A Multi-Level Monitoring Approach for the Dynamic Management of Private IaaS Platforms

Rodrigo Garcia-Carmona[1], Félix Cuadrado[2], Álvaro Navas[1], Antonio Celorio[1], Juan C. Dueñas[1]
[1]Departamento de Ingeniería de Sistemas Telemáticos, ETSI Telecomunicación,
Universidad Politécnica de Madrid, Spain
[2]School of Electronic Engineering and Computer Science,
Queen Mary University of London, United Kingdom, London
rodrigo@dit.upm.es, felix.cuadrado@eecs.qmul.ac.uk, {anavas, acelorio, jcduenas}@dit.upm.es

Abstract

To properly implement a dynamic management system in a cloud environment it is imperative to have complete, holistic and up to date information of the environment itself. However, current private cloud IaaS monitoring solutions are not sufficiently evolved to fulfill this task. Such a monitoring system needs to contemplate all stakeholders and dimensions of a cloud environment, while at the same time hiding irrelevant information. We propose an information model that offers a multi-level view of a private IaaS cloud, separates the concerns between consumers and providers, and enables the correlation of the collected data to have a complete picture of the environment. Using it we have designed a monitoring architecture, and built it using agents based on open-source monitoring platforms. We show the feasibility of our approach through two sample scenarios deployed into a private IaaS cloud. This monitoring provides the capabilities needed to be used in an autonomous management system.

Keywords: cloud computing, monitoring, autonomous systems, information model, IaaS.

1 Introduction

The widespread use of virtualization has transformed distributed computing platforms. The landscape is currently dominated by a few major public cloud computing providers, which exploit economies of scale to provide widely available elastic platforms. Public clouds enable rapid deployment with minor initial investment costs, and the capability to immediately scale up and down runtime services as needed. However, the control and management of sensible data and services could mandate a company to use its own private cloud, in order to complement public infrastructures.

Cloud computing research has focused on optimizing the usage of virtualized platforms, in order to minimize costs, improve energy efficiency and optimize performance, leading to more sustainable cloud solutions. The preferred approach to this matter are autonomous systems, which can dynamically change to adapt to the mutating needs of the environment, while at the same time optimizing the use of resources and therefore achieving a more sustainable cloud environment. This approach requires an efficient monitoring of the cloud environment: without trustworthy, complete, holistic and up to date information an autonomous system cannot reason. It will not know how to adapt to changes if it does not know that these changes are actually happening.

Unfortunately, private cloud solutions are significantly less evolved than the public offerings, and their current monitoring capabilities are very limited. There is no solution that retrieves all the key monitoring data needed for a meaningful autonomous management.

In this paper we aim to bridge this gap, by proposing an information model that represents all the relevant information from the environment. A private cloud involves multiple actors that need to monitor the runtime at different abstraction levels (e.g. the infrastructure provider will be concerned about physical resources partitioning, and availability, while an application provider will need to monitor the health of the virtual instances, and identify performance bottlenecks to scale up the components as needed). The presented information model represents the fundamental computing resources (processing, memory, storage, network), at both physical and virtual levels, in order to support the complete spectrum of management use cases.

Additionally, the model abstractions are designed to support multi-tenant usage of the monitoring information. Management agents concerned only with virtual machines and applications will operate with a partitioned subset of the information, while vertical analysis of the infrastructure is supported through explicit traceability between virtual and physical-level information.

In order to show the feasibility of our model we propose a monitoring architecture that instruments the multiple levels of the private cloud environment, and builds a global model of the runtime status. We

detail the technical choices we followed for implementing the architecture, and demonstrate its functionality with the execution of two representative scenarios (a Map/Reduce job, and a three-tier web application).

This paper is structured as follows: The next section analyzes the required monitoring information for a cloud environment, and proposes a thorough information model. Section 3 presents a monitoring system architecture that instruments both physical and virtual machines, creating a unified view of the state of the cloud. We follow up with our steps to validate our proposal, presenting a sample implementation of our architecture in a private cloud, and the monitoring results collected from scenarios with different types of applications. Section 5 compares our main contributions with previous research. The paper finishes with a discussion on the main conclusions extracted after this work, and the identification of the main lines for future work.

2 Information Model

2.1 Monitoring Information Requirements

A cloud ecosystem involves several stakeholders, each with different viewpoints and concerns about the environment. The monitoring system must retrieve the information that is relevant for all these stakeholders. Therefore, as an initial approach to the definition of the types of information monitored, we analyze the high-level cloud architectures proposed by the DMTF (Distributed Management Task Force) [1] and NIST (National Institute of Standards and Technologies) [2] and which stakeholders they contemplate.

The DMTF reference architecture identifies three actors: cloud service consumer, cloud service provider and cloud service developer. All of them revolve around the concept of service, which is the core element of this standard, and the exact role of an actor depends on the service model (IaaS, PaaS or SaaS) the service it is related to lies. The NIST architecture shares the same layer-dependent approach, and two of its actors (cloud consumer and cloud provider) have its exact correspondent in the DMTF architecture, but this standard ditches the cloud developer role while adding three other: cloud broker, cloud auditor, and cloud carrier. The roles and use cases of actors in both architectures depend on the service model the actor is involved with.

Since we are designing a monitoring system for an IaaS cloud, we will center our analysis on this layer. This leaves outside of the scope of this work the roles of the cloud carrier from the NIST architecture, and the cloud developer from the DMTF architecture. Also, for the sake of the information retrieved, we consider cloud brokers to be cloud consumers. This leaves us with the following stakeholders: *IaaS provider*, *IaaS provider auditor*, *IaaS consumer*, and *IaaS consumer auditor*. Observing them we decided to divide the information retrieved in two different levels: the *provider level* and the *consumer level*.

At the consumer level, actors are only concerned with the elements they are allowed to manipulate, that is, virtual resources. Moreover, each stakeholder should only be able to monitor the resources related to the services he has been entrusted. A consumer (or consumer auditor) will not have permission to access another consumer's resources.

Since the service model is IaaS, the elements contained in the consumer level are the virtual appliances and their corresponding virtual resources (computing, storage or network). To correctly manage them, a consumer will need both these resources' characteristics (like number of processing cores or amount of system memory) and their status at the present time and during the past. Furthermore, the virtual resources dedicated to a particular consumer could be used to power several

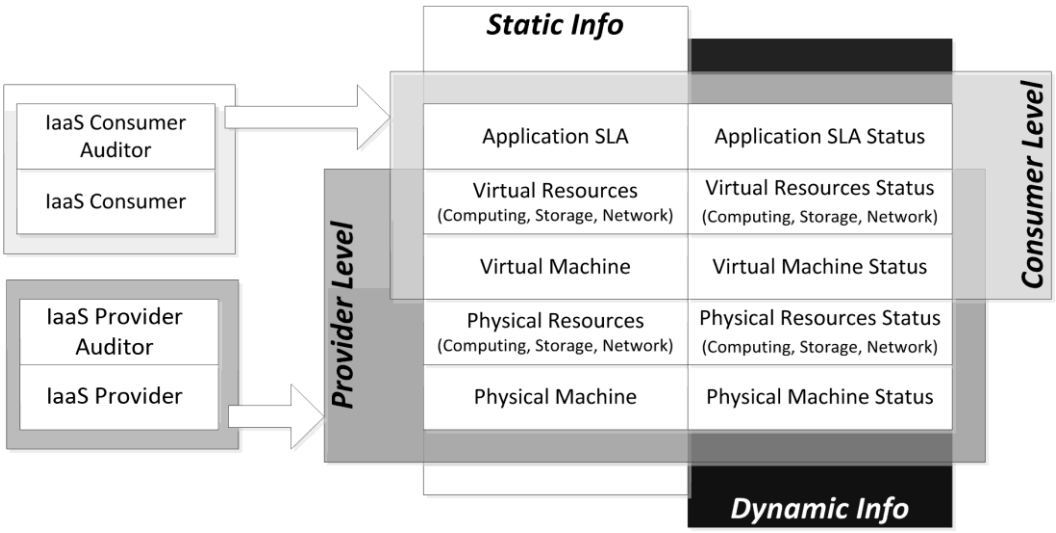

Figure 1: IaaS Cloud Monitoring Stakeholders and Information

applications, so this level should also include the status of the high level applications deployed over the virtual resources. This information must accurately represent if the application's SLAs are being respected, and therefore be application specific. This suggests a partitioning of the information retrieved, not only between stakeholders, but also between cloud applications.

