Prometheus: a flexible toolkit for the experimentation with virtualized infrastructures

Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000
DOI: xxx/xxxx

RESEARCH ARTICLE

Prometheus: a flexible toolkit for the experimentation with virtualized infrastructures

Cosimo Anglano | Massimo Canonico | Marco Guazzone*

[1]DiSIT, Computer Science Institute, University of Piemonte Orientale, Italy

Correspondence
*Marco Guazzone. Address: Viale T. Michel 11, 15121 Alessandria (Italy). Email: marco.guazzone@uniupo.it

Summary

A typical problem that arises when devising a novel resource management strategy for virtualized infrastructures is how to experimentally assess its ability of achieving its design goals and whether it advances the state-of-the-art. Among the available options, the use of a physical testbed is usually considered to be the most appropriate because of its high degree of accuracy. Unfortunately, however, physical testbeds are characterized by a limited controllability of the experimental conditions. Moreover, their implementation is usually a complex and time-consuming task, that requires the integration of many software components that need to interact among them in non-trivial ways.
In this paper, we address the above issues by proposing Prometheus, a toolkit specifically designed to support the configuration, deployment, and use of physical testbeds suitable to perform experimental studies of resource management strategies, and that provides a high degree of controllability and low implementation costs. We discuss the design, implementation and use of Prometheus, and we show how it can be used in practice to configure and deploy a physical testbed by providing various examples of experimental activities that can be carried out by means of it.

KEYWORDS:
Toolkit, resource management, experimental evaluation, physical testbed, virtualization

1 | INTRODUCTION

The advent of virtualization technologies able to run on commodity processors has reshaped the way we do computing. The possibility of efficiently multiplexing a physical system across different Virtual Machines (VMs) has indeed opened the door to cloud computing (1) first and, more recently, to fog computing (2). Virtualization brings indeed various advantages (3), such as partitioning (many VMs can share the same physical machine), isolation (these VMs can coexist without conflicting among them), and encapsulation (the whole software stack of a VM is encapsulated into a set of files).

However, what really makes virtualization a real option for production use, is the ability of efficiently multiplex a set of VMs on a physical machine in order to obtain, at the same time, good performance for individual applications/services running into a VM, performance isolation across different VMs, and high consolidation levels for physical hosts, in face of the time-varying and bursty workloads they must process.

To obtain this goal, it is crucial that resource management – for both physical and virtualized applications – is carried out in such a way to dynamically allocate physical resource capacity to individual VMs so that the above goals are met. However, in spite of the vast body of research carried out in the past years on this topic (see (4) for a survey), resource management for virtualized systems is still an open research problem, and represents a very active area of research.

A typical problem that arises when devising a novel resource management algorithm of this type is how to assess its ability of solving the problems mentioned before, and whether it advances or not the state-of-the-art (i.e., it performs better than available alternatives). Typically, this is carried out by means of an experimentation where the proposed algorithm is evaluated and compared with existing ones by means of a suitable methodology, that ensures both the accuracy of results (i.e., how well these results map to the actual results observed on a real system) and their reproducibility (i.e., whether a third-party replicating the experiments under the same conditions obtains the same results).

Among the methodology available to carry out the above experimentation, namely analytical modeling, (discrete-event) simulation, and usage of a physical testbed, the latter one is usually considered to be the most appropriate because it features a level of accuracy higher than the other ones. A physical testbed is indeed a real systems running real applications exposed to real workloads, while the other methodologies resort to abstract models of systems, applications, and workloads, whose accuracy is necessarily hindered by the simplifying assumptions required to make viable their analysis.

Unfortunately, however, physical testbeds are characterized by a limited controllability of the experimental conditions, that limits the reproducibility of results, since real applications, systems, and workloads cannot be completely controlled by the experimenter. Furthermore, they are also considerably hard to implement (5, 6), since typically the entire system must be implemented, albeit at a smaller scale than a system that would go in production use. To make physical testbeds a real experimentation option, it is thus necessary to properly address these problems that, at the best of our knowledge, have no solution in the current literature.

In this paper we aim to fill this gap, and we address the issues mentioned above by proposing Prometheus, a toolkit specifically conceived to support the configuration, deployment, and use of physical testbeds suitable to perform experimental studies of resource management algorithms, and that provides a high degree of controllability and low implementation costs.

Prometheus is designed and implemented in such a way to (a) allow to quickly provision a virtualized platform featuring user-defined VMs running multi-tier applications, (b) easily incorporate into it different resource management algorithms with limited implementation efforts, (c) carry out experiments where these algorithms operate on the above applications, and (d) precisely control the workload submitted to each one of these applications.

To achieve the above goals, Prometheus provides a set of easily customizable and extensible components that supply to its users the various mechanisms that the toolkit exposes to (a) implement user-defined resource management algorithms (e.g., dynamic resource allocation to VMs),

(b) generate user-defined and application-specific workloads, and (c) coordinate the various components in such a way to ensure full controllability and reproducibility of the experimental conditions. The ability of Prometheus to successfully support the experimental evaluation is shown in our previous works (7, 8, 9), where it was extensively used to assess the performance of various management algorithms for physical and virtualized resources.

Prometheus is written in standard C++, which ensures high performance and maximum portability. In order to foster further research and to provide reproducibility, the source code of Prometheus is publicly available on (10).

In the rest of this paper we discuss the design, implementation, and use of Prometheus, and we demonstrate its possible utilization by illustrating the type of experiments that it allows to carry out. We start by describing the architecture of Prometheus (Section 2), and then we provide internal details about the design of Prometheus (Section 3). In Section 4, we illustrate how it can be used in practice to configure and deploy a physical testbed. Then, we show examples of experimental activities that can be carried on a physical testbed by means of it (Section 5). Finally, we discuss related works (Section 6), and then we conclude the paper by outlining future research work (Section 7).

2 | ARCHITECTURE OF Prometheus

The configuration, deployment, and use of a physical testbed for the experimentation with virtualized applications requires the ability of performing the following tasks in an easy, quick, and effective way:

- Interacting with the virtualization platform (to manage VMs running on the physical testbed and the virtual resources allocated them).
- Generating the workload for the virtualized applications (to stress these applications with different and peculiar usage patterns).
- Gathering observations from these applications and the virtual resources allocated them (to monitor application performance and resource utilization).
- Enforcing a resource management policy (to assess its efficacy in achieving its design goals).

The Prometheus toolkit, whose architecture is depicted in Figure 1 with solid boxes (while dashed boxes represent the components of the virtualized infrastructures it handles), has been conceived in such a way to properly carry out all the above tasks.

As can be seen from this figure, we assume that a virtualized computing infrastructure, managed by a specific Virtualization Platform, runs a set of virtualized multi-tier applications $APP_1, \ldots, APP_k$, each one providing services to a population of clients that generate a set of requests. Each tier of an application A is usually deployed in a separate VM (although this is not a requirement, as Prometheus is able to handle also cases where all the tiers of an application run in a single VM), which is hosted on anyone of the physical servers of the infrastructure. Thus, VMs running tiers of the same application may be possibly hosted by different physical servers.

The architecture of Prometheus consists instead in the following subsystems:

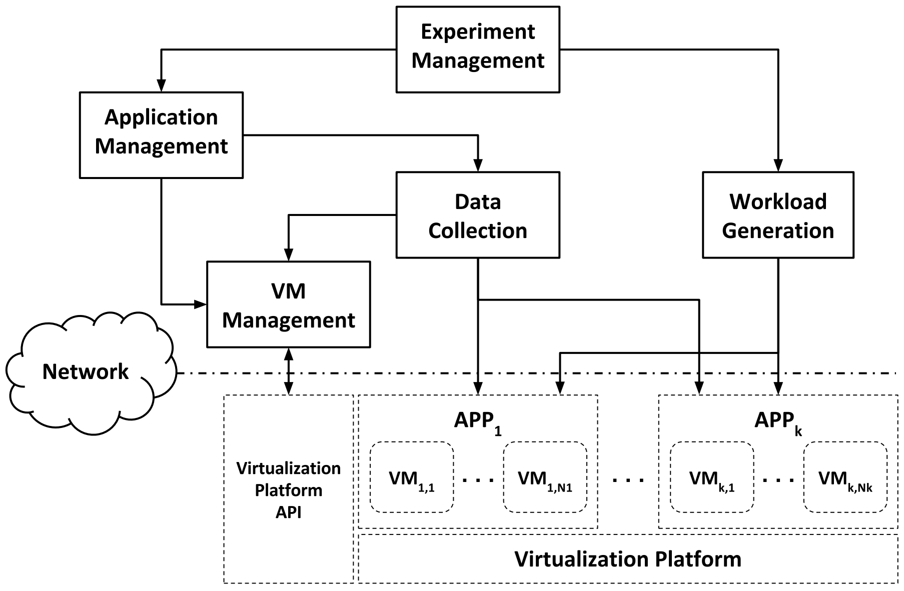

FIGURE 1 Architectural diagram of Prometheus. Dashed boxes represent components outside the scope of this paper. Arrows denote the direction of interaction between two components (i.e., $C_1 \to C_2$ means that component C_1 interacts with component C_2 to use its functionalities).

- the Experiment Management subsystem, that manages the life-cycle of the experimentations carried out over the virtualized system by interacting with both the Application Management and the Workload Generation subsystems;
- the Application Management subsystem, that implements suitable mechanisms that can be leveraged by a resource management strategy to dynamically allocate physical resource capacity to the various tiers of an application, possibly on the basis of the inputs provided by the Data Collection subsystem;
- the Data Collection subsystem, that provides various functionalities to gather and manipulate observations from the virtualized applications and the virtual resources allocated them;
- the Workload Generation subsystem, that provides various functionalities to generate the workload submitted to the applications running on the virtualized infrastructure;
- the VM Management subsystem, that enables interactions between the other subsystems of Prometheus and the VMs associated with the applications, by communicating directly with the Virtualization Platform.

