ISSN 22773061

Cloud based Cost Effective Resource Allocation Model for Software-as-a-service Deployments

Ibrahim A. Cheema[1], Mudassar Ahmad[2], Fahad Jan[1], Shahla Asadi[2]
[1]Department of Computer Science, University of Agriculture, Faisalabad, Pakistain.
[2]Department of Computing, Universiti Teknologi Malaysia, Johor Bahru, Malaysia.
[1]icheema313@gmail.com, [2]mudassar.utm@gmail.com, [1]fahad@uaf.edu.pk, [2]asadi.shahla2003@gmail.com

ABSTRACT

The Cloud Computing (CC) provides access to the resources with usage based payments model. The application service providers can seamlessly scale the services. In CC infrastructure, a different number of virtual machine instances can be created depending on the application requirements. The capability to scale Software-as-a-Service (SaaS) application is very attractive to the providers because of the potential to scale application resources to up or down, the user only pay for the resources required. Even though the large-scale applications are deployed on cloud infrastructures on pay-per-use basis, the cost of idle resources (memory, CPU) is still charged to application providers. The issues of saturation and wastage of cloud resources are still unresolved. This paper attempts to propose the resource allocation models for SaaS applications deployments over CC platforms. The best balanced resource allocation model is proposed keeping in view cost and user requirements.

Indexing terms/Keywords

Cloud Computing, Software-as-a-Service (SaaS), Cloud Resource Allocation

Academic Discipline And Sub-Disciplines

Networking, Computer Science, Communication, Application Software;

SUBJECT CLASSIFICATION

Computer Science Subject Classification;

TYPE (METHOD/APPROACH)

Empirical analysis; Experimental.

Council for Innovative Research
Peer Review Research Publishing System
Journal: INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY
Vol 9, No 2
editor@cirworld.com
www.cirworld.com, member.cirworld.com

ISSN 22773061

INTRODUCTION

The Cloud computing (CC) covers two main areas which include applications delivery over the Internet as a service and system software deployed in datacenters offering the services normally on pay-per-use basis pricing model [1]. With CC a variable number of virtual machine instances can be created depending on the application requirement and this is actually the elasticity feature of this computing technique [2]. The applications deployed on CC are 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 CC to create highly scalable applications, solving performance problems through CC 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, Infrastructure-as-a-Service (IaaS) 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 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 × 24 = 7200 server-hours; but since we must provision to the peak of 500 servers, we will pay for 500 × 24 = 12 000 server-hours, a factor of 1.7 more than what is needed.’’

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 plat-forms [9]. By imposing per-hour costs, CC 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 CC approaches such as on-demand resource allocation through Simple Object Access Protocol (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.

