requires only to assign λ_{short} and $\lambda_{long} < \lambda_{short}$ to describe the interarrival times respectively during flash crowds and normal user activity periods, together with the probabilities p_{short} and p_{long} of consecutive interarrival times being both short or both long, respectively. Once that these four parameters have been set up, the benchmark can generate interarrival time samples using the above pseudocode. For more details on MAPs, we direct the interested reader to [6, 7].

Yet, the problem that is not solved is the quantification of burstiness. The *index of dispersion* as a regulator of the intensity of traffic surges. The index of dispersion I

is a measure of burstiness in networking and service engineering [14, 30]. Consider a sequence of values (e.g., think times, inter-arrival times, service times) with variability quantified by the squared-coefficient of variation (SCV), where the difference in magnitude between consecutive values is summarized by the lag-k autocorrelations[8] ρ_k. Assuming that SCV and ρ_k do not change over time, then the index of dispersion is the quantity $I = SCV\,(1+2\sum_{k=1}^{\infty}\rho_k)$. For finite length sequences, this value can be estimated accurately, without resorting to an infinite summation, using the methods outlined in [14]. The index of dispersion I has the fundamental property that it grows proportionally with both variability and correlations, thus can be immediately used to identify burstiness in a workload trace.

When there is no burstiness, the value of I is equal to the squared coefficient-of-variation of the distribution, e.g., $I = SCV = 1$ for the exponential distribution, while it grows to values of thousands on bursty processes. Thus, a parameterization of I spanning a range from single to multiple digits can give a good sense of scalability between workloads with "no burstiness" and workloads with "dramatic burstiness", for examples of workloads and their index of dispersion, we direct the interested reader to [31].

Now the question becomes: how should the MAP process be parameterized in order to regulate burstiness using the index of dispersion as a measure? We present here a case study that considers a **closed system** and show how to determine a parameterization (λ_{long}, λ_{short}, p_{long}, p_{short}) that produces a sequence of surges in the traffic. We assume that the user gives the desired values of the mean interarrival time $E[Z]$ (typically, in closed systems terminology, this value can be considered as the mean user "think time" before a request is sent to the system and of the (desired) index of dispersion I. Again, using typical closed-system terminology, we assume that we have N circulating users in the system. For example, for a typical web server, N corresponds to the maximum number of client connections. One needs to also consider the average service demand $E[D_i]$ of each server i that can be estimated from utilization measurements [43]. This measure is important to provide, because the purpose of the benchmark is to keep the average system utilization to less than 100%, i.e., the system is never in over-saturation.

A MAP can fully define the think/interarrival time distribution other than the mean $E[Z]$ starting by the following parameterization equations:

$$\lambda_{short}^{-1} = (\textstyle\sum_i E[D_i])/f, \tag{14.1}$$

$$\lambda_{long}^{-1} = f\max(N(\textstyle\sum_i E[D_i]), E[Z]]). \tag{14.2}$$

Here, $f \geq 1$ is a free parameter, N is the maximum number of client connections considered in the benchmarking experiment, $\sum_i E[D_i]$ is the minimum time taken by a request to complete at all servers, and $N(\sum_i E[D_i])$ provides an upper bound to the time required by the system to respond to all requests. Eq. (14.1) states that, in order to create surges, the think times should be smaller than the time required

[8] Recall that the lag-k autocorrelation coefficient is a normalized measure of correlation between random variables X_t and X_{t-k}, with position in the trace differing by k lags. For a trace with mean μ and variance σ^2, $\rho_k = E[(X_t-\mu)(X_{t-k}-\mu)]/\sigma^2, k \geq 1$.