For the provider level we must consider the tasks and use cases of the providers and provider auditors. Both need information about the structure and status of the physical environment of the cloud they are working with: the physical resources that comprise the environment and their current and past states, together with how they are assigned to consumers. These resources are the physical nodes of the IaaS clouds and their computing, storage and network capabilities. This level must also include the virtual machines deployed on top of the physical nodes, and the share of resources assigned to them. With this information, the stakeholders at the provider level should be able to correlate between the status of the physical and the virtual resources.

Figure 1 depicts the two levels, the information they contain and the relevant stakeholders.

2.2 Cloud Standard Models

The current state of cloud computing standards is far from ideal. Although there have been some initiatives that try to bring different vendors and institutions together to define a common set of specifications, the fact is that most of the push in the advancement of cloud computing has come from the industry, with successful platforms becoming *de facto* standards such as the Amazon Web Services API. There are several standardization initiatives for providing a generic cloud management specification, but the models are still at an early stage. We present in this section the existing alternatives, and evaluate their maturity and suitability for capturing the monitoring information of IaaS clouds.

We have already introduced the two competing standards for the highest levels of management, from DMTF and NIST. Both define similar architectures and none has a clear advantage over each other. The problem of the overlap of these architectures has been confronted by Samba [3], who has defined a hybrid model that manages to bridge both. Still, although these standards offer a high-level reference architecture for the management of clouds, they are not specific enough to be implemented as is, with gaps in the lower levels that need to be filled with other specifications.

There is an effort from the DMTF to provide a lower level actual management interface in the form of the Cloud Infrastructure Management Interface (CIMI) specification [4]. This interface will provide a resource model that represents the artifacts needed to reproduce the use cases presented in the DMTF architecture for the IaaS service model, but is still in the development stage. Also, a CIM (Common Information Model) representation of these artifacts has been proposed, but it is not completely defined yet.

OCCI (Open Cloud Computing Interface) [5] began as a set of APIs for remote management of IaaS clouds and is being expanded to define a complete cloud management specification. The current standard is composed of OCCI Core (the core elements), OCCI Infrastructure (the IaaS domain elements) and OCCI HTTP Rendering (a REST management API).

The OCCI information model is based on the concept of resource, which represents any manageable element from the cloud infrastructure. A resource can be associated with others through the link element. OCCI defines a kind attribute for classifying both types of elements, and brings the concept of mixins, that can be used to add more capabilities at run-time. Finally, actions define what management operations can be applied to each element. The OCCI Infrastructure extension defines three classes of IaaS resources: compute, storage and network. All these resources also have a lifecycle associated with them.

The OCCI model provides a solid foundation for modeling cloud infrastructure resources, as well as the associated management APIs. However, these abstractions don't provide a complete view of the infrastructure. For instance, virtual and physical nodes are not represented, and although mixins enable the specification of details, OCCI does not have the vocabulary needed for a complete correlation between several information levels, or to establish the relationship between the cloud applications and the resources they use.

The OVF (Open Virtualization Format) [6] is a standard for specifying virtual images in a way that can be interpreted by multiple virtualization hypervisors. While it has been adopted by several private IaaS solutions, its scope does not cover the information required for monitoring IaaS clouds.

Also, since there is a relationship between the already presented CIMI and CIM (Common Information Model), there is an extension of CIM named System Virtualization Profile [7] that is concerned with the modeling of virtualized systems. However, this model is highly extensive, with too much information not relevant to the viewpoint of cloud consumers and providers. For its use inside a

cloud computing monitoring system some parts of this standard should be ignored. Yan et al. [8] have followed this approach, but their work was more concerned with providing a neutral API than with defining a complete cloud environment model.

Therefore, even if there are some standards that serve well for virtualization solutions, they are not fully extended to offer a comprehensive vision of a cloud computing environment. On the other hand, the specific cloud standards are either too high level or unable to correlate information between the multiple dimensions of a cloud environment.

2.3 Proposed Information Model

In this section we present our proposed information model that takes as reference the abstractions from the previously analyzed standards, and represents all the relevant monitoring information for all the IaaS cloud stakeholders. In order to simplify the model complexity, and ease its processing by autonomic management elements, we have made three assumptions about the IaaS cloud platform.

Our first assumption is that the physical CPUs won't be split into more virtual cores than the number of physical cores they have. This is to avoid having several virtual machines sharing one core, a situation that can have a big impact on performance, as shown by Sukwong et al. [9]. This is also important to enable the cloud manager to have full control over the resources.

We also assume that we know the throughput of physical network interfaces. In standard Ethernet LANs this is not the case, since all machines connected to the same network share the bandwidth of the channel. However, the architecture of modern datacenters is designed (thanks to a careful topology) with the aim of achieving uniform high capacity [10]: the maximum rate of a node to node traffic is limited only by the network interface cards.

Finally, we assume that there is only one stakeholder fulfilling the role of IaaS provider in a given cloud. It is very rare to have clouds with several administrators and, in fact, most private IaaS cloud solutions do not offer this feature.

Also, for the scope of this paper we don't model the application SLA and application SLA status information of the consumer level. We have concentrated our work in the other information types: virtual and physical resources and their status.

With these assumptions, and taking some ideas from the previously presented standards, we have defined an information model suitable for monitoring IaaS clouds. From OCCI we took the concept that every monitored element is a *resource*, with a lifecycle associated. We have streamlined the lifecycle down to three states: *active*, *inactive* and *suspended*. Active resources are running (and are

