Future Generation Computer Systems 29 (2013) 273–286

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures

Javier Espadas [a,*], Arturo Molina [b], Guillermo Jiménez [a], Martín Molina [b], Raúl Ramírez [a], David Concha [a]

[a] Tecnológico de Monterrey, Campus Monterrey, Mexico

[b] Tecnológico de Monterrey, Campus México City, Mexico

ARTICLE INFO

Article history:
Received 2 November 2010
Received in revised form
17 October 2011
Accepted 24 October 2011
Available online 29 October 2011

Keywords:
Cloud computing
Software-as-a-Service
Multi-tenancy
Virtualized resources
Resource allocation

ABSTRACT

Cloud computing provides on-demand access to computational resources which together with pay-per-use business models, enable application providers seamlessly scaling their services. Cloud computing infrastructures allow creating a variable number of virtual machine instances depending on the application demands. An attractive capability for Software-as-a-Service (SaaS) providers is having the potential to scale up or down application resources to only consume and pay for the resources that are really required at some point in time; if done correctly, it will be less expensive than running on regular hardware by traditional hosting. However, even when large-scale applications are deployed over pay-per-use cloud high-performance infrastructures, cost-effective scalability is not achieved because idle processes and resources (CPU, memory) are unused but charged to application providers. Over and under provisioning of cloud resources are still unsolved issues. Even if peak loads can be successfully predicted, without an effective elasticity model, costly resources are wasted during nonpeak times (underutilization) or revenues from potential customers are lost after experiencing poor service (saturation). This work attempts to establish formal measurements for under and over provisioning of virtualized resources in cloud infrastructures, specifically for SaaS platform deployments and proposes a resource allocation model to deploy SaaS applications over cloud computing platforms by taking into account their multi-tenancy, thus creating a cost-effective scalable environment.

1. Introduction

Cloud computing refers to both the applications delivered as services over Internet and the hardware and systems software in the datacenters that provide those services commonly in a pay-per-use pricing basis [1]. With cloud computing definition comes the term of elasticity which is the ability to create a variable number of virtual machine instances depending on the application demands [1,2]. The cloud applications themselves have long been known as Software-as-a-Service (SaaS). SaaS is a software delivery paradigm where the software is hosted off-premises, developed by service providers and delivered via Internet and the payment mode follows a subscription model [3]. For SaaS providers, having the power to scale up or down an application to only consume and pay for the resources that are required at that point in time is an attractive capability and if done correctly it will be less expensive than running on regular hardware from traditional hosting [1].

However, in spite of the advantages of using cloud computing to create highly scalable applications, solving performance problems through cloud computing is not a trivial decision if involved costs are analyzed [4]. For example, Amazon Web Services charges by the hour for the number of instances you occupy, even if your machine is idle. In 2008, the image-processing Animoto application deployed over Amazon EC2 infrastructure [5] experienced a demand surge that resulted in growing from 50 servers to 3500 servers in three days; after the peak subsided, traffic fell to a level that was well below the peak. Hence, scale-up elasticity was not a cost optimization strategy but an operational requirement, and scale-down elasticity allowed the steady-state expenditure to more closely match the steady-state workload. Indeed, Animoto's provider charge by 3500 virtual instances because a peak load occurred at a certain time frame and when this peak disappeared, it would pay for unused resources [4]. This effect is still a barrier for SaaS providers, whose applications have different peak loads and they are highly prone to suffer over and under provisioning of resources [6,7].

Over and underutilization of resources are problems that are presented because elasticity in pay-per-use cloud models has not been achieved yet [8]. An over provisioning effect happens by resource underutilization: even if peak loads are successfully

* Correspondence to: Tecnológico de Monterrey, Campus Monterrey, 64849, Monterrey, Nuevo Leon, Mexico. Tel.: +52 81 12343912.
E-mail address: mijail.espadas@itesm.mx (J. Espadas).

0167-739X/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2011.10.013

anticipated, resources are unused during nonpeak times. Armbrust et al. [1] provide a calculation of this problem:

"*…a service has a predictable daily demand where the peak requires 500 servers at a peak usage but it requires only 100 servers most of the time. As long as the average utilization over a whole day is 300 servers, the actual utilization over the whole day is* $300 \times 24 = 7200$ *server-hours; but since we must provision to the peak of 500 servers, we will pay for* $500 \times 24 = 12\,000$ *server-hours, a factor of 1.7 more than what is needed*."

A second problem is overutilization, which occurs when potential revenue from customers is lost by poor performance (saturation) and customers stop using the application permanently after experiencing poor service, resulting in a permanent loss of the revenue stream [1]. Unfortunately, while current cloud platforms allow for the instantiation of new virtual machines, their lack of agility fails to provide users with the full potential of a real elastic model.

Furthermore, current cloud virtualization mechanisms do not provide cost-effective pay-per-use model for Software-as-a-Service (SaaS) applications and just-in-time scalability is not achieved by simply deploying SaaS applications to cloud platforms [9]. By imposing per-hour costs, cloud computing encourages SaaS architects extra attention to efficiency (i.e., releasing and acquiring resources only when necessary) [1]. This is caused by the traditional approach that consists in scaling applications based on the number of users. As a result, with the current resource allocation models, SaaS providers will be charged for global resource usage without taking account of resources used by each tenant. Consequently, there exists the need to create a true elastic architecture to charge SaaS providers the actual resource usage [6]. To achieve cost-effective SaaS scalability, a level of automation is necessary, which translates in a more intelligent environment. A SaaS platform and its applications should be aware of how tenants use its resources [10]. In this sense, SaaS applications have an opportunity to improve this scenario by their multi-tenancy, which is the ability to offer one single application instance to several clients/providers (tenants). With the use of cloud computing approaches such as on-demand resource allocation through SOAP interfaces, it is possible to efficiently create virtualized resources for SaaS applications which allows to allocate and charge only consumed resources in a tenant-based environment.

This research work provides a twofold contribution: (1) establishes a formal measure for under and over provisioning of virtualized resources (CPU and memory) in cloud infrastructures specifically for SaaS platform deployments and (2) proposes new resource allocation mechanisms based on tenant isolation, VM instance allocation and load balancing in order to deploy SaaS applications over cloud computing platforms by taking into account their multi-tenancy and create a cost-effective scalable environment.

The rest of this paper is organized as follows. Section 2 introduces the background for multi-tenancy definition and related projects. Section 3 describes the platforms used to deploy test beds. It describes the architecture of a Software-as-a-Service (SaaS) platform and explains the setup of Eucalyptus cloud platform and how a Tomcat-based SaaS platform is deployed over it. Section 4 defines over and under utilization of cloud resources and how to generate workload and measure resource consumption of this Tomcat cluster. Section 4 also presents results obtained when traditional scaling over cloud computing is performed. Section 5 describes the proposed solution and Section 6 discusses how multi-tenant patterns were designed and implemented tenant isolation. Section 7 describes the mechanism to calculate the number of virtual machines needed when the SaaS platform is running and its tenants are consuming virtualized resources. Section 8 presents the tenant-based load balancer architecture and how this architecture implements the four components of a dynamic load balancer: load measurement, information exchange, initiation rule and load balancing operation. Section 9 describes the components for monitoring and accessing the *itesm-cloud* private infrastructure through SOAP interfaces. Section 10 outcomes the results and performs a statistic analysis of these results. Section 11 discusses the conclusions and Section 12 outlines further research.

2. Background

2.1. Multi-tenancy: definition and support

An important requirement for SaaS applications is the support of multiple tenants [11]. A tenant is a customer that uses or provides a SaaS application. In order to exploit economies of scale, i.e. allow SaaS providers to offer the one SaaS application instance to multiple tenants, a SaaS application must be multi-tenant aware [3,12,13]. Multi-tenant aware means that each tenant can interact with the application as if it were the only user of the application. In particular, a tenant cannot access or view the data of another tenant [12]. In a SaaS model, the multi-tenancy support can be applied to four different software layers [14]: the application, the middleware, the virtual machine (VM), and the operating system layers. In a multi-tenancy enabled service environment, user requests from different tenants are served concurrently by one or more hosted application instances based on the shared hardware and software infrastructure. There are generally two kinds of multi-tenancy patterns [11,15]: *multiple instances* and *native multi-tenancy*; the former supports each tenant with its dedicated application instance over a shared hardware, operating system or a middleware server in a hosting environment whereas the latter can support all tenants by a single shared application instance over various hosting resources. The two kinds of multi-tenancy patterns scale quite differently in terms of the number of tenants that they can support. Multi-instance is adopted to support a small number to hundreds of tenants, while native multi-tenancy is used to support a much larger number of tenants, usually in the hundreds or even thousands. It is interesting to note that the isolation level among tenants decreases as the scalability level increases [15].

In [16], the authors present a framework to deal with the issues of native multi-tenancy for SaaS applications. In [15], the challenges of SaaS applications for application vendors and providers are discussed, taking into account the need for customization of SaaS applications [17]. At the storage level, the traditional technique for implementing multi-tenancy is to add a tenant ID column to each table and share tables among tenants [13,18]. Another work is presented in [19], where the M-store system is proposed and developed which provides storage and indexing services for a multi-tenant database system. These techniques try to create an isolation environment for tenants by separating one tenant context from another. This tenant context isolation can be implemented from the data layer to an execution of a specific view.