Each subsystem of Prometheus consists in one or more software components, and has been designed with a pluggable and extensible architecture, meaning that its modules can be easily replaced or extended with new ones to provide new functionalities or to enhance those already available (see Section 4 for a discussion on how to carry out this extension).

Prometheus supports an experimentation model in which a set of experimental sessions are carried out sequentially, one after the other. Each session consists in the execution of one or more concurrent experiments, each one consisting in running an application for a given amount of time under a specific workload. In the same experimental session there can be multiple instances of the same subsystem, each one associated with a different experiment or implementing a different functionality. For instance, there can be multiple instances of the Workload Generation subsystem, each of which driving the workload of a different application and possibly using a different workload generator. The instances of the same subsystem that are associated with different experiments of the same session are run as separate and independent threads, that do not need to synchronize among them, or that have to synchronize rather infrequently, so as to avoid bottlenecks that would hinder scalability.

In the rest of this section, we describe the functionality provided by each subsystem of Prometheus.

2.1 | The Experiment Management Subsystem

The Experiment Management subsystem is in charge of the life-cycle management of the experimentations carried out over the target virtualized system. Specifically, it starts the workload generators (from the Workload Generation subsystem), to generate workloads for the virtualized applications, as well as the associated resource management strategies (from the Application Management subsystem), to manage these virtualized applications.

It also provides other functionalities, like execution tracking for the purpose of collecting summarizing statistics from applications and VMs (e.g., the average CPU capacity used by the VMs of the managed applications).

2.2 | The Application Management Subsystem

The Application Management subsystem is responsible for dynamically allocating physical resources to the various VMs of a given application, according to user-defined management policies. Thanks to the pluggable architecture of the Application Management, these policies are specified as `C++` plugins, that can be easily integrated in it. The pluggable architecture allows also an easy selection of the specific policy to be used in a given experiment, thus enabling experiments in which the same strategy is used to manage different applications (e.g., to assess the efficacy of a strategy to handle heterogeneous applications), or in which the same application can be managed by different strategies (e.g., to compare the performance of a strategy against other ones).

The Application Management subsystem interacts with the Data Collection subsystem, to obtain the information that are given as input to the above policies, and with the VM Management subsystem, to implement the decisions made by these strategies.

It is worth to mention that we already provide various application managers that implement the resource management policies presented in (7, 8, 9, 11, 12, 13, 14). In this way, an experimenter may easily compare her/his novel resource management policy against existing ones.

2.3 | The Data Collection Subsystem

The Data Collection subsystem provides sensors to gather measures that can be used to monitor the behavior of both the virtualized applications and the associated VMs. It also provides algorithms for data smoothing and data estimation to filter out noise from collected data, and compute statistics from them, respectively.

Specifically, this subsystem is responsible for gathering application-specific performance measures (e.g., response time or throughput) from the virtualized applications, and utilization measures for the resources allocated to the VMs. These measures can then be used by the Application Management subsystem to monitor application performance and make resource allocation decisions, such as increasing the number of virtual CPUs of some VMs when their utilization exceeds some prescribed threshold, and/or when the performance measures drop below given thresholds.

The Data Collection subsystem exposes to the other subsystems a uniform interface for the collection and retrieval of the above measures, which is independent from any specific virtualized application and virtualization platform. Hence, a different implementation of this interface must be provided for each one of the supported application and Virtualization Platforms. Clearly, in order to collect this data both the application and the Virtualization Platform must provide suitable mechanisms to do so, either natively or by properly instrumenting it at deployment time.

Currently, Prometheus provides several built-in sensors for gathering application-specific performance measures, that are based on the data logged by the workload generators it supports natively (namely, RAIN (15) and YCSB (16), see Sec. 2.4). These generators, indeed, write various application-specific data to their log files, so the corresponding sensors simply read periodically these files to extract from them the application performance measures of interest (independently from the specific application at hand). Prometheus also provides sensors for utilization measures for the resources allocated to the VMs that exploit the libvirt (17) API (and, in some cases, also some peculiarities of the operating system running in a VM) to get utilization measures for the virtual resources allocated to the VMs.

Finally, as already mentioned, Prometheus provides also the implementation of several data smoothing and estimation algorithms, including those for exponential smoothing (18) and for incremental quantile estimation (19, 20).

2.4 | The Workload Generation Subsystem

The Workload Generation subsystem is in charge of the workload generation for the applications hosted by the virtualized infrastructure. It consists of several pluggable components that provide different workload generation strategies through a common interface. The need of having different strategies is twofold.

First, a workload generation strategy is usually tied to a specific application (or to a class of applications) because each application defines unique workload characteristics and usage patterns. For instance, the operations that can be performed by users of an e-commerce Web application are usually very different from the ones of a data-analytics application (21). Thus, in this subsystem, multiple workload generators can run simultaneously, each one of them driving the workload of a different application.

Second, the same application may face different (time-varying) usage patterns that depends on specific parameters, like the number of its clients, the rate at which clients submit requests, and the mix of the requested operations, just to name a few. To assess the effectiveness of a given resource management strategy, it is important to vary such parameters in a controlled way. However, a single workload generator may not be able to model all such characteristics. For instance, burstiness and self-similarity are two workload characteristics that may have a serious impact on application performance (22) but, unfortunately, many workload generators do not model these characteristics or they focus on only one of them (23). Thus, the Workload Generation subsystem allows the experimenter to inject different load patterns into the same application by simply replacing the workload generator for that application with another one, implementing a different strategy or modelling different workload characteristics.

As mentioned before, in Prometheus we already support two widely-used workload generators, namely RAIN (15) and Yahoo! Cloud Serving Benchmark (YCSB) (16). These generators provide various workload generation strategies and are able to drive the workload for a large variety of real-world applications.

2.5 | The VM Management Subsystem

The VM Management subsystem is responsible for the management of the VMs run by the Virtualization Platform.

All the interactions between the subsystems of Prometheus and the VMs are carried out by this subsystem by means of the Application Programming Interface (API) exposed by the Virtualization Platform. Typical interactions include starting, stopping, restarting VMs, migrating VMs from a physical server to another one, changing the allocation of physical resources allocated to the VMs, and querying their status.

This subsystem hides the details about the real virtualization system used by exposing a common interface to the other subsystems of Prometheus. This way, Prometheus can support multiple Virtualization Platforms by simply replacing the actual concrete implementation of this interface with another one that provides the mechanisms to interact with a different virtualization system.

In Prometheus, we support several Virtualization Platforms by means of libvirt, which provides a hypervisor-agnostic virtualization API and supports the most common hypervisors like Xen (24), VMware ESXi (25) and KVM (26). Also, through the libvirt daemon (i.e., a server-side daemon that runs on the same physical machine of the hypervisor), we support remote interactions with the hypervisor, so that Prometheus and the Virtualization Platform can run on different machines.

3 | DESIGN OF Prometheus

In this section, we provide details about the internal design of Prometheus. In particular, in Section 3.1, we describe the classes composing the core of Prometheus, and in Section 3.2, we show the main interactions among them.

3.1 | Core Classes of Prometheus

Figure 2 shows the class diagram of Prometheus in Unified Modeling Language (UML) notation (27). [1] This diagram shows only the key classes of Prometheus (and their static relationships) that are used to model the core functionalities of the subsystems described in Section 2.

In the class diagram of Figure 2 , a class is represented with a box whose label is the class name. In particular, we distinguish between a concrete class (denoted by a box with a label in regular font) and an abstract class (denoted by a box with a label in italic font). In the following, unless

[1] In Figure 2 , we replaced underscore characters with spaces to enhance readability.