by the system to respond to requests. Thus, assuming that all N clients are simultaneously waiting to submit a new request, one may reasonably expect that after a few multiples of λ_{short}^{-1} all clients have submitted requests and the architecture has been yet unable to cope with the traffic surge. Conversely, (14.2) defines think times that on average give to the system enough time to cope with any request, i.e., the normal traffic regime. Note that the condition $\lambda_{long}^{-1} \geq fE[Z]$ is imposed to assure that the mean think time can be $E[Z]$, which would not be possible if both $\lambda_{short}^{-1} > \lambda_{long}^{-1} > E[Z]$ since $f > 1$ and in MAPs the moments $E[Z], E[Z^2], \ldots$ are

$$E[Z^k] = \frac{p_{long}}{p_{long} + p_{short}} \lambda_{short}^{-k} + \frac{p_{short}}{p_{long} + p_{short}} \lambda_{long}^{-k}. \tag{14.3}$$

The above formula for $k = 1$ implies that $E[Z]$ has a value in between of λ_{short}^{-1} and λ_{long}^{-1}, which is not compatible with $\lambda_{short}^{-1} \geq \lambda_{long}^{-1} \geq fE[Z]$. According to the last formula, the MAP parameterization can always impose the user-defined $E[Z]$ if

$$p_{long} = p_{short} \left(\frac{\lambda_{long}^{-1} - E[Z]}{E[Z] - \lambda_{short}^{-1}} \right), \tag{14.4}$$

condition which we use in the modified TPC-W benchmark to impose the mean think time.

In order to fix the values of p_{short} and f in the above equations, we first do a simple search on the space $(p_{short} \geq 0,\ f \geq 1)$ where at each iteration we check the value of the index of dispersion I and lag-1 autocorrelation coefficient ρ_1 from the current values of p_{short} and f. We stop searching when we find a MAP with an I that is within 1% of the target user-specified index of dispersion and the lag-1 autocorrelation is at least $\rho_1 \geq 0.4$ in order to have consistent probability of formation of surges within short time periods[9]. The index of dispersion of the MAP can be evaluated at each iteration as [10] [6, 7, 33]:

$$I = 1 + \frac{2\, p_{short}\, p_{long} (\lambda_{short} - \lambda_{long})^2}{(p_{short} + p_{long})(\lambda_{short} p_{short} + \lambda_{long} p_{long})^2}, \tag{14.5}$$

while the lag-1 autocorrelation coefficient is computed as

$$\rho_1 = \frac{1}{2}(1 - p_{long} - p_{short}) \left(1 - \frac{E[Z]^2}{E[Z^2] - E[Z]^2} \right), \tag{14.6}$$

where $E[Z^2]$ is obtained from (14.3) for $k = 2$. We direct the reader to [31] for a case study that illustrates how to generate arrival processes with various degrees of burstiness within the TPC-W benchmark.

[9] The threshold 0.4 has been chosen since it is the closest round value to the maximum autocorrelation that can be obtained by a two-state MAP.

[10] Note that Eq. (14.5) slightly differs in the denominator from other expressions of I, such as those reported in [14], because here we consider a MAP that is a generalization of an MMPP process.

14.6 Conclusion and Open Challenges

In this chapter, we proposed a set of metrics for describing self-aware computing system properties and grouped them by the MAPE-K paradigm. This set of metrics can be seen as a first step towards building benchmarks for self-aware computing systems, consisting of a well-defined measurement methodology (run rules), representative workloads and metrics.

The central remaining challenge is to properly evaluate the behavior of self-aware systems with this set of metrics, in other words to build a level-playing field for fair comparisons of self-aware system properties. This includes the challenges to define reference systems/behavior, and design measurement processes for repeatable and fair results. Ideas on how these challenges could be tackled can be found in related areas of research like self-adaptive systems, autonomic computing and similar. Starting from Section 14.3.1, we presented central aspects from related areas that may become relevant for benchmarking self-aware computing systems. As a more philosophical outlook and discussion on how self-aware system properties can be assessed, we refer the reader to Chapter 15.