2.2. Related projects

In [11], experts' knowledge of scaling application servers in the cloud is captured through profiles. Authors define a profile as an embodiment of knowledge and best practices of a commonly adopted computing environment with inherent just-in-time scalability. In [20], the authors propose a toolkit based in Java mechanisms that supports the fine granularity multi-tenancy mechanism; they use tenant contexts, runtime context elements that carries tenant-specific information. A *TenantContext* runtime object is used to carry the tenant's information within application servers. This object is tied to the Java Virtual Machine (JVM)

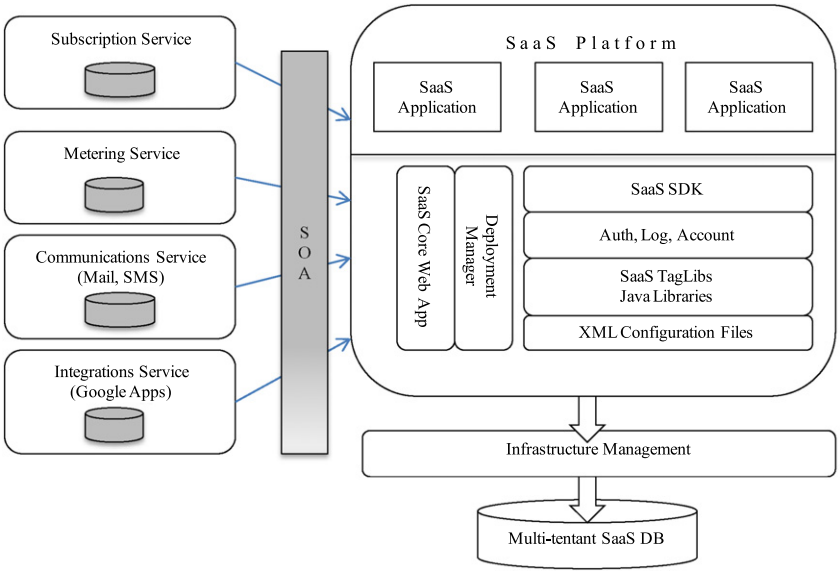

Fig. 1. SaaS platform architecture.

through the Java Instrumentation mechanism, which provides services that allow Java language agents to instrument programs running on the JVM. The basic idea of using JVM instrumentation is to start an agent listener when JVM initiates and then intercept values or fields that can be annotated as tenant-aware (called *isolation points*), and load them, according to the tenant's configuration.

Another related work is presented in [21] which focuses on two crucial problems: efficient VM image management and intelligent resource mapping. VM image management includes image preparation and local image management of physical resources. Ref. [22] presents iVIC, a platform for academic researchers to dynamically create customized virtual computing environments to launch various scientific computing, simulations and analysis, by leveraging VM technology. In iVIC, common resources (e.g., a set of workstations, PC servers and small clusters) are organized into a number of physical resource pools. Each physical machine is treated as a VM Container (VMC), which is responsible for providing VM environments. Each VMC exposes controlling and querying interfaces to upper resource level manager via a collection of SOAP interfaces. VM container interacting with SOAP interfaces is the mechanism for monitoring and measuring virtualized resources, as the selected cloud computing platform in this research, Eucalyptus, offers SOAP interfaces and make possible on-demand deployment of VM instances.

In [23] is proposed the use of a cloud computing mechanism as a raw computational on-demand resource for a grid middleware. In this work, authors use Eucalyptus to manage resources for a grid middleware implementation called DIET-Solve. In [24], the authors describe three key components, effectively covering "*measurement*, "*modeling*" and "*management*" (VM3) of shared resource implications on individual virtual machine performance. Authors also propose a decomposition model that allows estimating the potential performance loss when a virtual machine is consolidated with other virtual machines. Such a decomposition model consists of three major components: (a) virtualization overheads, (b) core contention overheads and (c) shared cache contention overheads. A relevant commercial tool is Amazon Auto Scaling [25]. It is a web service to automatically launch or terminate Amazon EC2 instances based on user-defined triggers. It claims to enable applications to scale up instances seamlessly and automatically when demand spikes and automatically shed unneeded instances when demand subsides. It uses proprietary commands to create *Auto Scaling Groups*, which is a representation of an application running on multiple instances. However, this mechanism is a closed proprietary mechanism which depends totally on Amazon EC2 platform. Second, it is based only on resource utilization but it does not take into account the nature of the applications. In this work, resource utilization is compared against performance (throughput) of web applications in order to determine whether or not a virtual machine is saturated.

3. Test bed platforms and architectures

The test bed deployment is composed of two main components: a Java-based SaaS platform and a private cloud platform, independent of each other. The first step of test beds is to deploy the SaaS platform over a cloud infrastructure in order to set a cluster of Tomcat servers over nodes of the cloud platform. The SaaS platform is developed as part of the *Rapid Product Realization for Developing Markets Using Emerging Technologies* research chair at Tecnológico de Monterrey University, Campus Monterrey.

The SaaS platform is composed of several components that allow the deployment of application as services (Fig. 1). Each component is integrated in an Apache Tomcat container as a Web application, a packaged library (.jar) or business services (Web application + Web Services) is defined a *service application* as the software application that will be delivered as a service. Each service application is deployed as a common web application within the Tomcat container and it manages its own resources, such as data sources, libraries, and views. The main difference from common web deployments is how the SaaS platform' components manage and interact with these web applications. The main interaction point of the service application with the platform is done through the SaaS SDK. The SaaS SDK provides the common libraries that are used by applications to access the basic SaaS services, such as authentication, account information, public resources and so on. In the view layer, the platform offers components (SaaS Tag Libraries) for an easy integration with the SaaS context (such as public/private menus, templates, layouts).

The Deployment Manager is a listener component that configures each application according to an XML configuration file. Every time an application service is deployed within the Web application container, the Deployment Manager reads the configuration file and analyzes the application code to detect updated or new modules, security roles or deployment changes. The access point

Table 1
Open SaaS platform technologies.

Requirement	Technology
Language platform	J2EE (Java 1.6)
Web container	Apache Tomcat 6
Web framework	Struts 2
Web services	Apache Axis2
Dependency injection	Spring 2
Dependency injection + Web services	Spring 2 + WSO2
Multi-tenancy layer	JoSQL + Java annotations
Persistence layer	Hibernate 3 and Java Persistence API (JPA)

to the SaaS platform is the SaaS Core Web Application (SCWA). This component is a web application that is used to access to all other applications and components. The SCWA is in charge of loading common resources and views, such as security context, authenticated user, view filters, etc. At the bottom of the architecture, we can find the Infrastructure Management and the physical data source of the SaaS platform.

As Table 1 outlines, the core components of the SaaS implementation are open source technologies. Fig. 1 shows a set of business components that are consumed by platform. These business components were designed, developed and deployed by following a Service Oriented Architecture (SOA) design in order to be completely decoupled from the SaaS platform [26]. Each business component is developed as a Web application, but it exposes a set of Web services through WSO2 framework[1] which integrates web services deployed through Apache Axis2 and dependency injection with Spring2. Each business component application implements its own Web services and they are referenced in the *applicationContext.xml* Spring file.

SaaS platform provides the App Metering Service which allows automatic and non-intrusive support for metering applications, tenant-based monitoring and virtual machine resource status. This service uses Java Management Extension (JMX) technology to provide information on performance and resource consumption of applications running in the Java platform. It also uses SIGAR (System Information Gatherer And Reporter) API[2] which provides a portable interface for gathering system information such as system memory, CPU loads and so on. App Metering Service application exposes a Web service interface that can be consumed by monitors or any other component that requests information about VM instance.

Eucalyptus Cloud Platform is used as Infrastructure-as-a-Service (IaaS) platform in this work for deploying the SaaS platform and run several tests. Eucalyptus is an open source platform originally developed at the University of California to deploy and run a cloud within commodity computing clusters [27]. Eucalyptus uses common computational and storage infrastructure available to academic research groups to provide a platform that is modular and open to experimental instrumentation and study [28]. This cloud platform will allow create on-demand virtual machines over a hardware infrastructure and deploy the SaaS platform presented previously. The reason why a public cloud platform, such as Amazon EC2, was not chosen is because we needed access to the cloud platform detailed information, such as real used resources (CPU, memory).

The SaaS platform is a Tomcat-based architecture, so it is possible to distribute one Tomcat instance per virtual machine thus enabling scalability based on the number of users. The deployment architecture is shown on the further Fig. 4 which depicts the whole test bed architecture. The Eucalyptus private cloud called *itesm-cloud* was installed over a 4-node cluster within Tecnológico de Monterrey university facilities. Through Eucalyptus platform it is possible to create as many SaaS platform Tomcat-node instances as required for a certain time period; however, creating virtual machines is done manually through Eucalyptus interfaces and tools. When running parallel clustered Tomcat servers, Web balance loader (HTTP component) is required on top to distribute the workload over the underlying Tomcat instances. Thus, the SaaS platform was configured for receiving requests from an Apache Web Server with default round-robin workload balancing. Below this HTTP component, Tomcat instances are distributed among each virtual machine (see Fig. 4). Details about setting up clustered Java applications are out of the scope of this work; however, it will be described how SaaS was deployed over an Eucalyptus cloud installation. As mentioned before, it is necessary to create on-demand virtual machines for each Tomcat instance we want to deploy. Thus, it a VM image was created with the SaaS platform to be deployed within each VM. This creation was performed through Eucalyptus tools set called *euca2ools*. As the cloud platform allows to determine how many resources will be available for each VM type (small, medium, large or extra large), in our test beds small VM instances were set with the similar characteristics as small Amazon Web Services instances [29]: 1 CPU core, 1 GB of memory and 3 GB for storage running a 32-bit operating system (Ubuntu 9.10 Karmic). These small VM instances will run Tomcat containers inside them. 800 MB of memory will be assigned to the Java virtual machine that contains the Tomcat server. A different virtual machine is used for deploying the HTTP server and a different physical server is used for deploying databases. Three different SaaS applications were deployed over the SaaS platform: a sales administration application, a bill control service and a contact manager. Each application will receive a set of HTTP requests to be executed in order to generate workload for the Tomcat-based deployment.

4. Defining and measuring over and underutilization

4.1. Overutilization (saturation)

For overutilization definition, the term "point of exhaustion" is used. For conventional load testing, the point of exhaustion is typically defined when a limiting resource (such as CPU, memory or storage) has reached 100% utilization [30]. In contrast, the point of exhaustion for cloud computing can be defined as the maximum useful payload that could be placed on a single virtual machine without adversely affecting the throughput [31]. Saturation or overutilization occurs whenever resource utilization gets above the point of exhaustion. The former means that at least one virtual machine must be monitored on each physical tier of the service being tested. In some cases, as workload begins to escalate, so do operating system-level activities such as thread context-switching, CPU consumption, virtual memory management and so on. For experimentation purposes, it must suffice to note that when resource utilization skyrockets, throughput (useful work) generally declines [6,32]. The SaaS platform uses the HTTP request throughput calculated by JMeter tool (explained later). This throughput value is calculated as requests/unit of time [30,33]. The time is calculated from the start of the first request call to the end of the last request call. This includes any intervals between requests, as it is supposed to represent the server's load. The formula is throughput = (number of requests)/(total time). Previous works [30–32,34–36] use the throughput to define an

[1] Mathew, T. (February, 2008). "*Hello World with WSO2 WSF/Spring*". WSO2—the Developer Portal for SOA. http://wso2.org/library/3208. Last access on June 2009.