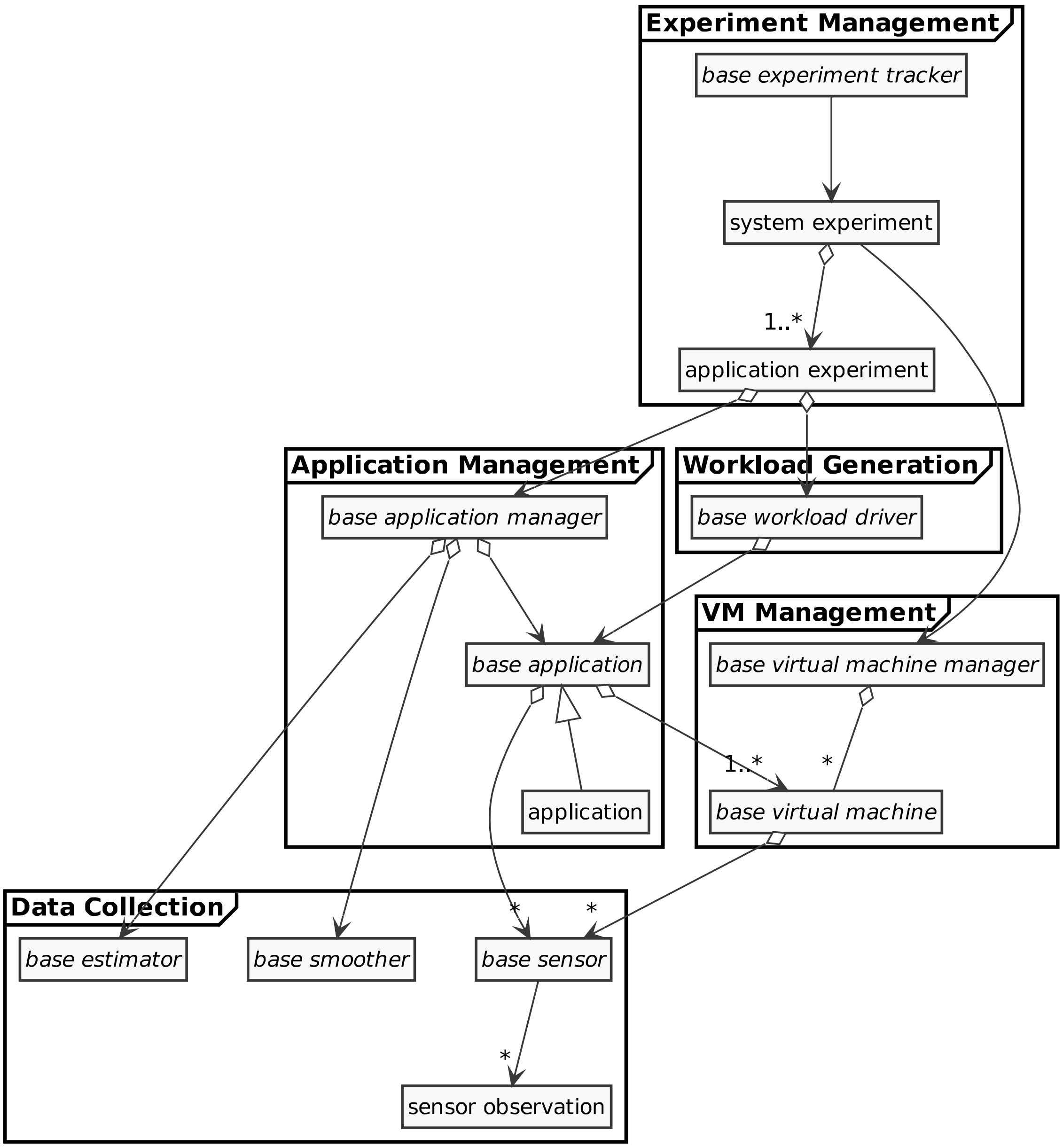

FIGURE 2 UML class diagram for the core classes of Prometheus.

explicitly stated, the terms "concrete class" and "class" are used interchangeably. In the same diagram, a static relationship among two classes is denoted by a line whose style varies according to the type of the UML relation.

For instance, from Figure 2 , we note that the `application` class is a concrete subclass of the `base_application` abstract class, and that an instance of this latter class may contain zero or more instances of the `base_sensor` abstract class, to which it communicates unidirectionally. Also, we observe that an instance of the `base_application` abstract class is in turn contained by an instance of the `base_application_manager` abstract class (to model the relationship that an application manager manages the resources assigned with a specific application) as well as by an instance of the `base_workload_driver` abstract class (to model the relationship that a workload driver generates the workload for a given application).

The diagram of Figure 2 also shows which subsystems of Prometheus these core classes belong to and thus how these subsystems are related to each other. For instance, the VM Management subsystem contains two abstract classes, namely `base_virtual_machine_manager` and `base_virtual_machine`, and depends on the Data Collection subsystem because an instance of `base_virtual_machine` contains zero or more instances of `base_sensor` (from the Data Collection subsystem), each one of them can be used to collect VM-specific data (e.g., the utilization of the allocated virtual resources).

To extend the functionality of a subsystem, concrete classes must implement its interface, which consists of all the abstract classes it contains. For instance, to support a new virtualization platform, concrete classes must implement both `base_virtual_machine_manager` and `base_virtual_machine`.

In the rest of this section, we provide, for each subsystem, an overview of the responsibility of the classes presented in Figure 2 .

3.1.1 | Classes of the Application Management Subsystem

The `application` Class.

The `application` class inherits from the `base_application` abstract class (see below) to provide an implementation suitable for modelling generic virtualized applications.

The `base_application` Class.

The `base_application` abstract class models an instance of a virtualized application as a set of VMs (e.g., one for each application tier). If multiple instances of the same application are running in the virtualized infrastructure, each one of them must be associated with a separate instance of this class.

The `base_application_manager` Class.

The `base_application_manager` abstract class represents a resource management strategy for a virtualized application. Typically, an instance of this class periodically collects data from the monitored application and from the associated VMs, and enforce the resource management policy it implements.

3.1.2 | Classes of the Data Collection Subsystem

The `base_estimator` Class.

The `base_estimator` abstract class models data estimation algorithms like, for instance, incremental quantile estimation. It can be used to compute an estimate of a given quantity based on the collected data. This class is typically used by instances of the `base_application_manager` class (Section 3.1.1) to compute an incremental estimate of the performance measure of interest from data collected from the monitored application.

The `base_sensor` Class.

The `base_sensor` abstract class models the concept of sensors gathering some type of raw measurements. In Prometheus, this class is used to model sensors for both application-specific performance measures (e.g., to collect response time measures from a given application) and VM-specific measures (e.g., to collect the utilization values of the virtual CPUs allocated to a VM).

The `base_smoother` Class.

The `base_smoother` abstract class models data smoothing strategies like, for instance, exponential smoothing. It can be used to filter out noise from data to capture relevant patterns. This class is typically used by instances of the `base_application_manager` class to smooth the collected "noisy" data (e.g., the utilization of a virtual resource) before using them to enforce the implemented management policy.

The `sensor_observation` Class.

The `sensor_observation` class models an observation collected by a sensor through an instance of the `base_sensor` class. An observation is characterized by three attributes, specifically:

- a timestamp (expressed as Unix epochs), representing the date and the time an observation has been collected;
- a textual label (e.g., used for tagging an observation according to its type or to the operation that generated it);
- a numeric real value, representing the collected value.

3.1.3 | Classes of the Experiment Management Subsystem

The `application_experiment` Class.

The `application_experiment` class models an application experiment (which consists in running an application for a given amount of time under a specific workload) and is in charge of starting the workload generator for the associated application and of periodically invoking the application manager for the same application. Thus, this class is coupled with an instance of the `base_workload_driver` class (Section 3.1.5), to inject a specific workload to the associated application, and with an instance of the `base_application_manager` class (Section 3.1.1) – and, hence, with an instance of the `base_application` class – to manage the resources of the associated application.

The `base_experiment_tracker` Class.

The `base_experiment_tracker` abstract class is responsible for tracking an experimental session represented by an instance of the `system_experiment` class (see below). For instance, a concrete classes may extend this abstract class to monitor the whole experimental session and thus to show, at the end of the session, a report containing summary statistics.

The `system_experiment` Class.

The `system_experiment` class drives the concurrent execution of one or more application experiments, each one of them registered to this class as an instance of the `application_experiment` class.

3.1.4 | Classes of the VM Management Subsystem

The `base_virtual_machine` Class.

The `base_virtual_machine` abstract class models a generic VM. It provides functionality for managing and querying a VM like, for instance, starting and gracefully shutting down a VM, or telling if a VM is currently running. It also provides methods for managing and querying virtual resources allocated to the associated VM like, for instance, setting/getting the maximum amount of memory or of the total physical CPU capacity that can be allocated to a VM.

The `base_virtual_machine_manager` Class.

The `base_virtual_machine_manager` abstract class models a generic VM hypervisor. This class provides functionality for querying both the hypervisor and the physical host where the hypervisor runs. For instance, there are functions for getting information about the status of the hypervisor (e.g., if the remote connection to the hypervisor is still alive) or about the physical host underneath (e.g., the host name or the maximum number of supported virtual CPUs).

3.1.5 | Classes of the Workload Generation Subsystem

The `base_workload_driver` Class.

The `base_workload_driver` abstract class represents a workload generator for virtualized applications.

3.2 | Interactions among Core Classes

The class diagram of Figure 2 only shows static relationships among the core classes of Prometheus. In this section, we show the dynamic behavior of instances of the above classes.

In Prometheus, class instances (hereafter, objects for short) may collaborate in various ways according to the scenario where they interact. One of the key scenarios is the execution of an experimental session to run a set of virtualized applications under specific workloads and particular resource management policies. We present the interactions for this scenario by means of the UML sequence diagram (27) shown in Figure 3 .

In this diagram, the interaction begins with `client` (i.e., a user of Prometheus) that tells `sys_1` (a previously configured instance of the `system_experiment` class) to start the experimental session by calling its `run()` method. Consequently, `sys_1` concurrently starts its previously registered instances of the `application_experiment` class, corresponding to participants `exp_1` and `exp_2` in the figure, by asynchronously calling for each one of them the `run()` method.