MATERIAL AND METHODS

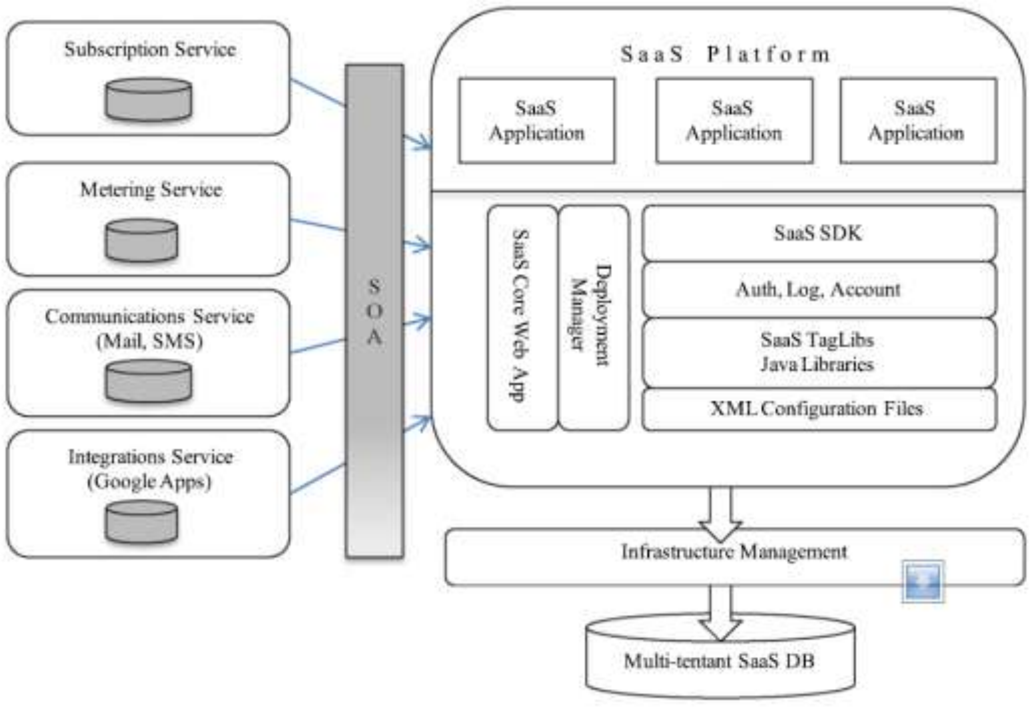

Fig. 1. SaaS platform architecture.

Test bed platforms and architectures

The test bed deployment has two main components: a Java-based SaaS platform and a private cloud platform, separately configured. As first step, the SaaS platform is deployed over a cloud infrastructure. The SaaS platform is composed of several components that allow the deployment of application as services (Fig. 1) [4]. 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 manages its own resources, such as data sources, libraries, and views.

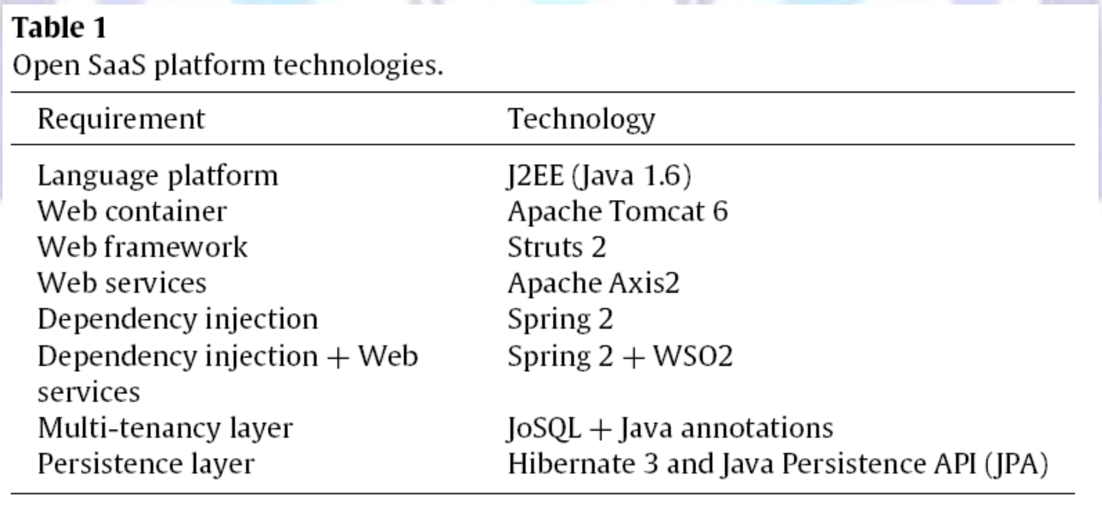

Table 1
Open SaaS platform technologies.

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)

As Table 1 [11] 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 framework1 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 application Context. 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) API2 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.

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 [7]. In contrast, the point of exhaustion for CC can be defined as the maximum useful payload that could be placed on a single virtual machine without adversely affecting the throughput [12]. 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 [13]. The SaaS platform uses the HTTP request throughput calculated by JMeter tool (explained later). This throughput value is calculated as requests/unit of time [14]. 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 [15] use the throughput to define an 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 [13]. By correlating the percentage of resource utilization at maximum throughput, it is possible to detect when resource utilization is saturated by the workload [15]. 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.