[2] Ryan Morgan, D. M. (2009, July). SIGAR-System Information Gatherer And Reporter. Retrieved March 2010, from http://support.hyperic.com/display/SIGAR/Home.

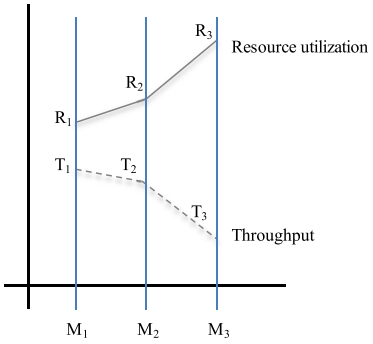

Fig. 2. Detecting inflection points.

inflection point as the percentage of utilization that is achieved when the throughput starts to decline.

Identifying those inflection points is the key for developing an accurate measurement for resource overutilization. The data point of greatest interest in this trend is the one that corresponds to the point of maximum throughput. If superimposed, the throughput trend on the utilization trend is possible to highlight this critical turning point where throughput and utilization become inversely related [30]. By correlating the percentage of resource utilization at maximum throughput, it is possible to detect when resource utilization is saturated by the workload [36]. This measurement is used in combination with a Tomcat-based cluster in order to determine the underutilization within virtual machines. Inflection points are measured by each virtual machine within the cloud-based Tomcat cluster.

Fig. 2 depicts the detection of an inflection point. Taking three measurement points ($M1$, $M2$ and $M3$), three consecutive values are gathered of certain resource utilization (i.e., CPU) (solid line, $R1$, $R2$ and $R3$) and three consecutive values of the throughput (dotted line, $T1$, $T2$ and $T3$). An inflection point occurs; when given these measurements, the following relationship is true: $R1 < R2 < R3$ and $T1 > T2 > T3$. The resource showed in Fig. 2 is saturated at time point M2 because by increasing its utilization, the throughput declines resulting in poor performance. During test execution, these measurements and inflection point detections are performed for the CPU and heap memory assigned to the Java virtual machine.

4.2. Underutilization (resource wasting)

Resource underutilization occurs whenever some resources are not being used by virtual machines within a cloud computing infrastructure and an application is being executed [4,37]. Resource underutilization can be measured by the amount of resources available to be used by potential virtual machines and applications. In this sense, work [4,31,37,38] is used to state that when resource utilization (CPU, memory or storage) of a single VM (original) can be allocated in another VM (destination) without exceeding the maximum quantity allowed for such resource, then resource of the original VM is being wasted (underutilization) [4]. Fig. 3 depicts a scenario where the used heap memory is measured within four virtual machines.

According to Fig. 3, there are at least two VM instances that can be released by reallocating their resources (VM1 and VM2 resource utilization can be allocated in VM3). In this research, the number of underused resources is obtained through a knapsack approach[3]

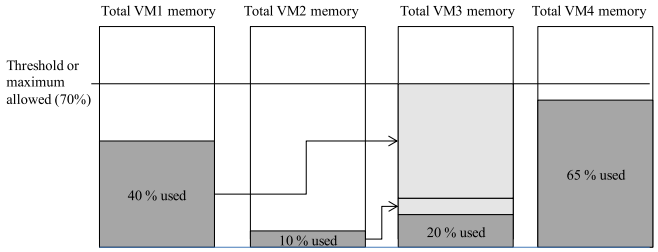

Fig. 3. Memory underutilization.

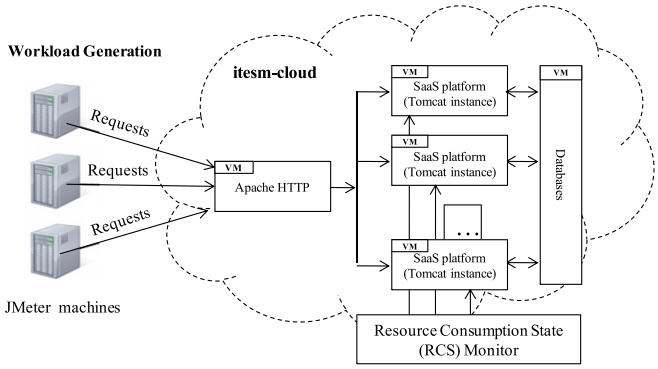

Fig. 4. Test bed architecture.

[8,39,40] by calculating the combination of VM instances that can be allocated by another single VM, according to the measured resource (CPU or heap memory) [41]. As it is not the aim of this work to detail or solve the knapsack problem, this research uses a simple tree-based Java program to calculate the allocation. Each VM is evaluated against the rest at a certain point in time. An algorithm has been developed, which takes the weights of the knapsack as the used resources in a given measurement. The values or profits are taken from the available quantity of such resources (maximum allowed minus used) of the rest of VM instances [8,42]. By doing these weight and value vector assignations, the knapsack implementation returns the maximum number of VM instances that can be released by maximizing resource availability and gives low weight to VM instances with low usage. Capacity of the knapsack is the available resource of the evaluated VM.

4.3. Generating workload

The Apache JMeter tool was selected for creating workload to the Tomcat cluster installed in a cloud environment and where the SaaS platform has been deployed. Apache JMeter is an open source software Java desktop application designed to load test functional behavior and measure performance [43,44]. It can be used to simulate a heavy concurrent load on a J2EE application and to analyze overall performance under various load types, it also allows graphical analysis of performance metrics (e.g. throughput, response time) [45]. Simulating concurrent users by JMeter can be employed in an independent computer; they can also be employed in a distributed testing framework [35]. For this work, JMeter will be configured to create distributed requests to simulate workload usage.

Fig. 4 depicts the test bed architecture that is used in this work. The distributed SaaS platform setup that was explained in the previous section will be stressed with several requests through different hosts running JMeter tests. The concept of Resource Consumption State (RCS) is used to define the state of the CPU and memory resources used by the Tomcat servers. Through a similar mechanism proposed by [46], while the JMeter machines run the tests, an RCS Monitor will be collecting information about

[3] In the knapsack problem, we are given a set of objects where each object has a weight and a value. We are given a container of a given capacity, imposing a weight constraint. The problem is then to place as many objects as possible into the container such that the weight constraint is not violated, and the sum of the values of the objects in the container is maximum.

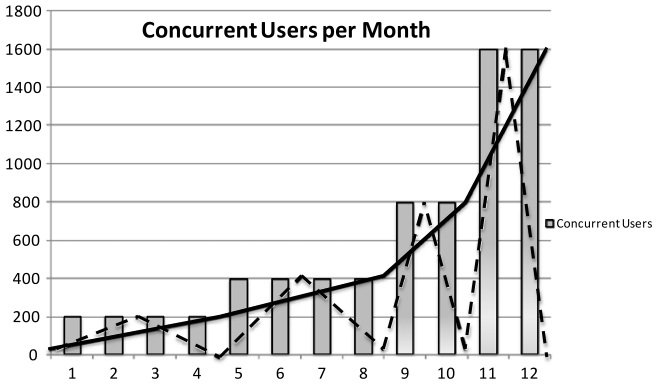

Fig. 5. Workload simulation.

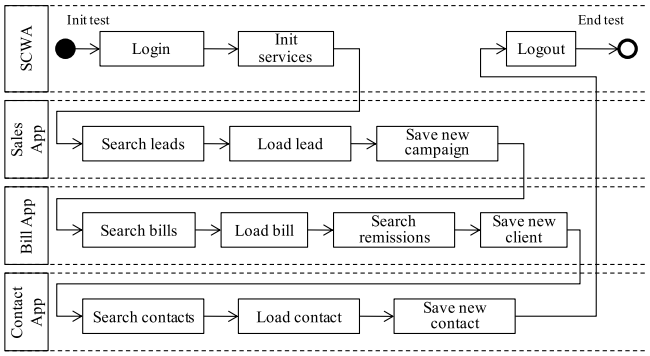

Fig. 6. Test plans flow.

Table 2
VM instances for workload simulation.

Season	VM instances	Maximum peak of users	Simulation hours
Jan–Apr	2	200	48
May–Aug	4	400	48
Sept–Oct	8	800	24
Nov–Dec	16	1600	24
TOTAL			**144** h

the resource utilization through Web services calls to the App Metering Service (explained earlier) by accessing to the resource status of each virtual machine. The amount of concurrent Tomcat users will depend on the server hardware (processors, memory), the types of resources being used within applications and what the applications are actually doing [47]. In Tomcat version 6.0 or newer, as used in the SaaS platform, a mechanism to configure the number of threads the Tomcat supports is via the *maxThreads* attribute of the *Executor* element in XML configuration files. The default setting for this attribute is 200, which should be enough to get most applications started, and according to [35,43,47], is enough to support at least a thousand simultaneous users. As in this research will be used small instances of virtual machines, it is established that each Tomcat server can handle 100 users as top [34]. Assuming different behaviors during twelve months, as presented in [48], different types of workload peaks are stated for SaaS requirements [1,15,30].

Two types of workload generation are as follows [8,15,34,47]:

- Incremental. For each time period, workload starts from the peak of the previous time period and increases until reaching the maximum peak of established users at the end of current period (see solid line in Fig. 5).
- Peak-based. For each time period, workload starts from zero users and increments until reaching the maximum peak of users at the middle of the period. Then, the workload starts to decrease until zero at the end of the period (see dotted line in Fig. 5).

One limitation is the fact that is not possible to run the test for the whole twelve months, due to time restrictions in this research. In this sense, the tests will be configured to run tests for 12 h (720 min) simulating an entire month (30 days) and 24 min will represent an entire day of execution (24 h) (see *Simulation hours* column in Table 2). In this sense, RCS Monitor will gather information every 10 s, simulating retrieve of resource consumption every 10 min in the presented scale (1 min = 1 h). As stated, one virtual machine instance will be deployed for each 100 simulated user. Therefore, the number of small instances to scale the workload depends on the maximum simultaneous users of the period.