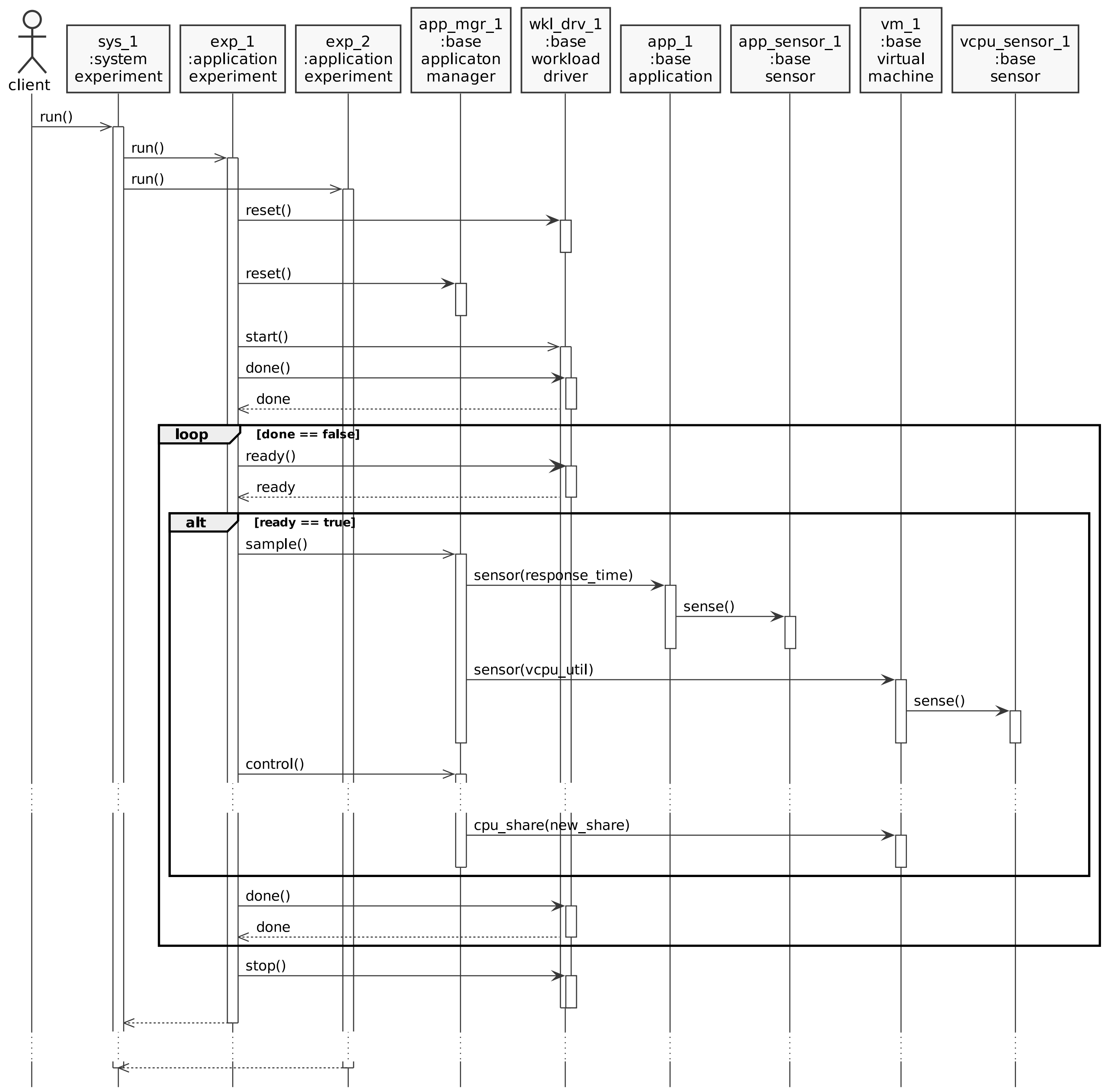

FIGURE 3 UML sequence diagram for the experimental session use-case in Prometheus.

When `exp_1` starts, (1) it resets the state of its associated workload generator, represented by the participant `wkl_drv_1`, and application manager, represented by the participant `app_mgr_1` (in the figure, this operation is shown as the `reset()` message sent to both participants), (2) it concurrently starts the workload generation (in the figure, see the `start()` asynchronous message sent to `wkl_drv_1`), and (3) it loops until workload generation is done to manage the associated application in face of the generated load (in the figure, see the `loop` combined fragment and the loop condition `done == false`).

At each iteration of the loop combined fragment, exp_1 checks if workload generation has reached a steady phase (in the figure, see the ready() message). If so (i.e., in the figure, when condition ready == true of the alt combined fragment is satisfied), it performs two concurrent tasks: (1) at every sample interval (a parameter previously defined by client) it tells app_mgr_1 to sample observations from either the managed application or from each associated VM or from both of them (in the figure, see the sample() message and those generated during its activation, like the sense() message sent by participant vm_1 to participant vcpu_sensor_1 to collect CPU utilization from a specific VM), and (2) at every control interval (another parameter previously set by client) it tells app_mgr_1 to enforce its resource management policy which, in this example, decides to change the CPU share of the VM represented by participant vm_1 by the quantity new_share (see the control() message and those generated during its activation, like the cpu_share(new_share) message sent by app_mgr_1 to vm_1).

When workload generation is done (i.e., in the figure, when condition done == false in the loop combined fragment is no longer satisfied), exp_1 tells wkl_drv_1 to stop the workload generation (in the figure, see the stop() message), and then terminates the run() operation by returning to sys_1.

Similar communications takes place concurrently for exp_2, but concerning different participants than those involved with exp_1. However, we do not report such interactions to keep the figure simple.

4 | PHYSICAL TESTBED IMPLEMENTATION WITH Prometheus

Configuring and deploying a physical testbed for the experimentation with virtualized applications is a complex and time-consuming task, that requires the integration of many software components that need to interact among them in non-trivial ways.

Prometheus has been specifically designed and implemented in such a way to reduce, as much as possible, these complexity and implementation costs thanks to the focus on reusability, so that any change in one of these components (e.g., to support different virtualization platforms or applications, or to add new resource management strategies) will not impact on the other ones.

This is achieved thanks to the adoption of a modular and extensible architecture (as discussed in Section 2), so that each one of the various tasks that need to be performed is carried out by a software component which is logically separated from the other ones, and whose implementation can be easily replaced or extended to support different functionalities.

In order to illustrate how Prometheus can be used in practice, in this section we discuss how to configure and deploy a testbed, by using as a case study the instantiation of Prometheus that we exploit in (9) to experimentally evaluate a resource management strategy named FCMS.

Prometheus is implemented as a set of classes, that define the interface of the various software components of each subsystem. To implement a given component, it is necessary to suitably implement these interfaces, that in turn requires to develop subclasses of the base classes defined in Prometheus. For the sake of readability, in this section we do not report the definition of base classes and interfaces (that are, instead, discussed in Section 3), but we limit ourselves to indicate the sub-classing operations, and to list the methods that must be implemented, using a simplified C++-like pseudo-code in which we omit implementation details like template parameters, class constructors and destructors.

The rest of this section is organized as follows. In Section 4.1, we present the case study we use throughout the whole section. In Section 4.2, we present how to implement the VM Management subsystem to support virtualization platforms. In Section 4.3, we discuss how to implement the Workload Generation subsystem to add workload generation strategies. Finally, in Section 4.4, we show how to implement the Data Collection subsystem to add data collection functionalities. In Section 4.5, we show how to implement the Application Management subsystem to add resource management strategies.

4.1 | The FCMS Case Study

FCMS (9) is a resource management strategy aimed at improving server consolidation while, at the same time, satisfying the Service Level Objectives (SLOs) of multi-tier cloud applications exposed to time-varying and bursty workloads.

With FCMS, each application tier is associated with a tier controller that periodically computes the amount of CPU capacity and of memory capacity to be allocated to the tier in order to meet the application SLO. In particular, at each activation, the tier controller (a) collects application-performance measures (as defined by the SLO) as well as CPU utilizations and memory demands from the corresponding tier, and (b) according to these collected data and to the SLO specification, it computes the new values of the CPU capacity and memory capacity to assign to the tier.

To assess the capability of FCMS to achieve its design goals, as well as to compare it against its state-of-the-art alternatives, we have configured and deployed a testbed by instantiating Prometheus, as shown in Figure 4, where we indicate (using rounded boxes) extensions of the various subsystems reported in Figure 1 (for instance, the box labeled *FCMS* inside *Application Management* denotes the extension of the Application Management subsystem that implements the FCMS resource management strategy).

As shown in the figure, the testbed relies on the Xen virtualization platform (24), whose credit scheduler and balloon driver are used to provide non-work-conserving CPU scheduling and dynamic memory management, respectively. Any interaction between Xen and Prometheus takes place through the libvirt daemon by means of the libvirt virtualization API, that has been integrated in Prometheus as an extension of the VM Management subsystem (see the *libvirt* box in Figure 4). The Application Management subsystem uses this libvirt component to set the CPU and memory capacity of the various VMs, which is also used by the Data Collection subsystem to harvest the CPU and memory utilization values for the various VMs (see the *vCPU Sensor* and *RAM Sensor* boxes in Figure 4, respectively).

To generate time-varying and bursty workloads for the applications used in our experimentation (and deployed on the testbed) we use the RAIN toolkit, that we have integrated in Prometheus as an extension of the Workload Generation subsystem (see the box *RAIN* in Figure 4).

Data collection is instead carried out by means of various sensors (as shown in the *Data Collection* box in Figure 4), that include extensions implementing both resource-specific (*vCPU Sensor* and *RAM Sensor*) and application-specific (*Response Time Sensor*) sensors. The former ones are implemented by suitable calls to the VM Management subsystem, while the latter one is implemented by an extension that reads response time observations from the log files generated by RAIN. The Data Collection subsystem has been also extended with the *t-digest Estimator* and the *SES Smoother* extensions, that provide algorithms (20, 18) for the estimation of the SLO-defined percentile of the response time and for the smoothing of the resource utilizations from the collected observations.

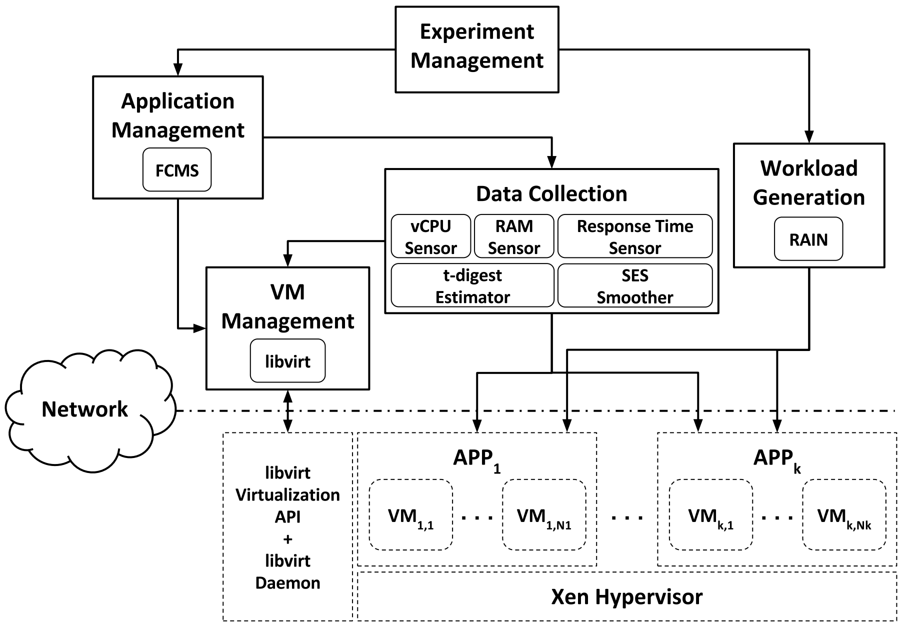

FIGURE 4 Architectural diagram of Prometheus as used in (9). Rounded solid boxes denote extensions to Prometheus's subsystems. Dashed boxes represent components outside the scope of this paper. Arrows denote the direction of interaction between two components (i.e., $C_1 \rightarrow C_2$ means that component C_1 interacts with component C_2 to use its functionalities).