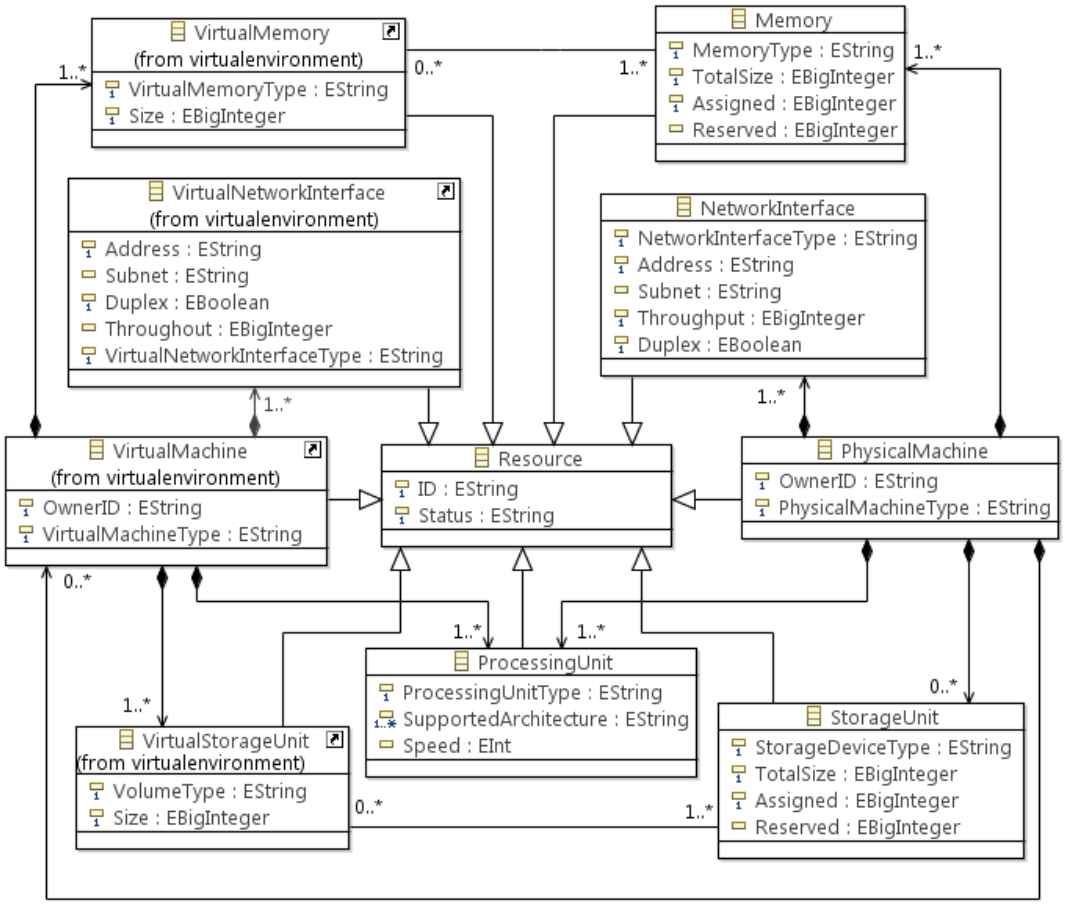

Figure 2: Cloud Environment Model

assigned if they are virtual); inactive is the state of a resource that has been turned off or otherwise completely deactivated; and suspended resources are not currently available but could be activated on a very short notice. Modeling suspended resources eases the implementation of energy efficiency policies (by turning off a resource or simply suspending it, depending on its expected future usage).

It could seem strange to have a *suspended* status for elements like memory, which can't actually be physically put into an energy saving state. However, there have been calls for this kind of behavior to be implemented [11], if truly energy effective computing wants to be achieved. Therefore, we have considered this possibility.

The environment model is divided into physical environment and virtual environment abstractions. This division has been defined so the IaaS consumers and IaaS consumer auditors can work only with their slice of the virtual environment, while the IaaS provider and any IaaS provider auditors monitor the physical resources. Figure 2 shows the most relevant elements of both models, and the relationships between them.

The top-level resources of the model are *physical machine* and *virtual machine*. A physical machine hosts one or more virtual machines. Both types of machines are characterized by the resources they offer. Following the example of OCCI, we split the resources contained in both physical and virtual machines into several broad types: *processing units*, *memory*, *network interfaces* and *storage units* for physical machines; and *processing units*, *virtual memory*, *virtual network interfaces* and *virtual storage units* for virtual machines. We have separated storage units and memory into different elements, even if they are very similar in concept. Their latencies and the way they are used are different enough to justify this division. The processing unit element is the same for both kinds of machines. This derives from our previous assumption of not the division of a physical CPU in more cores than it physically has.

Memory and storage units have a relationship with their virtual counterparts. These relationships model that these virtual elements are composed of one or more physical resources of the corresponding type. This could also happens in the opposite direction, with several virtual memory resources assigned to the same physical memory or several virtual storage units occupying the same physical storage. This fact is also indicated in the assigned and reserved attributes of memory and storage unit resources. The former represents the capacity assigned to the virtual elements making use of the resource, while the latter models the capacity reserved for the proper working of the physical

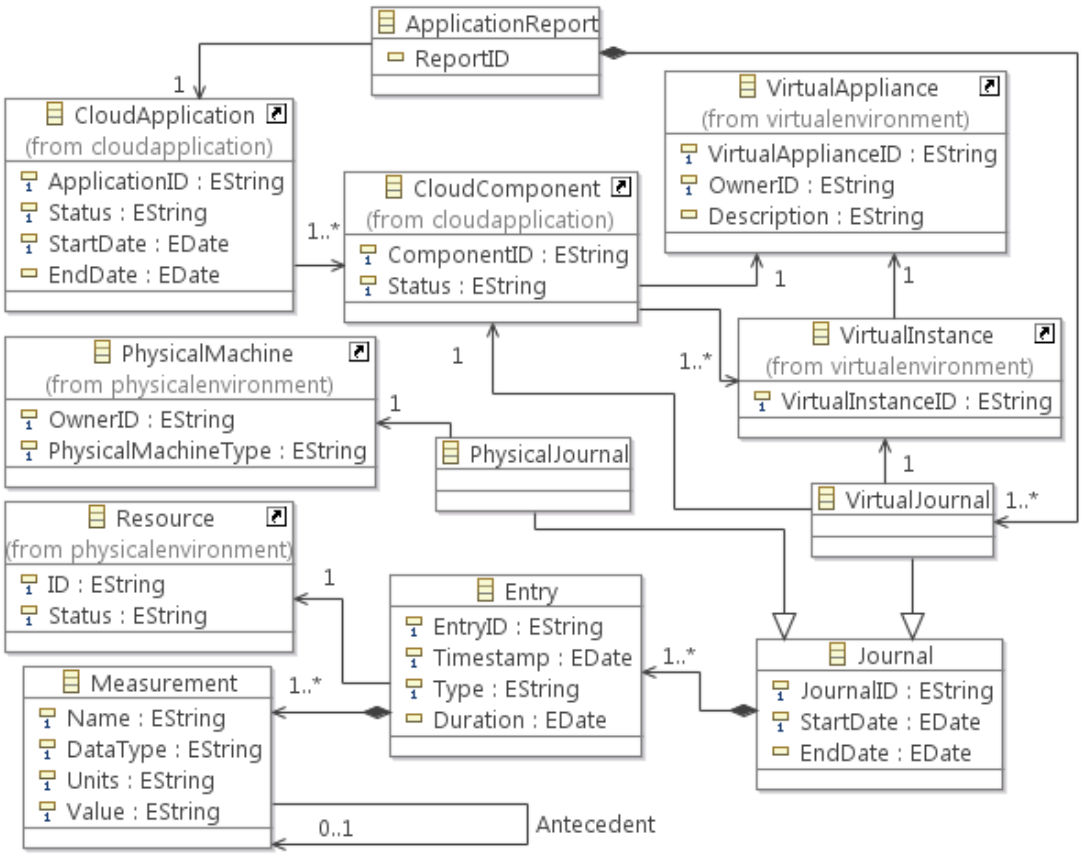

Figure 3: Dynamic Information Model

machine. It is important to note that a relationship like this does not exist between the physical and virtual network interfaces, since there is no correspondence between them.