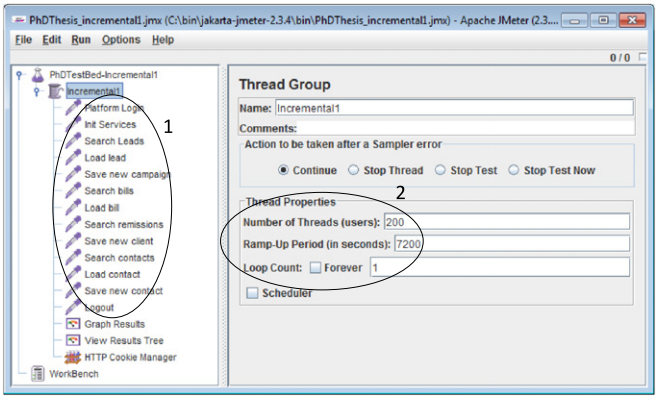

Fig. 7. JMeter tool configuration for test beds.

Fig. 6 shows the whole workflow for a set of HTTP requests that are defined within each test plan. Every test plan will execute the same set of requests but with different workload behavior. Each lane of the workflow diagram represents one different SaaS application. The first step consists of login to the platform through an e-mail/password mechanism which links to the user with a specific subscriber (tenant). For simulation purposes, this tenant assignation will be randomly performed by the platform authentication mechanism.

Fig. 7 depicts the configuration screen of JMeter tool. At the left panel (1) can be found the set of HTTP requests that were defined in Fig. 6, by accessing different SaaS applications. At the right panel (2) can be observed some parameters such as *Number of threads (users)* and *Ramp-Up Period* that define the stress behavior of the workload.

4.4. Test results

After running all the test plans all results were gathered. As explained, RCS Monitor gathered information every 10 s resulting in a total of 4320 measurements spanned in 720 min (30 days of simulated time, 1 month simulated). The following paragraphs present the results of such metrics according to the definition of RCS and each workload behavior. Underutilization and overutilization were metered by using the mechanisms explained before. For a certain simulated month, the underutilization will be the sum of the total wasted virtual machines calculated in all measurements. In the same way, the overutilization will represent the sum of all inflection points detected in the measurements of such simulated month.

Fig. 8 shows a chart of the throughput measurement results during the incremental (top screen) and peak-load (bottom screen) workload simulation. In order to calculate the throughput, JMeter

Table 3
Results of CPU and memory monitoring in traditional scaling.

Simulated month	VMs	Server-hours	Combined-incremental		Combined-Peak-based	
			UU (%)	OU (%)	UU (%)	OU (%)
Jan	2	1 440	24.57	4.78	25.93	9.86
Feb	2	1 440	13.09	12.52	13.10	14.24
Mar	2	1 440	10.37	17.94	8.58	23.77
Apr	2	1 440	8.15	33.88	34.85	7.44
May	4	2 880	27.32	2.96	37.27	3.46
Jun	4	2 880	16.76	9.43	16.91	21.34
Jul	4	2 880	9.09	14.07	14.54	30.48
Aug	4	2 880	7.59	22.09	42.92	8.73
Sept	8	5 760	31.61	6.13	55.19	0.46
Oct	8	5 760	11.57	14.18	26.07	10.03
Nov	16	11 520	40.55	9.12	51.93	2.74
Dec	16	11 520	20.30	11.96	22.17	14.10
	Averages		**18.42**	**13.26**	**29.12**	**12.22**

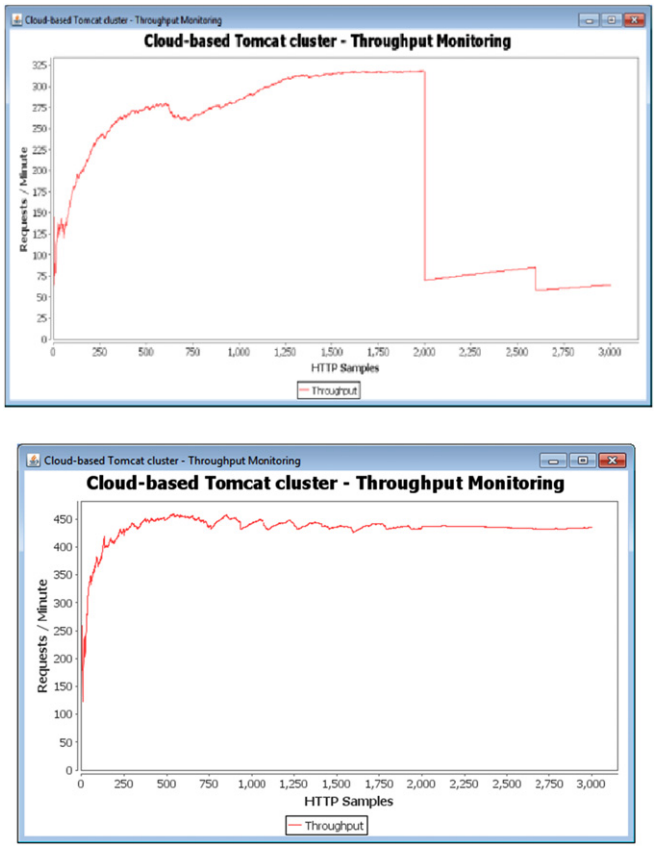

Fig. 8. Throughput measurement.

tool generates a set of *HTTP Samples* during test execution and evaluates the requests per minute that the Tomcat-cluster can process. As shown in Fig. 8, the behavior of throughput during simulation changes over time and it shows some declinations in the efficiency of the Tomcat cluster.

Table 3 shows the results of the measurements during both incremental and peak-based workload tests. The column labeled as *Combined* outlines the number of measurements where both CPU and memory are either saturated or underutilized. Last two columns calculate a percentage value by adapting formulas presented in [49]:

$$\%\text{UU(Underutilization)} = (\text{Combined UU/Measurements per hour})/\text{Server-hours}. \quad (1)$$

Formula (1) calculates the percentage of total available virtual machines that were wasted or idle during such time period. *Combined UU* is divided by *Measurements per hour* (6 in the tests) because we want to obtain the average of wasted virtual machines per hour. Then this average is divided by the total server-hours available during the test, which in this case is obtained by multiplying 720 h (30 days $*$ 24 h) with the number of virtual machines. For overutilization percentage we used the following calculation:

$$\%\text{OU(Overutilization)} = \text{Combined OU}/(\text{Measurements per month} * \text{Number of virtual machines}). \quad (2)$$

Overutilization percentage (formula (2)) is the percentage of the total measurements performed that have inflection points. This value is calculated by dividing the combined overutilization *Combined OU* by the total of measurements performed which is obtained by multiplying the number of measurements per month (4320 in the tests) with the number of available virtual machines. It can be observed that a total of 51 840 server-hours were provided for the whole time the SaaS platform was running over the cloud infrastructure.

5. The proposed solution model and architecture

To tackle under and over utilization issues, this work proposes a new model for allocating workload when deploying SaaS platform and its applications over cloud infrastructures. This model comprises three approaches that take advantage of the multi-tenancy nature of SaaS applications in order to improve workload distribution and instantiate the number of cloud resources that are really needed. The first approach is tenant-based isolation which creates tenant-level granularity and separates execution contexts for different tenants; isolation implementation is divided into tenant-based persistence and tenant-based authentication. The second approach is tenant-based VM allocation which implements mechanisms to calculate the number of VM instances needed, given a set of tenants and their weights in terms of active users. The third approach is tenant-based load balancing, which implements a distribution mechanism to process and dispatch workload requests concerning each tenant.

Fig. 9 depicts the architecture of the tenant-based VM instance creation model. The components of this architecture will be explained in the following sections. In addition to these components, a Cloud Communication Layer was implemented in order to get connected to the Eucalyptus private cloud and request VM instances on demand.

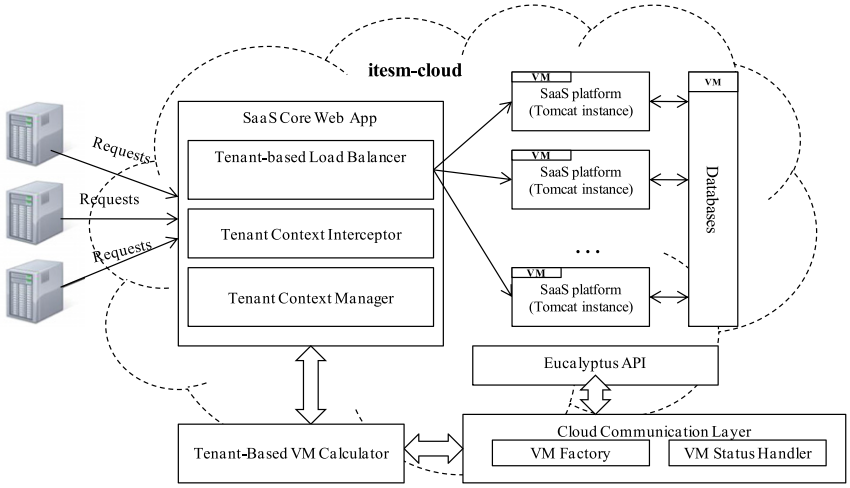

Fig. 9. Tenant-based VM instance creation architecture.

6. Tenant-based isolation

6.1. Tenant-based persistence

The SaaS applications are deployed in the SaaS platform by using *Shared Database–Shared Schema* mechanism [12,18], by logically separating the data corresponding to each tenant with a subscriber ID field in the database's tables. This shared schema approach has the lowest hardware and backup costs, because it allows serving the largest number of tenants per server [50]. Each service application implements its own database, separating the multitenant information with a tenant ID key as proposed by [18]. SaaS platform uses an aspect-oriented mechanism for multi-tenancy which is implemented in the application side and it is called *Multi-Tenant Persistence* layer. This layer uses JoSQL[4], a technology for performing SQL-like queries over collections and the ability of Struts2 for creating aspect-oriented interceptors that allow separating the information of each tenant in a logical way. The persistent layer is based on Object Relational Mapping technologies (JPA + Hibernate). Supposing is needed to retrieve the contacts from a given tenant, a simple object-oriented query can be used to do that:

```
//JPA query in the persistence layer
String sql = "SELECT contact FROM Contact contact"
Query query = em.createQuery(sql);
return query.getResultList();
```

Simple as this, but it is important to notice that there is no filter by tenant in the query sentence. The persistence layer will return a set of 'Contact' objects. It is possible to pre-process these results using the interceptor feature of Struts2 before they can be accessed by another application layer. Within a Struts2 action it is possible to declare an annotated property:

```
@Multitenant(attribute = "tenantId")
List ⟨Contact⟩ contacts;
```

Last code declares that this particular list of objects will be filtered before they are accessible to another application component (a Java Server Page view for example). This pre-processing is achieved by setting a Struts2 interceptor in the call stack. This interceptor can access the invocation action as follows:

```
Object action = invocation.getProxy().getAction();
//getting the subscriber ID from the SCWA
long subscriberId = Auth.getSubscriberId();
for (Field field: clazz.getDeclaredFields()){
        if (field.isAnnotationPresent(Multitenant.class)){
        Multitenant filter = (Multitenant)field.getAnnotation(Multitenant.class);
        String attribute = filter.getAttribute();
        String property = field.getName();
//obtains the list
Object objList = BeanUtils.getProperty(action, property);

//obtains the name of the class
String className = getClassName(objList);

        // create and perform a query over the object list
        Queryq = new Query ();
        q.parse("SELECT * FROM "+className+" WHERE
            "+attribute +" = "+subscriberId);
        QueryResults qr = q.execute (list);
        List newList = qr.getResults();

        //setting back the filtered list by tenant
        BeanUtils.setProperty(action,property,newList);
        }
}
```

In the former example, the '*Contacts*' list will be reduced to only the objects which their "*tenantId*" property matches with the authenticated tenant's user. With this mechanism it is possible to create multi-tenant pre-processing behavior within the SaaS applications. In fact, it is feasible to create a transparent support for multi-tenant persistence without affecting the legacy on-premise applications.