4.2 | Supporting Virtualization Platforms

When implementing a physical testbed for virtualized applications, it is necessary to interact with the virtualization platform in order to manage the VMs hosting the applications and the virtual resources allocated to them. Typical interactions comprise the life-cycle management of the VMs (e.g., to start and stop a VM) and of the virtual resources allocated to the VMs (e.g., to increase and decrease the number of virtual CPUs allocated to a VM), as well as the retrieval of state information of the physical machine hosting these VMs (e.g., to know limits about the capacity of physical resources).

To do so, it is necessary to interact with the hypervisor, typically through the API provided by the virtualization platform or by an intermediate software layer.

In Prometheus, we encapsulate this functionality in the VM Management subsystem, so that supporting a virtualization platform simply requires implementing the interface of this subsystem to provide the required functions. In particular, this corresponds to implement a subclass of the `base_virtual_machine` class, and one of the `base_virtual_machine_manager` class (see Section 3.1.4).

For instance, the integration of the *libvirt* toolkit requires the implementation of the `virtual_machine` and `virtual_machine_manager` classes as concrete subclasses of the `base_virtual_machine` and `base_virtual_machine_manager` classes, whose details are shown in Figure 5 .

From Figure 5 , we see that the `virtual_machine` class must implement methods for managing and querying a VM. For instance, the method `do_start()` commands the hypervisor to start the VM, the method `do_shutdown()` commands the operating system running inside the VM to shut down itself, and the method `do_is_running()` tells whether the VM is currently running. In addition, the concrete subclass must implement methods for managing and querying the virtual resources allocated to a VM. For instance, the method `do_max_memory()` sets or returns the

```
class virtual_machine: public base_virtual_machine {
    void do_start() { ... } // Starts this VM
    void do_shutdown() { ... } // Commands the VM to gracefully shut down itself
    bool do_is_running() const { ... } // Tells whether this VM is running
    void do_max_memory(unsigned long value) { ... } // Sets the maximum amount of memory this VM can use
    unsigned long do_max_memory() const { ... } // Returns the maximum amount of memory this VM can use
    void do_cpu_share(double value) { ... } // Sets the CPU share for this VM
    double do_cpu_share() const { ... } // Returns the CPU share assigned to this VM
    void do_memory_share(double value) { ... } // Sets the memory share to this VM
    double do_memory_share() const { ... } // Returns the memory share assigned to this VM
    ...
};
class virtual_machine_manager: public base_virtual_machine_manager {
    bool do_alive() const { ... } // Tells whether the connection to the hypervisor is valid
    unsigned long do_max_supported_num_vcpus() const { ... } // Returns the maximum number of virtual CPUs
    std::string do_hostname() const { ... } // Returns the name of the physical node hosting the hypervisor
    ...
};
```

FIGURE 5 Implementation outline for the integration of the libvirt virtualization toolkit (see files `libvirt/virtual_machine.hpp` and `libvirt/virtual_machine_manager.hpp` in (10) for more details).

maximum amount of memory that can be allocated to a given VM, while the method `do_cpu_share()` sets or returns the fraction of the total capacity of the CPUs allocated to a given VM.

Also, from the same figure, we see that the `virtual_machine_manager` class must implement methods for managing and querying the hypervisor as well as the physical host where it runs. For instance, the method `do_max_supported_num_vcpus()` returns the maximum number of virtual CPUs that can be allocated to the VMs running on the same physical machine, while the method `do_alive()` tells whether the connection to the real hypervisor is still valid.

For implementing all these methods, we rely on the various functions provided by the libvirt API. For instance, in the `do_start()` method of the `virtual_machine` class, we call the `virDomainCreate()` function of libvirt by passing as parameters the internal handles used by libvirt to represent the connections to the VM to be started and to the hypervisor that hosts this VM.

4.3 | Adding Workload Generators

Adding a workload generator to the testbed requires to implement the interface of the Workload Generation subsystem (see Section 2.4).

In the case of *RAIN*, which is the generator used in our example testbed, this entails to implement the `workload_driver` class as a concrete subclass of `base_workload_driver` (see Section 3.1.5), as shown in Figure 6 . In this class, the RAIN toolkit is run as an external process and any interaction with RAIN take places by sending signals to this process and by reading log files generated by RAIN. These log files are monitored by a separate thread to keep track of the progress of the RAIN's workload generation (e.g., to know if it has already reached the desired load level).

As shown in Figure 6 , a concrete subclass of `base_workload_driver` must implement the following methods:

```
class workload_driver: public base_workload_driver {
    void do_reset() { ... } // Resets the state of the workload driver
    void do_start() { ... } // Starts workload generation
    void do_stop() { ... } // Stops workload generation
    bool do_done() const { ... } // Tells whether workload generation is done
    bool do_ready() const { ... } // Tells whether workload generation has reached the prescribed load level
};
```

FIGURE 6 Implementation outline for the integration of the RAIN toolkit (see file `rain/workload_driver.hpp` in (10) for more details).

- `do_reset()`: clears the workload generator's state so that it can be restarted. This method is used by Prometheus to reset the state of the workload generator for a given application before running a new experimental session. In the `workload_driver` class, we reset the state of RAIN by stopping the currently running RAIN process (if any) as well as the thread used for monitoring the RAIN's log files.
- `do_start()`: starts generating the workload for the configured application. This method is used by Prometheus to start generating the workload for a given application. In the `workload_driver` class, we start RAIN as an external process as well as the thread used for monitoring its log files.
- `do_stop()`: stops generating the workload for the configured application. This method is used by Prometheus to stop generating the workload for a given application. For the `workload_driver` class, we stop the workload generation by terminating both the RAIN process and the thread that have been launched in the `do_start()` method.
- `do_done()`: tells whether the workload generation is done. This method is used by Prometheus to know when workload generation for a given application has finished. For the `workload_driver` class, we return `true` if the RAIN process has terminated, or `false` otherwise.
- `do_ready()`: tells whether the workload generator has reached the desired load level. This method is used by Prometheus to know when it can start sampling data from application and VMs and enforcing the resource management policy. For the `workload_driver` class, we return `true` if the workload generation in RAIN has already left its ramp-up phase, or `false` otherwise. This information is provided by the thread that monitors the RAIN's log files.

4.4 | Adding Data Collection Functionality

Adding data collection functionality involves implementing the interface of the Data Collection subsystem (see Section 2.3), that entails to implement a concrete subclass of (a) the `base_sensor` class to add sensors, (b) the `base_estimator` class to add data estimators, and (c) the `base_smoother` class to add data smoothers (see Section 3.1.2).

For instance, as shown in Figure 7 , the *Response Time Sensor*, which is used in our case study to collect response time observations from a running application, is implemented by the `response_time_sensor` class as a concrete subclass of the `base_sensor` class.

From this figure, we see that a concrete class of `base_sensor` must implement the following methods:

```
class response_time_sensor: public base_sensor {
    void do_reset() { ... } // Resets the state of the sensor
    void do_sense() { ... } // Collects data
    bool do_has_observations() const { ... } // Tells whether there are observations ready to be consumed
    std::vector<observation> do_observations() const { ... } // Returns the last collected observations (if any)
};
```

FIGURE 7 Implementation outline of the response time sensor (see file `rain/sensors.hpp` in (10) for more details).

- `do_reset()`: resets the state of the sensor. This method is used by Prometheus to reset the sensor before running a new experimental session. For the `response_time_sensor` class, we clear all the previously collected data.
- `do_sense()`: gathers observations from the monitored entity. This method is invoked by Prometheus to periodically collect application-specific or VM-specific observations. For the `response_time_sensor` class, we gather response time observations from the log files generated by RAIN and store them in an internal array.
- `do_observations()`: returns the last sensed data (if any). This method is used by Prometheus to retrieve the most recently collected observations. For the `response_time_sensor` class, we return the values that we previously stored in the internal array in the `do_sense()` method.
- `do_has_observations()`: tells whether there are observations waiting to be consumed. For the `response_time_sensor` class, we return `true` if the internal array (used to hold the last collected observations) is not empty, or `false` otherwise.

As for the *t-digest Estimator* and the *SES Smoother*, which in our example testbed are used for application performance estimation and data smoothing, their implementation is outlined in Figure 8 and in Figure 9 , respectively.

In particular, from Figure 8 , we see that the *t-digest Estimator* is implemented by the `dunnings2013_tdigest_estimator` class as a concrete subclass of the `base_estimator` class which requires to implement the following methods:

```
class dunning2013_tdigest_quantile_estimator: public base_estimator {
    void do_reset() { ... } // Resets the state of the data estimator
    void do_collect(const std::vector<double>& data) { ... } // Collects the given data
    double do_estimate() const { ... } // Applies the data estimation algorithm
};
```

FIGURE 8 Implementation outline of the t-digest algorithm (see file `data_estimators.hpp` in (10) for more details).