The model can also represent secondary processing units and their associated memory banks, such as GPUs used for stream computing and scientific applications or other special purpose hardware like digital signal processors. In particular, the use of GPUs is getting more popular with time, and some cloud providers (like Amazon) already offer them in some of the virtual instances. This is done through the processing unit and memory resources, changing their type to the appropriate one (like "GPU" or "DSP").

The previous abstractions characterize the static information of the cloud environment, but for the dynamic information (the status of resources) we present a complementary model. Figure 3 shows its fundamental abstractions.

To fulfill the separation of information between several consumers in the consumer level, we have articulated this model around the idea of *cloud applications*: any set of software components that can be deployed into the cloud and together offer a high-level functionality. For instance, a Hadoop cluster or a three tier web application. Each cloud application is composed of several *components*, realized in the form of one or more *virtual instances*.

The basic element of the dynamic model is the *entry* element, which represents a set of related *measurements* gathered for a *resource* in a specific instant of time (defined by a timestamp and, optionally, duration). The structure of entries and measurements are inspired in the elements of the CIM Metrics model [12], a standard designed to represent the information captured from applications at runtime.

The entries corresponding to a resource are compiled into a *journal*. Depending on the monitored resource, entries are collected into journals of two different types: *virtual journals* for a virtual instance realizing a particular component (a part of a cloud application), and *physical journals* for a physical machine. Virtual journals are aggregated into *application reports*, which collect all the information relevant to a specific cloud application.

The combination of several features of our model (the separation of entries in two journal types, the links that tie each entry to a specific resource, and the relationships between resources themselves) enable the correlation between the two information levels (consumer and provider) and the two information types (static and dynamic). This correlation is fundamental, since it makes possible the creation of a true global image of the environment, while at the same time separating between the consumer and provider concerns.

3 Monitoring

We propose in this section a generic monitoring architecture that can instrument a private cloud environment and create instances of our information model to report on the infrastructure status. We describe the architecture in two stages. First we explain the different agents that constitute the system, and specify their responsibilities and relationships. Once the main concepts have been clearly explained, we provide further detail on the technical concerns of the instrumentation agents.

3.1 Agents Infrastructure

The monitoring architecture collects information from the physical and virtual levels that were presented in the previous section. In both cases, information about the status of the machines and resources is obtained.

The information is retrieved from the physical and virtual machines by pairs of instrumentation agents named *monitoring clients* and *monitoring servers*. The combination of both types of elements adapt the specific management APIs and models from each type of managed resource, and offer a common interface to the remaining elements of the monitoring architecture.

An important point when discussing agents is the opaqueness of the monitoring, that is, the advantages and disadvantages of black box vs. white box monitoring. With black-box monitoring, the information about the surveyed element is obtained from observing only its external responses to stimulus, with the monitoring system being oblivious at what happens inside it. This is the easiest way of implementing a monitoring system, since no changes need to be performed in the monitored elements. However, the information provided is less complete or up to date than with white box monitoring methods, where sensors are placed inside the managed element, with access to its inner workings. The information needed to populate our information model can only be provided by a white-box approach, and therefore we decided to instrument the resources monitored this way.

Monitoring clients instrument the monitored resources. Clients retrieve the information directly, and relay the data to the servers. Monitoring clients should not put a big burden in the monitored resources, so they need to be as lightweight as

possible. Because of this, the only component of the monitoring client is the client of the chosen monitoring technology itself. The proposed architecture does not add to the clients any functionality not already provided by the specific monitoring technology, and the only requirement they need to fulfill is to be able to communicate with another host. Its implementation details and the communication protocol chosen is not specified in the architecture. Examples of existing technologies suitable for this task are CollectD, Nagios or Zabbix. There are also similar technologies, like MBeans or MAF, capable of monitoring application SLAs, even though they are not limited simply to monitoring tasks.

Another technology-specific component (with whom the monitoring clients communicate) lies inside the monitoring servers, and its task is to aggregate the information retrieved by all the clients paired with it. On top of this, monitoring servers also have the responsibility of processing the raw data retrieved, translate it into the information model, maintaining it correctly populated and assessing the relevance of the information received. This task is performed by another part of the monitoring server, the *model transformation module*.

Going back to the idea of the monitoring levels, the proposed architecture defines two types of components that manage the information from either the physical or the virtual level. These elements use monitoring servers for collecting the runtime information. The Infrastructure Manager (from now on IM) monitors all the physical machines that constitute the infrastructure cloud. For the virtual machines and resources, each cloud application running inside the cloud is monitored by an Application Manager (from now on AM), which monitors the set of components (and their virtual instances) that comprise the application. An AM is oblivious to the physical level, or the other applications deployed in the cloud. However, while we are considering the physical and virtual levels separately, AMs also relay to the IM in order to populate the model with virtual resources information.

AMs are virtual appliances themselves, and are deployed into the same environment as the applications they manage. Thanks to this, they can be monitored using the same infrastructure, agents and information model already deployed for the cloud environment itself.

Figure 4 shows how a simple private cloud environment is monitored by the presented architecture. The bottom half of the picture depicts two physical nodes with several virtual machines that constitute three virtual applications. Each virtual machine has one or more monitoring clients connected to the corresponding monitoring servers of an AM. In a similar way, the IM has one or more monitoring servers, each paired with a monitoring client deployed inside a physical machine of the environment.

Communication between the IM and AMs takes place through a REST interface. We define two modes of communication that support the traditional push / pull operation modes. Therefore,