6.2. Tenant-based authentication

The access point for SaaS platform and its deployed applications is known as SaaS Core Web App (SCWA). This component is a Web application with specific characteristics for managing tenant-based authentication, security and access list control. Each user belongs to one or more subscriber or tenant (these terms are used indistinctly). Once the user has been authenticated through an email and password, SCWA links the user to its tenant ID [14]. If the user belongs to two or more subscribers, a selection screen is displayed to select which to work with. After that, SCWA searches for the user within an Access Control List (ACL) to retrieve permissions for that subscriber and for the SaaS applications that the subscriber has contracted. Then SCWA creates a session cookie with all this information and stores it within the user session

[4] Steven Haines. (2005, Sept) JoSQL–SQL for Java Objects. [Online]. http://www.informit.com/guides/content.aspx?g=java&seqNum=230.

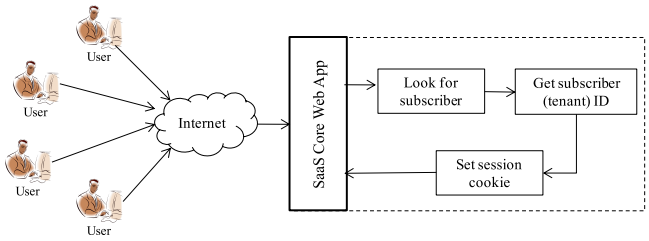

Fig. 10. Tenant-based authentication.

(Fig. 10). In this way, each user is linked to this tenant-based information and all subsequent requests are identified through this session cookie.

It is necessary to retrieve this information from several SaaS applications, even when deployed among different machines over a cluster. In order to achieve this, all Web applications should have access to the SCWA context information and should retrieve the cookies from it. The tenant-based information is stored in the SCWA context session and the rest of the applications can access it through the following mechanisms:

```
public static UserVO getAuthenticatedUser() throws NotAuthenticatedUserException {
        HttpServletRequest request = ServletActionContext.getRequest();
        String SAASADMIN_SESSIONID = getCookieValue(request,AuthConstants
          .SIDEL_SESSION_ID);
        ServletContext contextAuth = request.getSession().getServletContext();
        UserVO authUser =getUserFromAdminContext(contextAuth,SAASADMIN
        _SESSIONID);
if (authUser == null){throw new NotAuthenticatedUserException();}
          return authUser; }

private static UserVO getUserFromAdminContext(ServletContext context, String
  ssosessionid) {
        ServletContext sidelcontext = context.getContext(SAAS_CORE_APP);
          Hashtable<String, UserVO>shareddata = (Hashtable<String, UserVO>)
            sidelcontext.getAttribute(AuthConstants.SAAS_USERS);
if(shareddata! = null&& ssosessionid! = null) {
              //get the right User using the sessionid
              return (UserVO)shareddata.get(ssosessionid);}
          else return null;
}
```

The static method *getAuthenticatedUser*() can be called from any application and it retrieves the session cookie of the authenticated user from the SCWA context (represented by *SAAS_CORE_APP* variable). *UserVO* is the value object that holds information about the subscriber and the authenticated user.

Given this tenant-based isolation, a *Tenant Context* object is been conceptualized as a part of the SCWA component, as proposed in [20]. This object is created or destroyed for each tenant that is being accessed within the platform through user requests. Each *Tenant Context* object holds information (see Fig. 11) about tenant status such as active users, logged users, subscriber ID, total of requests, etc. A *Tenant Context Manager* object holds information about all *Tenant Context* objects that exist within the running SaaS platform and it is in charge of their creation or destruction. Both objects are creating by using the ability of Struts2 to create Aspect-Oriented interceptors that allows to separate in a logical way the information of each tenant. A pre-processing implementation is achieved by setting a Struts2 interceptor in the call stack. Every request is then filtered and analyzed by a *Tenant Context Interceptor*. When the interceptor is executed, it calls the *getAuthenticatedUser*() method and updates the *Tenant Context Manager* and its *Tenant Context* objects with the information about user's tenant.

7. Tenant-based VM allocation

In order to describe the topology and characteristics of the deployed cloud and server cluster, a profile-based approach, as

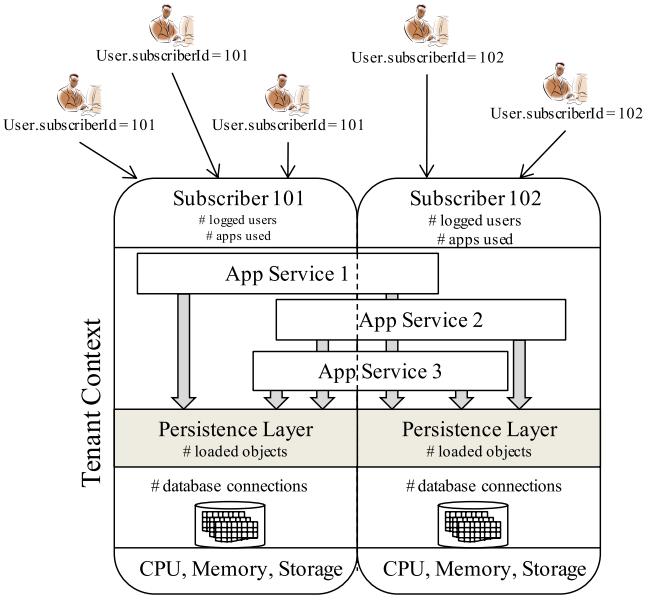

Fig. 11. Tenant context.

proposed by [9], is implemented. For example, the profile of test bed in this work is as follows: it uses small virtual machine types (1 CPU core, 1 GB of memory) and 800 MB to the Java heap memory for running Tomcat instances. Also, it was set in Tomcat configuration that each server can handle 100 users by setting its *maxThreads* configuration attribute to 200. Other server deployments can represent different profiles depending on the application provider needs. Based on the number of *Tenant Context* objects, their number of current active users and the profile of the deployed cluster, a number of virtual machines is calculated to support the current workload. Each tenant has its own resource requirements depending on the number of its active users and the applications that are been accessed. In order to assign values for VM allocation, each tenant context is given a weight that is calculated by the *Tenant Context Manager* component as showed in formula (3)

$$\text{Tenant Context}_{\text{weight}} = \text{Active users} * (\text{heap size} / \text{maxThreads}). \quad (3)$$

Java memory heap size assigned to Tomcat is used as a profile parameter for the calculation. Active users are those whose session timeout has not expired. This number is multiplied by the average memory size per thread. For example, if a *Tenant Context* contains 20 active users, and based on the profile information of the deployed cluster we have a heap size of 1024 MB and a *maxThread* value of 256, we get that the tenant context weight is $20 * (1024/256) = 80$. The VM capacity is calculated as described in formula (4):

$$\text{VM}_{\text{capacity}} = \text{Heap size} - ((\text{heap size}/\text{maxThreads}) * \text{platform threads}). \quad (4)$$

The VM capacity is determined subtracting the amount of memory used by the platform from the total Java memory heap size. Java Management Extension (JMX) implementation of the App Metering Service is used to calculate the number of threads used by Tomcat server plus the platform. By using formulas (3) and (4), a *Tenant-Based VM Calculator* component performs calculations to obtain certain number of VM instances. As most resource allocation problems, VM instances allocation problem is related to knapsack problem [8]. The problem to solve is to calculate the minimum number of virtual machine instances with specific and

homogeneous capacity (same VM type) in order to allocate a set of tenant context weights. This type of allocation problem is known as multi-objective optimization (MO) problem [51]. The allocation problem can be expressed as showed in formula (5) (adapted from [8,51]):

$$f(x) = \sum \square \left(\max \sum_{j=1}^{\text{w.length}} Wj \text{ where } \sum_{j=1}^{\text{w.length}} Wj \leq \text{VM}_{\text{capacity}} \right)$$
$$W \in \text{Tenant context weight vector.} \quad (5)$$

The goal of formula (5) is not to obtain an assignation vector as traditional allocation mechanisms do, but to determine the minimum number of VM instances that are needed to allocate the entire weight vector given homogeneous VM capacity. In order to solve this calculation, authors propose a simplistic iterative algorithm by using the same tree-based Java library presented in Section 4.2 for solving simple knapsack allocations. Tenant-based allocation uses a vector of tenant context weights retrieved from *Tenant Context Manager*. The values to maximize are the same than weights in such a way that knapsack function allocates the maximum number of tenant context weights and maximizes resource usage. The first iteration of the proposed algorithm will allocate as many weights as it can within an initial VM. The remaining weights that could not be allocated in the first iteration will be used for a second iteration. Iterations continue until the remaining tenant context weights vector has a length of zero. The number of iterations represents the number of VM instances that need to be running to allocate the whole tenant workload. The following code snippet shows this recursive function from *Tenant-Based VM Calculator* component:

```
int VMs = 0;
public int calculateVMs(int capacity,int []tenantWeights){
   //STEP_1 Initial VMs for overweight tenants
   VMs = preProcessWeights(capacity, tenantWeights);
   //STEP_2 Eliminate overweight values
   tenantWeights = processWeightArray(capacity,tenantWeights);
   //STEP_3 Calculate VMs
   calculateVmsAllocation(capacity, tenantWeights);
   return VMs;
}//end of function
private void calculateVmsAllocation(int capacity,int []tenantWeights){

if (tenantWeights.length == 0) return;
else{
//EVERY ITERATION INCREMENTS VM INSTANCES NUMBER
VMs++;
//SOLVING KNAPSACK
Knapsack KS = new Knapsack(capacity, tenantWeights, tenantWeights);
KS.search(0,0,0);
int [] take = KS.getBestSolution();

  //FILTERING NOT-TAKEN ELEMENTS
  List<Integer>newWeightList = new ArrayList<Integer>();
  for (intc = 0; c < take.length;c++){
            if (take[c]==0){//not taken
                    newWeightList.add(tenantWeights[c]);
            }
}
//REDUCING ARRAY
tenantWeights = convertToIntArray(newWeightList);
//RECURSIVE TO NEXT ITERATION
calculateVmsAllocation(capacity,tenantWeights);
   }
} //end of function
```