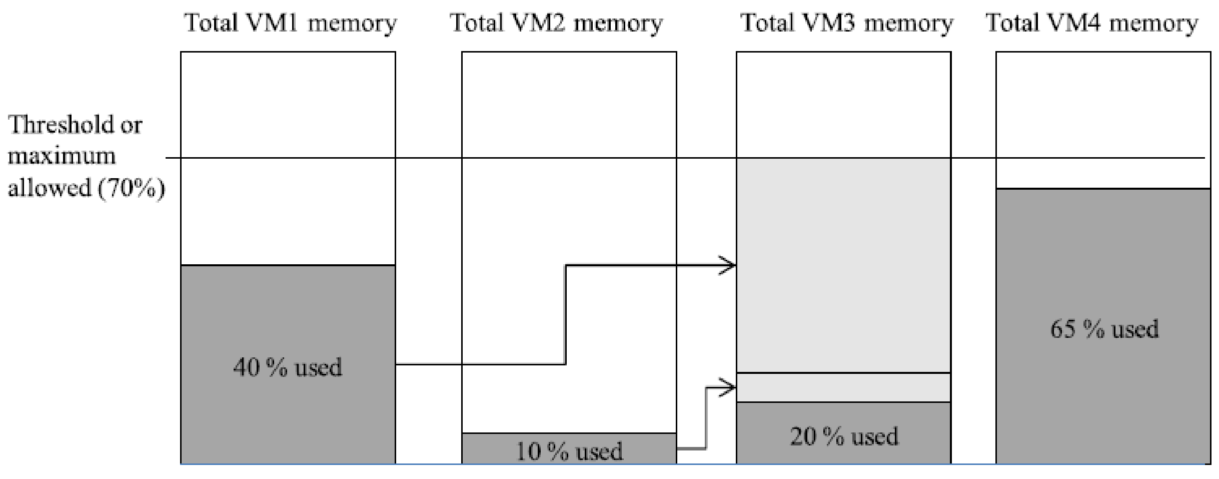

Fig. 2. Memory underutilization.

Underutilization (resource wasting)

Resource underutilization occurs whenever some resources are not being used by virtual machines within a CC infrastructure and an application is being executed [16]. Resource underutilization can be measured by the amount of resources available to be used by potential virtual machines and applications. In this sense, work [17] 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. 2 [14] depicts a scenario where the used heap memory is measured within four virtual machines. According to Fig. 2, 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 approach3 [18] by calculating the combination of VM instances that can be allocated by another single VM, according to the measured resource (CPU or heap memory) .

As it is not the aim of this work to detail or solve the knapsack problem, this work 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.

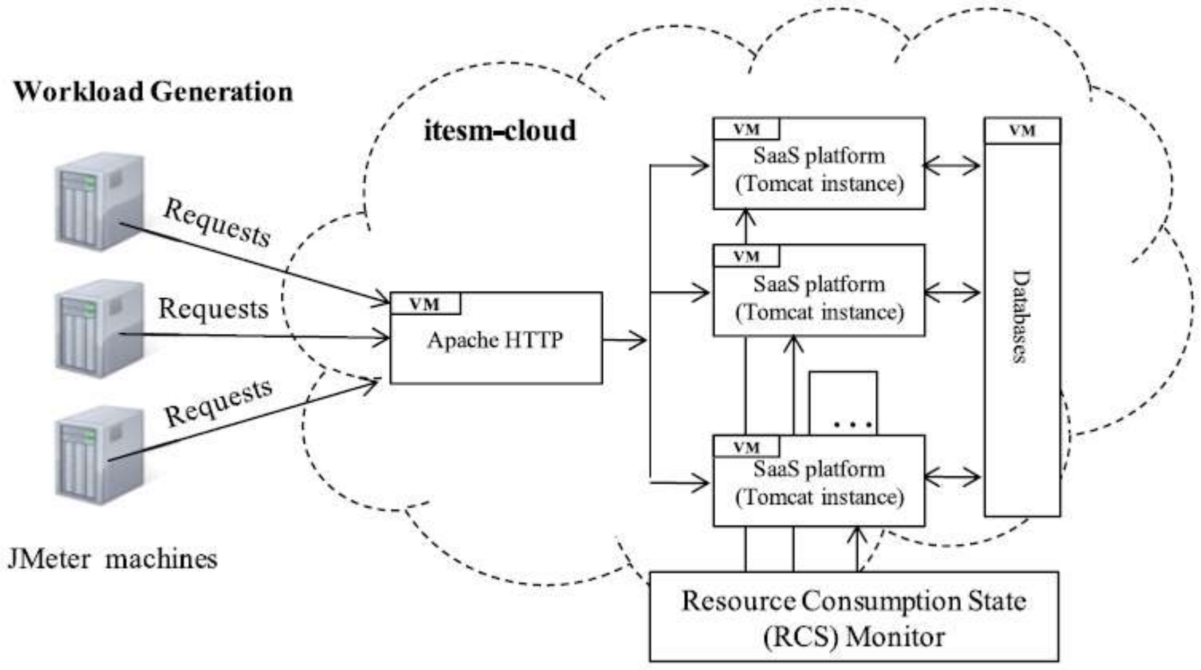
Workload Generation
itesm-cloud
Requests
Requests
Requests
JMeter machines
VM
Apache HTTP
VM
SaaS platform (Tomcat instance)
VM
SaaS platform (Tomcat instance)
...
VM
SaaS platform (Tomcat instance)
VM
Databases
Resource Consumption State (RCS) Monitor