References

1. LIMBO: Load Intensity Modeling Framework. `http://descartes.tools/limbo`, 2015.
2. Rodrigo F Almeida, Flávio RC Sousa, Sérgio Lifschitz, and Javam C Machado. On Defining Metrics for Elasticity of Cloud Databases. In *Proceedings of the 28th Brazilian Symposium on Databases*, 2013.
3. Enrico Bini, Marko Bertogna, and Sanjoy Baruah. Virtual multiprocessor platforms: Specification and use. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pages 437–446, 2009.
4. Robert Birke, Mathias Björkqvist, Lydia Y. Chen, Evgenia Smirni, and Ton Engbersen. (big)data in a virtualized world: volume, velocity, and variety in cloud datacenters. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014*, pages 177–189, 2014.
5. Robert Birke, Andrej Podzimek, Lydia Y. Chen, and Evgenia Smirni. State-of-the-practice in data center virtualization: Toward a better understanding of VM usage. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, pages 1–12, 2013.
6. Giuliano Casale, Eddy Z. Zhang, and Evgenia Smirni. Kpc-toolbox: Best recipes for automatic trace fitting using markovian arrival processes. *Perform. Eval.*, 67(9):873–896, 2010.
7. Giuliano Casale, Eddy Z. Zhang, and Evgenia Smirni. Trace data characterization and fitting for markov modeling. *Perform. Eval.*, 67(2):61–79, 2010.
8. Dean Chandler, Nurcan Coskun, Salman Baset, Erich Nahum, Steve Realmuto Masud Khandker, Tom Daly, Nicholas Wakou Indrani Paul, Louis Barton, Mark Wagner, Rema Hariharan, and Yun seng Chao. Report on Cloud Computing to the OSG Steering Committee. Technical report, April 2012.
9. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

10. Thibault Dory, Boris Mejías, Peter Van Roy, and Nam-Luc Tran. Measuring Elasticity for Cloud Databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.
11. Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In Raghunath Nambiar and Meikel Poess, editors, *Selected Topics in Performance Evaluation and Benchmarking*, volume 7755 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2012.
12. John E. Gaffney and Jacob W. Ulvila. Evaluation of intrusion detectors: a decision theory approach. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 50–61, 2001.
13. Guofei Gu, Prahlad Fogla, David Dagon, Wenke Lee, and Boris Skorić. Measuring intrusion detection capability: an information-theoretic approach. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security (ASIACCS)*, pages 90–101, New York, NY, USA, 2006. ACM.
14. R. Gusella. Characterizing the variability of arrival processes with indexes of dispersion. *IEEE JSAC*, 19(2):203–211, 1991.
15. Mike Hall and Kevin Wiley. Capacity verification for high speed network intrusion detection systems. In *Proceedings of the 5th International Conference on Recent Advances in Intrusion Detection (RAID)*, pages 239–251, Berlin, Heidelberg, 2002. Springer-Verlag.
16. Amin Hassanzadeh and Radu Stoleru. Towards Optimal Monitoring in Cooperative IDS for Resource Constrained Wireless Networks. In *Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–8, August 2011.
17. Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What it is, and What it is Not (short paper). In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013)*. USENIX, June 2013.
18. Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. BUNGEE: An Elasticity Benchmark for Self-adaptive IaaS Cloud Environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 46–56, Piscataway, NJ, USA, 2015. IEEE Press.
19. Karl Huppler. Performance Evaluation and Benchmarking. chapter The Art of Building a Good Benchmark, pages 18–30. Springer-Verlag, Berlin, Heidelberg, 2009.
20. Karl Huppler. Benchmarking with Your Head in the Cloud. In Raghunath Nambiar and Meikel Poess, editors, *Topics in Performance Evaluation, Measurement and Characterization*, volume 7144 of *Lecture Notes in Computer Science*, pages 97–110. Springer Berlin Heidelberg, 2012.
21. Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 85–96, New York, NY, USA, 2012. ACM.
22. Jóakim V. Kistowski, Nikolas Herbst, Daniel Zoller, Samuel Kounev, and Andreas Hotho. Modeling and extracting load intensity profiles. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 109–119, Piscataway, NJ, USA, 2015. IEEE Press.
23. Anita Komlodi, John R. Goodall, and Wayne G. Lutters. An Information Visualization Framework for Intrusion Detection. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, page 1743, New York, NY, USA, 2004. ACM.
24. Michael Kuperberg, Nikolas Roman Herbst, Joakim Gunnarson von Kistowski, and Ralf Reussner. Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms. Technical report, Karlsruhe Institute of Technology (KIT), 2011.
25. Hennadiy Leontyev, Samarjit Chakraborty, and James H. Anderson. Multiprocessor extensions to real-time calculus. *Real-Time Syst.*, 47(6):562–617, December 2011.
26. Zheng Li, L. O'Brien, He Zhang, and R. Cai. On a Catalogue of Metrics for Evaluating Commercial Cloud Services. In *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*, pages 164–173, Sept 2012.
27. Martina Maggio, Juri Lelli, and Enrico Bin. Analysis of os schedulers with rt-muse. In *RTSS@Work (Real-Time Systems Symposium Demo Session)*, 2015.