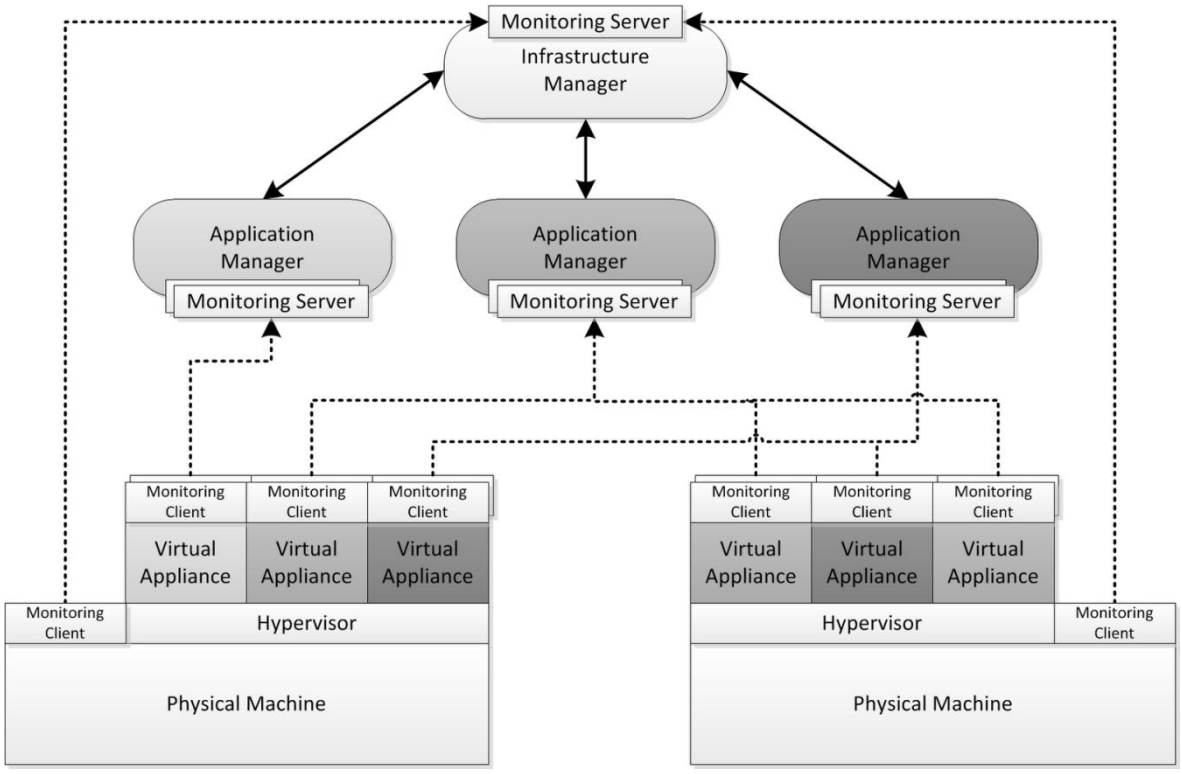

Figure 4: Monitoring Architecture

communication can be started both ways: with the IM asking the AMs for new data or a particular information from the past, or with the AMs relaying potentially important information the instant it is captured instead of waiting for a periodic update. Figure 5 shows an example of both modes of interaction.

3.2 Monitoring Instrumentation

In this section we present how we have solved the instrumentation of our implementation of the proposed architecture and the technologies we have used to implement it.

Regarding the information monitored, both physical and virtual levels must collect the same kind of data: the basic status and usage of the resources of the system. In particular (and following the model): processing core, memory, network, and storage usage. Therefore, we must choose a set of agents with the capability for retrieving this data and a server-client architecture that enables their use as our monitoring servers and clients.

CollectD is an open source UNIX daemon that collects system performance data periodically. It has a modular design, with several plugins, each built to collect data from a specific application or service, or to write data to a particular storage medium. One of these, the network plugin, can listen on the network for other CollectD daemons transmitting their data, or send its collected data via the network.

Our data requirements translated into a CollectD daemon with the following plugins: network, cpu, memory, df (for disk usage) and interface. The network plugin is configured to relay the data collected to the monitoring servers. On every monitoring server a CollectD daemon is also installed, with the network and rrdtool (for round robin databases) plugins active. These CollectD daemons listen to the network for incoming data and store said information in a round robin database (one for each monitored machine). We chose round robin databases because they are specially designed for retrieved monitoring data. This structure is used both for virtual and physical levels.

Deploying these agents in a cloud environment presents several problems, since they must be installed in virtual appliances. To deploy a CollectD agent on every virtual machine of the cloud, there are two possible solutions:

- To have CollectD preinstalled on every virtual machine.
- To install and configure CollectD at boot time.

Modifying every image is not a trivial task and is a considerable burden for the developer responsible of preparing the virtual image. On top of that, the configuration of the CollectD daemon is still needed, so we opted for the second solution. Although easier, this approach still presents some difficulties that need to be solved.

Installing software on virtual instances during boot time can be achieved by several means. The easiest way to perform this task is through cloud-init, which comes preinstalled in most Linux distributions

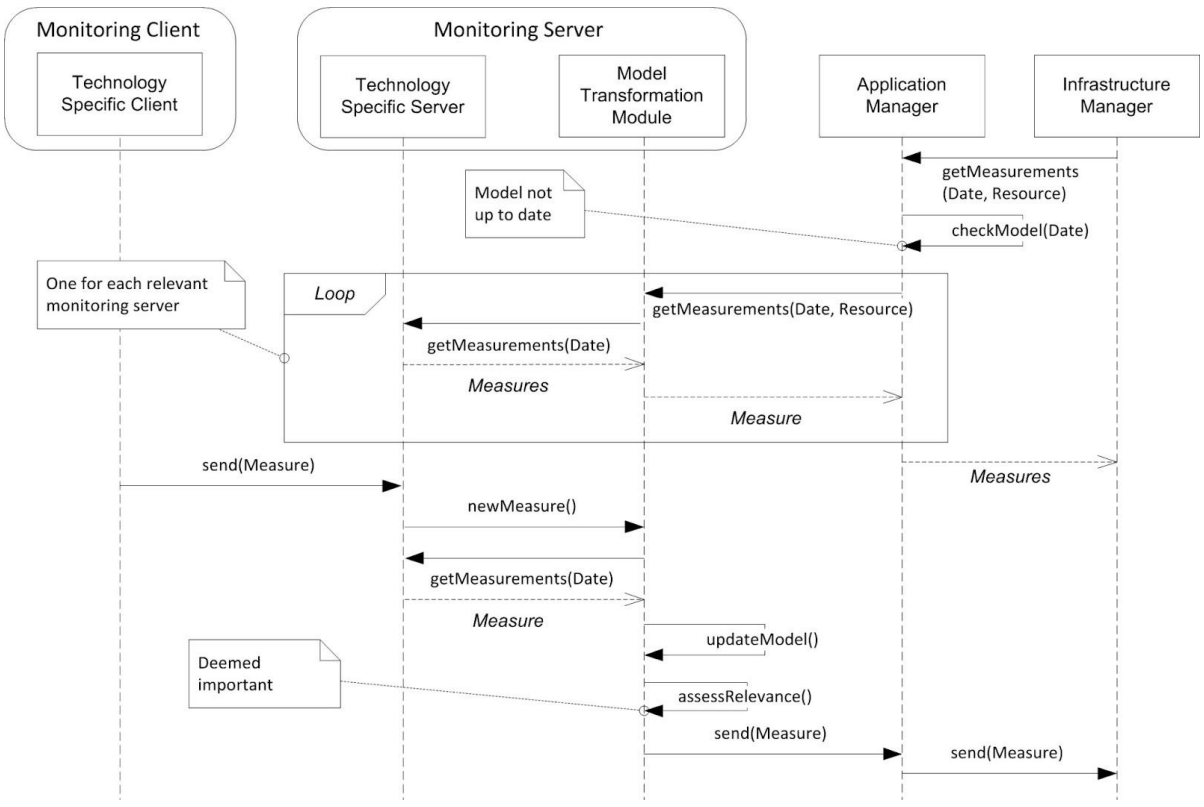

Figure 5: IM and AMs Communication Modes

prepared for cloud environments. A cloud-init script can be passed on boot to the instance to install a CollectD agent, activate the appropriate plugins, and configure the network parameters. This approach is limited to the selectable virtual images to those with cloud-init preinstalled, and is only applicable to the cloud infrastructure managers that support it (EC2, OpenStack and Eucalyptus, at the very least).

In cloud infrastructures where cloud-init is not available, it is usual to at least support executing a user script at boot time in the virtual machines. This script can be used to install and configure CollectD. Regarding compatibility with the instrumented operating systems, CollectD is available for Linux (the SO we used) and other UNIX flavors, but there is also a Windows product (SSC Serv) that sends data in the same format as CollectD.

3.3 Retrieved Data

Using CollectD to populate our model we defined a core measurement for each resource type, and several other measurements that offer a more detailed view. The core measurement is derived from the rest. Regarding processing units, the core measurement was CPU usage as a percentage. This was collected via the CPU plugin, which measurements the amount of time spent by the CPU in several states: Idle (idle and without and outstanding I/O request), User (executing code at the user level), Nice (executing at the user level with nice priority), System (executing at the system level), IOwait (idle with an outstanding I/O request), SoftIRQ (servicing a software interrupt), IRQ (servicing an interrupt), and Steal (spent on other OSs, usually virtualized ones).