Fig. 3. Test bed architecture.

Generating workload

The Apache JMeter tool was useded 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 [14]. 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) [5]. Simulating concurrent users by JMeter can be employed in an independent computer; they can also be employed in a distributed testing framework [15]. For this work, JMeter will be configured to create distributed requests to simulate workload usage. Fig. 3 [19] 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 [6], while the JMeter machines run the tests, an RCS Monitor will be collecting information about 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 [17]. 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 [11], 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 [14]. Assuming different behaviors during twelve months, as presented in [8], different types of workload peaks are stated for SaaS requirements [1,15].

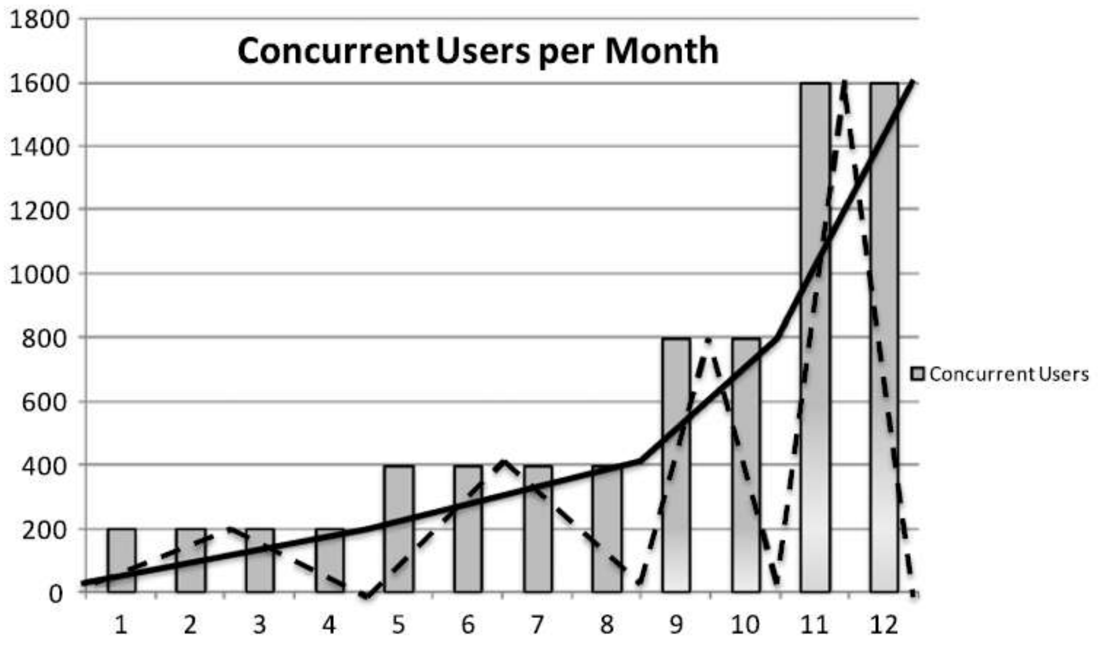

Fig. 4. Workload simulation.