28. Roy A. Maxion and Kymie M.C. Tan. Benchmarking anomaly-based detection systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 623–630, 2000.
29. Yuxin Meng. Measuring intelligent false alarm reduction using an ROC curve-based approach in network intrusion detection. In *IEEE International Conference on Computational Intelligence for Measurement Systems and Applications (CIMSA)*, pages 108–113, July 2012.
30. Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *ACM/IFIP/USENIX 9th International Middleware Conference (Middleware'08)*, Leuven, Belgium, 2008. The prelimilary paper appeared in the HotMetrics 2008 Workshop.
31. Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *Proceedings of the 6th International Conference on Autonomic Computing, ICAC 2009, June 15-19, 2009, Barcelona, Spain*, pages 149–158, 2009.
32. Ningfang Mi, Qi Zhang, Alma Riska, Evgenia Smirni, and Erik Riedel. Performance impacts of autocorrelated flows in multi-tiered systems. *Perform. Eval.*, 64(9-12):1082–1101, 2007.
33. M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. Marcel Dekker, New York, 1989.
34. Object Management Group, Inc. UML Profile for Schedulability, Performance, and Time (SPT), version 1.1. `http://www.omg.org/spec/SPTP/1.1/`, 2005.
35. Object Management Group, Inc. UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1. `http://www.omg.org/spec/MARTE/1.1/`, 2011.
36. Dorin Bogdan Petriu and C. Murray Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Springer Software and System Modeling (SoSym)*, 6(2):163–184, 2007.
37. Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX conference on System Administration (LISA)*, pages 229–238. USENIX Association, 1999.
38. Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation (NSDI '06)*, pages 18–18. USENIX Association, 2006.
39. Jaydip Sen, Arijit Ukil, Debasis Bera, and Arpan Pal. A distributed intrusion detection system for wireless ad hoc networks. In *16th IEEE International Conference on Networks (ICON)*, pages 1–6, 2008.
40. Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.
41. Christian Tinnefeld, Daniel Taschik, and Hasso Plattner. Quantifying the Elasticity of a Database Management System. In *DBKDA 2014, The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 125–131, 2014.
42. Joe Weinman. Time is Money: The Value of "On-Demand", 2011. (accessed July 9, 2014).
43. Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Fourth International Conference on Autonomic Computing (ICAC'07), Jacksonville, Florida, USA, June 11-15, 2007*, page 27, 2007.
44. S. Zhang, Z. Qian, Z. Luo, J. Wu, and S. Lu. Burstiness-aware resource reservation for server consolidation in computing clouds. *IEEE Trnascations on Parallel and Distributed Systems*, to appear 2015.