CPU usage is the addition of all these parameters with the exception of Idle. It is important to note that the Linux kernel collects statistics measured in jiffies (units of scheduling) instead of percentage. On most Linux kernels there are 100 jiffies to the second, but depending on both internal and external variables, this might not be always true. However, it is a reasonable approximation.

For both virtual and physical memory resources we defined the percentage used as the core measurement, which was built using the following measurements, captured using the CollectD memory plugin: Used (used by applications), Buffered (block devices' caches), Cached (used to park file data), and Free (not being used).

The Linux kernel tends to minimize free memory by growing the cache as much as it can, but when an application demands extra memory, cached and buffered memory are released to give way to applications. Therefore we decided to consider for the percentage used only the memory used by applications.

For storage units (again virtual and physical) we measured the disk free space, using the CollectD df plugin. Finally, for network interfaces we defined network traffic as the core measurement, using the CollectD interface plugin. We chose bytes per second as the unit and made the distinction between inbound and outbound traffic.

4 Validation

To validate our proposal we have deployed the monitoring infrastructure on two different cloud scenarios. In this section we detail their characteristics and analyze the information retrieved. We deployed a three-node OpenStack cloud with the following characteristics: A Cloud Controller (nova-volume, nova-network, nova-scheduler and nova-compute services) on a Intel Core 2 Quad Q9400 with 4 GB of RAM, with one nova-compute node (nova-compute service only) on a Intel Core 2 Duo E6600 with 2 GB of RAM, and another node (nova-compute service only as well) on a Intel Core i7 2600k with 8 GB of RAM. All nodes were connected via a single 100 Mbps Ethernet link. Monitoring, cloud management and virtual machine traffic shared the same physical link.

The virtual machines were created containing a 10 GB root filesystem and a 20 GB /mnt partition. Applications that were considered to be likely to write to disk were configured to use the 20 GB partition. The data were collected on this one.

4.1 Scenario 1: Web Application

The first scenario was a cloud multi-tier web application. We used Apache Olio, a sample web 2.0 application written in PHP that simulates an application used to organize conferences. Three different virtual appliances are part of this cloud application: the Olio PHP web application running over an Apache server, a MySQL database for data storage, and a mock service for geolocalization running over a Tomcat application server (since each user profile or event page includes a map with its location), which we will call Geocacher. We used Apache JMeter to test the load capacity of the application. JMeter tests were executed from another virtual machine inside the same cloud. This virtual machine had double the CPU capacity (two cores assigned) than the others and the same memory and disk space.

The test creates a user and then proceeds to login as that user. It then registers a new event, searches for

events with a specific tag and signs up to attend a specific event. To test Olio's load limit in our setup we ran two different set of tests. During the first one, several sequential tests with a duration of 50 seconds each were run, increasing the number of clients after each iteration. We ran 100, 500, 1000, 1500 and 2000 users, starting each test at the closest even minute after the previous test ended. We extracted the network data from the cloud application's virtual journal and compared it to data extracted from the physical node journal. Figure 6.a shows the data received by each network interface, including the virtual interfaces in the host. Each pair of virtual network interface in the physical machine and its corresponding interface in a virtual appliance has been shaded with the same hue, so that the correlation between them can be better appreciated. The VMs are listed in the same order in which they were created.

The second test consisted of starting 2000 user sessions in a five-minute window to check if the application was able to handle that load. The test came back positive, so we proceeded to launch a series of sequential test with the same parameters and little to no waiting time between them. The objective of this test was to put the application under heavy stress, so it would show a growing resource consumption slope. When we reached the maximum load capacity of the virtual machine with the Apache server (limited by memory), the machine ceased to respond. This showed one limitation of our monitoring approach: if the monitored node is under a high load the CollectD monitoring agent will stop reporting to its corresponding monitoring server. Figure 6.b shows the evolution of memory consumption for virtual and physical memories involved in the test. It should be noted that while the virtual memory is occupied and released after use, physical memory is not readily released. This means that it safe to assume that after the second peak in Apache + Olio's memory, the physical node started to use swap memory.

4.2 Scenario 2: Hadoop Cluster

The other scenario was a cloud application consisting of a Hadoop cluster made of three separate instances serving as nodes, one master and two slaves. We acknowledge the limitations of running a Hadoop cluster over a set of virtual machines, but this example serves the purpose of showing a different kind of application from the usual multi-tier web. The master node (NameNode and JobTracker) and one