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

Two types of workload generation are as follows [14, 18]:

– 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. 4 [12]).

– 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. 4).

Test bed results

The test plans results are elaborated as follows. 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. 5 [4] 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 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. 5, 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

ISSN 22773061

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 [20]:

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	1440	24.57	4.78	25.93	9.86
Feb	2	1440	13.09	12.52	13.10	14.24
Mar	2	1440	10.37	17.94	8.58	23.77
Apr	2	1440	8.15	33.88	34.85	7.44
May	4	2880	27.32	2.96	37.27	3.46
Jun	4	2880	16.76	9.43	16.91	21.34
Jul	4	2880	9.09	14.07	14.54	30.48
Aug	4	2880	7.59	22.09	42.92	8.73
Sept	8	5760	31.61	6.13	55.19	0.46
Oct	8	5760	11.57	14.18	26.07	10.03
Nov	16	11520	40.55	9.12	51.93	2.74
Dec	16	11520	20.30	11.96	22.17	14.10
	Averages		**18.42**	**13.26**	**29.12**	**12.22**

%UU(Underutilization) = (Combined UU/Measurements per hour)/Server-hours. (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 serverhours 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:

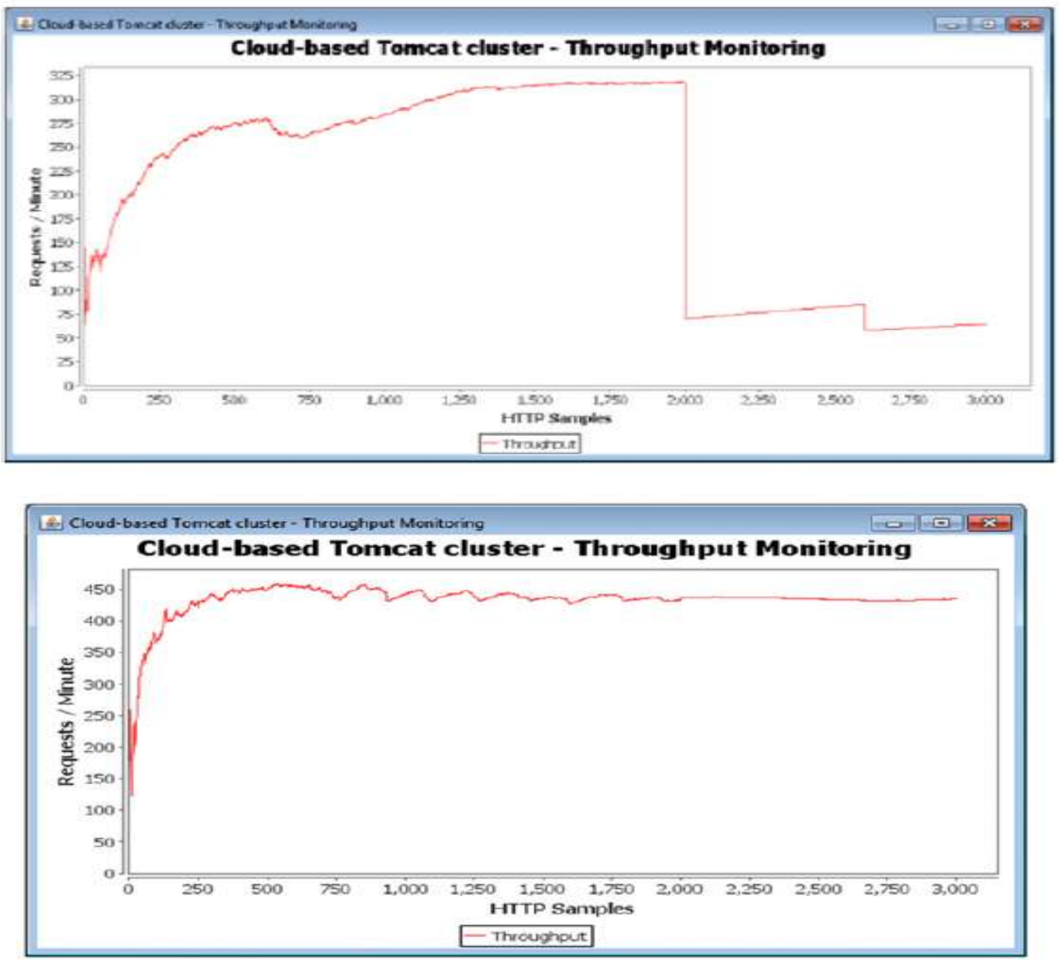

Fig. 5. Throughput measurement.

%OU(Overutilization) = Combined OU/(Measurements per month * Number of virtual machines). (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.

RESULTS AND DISCUSSION