- `do_reset()`: clears the state of the data estimator. This method is used by Prometheus to reset the data estimator before running a new experimental session. For the `dunning2013_tdigest_quantile_estimator` class, we reinitialize the internal data structures used by the percentile estimation algorithm.

- `do_collect(data)`: updates the state of the data estimator with the provided data. This method is used by Prometheus to update the data estimator when new data is available (e.g., when it is gathered from a sensor). For the `dunning2013_tdigest_quantile_estimator` class, we add the provided data to the set of values that will be used for the percentile estimation.

- `do_estimate()`: applies the estimation algorithm and returns the computed estimate. This method is used by Prometheus to get the current value of the estimate. For the `dunning2013_tdigest_quantile_estimator` class, we use the collected data to compute and return the percentile estimate.

As for Figure 9 , we note that the *SES Smoother* is implemented by the `brown_single_exponential_smoother` class as a concrete subclass of the `base_smoother` class which requires to implement the following methods:

```
class brown_single_exponential_smoother: public base_smoother {
    void do_reset() { ... } // Resets the state of the data smoother
    double do_smooth(const std::vector<double>& data) { ... } // Applies the smoothing algorithm
    double do_forecast(unsigned int t) const { ... } // Predicts the next value at time t (t >= 0)
    bool do_ready() const { ... } // Tells whether the data smoother is ready to forecast values
};
```

FIGURE 9 Implementation outline of the SES algorithm (see file `data_smoothers.hpp` in (10) for more details).

- `do_reset()`: resets the state of the data smoother. This method is used by Prometheus to reset the data smoother before running a new experimental session. For the `brown_single_exponential_smoother` class, we clear the value of the SES statistic as well as some internal flags.

- `do_smooth(data)`: feeds the smoothing algorithm with the provided data. This method is used by Prometheus to update the state of the data smoother when new data is available (e.g., when it is gathered from a sensor). For the `brown_single_exponential_smoother` class, we apply the SES algorithm to the given data.

- `do_forecast(t)`: predicts the next smoothing value at the given time `t`. This method is used by Prometheus to get the current value of the smoothed statistic. For the `brown_single_exponential_smoother` class, we return the most recent value of the SES statistic.

- `do_ready()`: tells whether the data smoother is ready to forecast values. For the `brown_single_exponential_smoother` class, we return the value of the internal flag that we set the first time the SES algorithm is applied after its last reset.

4.5 | Adding Resource Management Strategies

Adding resource management strategies requires implementing the interface of the Application Management subsystem, that entails in implementing a concrete subclass of (a) the `base_application` class to add applications, and (b) the `base_application_manager` class to add resource management strategies (see Section 3.1.1).

For instance, as outlined in Figure 10 , the *FCMS* resource management strategy used in the case study is implemented by the `fcms_application_manager` class as a subclass of the `base_application_manager` class.

```
class fcms_application_manager: public base_application_manager {
    void do_reset() { ... } // Resets the state of the application manager
    void do_sample() { ... } // Samples application-specific and/or VM-specific observations
    void do_control() { ... } // Applies the management strategy to the controlled application
};
```

FIGURE 10 Implementation outline of the FCMS resource management strategy (see file `fcms_application_manager.hpp` in (10) for more details).

Specifically, as shown in this figure, a concrete subclass of `base_application_manager` must implement the following methods:

- `do_reset()`: (re)initializes the state of the application manager so that any previously stored data used for making resource management decisions is cleaned. This method is used by Prometheus to reset the state of the application manager before running a new experimental session. In the `fcms_application_manager` class, we reset the state of the tier controllers and of their related *vCPU Sensors* and *RAM Sensors* as well as the state of the *Response Time Sensor*, and initialize the *SES Smoothers*.

- `do_sample()`: collects observations from either application-specific sensors or VM-specific sensors or both of them. This method is periodically invoked by Prometheus to collect data from the sensors associated with the monitored application and its VMs. In the `fcms_application_manager` class, we collect response time observations with the *Response Time Sensor* and we pass them to the *t-digest Estimator*. Also, we collect CPU and memory utilization data from application's VMs by means of the *vCPU Sensor* and the *RAM Sensor*, respectively, and we pass them to the respective *SES Smoothers*.

- `do_control()`: applies the resource management strategy, usually by making decisions on the basis of data collected so far in the `do_sample()` method. This method is periodically invoked by Prometheus to apply the resource management strategy to the monitored application. In the `fcms_application_manager` class, (a) we first estimate the achieved application SLO value by means of the *t-digest Estimator*, (b) then for each application's VM we predict the next CPU and memory utilization values by means of the *SES Smoothers*, and (c) finally we set these values as input to the tier controllers in order to compute the right CPU and memory shares to assign to the application's VMs for satisfying SLOs and maximizing the server consolidation level.

5 | EXPERIMENTAL EVALUATION

To demonstrate the effectiveness of Prometheus to support experimental activities involving virtualized applications, in this section we show how to employ a testbed deployed using it to carry out two distinct experimental campaigns. The first one focuses on application performance profiling, and is aimed at characterizing the performance of a virtualized application for varying levels of workload intensity and amounts of allocated resource capacity. The second one focuses instead on resource management, and is aimed at comparing the effectiveness of several resource management strategies when dealing with a set of distinct applications that compete for the same physical resources.

In the rest of this section, we first describe the setup of the physical testbed used in these experiments (Section 5.1), then we discuss the application profiling experimental campaign (Section 5.2), and finally we conclude with the resource management one (Section 5.3).

5.1 | Experimental Setup

For both the experimental campaign discussed in this section, we implement the testbed depicted in Figure 4 and described in Section 4.1, and we deploy it on two Fujitsu Server PRIMERGY RX300 S7, connected via a Gigabit Ethernet switch, each one equipped with two 2.4 GHz Intel Xeon E5-2665 processors with 8 cores and 96 GiB of RAM, and running the Linux kernel version $4.1.13$ and Xen 4.4. One of these machines (henceforth, "Host A") is used to run all the VMs corresponding to the cloud applications used in our evaluation, while the other one (hereafter, "Host B") is used to run all the software components of Prometheus.

For our experiments, we consider two applications that are representative of those that run on current cloud computing infrastructures (28), namely:

- Olio (29), a two-tier PHP-based Web 2.0 Internet application for social events modeled after the Yahoo! Upcoming service.
- RUBBoS (30), a two-tier PHP-based Web 1.0 Internet application that implements a bulletin board system modeled after an online news forum like Slashdot.org.

We deploy and run each application tier inside a separate VM, each one equipped with a single virtual CPU (vCPU).

The workload for these applications is driven by the RAIN toolkit (15), using the implementation discussed in Section 4.3. The workloads of both applications is generated by a population of N clients, each one issuing a request and waiting for the response, then sleeping for a given amount of time (the think time), and then repeating the process again (the details of the workload used in each experimental campaign are given in the corresponding subsection).

As described in Section 4, data collection is carried out by means of the built-in sensors of Prometheus, namely the *Response Time Sensor*, the *vCPU Sensor* and the *RAM Sensor*. These sensors consume very little system resources as can be observed in Table 1 , where we report the results of the profiling of 1000 invocations of each sensor, and where we note that each invocation takes, on average, less than 200 μsecs of CPU time (which is comparable with the mean time taken by a context-switch on modern computing systems (31)). Furthermore, this CPU time is consumed only occasionally because sensors are invoked at user-defined time intervals which are usually set to a value larger than the time taken by each invocation (e.g., every 2 secs, in our experiments). Finally, from the same table, we observe that the *vCPU* and *RAM Sensors* are the only sensors that may affect the performance of the hosted applications since they are the only ones that may contend resources with the VMs of the physical testbed (because they interact with the virtualization platform running on Host A). However, considering their low resource consumption, we conclude that the use of these sensors has a negligible impact on the performance of the applications hosted by the physical testbed.

TABLE 1 Mean CPU time taken by the invocation of each sensor. Values inside parenthesis denote standard deviations.

Sensor	Affected Host	CPU Time (μs)
vCPU Sensor	Host A	147.01 (237.30)
RAM Sensor	Host A	192.64 (18.16)
Response Time Sensor	Host B	167.90 (5.20)

5.2 | Application Performance Profiling

Performance profiling consists in studying how the performance of a given application is affected by changes in its workload, in the amount of resource capacity allocated to it, or both. Performance profiling has various uses, including determining the right amount of physical resources that must be allocated in order for the application to meet its performance goals, or to build a performance model that is able to dynamically predict its performance under given operational conditions.

Prometheus offers two built-in application managers to carry out application performance profiling for varying levels of workload intensity and for changing amounts of allocated resource capacity, namely the *dummy_am* and the *sysid_am* application managers.

The *dummy_am* application manager supports performance profiling as the workload changes, while keeping constant the amount of allocated resources. This application manager periodically collects application performance and resource utilization data from the managed application, without performing any resource allocation adjustment.

To give an example, let us assume we are interested in profiling the performance of RUBBoS when the number of users is increased from 100 to 200 and, finally, to 300 (while the think times of users are sampled from a random variable that is exponentially distributed with mean 7 sec). To carry out this task, we configure RAIN to generate the above workload, and we start the Experiment Management subsystem, that starts in turn the *dummy_am* manager and the RAIN extension of the Workload Generation subsystem.

The results of these profiling experiments are shown in Table 2 , where we report summary statistics for the associated response time empirical distribution, which have been computed using the data collected by *dummy_am*.