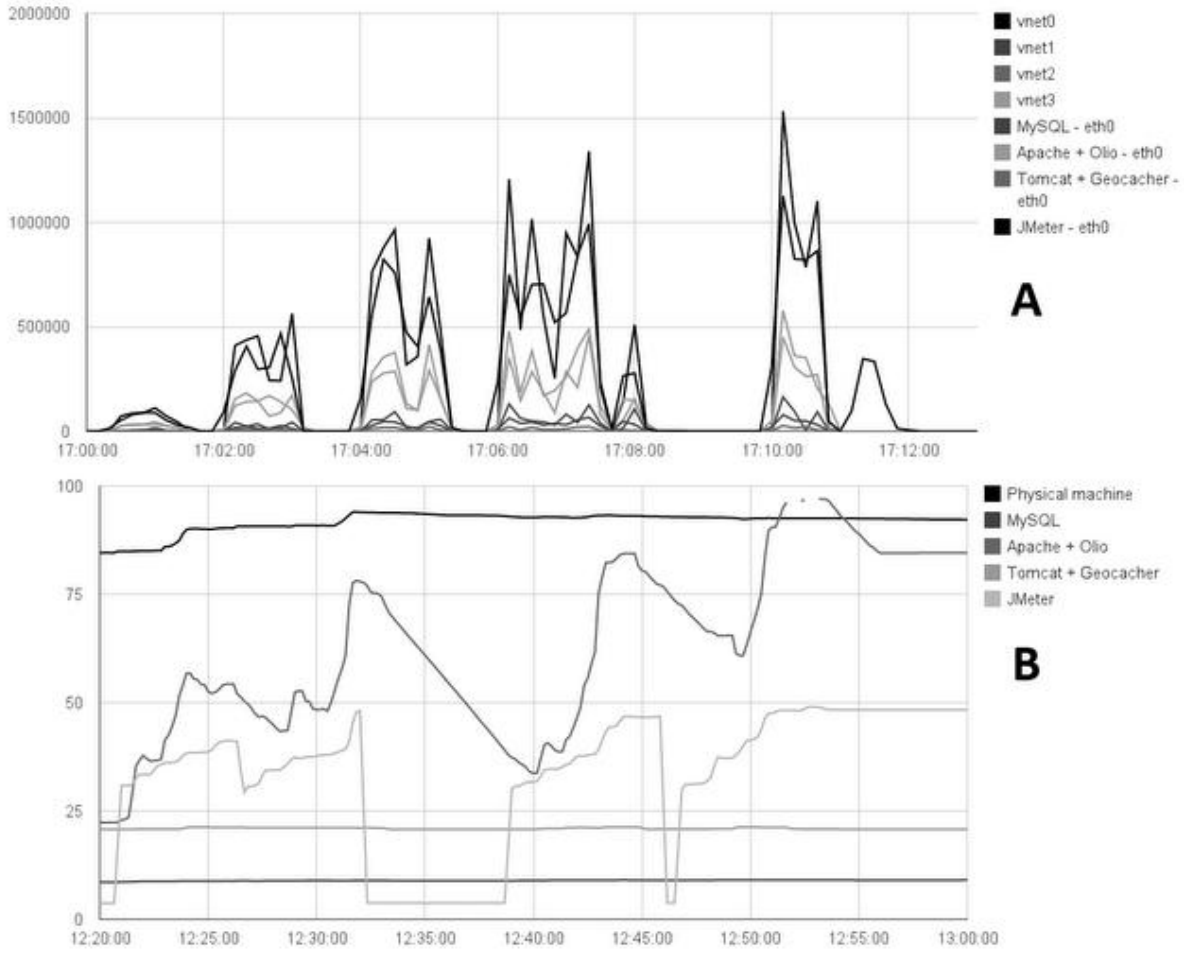

Figure 6: Web Application Scenario Retrieved Information

slave node (DataNode and TaskTracker) instances

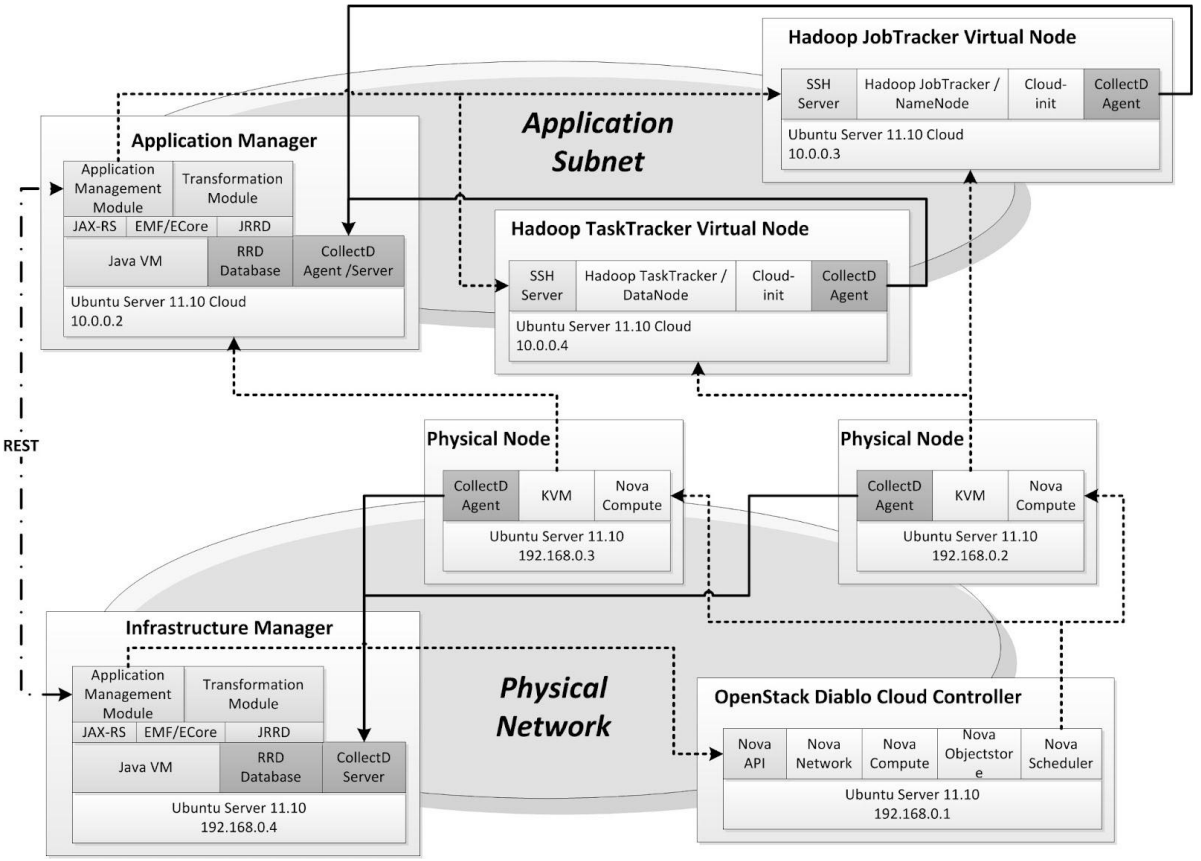

Figure 7: Validation Scenario 2

were running on one physical node, while the other slave node instance was running on another physical machine. This scenario can be seen in Figure 7. Solid lines represent monitoring interactions, and dashed lines management actions.

On this Hadoop cluster, a benchmarking process was started. Using the tools included in the Hadoop suite, we generated 10GB of random data. Then we proceeded to sort the data and finally we used the *testmapredsort* tool to validate the sorting. The monitoring system collected information from the physical and the virtual machines for the length of the jobs.

These data could then be visualized and reviewed to study the application's behavior and how it affected the virtual machines and the physical nodes underneath. In this particular case, the problems inherent to a heterogeneous Hadoop cluster became very evident.

First, we launched a *randomwriter* job, which initiated 10 mappers that wrote approximately 1GB each. All the data ended up on just one of the nodes, the fastest one, and, due to additional temporary files being needed between the map and reduce cycles, the total turned out to be too large to be sorted on a 20 GB partition, so 4 GB were deleted to end up with only 6 GB of random data to be sorted subsequently. When running the test case's second stage, sort, the virtual machine running on the first compute node (Core 2 Duo E6600) needed a much longer time to finish its share of jobs than the Hadoop node running on the second compute node (Core i7 2600k). When this second Hadoop node finished its tasks, it sat idly for a long time, waiting for the slow node to finish its work. On the third and final stage, validating the sorting, a very similar situation arose where a node was being used to its full extent while the other had plenty of available resources. This can be seen in Figure 8. It can be concluded that actively monitoring all parts involved in a cloud application might enable a manager to react to anomalous situations where resources are not being optimally allocated.

A smart Application Manager could take actions in response to this behavior (for present or future tasks) such as reallocating virtual machines to a faster node or creating new virtual machines on those nodes that are being underused to take advantage of the idle cpu processors. Having different data collected and establishing the relationships among them helps an intelligent manager make an informed decision: what the current situation really is and what actions can be taken to correct or improve it with the available

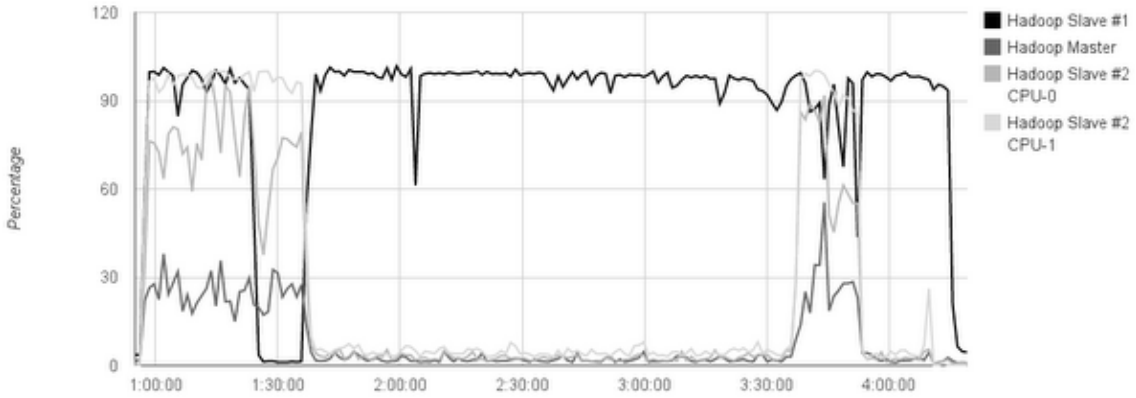

Figure 8: Hadoop Scenario Retrieved Information

resources. Arguably, even real time decisions could be taken based on the information gathered, but due to the special nature of Hadoop jobs and clusters, in this case it would not have been useful.