To address under and over utilization issues, this work recommends a model for allocating workload when deploying SaaS platform and its applications over cloud infrastructures. This model comprises three approaches that take advantage of the multitenancy 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. 6 [21] depicts the architecture of the tenant-based VM instance creation model.

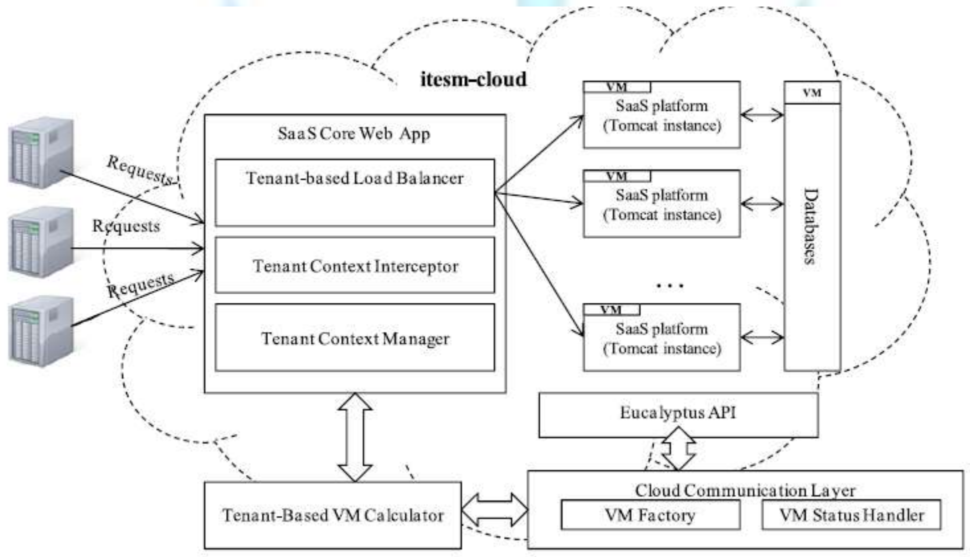
itesm-cloud
Requests
Requests
Requests
SaaS Core Web App
Tenant-based Load Balancer
Tenant Context Interceptor
Tenant Context Manager
VM
SaaS platform (Tomcat instance)
VM
SaaS platform (Tomcat instance)
VM
SaaS platform (Tomcat instance)
VM
Databases
Eucalyptus API
Tenant-Based VM Calculator
Cloud Communication Layer
VM Factory
VM Status Handler

Fig. 6. Tenant-based VM instance creation architecture.

VM allocation

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

Tenant Contextweight = Active users $*$ (heap size / maxThreads). (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):

VMcapacity = Heap size − ((heap size/maxThreads) $*$ platform threads). (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-BasedVMCalculator 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 282 J. Espadas et al. / Future Generation Computer Systems 29 (2013) 273–286 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 [22]. The allocation problem can be expressed as showed in formula (5) (adapted from [8, 22]):

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

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.

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 gainst 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 ofVMinstances that were created through tenant-based demand.