In the last code snippet, there is a comment labeled as STEP_1; in this step is calculated the number of VM instances that is needed for those tenant contexts that exceed the VM capacity. For example, assuming 3 tenant context weights as follows: {254, 21, 434} and a VM capacity of 100, the mechanism in the STEP_1 will detect that first and third weight values are overweight and it calculates how many VM instances are needed initially for such tenant weights. This calculation is performed by adding all the integer values obtained from the division *exceeded_weight/capacity*. In the example, this calculation is performed as follows: (integer)(254/100) + (integer)(434/100) = 2 + 4 = 6 initial VM instances. Once the overweight is calculated, the next step, labeled as STEP_2, is to process the weight vector in order to eliminate such overweight of exceeded tenants. This calculation is done by interchanging the exceeded weight by the modulo of *exceeded_weight/capacity*. For instance, in the former example, 254 weight is changed by 54 (254%100) and 434 by 34 (434%100), therefore the new weight vector will contain {54, 21, 34} values. After these two calculations, the allocation algorithm performs the VM allocation with the processed weight vector. *Tenant-Based VM Calculator* performs all these calculations every determined time and interacts with the Cloud Communication Layer in order to request the VM instances as needed.

8. Tenant-based load balancing

Attention now turns to the design of a tenant-based load balancer to distribute requests concerning each tenant. Tenant-based load balancing is proposed to take advantage of isolation parallelism in Web applications, meaning that requests from users are totally independent of each other. A load balancer allows coordinating the operation of the multiple servers [52] and load balancing strategies involve the adjustment of the distribution of the work load among the participating servers if the distribution is expected to result in a reduction of the total execution time [49,53]. In this sense, instead of using traditional load balancers on top of the Tomcat cluster, the design of a component that receives, identifies, analyzes and dispatches requests received to the SaaS platform is suggested. This kind of load balancing is known as dynamic load balancing because it is not based on previous knowledge of received workload [54].

Tenant-based load balancer is part of the SCWA component and it uses *Tenant Context Manager* to retrieve information about tenant statuses. This component allocates requests of the same tenant to an unique virtual machine, depending on the capacity obtained from the profile information (explained before). If the capacity of a VM is saturated [53], the tenant-based load balancer will allocate new tenant requests in different VMs if available, if not, it will notify the *Tenant-Based VM Calculator* that a new VM is needed.

Fig. 12 shows the modules and interactions of the tenant-based load balancer. Previous research work has developed adaptive models of load balancing [49,55,56] and this proposal is based on such research. The tenant-based load balancer has the following elements:

- *Request Processor* receives requests from HTTP or HTTPS ports and interprets the request method (GET, POST, etc.).
- *Server Preparer* instantiates a new HTTP client, sets the request headers and establishes timeouts.
- *Cookie Manager* copies cookie information into the request.
- *Response Parser* returns the server answer meta data to the client.
- *Tenant Request Scheduler* communicates with *Tenant Context Manager*component in order to determine to which VM the request is dispatched.

The tenant-based load balancer retrieves the information of the available VM instances from the Cloud Communication Layer. This balancer implements the four elements described by [54] to achieve dynamic load balancing when parallel nodes are used: load measurement, information exchange, initiation rule and load balancing operation. The following paragraphs describe the general algorithm for scheduling and dispatching policy.

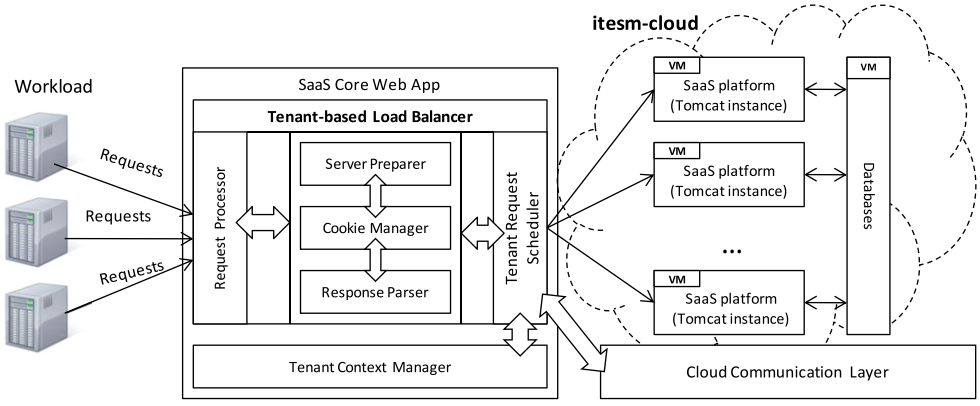

Fig. 12. Tenant-based load balancing.

Load measurement. Dynamic load balancing algorithms rely on the workload and capacity information of nodes. The workload information is commonly quantified by a load index, a non-negative variable taking zero value if the node is idle, and taking on increasing positive values as the load increases [57], in SaaS environment, the load index can be taken as the number of users of each tenant. A load index should be a good estimate of the response time of the resident processes within the node. Since the measure of the load would occur frequently, its calculation must be highly efficient [54]. Previous studies [57,58] have shown that the choice of a load index has considerable effect on the performance of load balancing and that simple load indices such as the number of ready processes are particularly effective. Retrieving information about the Tenant Context allows to determine the workload needed for a certain tenant. When a new request arrives, load balancer reads the information from the Tenant Context Manager (explained in the previous section) and then invokes to Tenant Request Scheduler class that calculates in which VM the incoming request will be dispatched.

Information exchange. The information exchange rule specifies how to collect and maintain the workload information of nodes necessary for making the load balancing decisions. A desirable information exchange rule should strike a balance between incurring a low cost for the collection of system-wide load information and maintaining an accurate view of the system state. This tradeoff is better captured in the periodical information exchange rule, where the nodes periodically are requested their workload information from a load balancer component, regardless of whether the information is useful to others or not [54]. This work proposes an alternative to distributed approaches, a centralized rule in which a dedicated component collects and maintains the system's workload information [59,60]. Look up within tenant-based map to get the IP address of the VM instance by using tenant ID as key. If no value is returned from the map, load balancer requests the Tenant-Based VM Calculator a VM instance IP address. This calculation is performed by Tenant Request Scheduler component.

Initiation rule. An initiation rule dictates when to initiate a load balancing operation. The execution of a load balancing operation incurs non-negligible overhead; its invocation decision must weigh its overhead cost against its expected performance benefit. Generally, load balancing operations can be initiated either by an overloaded node or an under loaded node [61]. Periodic remapping is a common practice in distributed parallel computations. In this proposal, the initiation rule is implemented by Server Preparer component. ServerPreparer class creates a HttpClient object and establishes a connection to the VM instance. Then, the request headers are copied to the scheduled request.

Load balancing operation. A load balancing operation is defined by three rules: location rule, distribution rule and selection rule [54]. The location rule determines the partners of the balancing operation such as the nodes to involve in the balancing operation. The set of nodes that will participate in the operation are known as balancing domain. The distribution rule determines how to redistribute workload among nodes in the balancing domain (VM for same tenant). The selection rule selects the most suitable requests for transfer among nodes to realize the distribution decision. In this work, location rule is performed when the load balancer retrieves the tenant information and selection rule is performed when the Tenant Request Scheduler calculates or requests the VM IP address. The distribution rule is applied when the load balancer dispatches the request through an HTTP client.

As mentioned, policy for scheduling and dispatching workload is based on the tenant's information and available VM instances in the VM cluster. It basically holds a tenant-based map where requests for each tenant or tenant group is been allocated according to tenant ID field (key of the values map). This map contains the statuses of current assignations for the tenants and their corresponding VM instances. If dispatch fails, the load balancer will try again to process the scheduled request to the same VM node. If it fails again, it will ask for other VM instance IP to the Cloud Communication Layer. The load balancer will try to process the scheduled requests until a MAX_RETRIES configuration value has been achieved.

9. Cloud communication layer

The *Cloud Communicator Layer* contains the components for monitoring and accessing the *itesm-cloud* private infrastructure through SOAP interfaces. It uses jClouds[5] library which is a Java open source framework that implements portable abstractions to get connected to public clouds such as Amazon, Azure, Rackspace or private clouds like Eucalyptus. The jClouds API allows to remotely create and shutdown virtual machine instances and obtain their status. Cloud Communicator Layer has two components: (1) *VM Status Handler*, which holds and retrieves information about virtual machine instances, their IP addresses and statuses (starting, running, shutdown) and (2) *VM Factory*, the component that communicates to the Eucalyptus API through SOAP requests. It implements the mechanism for creating or releasing VM instances according to the *Tenant-Based VM Calculator* component.

//initialize credentials
String accesskeyid = "…";
String secretkey = "…";

[5] Adrian Cole. (October, 2010). "Introducing jclouds" [Online] http://code.google.com/p/jclouds/.

Table 4
Results with tenant-based components.