The *sysid_am* application manager instead supports application performance profiling when the amount of resources allocated to the application is dynamically changed according to a given pattern (e.g., according to a sinusoidal-like function or by generating random values from a certain

TABLE 2 Response time statistics of RUBBoS under different number of users. All VMs were equipped with 3GiB of RAM. All values are expressed in sec.

Number of	Summary Statistics for the Response Time Empirical Distribution					
Users	Min	1st Quartile	Median	Mean	3rd Quartile	Max
100	0.0000035	0.0027554	0.0066064	0.0173767	0.0173567	0.9561026
200	0.0000037	0.0037358	0.0102574	0.0376509	0.0446460	1.6610171
300	0.0000031	0.0028206	0.0082611	0.0403694	0.0452455	1.7688789

probability distribution). To do so, *sysid_am* periodically varies the amount of the capacity of the virtual resources allocated to the application's VMs according to the given pattern, and registers the effects of these changes by collecting performance and resource utilization observations from the interested application.

To give an example, let us consider the case where we want to quantify the effects on application performance of changes in the amount of RAM allocated to the VM hosting that application. As reported in the literature (32, 33), indeed, the actual dynamic memory management mechanisms used by a virtualization platform causes abrupt changes in the utilization of its vCPU, lasting a variable amount of time, when the amount of memory allocated to a VM is changed.

To quantify these effects, we run a set of experiments on the physical testbed where, by means of the *sysid_am* manager, we dynamically change the memory allocated to a VM by a fixed magnitude and collect the utilization of its vCPU and its memory. In particular, we fix the minimum (1.6 GiB) and maximum (16 GiB) size of the VM memory, and then we run a sequence of experiments in which the RAM size of the VM is first decreased from the maximum to the minimum values using increasing steps m from 10% to 90% of the maximum value, and then it is increased until the maximum value is hit using steps of the same magnitude.

The results of these experiments are reported in Table 3 , and confirm the effects reported in the literature, that is we observe that increasing or decreasing allocated memory make vCPU utilization to increase, and that the increase and its duration are proportional to the magnitude of the variation. For instance, halving the memory size (i.e., $m = 50\%$) raises the vCPU utilization up to 69.79% for 9 sec, while reducing it of just $m = 10\%$ yields a vCPU utilization of 42.97% for 3.03 sec. A similar behavior can also be observed when memory size is increased.

5.3 | Comparing resource management strategies

An activity often recurring when devising a novel resource management strategy, is the evaluation of its effectiveness and the comparison against existing alternatives. Prometheus effectively supports this activity by providing a simple mean to add a new resource management strategy (see Section 4.5), and by providing a set of built-in modules implementing alternative solutions already published in the literature, as listed in Table 4 . All these modules use the mechanisms Prometheus offers to collect application performance and resource utilization measures, as well as those to dynamically change CPU and memory capacity associated with a VM according to the implemented resource management strategy.

TABLE 3 Effects of dynamic memory management on vCPU utilization. Values inside parenthesis denote standard deviations.

Magnitude of Change m (%)	Memory Increment		Memory Decrement	
	Mean vCPU Utilization (%)	Mean Duration (s)	Mean vCPU Utilization (%)	Mean Duration (s)
10	42.28 (59.04)	2.03 (0.17)	42.97 (39.82)	3.03 (0.17)
20	52.71 (50.14)	3.00 (0.00)	59.85 (39.80)	4.00 (0.00)
30	64.16 (46.90)	4.13 (0.35)	63.64 (31.72)	6.07 (0.26)
40	66.22 (47.52)	5.00 (0.00)	64.20 (29.86)	8.00 (0.00)
50	72.21 (44.80)	6.25 (0.50)	69.79 (26.35)	9.00 (0.00)
60	79.44 (39.92)	6.00 (0.00)	69.07 (23.06)	11.33 (0.58)
70	79.52 (38.56)	7.00 (0.00)	68.30 (23.30)	13.00 (0.00)
80	79.96 (38.57)	8.00 (0.00)	69.27 (22.69)	15.50 (0.71)
90	81.74 (37.28)	9.00 (0.00)	71.32 (23.60)	16.50 (0.71)

TABLE 4 Library of available application managers in Prometheus.

Module name	**Resource manager name**	**Reference**
fuzzyqe_am	Fuzzy-Q&E	(7)
fc2q_am	FC2Q	(8)
fcms_am	FCMS	(9)
appleware_am	APPLEware	(12)
autocontrol_am	AutoControl	(13)
dynaqos_am	DynaQoS	(34)
fmpc_am	FMPC	(14)

As an example, consider an experimentation aimed at comparing the effectiveness of FCMS, APPLEware, and FMPC in maximizing the consolidation level of the physical servers running a set of cloud applications that compete for the same physical resources without impacting on application performance goals.

For this experimental campaign, we consider the RUBBoS and Olio applications, using the settings reported in Section 5.1, and we configure the testbed to allocate competing VMs on the same CPU core using vCPU pinning. For all VMs, the maximum amount of RAM is set to 3 GiB, but the applications managers are free to reduce this value when they deem it necessary. The performance goal of each application is set by fixing a threshold value R for the response time (0.37 sec and 0.17 sec for Olio and RUBBoS, respectively), and by requiring that the 95^{th} percentile $\widehat{R}$ of the observed response time values is below this threshold. We compute these thresholds in a way to make them achievable with the physical resources available in our physical testbed and a nontrivial target to meet. Specifically, we profile each application by running it in isolation under the maximal workload intensity, we collect its response time observations, and then we compute the SLO value as the 95^{th} percentile of the empirical distribution built from these data (see (8, 9) for more details).

The following metrics are used to quantify the effectiveness of the resource managers:

- Mean CPU Capacity (MCC): the average CPU capacity allocated to each application's VM.

- Mean Memory Capacity (MMC): the average memory capacity allocated to each application's VM.
- Percent Error $\widehat{E}$: relative percentage change between the observed 95^{th} percentile $\widehat{R}$ of the response time values and the threshold R, that is:

$$\widehat{E} = 100\frac{\widehat{R} - R}{R} \tag{1}$$

These metrics are used to rank the approaches as follows: if $\widehat{E} > 0$, then the approach is violating the performance goal, so it is placed at the bottom of the ranking. If instead $\widehat{E} \leq 0$, then the lower MCC and MMC values, the better the approach (i.e., among two approaches that both meet the SLO, the best one is that which uses less CPU and memory capacity, thus allowing a higher consolidation level).

The results of the experiments are reported in Tables 5 and 6 , for RUBBoS and Olio, respectively.

TABLE 5 Results for the RUBBoS application.

	Performance goal		MCC		MMC	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)	Web (%)	DB (%)
FCMS	**Yes**	$\mathbf{-3.90}$	**45.84**	**54.52**	**86.05**	**37.68**
APPLEware	No	2183.36	7.91	7.24	67.79	19.60
FMPC	No	3079.03	18.99	25.25	63.58	45.35

TABLE 6 Results for the Olio application.

	Performance goal		MCC		MMC	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)	Web (%)	DB (%)
FCMS	**Yes**	$\mathbf{-11.95}$	**33.10**	**16.01**	**73.32**	**89.08**
APPLEware	No	1053.62	22.85	7.04	54.61	73.15
FMPC	No	526.77	22.23	14.28	53.32	68.49

From these results, we note the FCMS is able to meet the performance goal of both applications, while APPLEware and FMPC both fail to do so. The interested reader is referred to (9) for a more detailed analysis of the effectiveness of these resource management strategies.

6 | RELATED WORKS

Although in the last years many papers discussing resource management algorithms – and evaluating their performance using physical testbeds – have been published in the literature (see (4) for a thorough survey), the problem of devising suitable toolkits simplifying the deployment, configuration, and use of these testbeds has not been directly addressed.

Most of the above works (e.g., (35, 36, 37) base their experimental evaluation on a purposely-built testbed, that is usually assembled ad-hoc for the specific case, but it cannot be easily customized, or extended for algorithms and systems different from those considered for that case.

In contrast, Prometheus is a generic, extensible toolkit that allows to easily configure, deploy, and use a physical testbed for any combination of virtualization platforms, applications, workloads, and resource management algorithms.

Other efforts have been devoted to the development of individual components needed to deploy and use a physical testbed, such as workload generation (e.g., C-Meter (38) and RAIN (15)), or VM management (e.g., Amazon Web Services (39), Google Cloud Platform (40), and OpenStack (41)). However, none of these systems integrates into a single toolkit all the components needed to configure, deploy, and use a physical testbed for the experimental evaluation of resource management algorithms, as instead done by Prometheus.

7 | CONCLUSION

Physical testbeds are usually considered the most accurate option for experimentally assessing the effectiveness of a resource management algorithm for virtualized infrastructure, as well as to compare it against state-of-the-art alternatives. However, using a physical testbed is a complex and time-consuming task, because of the difficulty in implementing, configuring, and deploying the various software components needed by the testbed, and of the complexity in running the various experiments in a controlled way.

To address the above issues, in this paper we have proposed Prometheus, a highly modular and extensible toolkit specifically devised to support the configuration, deployment, and use of physical testbeds for the experimental evaluation of resource management algorithms, which provides low implementation costs and high controllability. To show the potential and the features of Prometheus, we have presented various practical examples where it has been used to carry out experimental activities on a physical testbed.

As future works, we plan to extend Prometheus to support other virtualization and cloud platforms (like OpenStack (41), Google Cloud Platform (40), and Amazon Web Services (39)), by interacting with them either directly through their native APIs or by means of intermediate software layers like (42). Furthermore, we intend to add new functionalities to Prometheus as those for the monitoring and the management of physical servers, thus enabling it to also support the experimental assessment of server-level resource management policies, like those designed to reduce energy consumption in a physical infrastructure (43, 44).

References

[1] Armbrust Michael, Fox Armando, Griffith Rean, et al. A View of Cloud Computing. *Commun. ACM.* 2010;53(4):50–58.