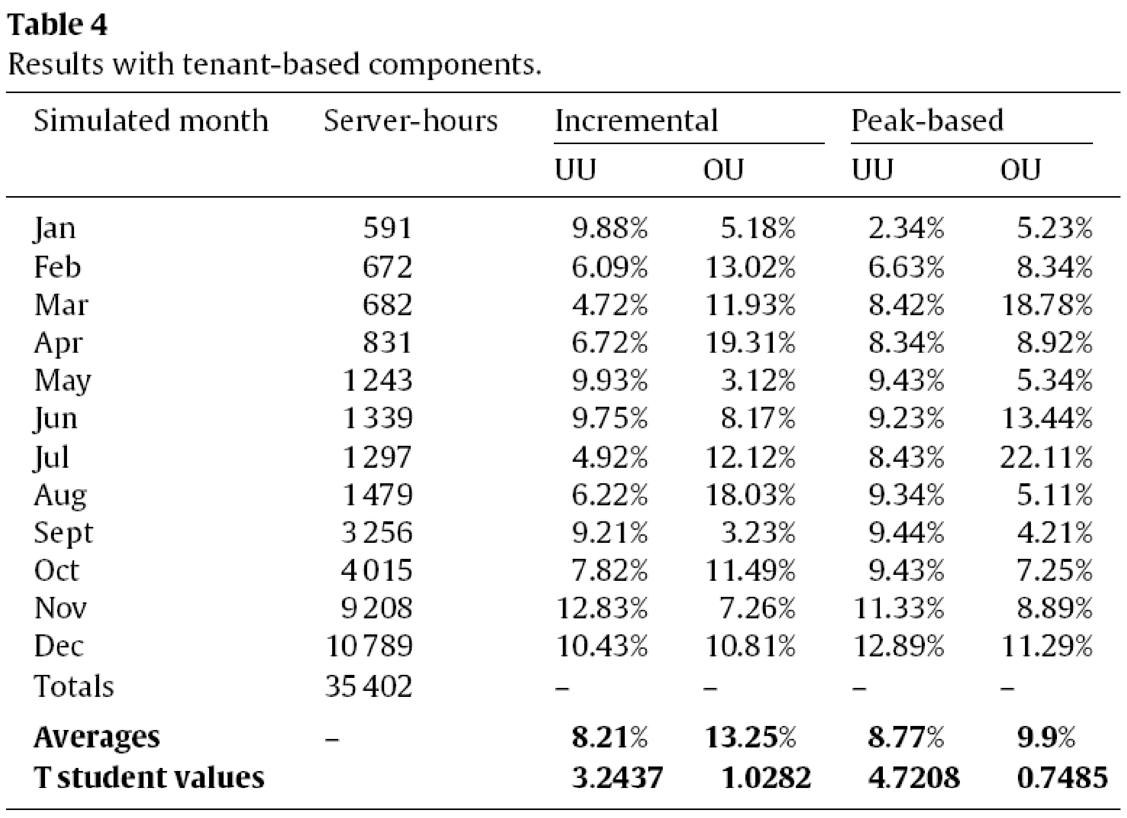

Table 4
Results with tenant-based components.

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**

In Table 4, the server-hours were reduced from 51 840 to 35 402, 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 [13], if averages of Table 4 are statistically less than those in Table 3. With N as a number of months (samples), we have (N1 + N2 − 2) = (12 + 12 − 2) = 22 degrees of freedom and we set an accuracy of 99.5% (α = 0.005 of significance), t-student distribution table produces a value of tα = 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 tα 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 tstudent test is represented in formula (6) [21].

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

where X1 is the average and S1 the standard deviation of columns of Table 4 results. X2 and S2 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 tα (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 tα; even if the averages were reduced, we cannot conclude that there was a statistical improvement for such behavior.

CONCLUSION

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.

FUTURE WORK

Resource allocation has a significant impact in CC, 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, 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.