Simulated month	Server-hours	Incremental		Peak-based	
		UU	OU	UU	OU
Jan	591	9.88%	5.18%	2.34%	5.23%
Feb	672	6.09%	13.02%	6.63%	8.34%
Mar	682	4.72%	11.93%	8.42%	18.78%
Apr	831	6.72%	19.31%	8.34%	8.92%
May	1243	9.93%	3.12%	9.43%	5.34%
Jun	1339	9.75%	8.17%	9.23%	13.44%
Jul	1297	4.92%	12.12%	8.43%	22.11%
Aug	1479	6.22%	18.03%	9.34%	5.11%
Sept	3256	9.21%	3.23%	9.44%	4.21%
Oct	4015	7.82%	11.49%	9.43%	7.25%
Nov	9208	12.83%	7.26%	11.33%	8.89%
Dec	10789	10.43%	10.81%	12.89%	11.29%
Totals	35402	–	–	–	–
Averages	–	**8.21%**	**13.25%**	**8.77%**	**9.9%**
T student values		**3.2437**	**1.0282**	**4.7208**	**0.7485**

```
//connect to the Eucalyptus itesm-cloud
Properties overrides = new Properties();
overrides.setProperty("eucalyptus.endpoint",
"http://itesm-cloud:8773/services/Eucalyptus");
ComputeServiceContext context = new ComputeServiceContext
Factory().createContext("eucalyptus", accesskeyid ,secretkey,
ImmutableSet.< Module > of(new Log4JLoggingModule(), new JschSsh-
ClientModule()), overrides);

//create a VM template with SaaS platform image and small VM type
Template template =
context.getComputeService().templateBuilder().hardwareId("m1.
small").imageId("Eucalyptus/emi-9ACB1363").build();

//set security params
template.getOptions().as(EC2TemplateOptions.class).
securityGroups("default");
template.getOptions().as(EC2TemplateOptions.class).noKeyPair();

//creating N instances
Set<? extends NodeMetadata> nodes = context.getComputeService().
runNodesWithTag("saasplatform", N, template);
...
//or shutdown an instance by ID
context.getComputeService().destroyNode("Eucalyptus/i-3FB706CB");
```

Last code snippet shows abstractions that are used in most cloud infrastructures, such as authentication through key credentials, creating templates that match with VM images and running or destroying instances.

10. Results and analysis

After setting up the tenant-based components and deploying them over the test bed, all the simulation and tests were run again. Over and underutilization measurements where performed against workload tests in traditional load balancing (incremental and peak-based). Similar to Tables 3 and 4 shows the results of combined percentages. A main difference among results is the measurement of server-hours given by the number of VM instances that were created through tenant-based demand.

In Table 4, the server-hours were reduced from 51840 to 35402, a reduction of 32%. Also, it can be observed that averages of over and underutilization has been reduced, but in order to demonstrate a statistical significance improvement of previous averages (Table 3, known as control group) against new values in Table 4 (experimental group), a t-student test is carried out. The t-student test allows to determine if two averages are significantly different, in this case [62], if averages of Table 4 are statistically less than those in Table 3. With N as a number of months (samples), we have $(N_1 + N_2 - 2) = (12 + 12 - 2) = 22$ degrees of freedom and we set an accuracy of 99.5% ($\alpha = 0.005$ of significance), t-student distribution table produces a value of $t_\alpha = 2.8188$ as base parameter. Next step is to calculate t-student values for each corresponding column pair (for example, underutilization for incremental workload of Tables 3 and 4). t-student test dictates that if calculated value is greater than t_α parameter, we can say with a 99.5% of certainty that second column (Table 4) is statistically less than first column (Table 3). The calculation for *t-student* test is represented in formula (6) [62].

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{N_1} + \frac{S_2^2}{N_2}}} \tag{6}$$

where X_1 is the average and S_1 the standard deviation of columns of Table 4 results. X_2 and S_2 is average and standard deviation, respectively, when tenant-based components are used (Table 4). The last row of Table 4 shows the calculated t values. For example, taking the columns of the underutilization (UU) during incremental workload of both tables, the calculated t-student value is 3.2437. This value is greater than 2.8188 parameter, meaning that averages for underutilization before tenant-based components have been statistically improved. The t-student value for underutilization (UU) during peak-based is higher than t_α ($4.7208 > 2.8188$) and we can say that this behavior was, statistically speaking, improved as well. On the other hand, both t-student values corresponding to overutilization (OU) are not higher than t_α; even if the averages were reduced, we cannot conclude that there was a statistical improvement for such behavior.

11. Conclusions

In spite of cloud computing advantages for offering on-demand resources, there is still the need for certain automation when specific platforms are deployed and scaled over virtualized environments. This is the case of Software-as-a-Service (SaaS) platforms and their applications, where over and underutilization of resources occur due lower and higher workload pikes and because the number of virtual machine instances deployed for scaling applications are traditionally based on the maximum simultaneous users. In this matter, a tenant-based model is presented to tackle over and underutilization when SaaS platforms are deployed over cloud computing infrastructures. This model contains three complementary approaches: (1) tenant-based isolation which encapsulates the execution of each tenant, (2) tenant-based load balancing which distributes requests according to the tenant information, and (3) a tenant-based VM instance allocation which determines the number of VM instances needed for certain workload, based on VM capacity and tenant context weight. After running all tests and simulations, the results were gathered and averages were calculated. In general, over and underutilization averages were reduced but only averages for underutilization were statistically improved.

12. Future work

Resource allocation has a significant impact in cloud computing, especially in pay-per-use deployments where the number of resources are charged to application providers. As further research of the tenant-based resource allocation model, authors recommend some work to be done to improve and continue validating the proposed solution. It is recommendable to deploy a different platform

over the cloud infrastructure, such as High-Performance Computing (HPC) or scenarios such as online transactional applications. Moreover, other kind of resources can be defined to be metered such as bandwidth, storage, transferred data or database connectivity. In this way, new models and mechanisms for measuring virtual machine statuses must be defined and implemented in order to gather results for these resources. Additionally, test bed architecture used in this work implements a single database that SaaS applications access to persist data. An improved test bed architecture can implement clustered database design in order to distribute the information of applications. Two workload generation behaviors were used in this work: an incremental workload and a peak-based workload. Different approaches can be tested for different measurements. A non-deterministic workload can be generated through different mechanisms. Finally, this work used Eucalyptus to deploy a private cloud infrastructure over commodity computers but is important to remark that any other cloud infrastructure such as Microsoft Azure or Amazon EC2 can be adapted to be measured and tested in the similar way. The deployment of the SaaS platform over different cloud environment might require different configuration and test bed architecture.

Acknowledgments

The research presented in this document is a contribution for the "Rapid Product Realization for Developing Markets Using Emerging Technologies" Research Chair, ITESM, Campus Monterrey, and for the "Technological Innovation" Research Chair, ITESM, Campus Mexico City.

References

[1] M. Armbrust, et al. Above the clouds: a Berkeley view of cloud computing, electrical engineering and computer sciences, Technical Report No. UCB/EECS-2009-28, University of California at Berkeley, February 2009.
[2] R. Buyya, C. Shin Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.
[3] J. Espadas, D. Concha, A. Molina, Application development over software-as-a-service platforms, in: The Third International Conference on Software Engineering Advances ICSEA, 2008, pp. 97–104.
[4] J. Napper, P. Bientinesi, R. Iakymchuk, Underutilizing resources for HPC on clouds, Aachen Institute for Advanced Study in Computational Engineering Science, 2010.
[5] Animoto Blog, Amazon CEO Jeff Befos on Animoto, April 2008. http://animoto.com/blog/company/amazon-com-ceo-jeff-bezos-on-animoto/.
[6] J. Wong, G. Iszlai, M.L. Ye Hu, Resource provisioning for cloud computing, in: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research, November 2009.
[7] C. Isci, J. Kephart, L. Zhang, E. Bouillet, D. Pendarakis, X. Meng, Efficient resource provisioning in compute clouds via VM multiplexing, in: Proceeding of the 7th International Conference on Autonomic Computing, June 2010.
[8] M. Stillwella, D. Schanzenbacha, F. Vivienb, H. Casanova, Resource allocation algorithms for virtualized service hosting platforms, Journal of Parallel and Distributed Computing 70 (9) (2010) 962–974.
[9] Y. Jie, Q. Jie, L. Ying, A profile-based approach to just-in-time scalability for cloud applications, in: CLOUD'09, IEEE International Conference on Cloud Computing, September 2009.
[10] G.V. Mc Evoy, B. Schulze, Using clouds to address grid limitations, in: MGC'08: Proceedings of the 6th International Workshop on Middleware for Grid Computing, December 2008.
[11] C.J. Guo, W. Sun, Y. Huang, W. Zhi Hu, B. Gao, A framework for native multi-tenancy application development and management, in: The 9th IEEE International Conference on E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007, pp. 551–558.
[12] R. Mietzner, F. Leymann, M.P. Papazoglou, Defining composite configurable saas application packages using SCA, variability descriptors and multi-tenancy patterns, in: ICIW '08, Third International Conference on Internet and Web Applications and Services, June 2008.
[13] W. Sun, X. Zhang, C. Jie Guo, P. Sun, H. Su, Software as a service: configuration and customization perspectives, in: IEEE Congress on Services Part II, September 2008.
[14] T. Kwok, T. Nguyen, L. Lam, A software as a service with multi-tenancy support for an electronic contract management application, in: SCC'08 IEEE International Conference on Services Computing, vol. 2, July 2008.
[15] G. Carraro, F. Chong, Architecture strategies for catching the long tail, April 2006. http://msdn.microsoft.com/en-us/library/aa479069.aspx.
[16] R. Badonnel, A. Keller, Automating the provisioning of application services with the BPEL4WS workflow language, in: Proceedings of DSOM, 2004.
[17] R. Mietzner, A. Metzger, F. Leymann, K. Pohl, Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications, in: PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, May 2009.
[18] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, J. Rittinger, Multi-tenant databases for software as a service: schema-mapping techniques, in: SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, 2008.
[19] M. Hui, D. Jiang, G. Li, Y. Zhou, Supporting database applications as a service, in: IEEE 25th International Conference on Data Engineering, 2009, pp. 832–843.
[20] H. Cai, et al. An end-to-end methodology and toolkit for fine granularity saas-ization, in: IEEE International Conference on Cloud Computing, 2009, pp. 101–108.
[21] Y. Chen, T. Wo, J. Li, An efficient resource management system for on-line virtual cluster provision, in: IEEE International Conference on Cloud Computing, 2009, pp. 72–79.
[22] J. Huai, Q. Li, C. Hu, CIVIC: a hypervisor based virtual computing environment, in: International Conference on Parallel Processing Workshops, 2007, p. 51.
[23] E. Caron, F. Desprez, D. Loureiro, Cloud computing resource management through a grid middleware: a case study with DIET and eucalyptus, in: IEEE International Conference on Cloud Computing, 2009, pp. 151–154.
[24] R. Iyer, R. Illikkal, O. Tickoo, L. Zhao, P. Apparao, D. Newell, VM3: measuring, modeling and managing VM shared resources, The International Journal of Computer and Telecommunications Networking (2009) 2873–2887.
[25] Amazon Web Services—auto scaling. 2010. http://aws.amazon.com/autoscaling/.
[26] J. Sedayao, Implementing and operating an internet scale distributed application using service oriented architecture principles and cloud computing infrastructure, in: iiWAS'08: Proceedings of the 10th International Conference on Information Integration and Web-Based Applications & Services, November, 2008.
[27] C. Baun, M. Kunze, Building a private cloud with eucalyptus, in: 5th IEEE International Conference on E-Science Workshops, 2009, pp. 33–38.
[28] D. Nurmi, et al. The eucalyptus open-source cloud-computing system, in: CCGRID'09, 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, May 2009.
[29] Amazon Web Services, May 2010, http://aws.amazon.com/ec2/#pricing.
[30] C. Devlin, SaaS capacity planning: transaction cost analysis revisited, MSDN Library, February 2008. http://msdn.microsoft.com/en-us/library/cc261632.aspx.
[31] C. Isci, J. Kephart, L. Zhang, E. Bouillet, D. Pendarakis, X. Meng, Efficient resource provisioning in compute clouds via VM multiplexing, in: Proceeding of the 7th International Conference on Autonomic Computing, June 2010.
[32] A.K. Mishra, J.L. Hellerstein, W. Cirne, C.R. Das, Towards characterizing cloud backend workloads: insights from Google compute clusters, SIGMETRICS Performance Evaluation Review 37 (4) (2010).
[33] Apache Software Foundation, Apache JMeter, 2010. http://jakarta.apache.org/jmeter/usermanual/glossary.html.
[34] G. Paroux, B. Toursel, R. Olejnik, V. Feleax, A Java CPU calibration tool for load balancing in distributed applications, in: Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Parallel and Distributed Computing, 2004, pp. 155–159.
[35] Q. Wu, Y. Wang, Performance testing and optimization of J2EE-based web applications, in: Second International Workshop on Education Technology and Computer Science, ETCS, vol. 2, 2010, pp. 681–683.
[36] S. Wee, H. Liu, Client-side load balancer using cloud, in: Proceedings of the 2010 ACM Symposium on Applied Computing, March 2010.
[37] D. Dyachuk, R. Deters, A solution to resource underutilization for web services hosted in the cloud, in: Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on the Move to Meaningful Internet Systems: Part I, November 2009.
[38] C. Matthews, Y. Coady, Virtualized recomposition: cloudy or clear? in: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing, Washington, DC, USA, May 2009, pp. 38–43.
[39] O.C. Granmo, B.J. Oommen, S.A. Myrer, M.G. Olsen, Learning automata-based solutions to the nonlinear fractional knapsack problem with applications to optimal resource allocation, IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 37 (2007) 166–175.
[40] D.C. Vanderster, Resource allocation and scheduling strategies using utility and the knapsack problem on computational grids, Dept. of Electrical and Computer Engineering, British Columbia Canada, 2008.
[41] D.C. Vanderster, N.J. Dimopoulos, R. Parra-Hernandez, R.J. Sobie, Resource allocation on computational grids using a utility model and the knapsack problem, Future Generation Computer Systems (2009) 35–50.
[42] R. Parra-Hernandez, D. Vanderster, N.J. Dimopoul, Resource management and knapsack formulations on the grid, in: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Washington, DC, USA, November 2004, pp. 94–101.
[43] B. Weis, A. Buchmann, S. Kounev, Performance tuning and optimization of J2EE applications on the JBoss platform, Journal of Computer Resource Management 113 (2004).
[44] A. Zaidman, B. Du Bois, S. Demeyer, How webmining and coupling metrics improve early program comprehension, in: 14th IEEE International Conference on Program Comprehension, ICPC, 2006, pp. 74–78.