5 Related Work

Most of the research related to cloud computing systems focuses on the algorithms for providing scalability, elasticity, and self-optimization of the available resources. On the other hand, cloud monitoring is a field significantly less explored by the research community. Hasselmeyer et al. [13] describe a set of requirements for a cloud monitoring system (multi-tenancy, scalability, dynamism, simplicity and comprehensiveness) and propose an agent-based architecture that fulfills them. However, the complexity of the proposed solution greatly complicates its realization and has not been validated with a complete implementation yet. On top of that, the information retrieved is classified in a generic data structure that, although extensible, does not have the capability to show the relationships between the monitored elements or the data retrieved.

A simpler approach, focused on private cloud monitoring, is presented by De Chaves et al. [14]. This work ignores some of the requirements of the previous work (like multitenancy) and focuses on private clouds. It is based upon previous work in the field of distributed computing monitoring and has seen a basic implementation built upon Nagios. The main problem with this approach is that it does not define an information model. Therefore, the retrieved data is shoehorned into a traditional system administration view, which lacks the language to express some cloud concerns.

In the commercial field, some public IaaS providers offer basic monitoring capabilities to their clients, but they are mostly focused on billing-related metrics. The exception is Amazon CloudWatch, which also provides other features like alarms connected with Amazon Auto Scaling to automatically add or remove instances. However, the metrics offered by this solution are pre-selected and might not feed the needs of some customers. On the other hand, most private IaaS solutions lack even basic monitoring support.

An important factor to take into account when monitoring clouds is the amount of communication resources the monitoring process could take from both the monitored nodes and the underlying network infrastructure. The work of Meng et al. [15] tries to solve this problem using time windows. We have not experienced this problem in our test scenarios, but we still need to perform tests with more complex environments.

Another topic of interest is the white-box vs black-box approaches in monitoring. For some aspects of virtualization, a hybrid approach called grey-box has been proposed instead, aiming to reduce the burden in the monitored resources [16].

6 Conclusions and Future Work

In this article we have proposed a monitoring architecture for private IaaS clouds. We have defined an information model designed to take into account the several stakeholders that operate in a cloud environment and their specific needs and use cases. To fulfill this, the proposed information model covers both the physical and virtual levels of a cloud environment, and its static and dynamic state.

The model provides a complete picture of the cloud environment, separating the information relevant to each stakeholder, but at the same time

enabling the traceability between the data retrieved by different agents. The model is aligned with the main cloud management standards. We have extended their base concepts in order to fulfill all the requirements for a private cloud monitoring system.

To populate this information model we have defined a generic monitoring architecture that instruments the physical and virtual platforms with pairs of monitoring clients and servers. The proposed architecture is structured around Application Managers and a Infrastructure Manager, which separate the physical and virtual layers and collaborate to aggregate and correlate the monitoring information.

We have implemented our monitoring architecture using open-source systems-level monitoring tools, and we have demonstrated its application through two scenarios (a Hadoop cluster and a traditional web application) deployed into a private IaaS cloud built using OpenStack. We have shown that our approach is able to correlate the information, create a complete view of the environment, and separate data between several stakeholders. To summarize: it is able to fulfill the role of the monitoring component of an autonomous cloud management system.

We plan to test the monitoring architecture in larger cloud environments, developing instrumentation agents for additional cloud platforms. It is important to assess the behavior of the architecture in more complex situations and test its scalability for larger environments. More specifically, we aim to study the impact of monitoring traffic in several scenarios. With this knowledge, we could devise a strategy to select windows for sending monitoring traffic, reducing the burden for the network of the cloud environment. This could be extended to public clouds through the use of an intermediate layer like Xen- Blanket [17].

We will also be extending this work with the modeling of relevant information at application level (such as SLA compliance related metrics), and the development of additional monitoring agents that can collect the higher-level information and merge it with the remaining information.

Finally, and more importantly, we will develop the rest of the components of the autonomous cloud management system: the actuators, and the reasoning engine. They together will be able to dynamically manage a private cloud IaaS environment.

References

[1] Interoperable Clouds Version 1.0; Available at http://www.dmtf.org/sites/default/files/standards/documents/DSP-IS0101_1.0.0.pdf

[2] NIST Cloud Computing Reference Architecture Version 1.0; Available at http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/ReferenceArchitectureTaxonomy/NIST_SP_500-292_-_090611.pdf

[3] A. Samba, Logical Data Models for Cloud Computing Architectures, IT Professional, Vol.14 No.1, 2012, pp.19-26.

[4] Cloud Infrastructure Management Interface (CIMI) Model and REST Interface over HTTP Version 0.0.35; Available at http://dmtf.org/sites/default/files/standards/documents/DSP0263_1.0.0a.pdf

[5] Open Cloud Computing Interface – Core Version 1.1; Available at http://ogf.org/documents/GFD.183.pdf

[6] Open Virtualization Format (OVF) specification version 1.1.0; Available at: http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf

[7] DMTF System Virtualization Profile Version 1.0; Available at http://dmtf.org/sites/default/files/standards/documents/DSP1042_1.0.0_0.pdf

[8] S. Yan, B.S. Lee, S. Singhal, A Model-Based Proxy for Unified IaaS Management, Proc. 4th International DMTF Academic Alliance Workshop on Systems and Virtualization Management, 2010, pp.15-20.

[9] O. Sukwong, A. Sangpetch, H. S. Kim, SageShift: Managing SLAs for Highly Consolidated Cloud, Proc. 31st Annual International Conference on Computer Communications, 2012, pp.208-216.

[10] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, Journal of Internet Services and Applications, Vol.1, No.1, 2010, pp.7-18.

[11] L.A. Barroso, Warehouse-Scale Computing: Entering the Teenage Decade, Proc. 38th annual international symposium on Computer architecture, 2011.

[12] Common Information Model (CIM) Metrics model Version 2.7; Available at http://dmtf.org/sites/default/files/standards/documents/DSP0141.pdf

[13] P. Hasselmeyer, N. d'Heureuse, Towards holistic multi-tenant monitoring for virtual data centers, Proc. IEEE/IFIP Network Operations and

Management Symposium Workshops, 2010, pp.350-356.
[14] S.A. de Chaves, R.B. Uriarte, C.B. Westphall, Toward an Architecture for Monitoring Private Clouds, IEEE Communications Magazine, Vol.49, No.12, 2011, pp.130-137.
[15] S. Meng, L. Liu, T. Wang, State Monitoring in Cloud Datacenters, IEEE Transactions on Knowledge and Data Engineering, Vol.23, No.9, 2011, pp.1328-1344.
[16] T. Wood, P. Shenoy, A. Venkataramani, M. Yousif, Black-box and Gray-box Strategies for Virtual Machine Migration, Proc. 4th USENIX Symposium on Networked Systems Design & Implementation, 2007, pp.229-242.
[17] D. Williams, H. Jamjoom, H. Weatherspoon, The Xen-Blanket: virtualize once, run everywhere, Proc. 7th ACM european conference on Computer Systems, 2012, pp.113-126.