REFERENCES

1. Fox, A., et al., Above the clouds: A Berkeley view of cloud computing. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 2009. **28**.
2. Buyya, R., et al., Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation computer systems, 2009. **25**(6): p. 599-616.
3. Espadas, J., D. Concha, and A. Molina. Application development over software-as-a-service platforms. in Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on. 2008: IEEE.
4. Iakymchuk, R., J. Napper, and P. Bientinesi, Underutilizing Resources for HPC on Clouds. 2010.
5. Blog, A., Amazon CEO Jeff Befos on Animoto, in http://animoto.com/blog/company/amazon-com-ceo-jeff-bezos-on-animoto/. April 2008.
6. Hu, Y., et al. Resource provisioning for cloud computing. in Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research. 2009: ACM.
7. Meng, X., et al. Efficient resource provisioning in compute clouds via vm multiplexing. in Proceedings of the 7th international conference on Autonomic computing. 2010: ACM.
8. Stillwell, M., et al., Resource allocation algorithms for virtualized service hosting platforms. Journal of Parallel and Distributed Computing, 2010. **70**(9): p. 962-974.
9. Jie, Y., Q. Jie, and L. Ying. A profile-based approach to just-in-time scalability for cloud applications. in Cloud Computing, 2009. CLOUD'09. IEEE International Conference on. 2009: IEEE.
10. Mc Evoy, G.V. and B. Schulze. Using clouds to address grid limitations. in Proceedings of the 6th international workshop on Middleware for grid computing. 2008: ACM.
11. Parra-Hernandez, R., D. Vanderster, and N.J. Dimopoulos. Resource management and knapsack formulations on the grid. in Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on. 2004: IEEE.
12. Paroux, G., et al. A java cpu calibration tool for load balancing in distributed applications. in Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on. 2004: IEEE.
13. Mishra, A.K., et al., Towards characterizing cloud backend workloads: insights from google compute clusters. ACM SIGMETRICS Performance Evaluation Review, 2010. **37**(4): p. 34-41.
14. Wu, Q. and Y. Wang. Performance testing and optimization of J2EE-based web applications. in Education Technology and Computer Science (ETCS), 2010 Second International Workshop on. 2010: IEEE.
15. Wee, S. and H. Liu. Client-side load balancer using cloud. in Proceedings of the 2010 ACM Symposium on Applied Computing. 2010: ACM.

16. Dyachuk, D. and R. Deters, A solution to resource underutilization for web services hosted in the cloud, in On the Move to Meaningful Internet Systems: OTM 2009. 2009, Springer. p. 567-584.

17. Matthews, C. and Y. Coady. Virtualized recomposition: Cloudy or clear? in Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing. 2009: IEEE Computer Society.

18. Vanderster, D.C., et al., Resource allocation on computational grids using a utility model and the knapsack problem. Future Generation computer systems, 2009. **25**(1): p. 35-50.

19. Kounev, S., B. Weis, and A. Buchmann, Performance tuning and optimization of J2EE applications on the JBoss platform. Journal of Computer Resource Management, 2004. **113**(113-116): p. 41.

20. Cao, J., et al., Grid load balancing using intelligent agents. Future Generation computer systems, 2005. **21**(1): p. 135-149.

21. Harik, L. and A. Kayssi. FPGA-based load balancer for Internet servers. in Microelectronics, The 14th International Conference on 2002-ICM. 2002: IEEE.

22. Zitzler, E., M. Laumanns, and L. Thiele, SPEA2: Improving the strength Pareto evolutionary algorithm. 2001, Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK).

Author' biography with Photo

Ibrahim A. Cheema is a MS student at Department of Computer Science, University of Agriculture, Faisalabad. His research interest area is Cloud Computing (CC), Software-as-a-Service (SaaS) and Mobile Cloud Computing (MCC). He has worked in manufacturing industry for development of IT Infrastructure specially Databases and Virtualization.

Mudassar Ahmad has completed his MS computer Sciences from University of Agriculture, Faisalabad. He is doing Ph.D. in Computer Sciences from Universiti Teknologi Malaysia, Johor Bahru, Malaysia. His core research area is Network Protocols.

Fahad Jan is MS in Computer Sciences and Lecturer at Faculty of Sciences, Department of Computer Science, University of Agriculture, Faisalabad. His main area of research is in the design and implementation of advanced programming languages, distributed databases and information integration.

Shahla Asadi is a PhD student in University Technology Malaysia in Information Technology Management. She has completed her Master from University Technology Malaysia in 2013 and B.S Computer Science in 2005. She taught the students in Database Lab, Department of Computer Engineering, Tarbiat Moallem University of Tehran in 2006.