[45] T. Parsons, L.M. Patcas, J. Murphy, A. Ufimtsev, Introducing performance engineering by means of tools and practical exercises, in: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative research, New York, NY, USA, 2006.
[46] Ruay-Shiung Chang, Wu-Chun Chung, A new mechanism for resource monitoring in Grid computing, Future Generation Computer Systems 25 (1) (2009) 1–7.
[47] Y. Zhang, G. Huang, X. Liu, H. Mei, Integrating resource consumption and allocation for infrastructure resources on-demand, in: 2010 IEEE 3rd International Conference on Cloud Computing, CLOUD, Miami, FL, 2010, pp. 75–82.
[48] Amazon Web Services, Deploying distributed J2EE spplications using Amazon EC2, December 2007. http://www.imaginea.com/docs/Scaling%20JEE%20apps%20on%20ec2.pdf.
[49] J. Caoa, D.P. Spooner, S.A. Jarvisb, G.R. Nuddb, Grid load balancing using intelligent agents, Future Generation Computer Systems 21 (1) (2005) 135–149.
[50] G. Carraro, R. Wolter, F. Chong, Multi-tenant data architecture, MSDN Library, June 2006. http://msdn.microsoft.com/en-us/library/aa479086.aspx.
[51] C.A. Correa, R.A. Bolaños, A. Molina, Multiobjective knapsack problem using NSGA-II algorithm, Scientia et Technica, No. 39, September 2008.
[52] L. Harik, A. Kayssi, FPGA-based load balancer for Internet servers, in: The 14th International Conference on Microelectronics 2002—ICM, 2002, pp. 190–193.
[53] C.C. Myint, K.M. Lar Tun, A framework of using mobile agent to achieve efficient load balancing in cluster, in: 6th Asia-Pacific Symposium on Information and Telecommunication Technologies, 2005, APSITT 2005 Proceedings, 2005, pp. 66–70.
[54] C. Xu, L.C.M. Francis, Load Balancing in Parallel Computers, Kluwer Academic Publishers, 1997.
[55] G. Lodi, F. Panzieri, D. Rossi, E. Turrini, SLA-driven clustering of QoS-aware application servers, IEEE Transactions on Software Engineering 33 (3) (2007) 186–197.
[56] Y. Liu, L. Wang, S. Li, Research on self-adaptive load balancing in EJB clustering system, in: ISKE 2008, 3rd International Conference on Intelligent System and Knowledge Engineering, Xiamen, 2008, pp. 1388–1392.
[57] D. Ferrari, S. Zhou, An empirical investigation of load indices for load balancing applications, in: Proceedings of 12th Annual International Symposium of Computer Performance Modeling, Measurement and Evaluation, 1987, pp. 515–528.
[58] T. Kunz, The influence of different workload descriptions of a heuristic load balancing scheme, IEEE Transactions on Software Engineering (1994) 60–79.
[59] F. Bonomi, A. Kumar, Adaptive optimal loadbalancing in a heterogeneous multiserver system with a central job scheduling, IEEE Transactions on Computers (1990) 1232–1250.
[60] W. Shu, M.Y. Wu, Runtime incremental parallel scheduling (RIPS) on distributed memory computers, IEEE Transactions on Parallel and Distributed Systems (1996) 637–649.
[61] F. Lin, R. Keller, The gradient model load balancing method, IEEE Transactions on Software Engineering (1987) 32–38.
[62] M.R. Spiegel, L.J. Stephens, Estadística, 3rd ed., McGraw Hill, Mexico, 2002.

Javier Espadas is a Ph.D. graduate of the Information and Communication Technologies Program at the Tecnológico de Monterrey, Campus Monterrey. He is also a Researcher and Software Architect at Center of Innovation in Design and Technology of the same institute since April 2004. He has worked in different national and international projects regarding the development of IT-platforms based-on state-of-the-art approaches such as service-oriented architectures (SOA) and Software-as-a-Service (SaaS) models in collaborative environments. He was awarded by the Microsoft Latin American Internship Program 2009 with a 3-month internship at the Cloud Computing Futures Group at Microsoft Research in Redmond, Washington.

Arturo Molina is the General Director of the Tecnológico de Monterrey, Campus Mexico City. He received his Ph.D. degree in Manufacturing Engineering at Loughborough University of Technology, England (1995), his University Doctor Degree in Mechanical Engineering at the Technical University of Budapest, Hungary (1992), and his Master's Degree in Computer Science from Tecnológico de Monterrey, Campus Monterrey, Mexico (1992). He is member of the National Researchers System of Mexico (SNI-Level II), Mexican Academy of Sciences, and member of IFAC TC-WG 5.3 on Enterprise Integration and Enterprise Networking, IFIP WG 5.12 on Enterprise Integration Architectures and IFIP WG 5.3 Cooperation of Virtual Enterprises and Virtual Organizations.

Guillermo Jiménez received his Ph.D. in Computer Science from Tecnológico de Monterrey, where he is full professor since 1993 at the Computer Science Department. His areas of interest are Component Based Software Development, Software Product Lines, Service Oriented Computing, and Architectures for Enterprise Integration.

Martín Molina is the Chairman of the Computer Science Graduate Programs Department at Tecnológico de Monterrey, Campus Mexico City. He received his Ph.D. and Master degrees in Computer and Telecommunications at Institut National Polytechnique de Toulouse (INPT), France (2003/1999). His research areas include coordination, cooperation and communication on mobile and distributed systems, SOA and cloud computing. Dr. Molina has participated in several national and international research projects.

Raúl Ramírez finished a Chemical & Industrial Engineering degree in May, 1988 at Instituto Tecnólogico y de Estudios Superiores de Monterrey (ITESM), Campus Monterrey, Mexico. He finished a Master Degree on Science with specialty in Computer Science in July 1991. He has been full time professor at ITESM since August de 1991. He reached Associate Professor Level in 2003. He finished a doctorate degree at ITESM in February 2004 in Information Technology with specialty in Multimedia Distributed Systems. He has done research in collaborative distributed systems, information technology architecture and infrastructure engineering, computer graphics and animation, and networked multimedia.

David Concha is a Researcher and Programmer at Center of Innovation in Design and Technology of the Tecnológico de Monterrey, Campus Monterrey since April 2004. From this institute, he also received his Master's Degree in Information Technologies (2006). As part of the research center, he has worked in different national and international projects related to on-demand databases, business process management and IT-platforms development. He was an invited researcher to the Hewlett-Packard (HP) Lab at Palo Alto, California from April 2007 to April 2008.