[2] Yi Shanhe, Li Cheng, Li Qun. A Survey of Fog Computing: Concepts, Applications and Issues. In: Proceedings of the 2015 Workshop on Mobile Big Data:37–42; 2015; Hangzhou, China.

[3] Rosenblum Mendel. The Reincarnation of Virtual Machines. *Queue.* 2004;2(5):34–40.

[4] Jennings Brendan, Stadler Rolf. Resource Management in Clouds: Survey and Research Challenges. *J. Netw. Syst. Manage..* 2015;23(3):567–619.

[5] Anglano Cosimo, Canonico Massimo, Guazzone Marco, et al. Peer-to-Peer Desktop Grids in the Real World: The ShareGrid Project. In: Proc. of the 2008 8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID):609–614; 2008; Lyon, France.

[6] Anglano Cosimo, Canonico Massimo, Guazzone Marco. The ShareGrid Peer-to-Peer Desktop Grid: Infrastructure, Applications, and Performance Evaluation. *Journal of Grid Computing.* 2010;8(4):543–570.

[7] Albano Luca, Anglano Cosimo, Canonico Massimo, Guazzone Marco. Fuzzy-Q&E: Achieving QoS Guarantees and Energy Savings for Cloud Applications with Fuzzy Control. In: 2013 International Conference on Cloud and Green Computing:159–166; 2013; Karlsruhe, Germany.

[8] Anglano Cosimo, Canonico Massimo, Guazzone Marco. FC2Q: exploiting fuzzy control in server consolidation for cloud applications with SLA constraints. *Concurrency and Computation: Practice and Experience.* 2015;27(17):4491–4514.

[9] Anglano Cosimo, Canonico Massimo, Guazzone Marco. FCMS: A fuzzy controller for CPU and memory consolidation under SLA constraints. *Concurrency and Computation: Practice and Experience.* 2017;29(5):e3968–n/a.

[10] Anglano Cosimo, Canonico Massimo, Guazzone Marco. Repository for the source code of Prometheus Available from: https://github.com/sguazt/prometheus. Accessed: July 26, 2017; .

[11] Guazzone M., Anglano C., Canonico M.. Energy-Efficient Resource Management for Cloud Computing Infrastructures. In: Proc. of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science:424–431; 2011; Athens, Greece.

[12] Lama Palden, Guo Yanfei, Jiang Changjun, Zhou Xiaobo. Autonomic Performance and Power Control for Co-located Web Applications in Virtualized Datacenters. *IEEE Transactions on Parallel and Distributed Systems.* 2016;27(5):1289–1302.

[13] Padala Pradeep, Hou Kai-Yuan, Shin Kang G., et al. Automated Control of Multiple Virtualized Resources. In: Proc. of the 4th ACM European Conference on Computer Systems:13–26; 2009; Nuremberg, Germany.

[14] Wang Lixi, Xu Jing, Duran-Limon H. A., Zhao Ming. QoS-Driven Cloud Resource Management through Fuzzy Model Predictive Control. In: Proc. of the 2015 IEEE International Conference on the Autonomic Computing (ICAC):81-90; 2015.

[15] Beitch Aaron, Liu Brandon, Yung Timothy, Griffith Rean, Fox Armando, Patterson David A.. *RAIN: A Workload Generation Toolkit for Cloud Computing Applications.* Technical Report UCB/EECS-2010-14: University of California at Berkeley; 2010.

[16] Cooper Brian F., Silberstein Adam, Tam Erwin, Ramakrishnan Raghu, Sears Russell. Benchmarking Cloud Serving Systems with YCSB. In: Proc. of the 1st ACM Symposium on Cloud Computing:143–154; 2010; Indianapolis, Indiana, USA.

[17] Red Hat, Inc. . *Libvirt Virtualization API.* Available from: http://libvirt.org. Accessed: July 26, 2017; 2017.

[18] Holt Charles C.. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting.* 2004;20(1):5–10.

[19] Jain Raj, Chlamtac Imrich. The P^2 algorithm for dynamic calculation of quantiles and histograms without storing observations. *Communications of the ACM.* 1985;28(10):1076–1085.

[20] Dunning Ted. *t-Digest: an algorithm for computing extremely accurate quantiles.* Available from: https://github.com/tdunning/t-digest. Accessed: July 26, 2017; 2015.

[21] Feitelson Dror G.. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press; 2015.

[22] Mi Ningfang, Casale Giuliano, Cherkasova Ludmila, Smirni Evgenia. Burstiness in Multi-tier Applications: Symptoms, Causes, and New Models. In: Proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware:265–286; 2008; Leuven, Belgium.

[23] Yin J., Lu X., Zhao X., Chen H., Liu X.. BURSE: A Bursty and Self-Similar Workload Generator for Cloud Computing. *IEEE Transactions on Parallel and Distributed Systems.* 2015;26(3):668–680.

[24] Barham Paul, Dragovic Boris, Fraser Keir, et al. Xen and the Art of Virtualization. In: Proc. of the 19th ACM Symposyum on Operating Systems Principles:164–177; 2003; Bolton Landing, NY, USA.

[25] VMware, Inc. . *VMware vSphere Hypervisor.* Available from: http://www.vmware.com/products/vsphere-hypervisor.html. Accessed: July 26, 2017; 2017.

[26] Kivity Avi, Kamay Yaniv, Laor Dor, Lublin Uri, Liguori Anthony. KVM: the Linux virtual machine monitor. In: Proc. of the Ottawa Linux Symposium, vol. 1: :225–230; 2007; Ottawa, Canada.

[27] OMG et al. . *OMG Unified Modeling Language (OMG UML), Version 2.5.* Specification formal/2015-03-01: Object Management Group, Inc.; 2015. Available from: http://www.omg.org/spec/UML/2.5. Accessed: July 26, 2017.

[28] Ferdman Michael, Adileh Almutaz, Kocberber Onur, et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS):37–48; 2012.

[29] Apache Software Foundation . *Olio Web 2.0 Toolkit.* Available from: https://incubator.apache.org/projects/olio.html. Accessed: July 26, 2017; 2011.

[30] OW2 Consortium . *RUBBoS: Rice University Bulletin Board System.* Available from: http://jmob.ow2.org/rubbos.html. Accessed: July 26, 2017; 2004.

[31] Li Chuanpeng, Ding Chen, Shen Kai. Quantifying the Cost of Context Switch. In: Proc. of the 2007 Workshop on Experimental Computer Science (ExpCS'07); 2007; San Diego, California.

[32] Lloyd W., Pallickara S., David O., Lyon J., Arabi M., Rojas K.. Performance Implications of Multi-tier Application Deployments on Infrastructure-as-a-Service Clouds: Towards Performance Modeling. *Future Generation Computer Systems.* 2013;29(5):1254–1264.

[33] Liu H., Jin H., Liao X., Deng W., He B., Xu C.. Hotplug or Ballooning: A Comparative Study on Dynamic Memory Management Techniques for Virtual Machines. *IEEE Transactions on Parallel and Distributed Systems.* 2015;26(5):1350–1363.

[34] Rao Jia, Wei Yudi, Gong Jiayu, Xu Cheng-Zhong. QoS Guarantees and Service Differentiation for Dynamic Cloud Applications. *IEEE Transactions on Network and Service Management.* 2013;10(1):43–55.

[35] Babaioff Moshe, Mansour Yishay, Nisan Noam, et al. ERA: A Framework for Economic Resource Allocation for the Cloud.. In: WWW (Companion Volume):635-642; 2017.

[36] Garg Saurabh Kumar, Yeo Chee Shin, Buyya Rajkumar. Green Cloud Framework for Improving Carbon Efficiency of Clouds.. In: Euro-Par (1):491-502; 2011.

[37] Marozzo Fabrizio, Talia Domenico, Trunfio Paolo. A Cloud Framework for Parameter Sweeping Data Mining Applications.. In: CloudCom:367-374; 2011.

[38] Yigitbasi Nezih, Iosup Alexandru, Epema Dick H. J., Ostermann Simon. C-Meter: A Framework for Performance Analysis of Computing Clouds. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18-21 May 2009:472–477; 2009.

[39] Amazon Web Services, Inc. . Amazon Web Services Available from: https://aws.amazon.com. Accessed: July 26, 2017; .

[40] Google, Inc. . Google Cloud Platform Available from: https://cloud.google.com. Accessed: July 26, 2017; .

[41] OpenStack . OpenStack: Open Source Cloud Computing Software Available from: https://www.openstack.org. Accessed: July 26, 2017; .

[42] Canonico Massimo, Monfrecola Davide. CloudTUI-FTS: A user-friendly and powerful tool to manage Cloud Computing Platforms. *EAI Endorsed Transactions on Cloud Systems.* 2016;16(6).

[43] Guazzone Marco, Anglano Cosimo, Canonico Massimo. Exploiting VM Migration for the Automated Power and Performance Management of Green Cloud Computing Systems. In: Huusko Jyrki, Meer Hermann, Klingert Sonja, Somov Andrey, eds. *Energy Efficient Data Centers: First International Workshop, E2DC 2012*, Lecture Notes in Computer Science, vol. 7396: Berlin, Heidelberg: Springer Berlin Heidelberg 2012 (pp. 81–92).

[44] Guazzone Marco, Anglano Cosimo, Sereno Matteo. A Game-Theoretic Approach to Coalition Formation in Green Cloud Federations. In: Proc. of the 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID):618–625; 2014; Chicago, IL, USA.

How cite this article: C. Anglano, M. Canonico, and M. Guazzone (2018), Prometheus: a flexible toolkit for the experimentation with virtualized infrastructures, *Concurrency Comput.: Pract Exper., 30(11):e4400*, 2018. https://doi.org/10.1002/cpe.4400