A Tenant-based Resource Allocation Model Application in a Public Cloud

Wojciech Stolarz, Marek Woda

Tieto Poland Sp. z o.o., Aquarius, Swobodna 1, 50-088 Wroclaw, Poland
Department of Computer Engineering, Wroclaw University of Technology,
Janiszewskiego 11-17, 50-372 Wroclaw, Poland
wojciech.stolarz@tieto.com, marek.woda@pwr.edu.pl

Abstract: The aim of this study is to check the proposed tenant-based resource allocation model in practice. In order to do this, two SaaS systems are developed. The first system utilizes traditional resource scaling based on a number of users. It acts as a reference point for the second system. Conducted tests were focused on measuring over- and underutilization in order to compare cost-effectiveness of the solutions. The tenant-based resource allocation model proved to decrease system's running costs. It also reduces system resource underutilization. Similar research has been done before, but the model was tested only in a private cloud. In this work, the systems are deployed into commercial, public cloud.

Keywords: Cloud computing; multi-tenancy; SaaS; TBRAM

1   Introduction

Cloud computing is gaining more and more interest every year. Cloud computing is not a new technology. It is rather a mixture of technologies existing before, like: grid computing, utility computing, virtualization or autonomic computing [14], and it finds application in other seemingly indirectly related areas, like hybrid wireless networks [16] integration of information systems [17] or high performance simulation [18].

The National Institute of Standards and Technology (NIST)[1] defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

---

[1]   NIST Cloud Computing Standards Roadmap, Special Publication 500-291, Version 2, July 2013

This cloud model is composed of five essential characteristics, three service models, and four deployment models".

This approach allows Internet based applications to work in distributed and virtualized cloud environment. It is characterized by on-demand resources and pay-per-use [1] pricing. Nowadays, every respected IT-company has started to think about providing its services in the cloud [5]. Currently Cloud computing is one of major enablers for the manufacturing industry [3]. It became widely used to enhance many other aspects of industrial commerce by moving business processes to the cloud to improve the companies' operational efficiency. Currently, a new trend can be observed, inspired by cloud computing, it is illustrated in the movement from production-oriented manufacturing to service-oriented manufacturing. It converges networked manufacturing, manufacturing grid (MGrid), virtual manufacturing, agile manufacturing and Internet of Things into cloud manufacturing [2], [3], where distributed resources provided by cloud services are managed in a centralized way. Clients can use the cloud services according to their requirements. Cloud users can request services ranging from product design, manufacturing, testing, management and all other stages of a product life cycle [3].

Software-as-a-Service (SaaS) is software delivery model in which entire application (software and data) is hosted in one place (usually in the cloud). The SaaS application is typically accessed by the users via a web browser. It is the top layer in cloud computing stack. SaaS evolved from *SOA (Service Oriented Architecture)* and manages applications running in the cloud. It is also seen as a model that extends the idea of *Application Service Providers (ASP)*. *ASP* is primary centralized computing model from the 1990s. SaaS platform can be characterized by: service provider development, Internet accessibility, off-premises hosting and pay-as-you-go pricing [3]. The SaaS platform supports hosting for many application providers. As opposed to ASP model, SaaS provides fine-grained usage monitoring and metrics [8]. It allows tenants to pay according to the usage of specific cloud resources. SaaS applications often conform to multi-tenant architecture, which allows a single instance of a program to be used by many tenants (subscribers) [3]. This architecture also helps to serve more users because of a more efficient resource management than in multiple instances approach [4].

In spite of the fact that the idea of cloud computing utilization has become a reality, questions like how to enhance resource utilization and reduce the resource and energy consumption are still not effectively addressed [1].

Since most cloud services providers charge for the resource use, it is important to create resource efficient applications. One of the ways to achieve this is multi-tenant architecture of SaaS applications. It allows the application for efficient self-management of the available resources.

Despite the fact, that in the cloud, one can automatically receive on-demand resources, one can still encounter problems related to inappropriate utilization of the available resource pool at a particular time. These issues manifest in over- and underutilization, which exists, because of the "not fully elastic", pay-per-use model used nowadays [10]. Over provisioning arises when, after receiving additional resources (in reply for peak loads), one keeps them, even if they are not needed any more. Such a situation is called underutilization. Under provisioning (saturation) arises when one cannot deliver required level of service because of insufficient performance. This is also known as an overutilization. It leads to customers turnover and revenue losses [1]. *Amazon Elastic Cloud Computing (EC2)* service charges users for every partial hour they reserve each EC2 node. Paying for server-hours is common among cloud providers. That is why it is very important to utilize fully given resources in order to really pay just for what we use.

Due to the fact that we are still in early stages of cloud computing development, we cannot expect cost-effective pay-per-use model for SaaS applications after just deploying it in the cloud. What is more, automatic cloud scalability will not work efficiently if applications consume resources, which are indispensable to meet the desired performance levels [13]. To achieve desired scalability we need to design our SaaS application with that in mind. In order to do so, the application must be aware how it is used [7]. We can use multi-tenant architecture [9] to manage the application behavior. It allows using a single instance of the program for many users. It works in similar way like a singleton class in object programming languages, which can supervise creation and life cycle of objects derived from that class. Supporting multiple users is a very important design step for SaaS applications [2]. We can distinguish two kinds of multi-tenancy patterns: multiple instances (every tenant has got their own instance running on shared resources) and native multi-tenancy (single instance running on distributed resources)[2] [2]. The first pattern scales well for a small number of users, but if there are more than hundreds of users, it is better to use the second pattern.

The one of most recent solutions for over- and under- utilization problems may be a tenant-based resource allocation model (TBRAM) for SaaS applications. That solution was introduced and tested with regard to CPU and memory utilization by various authors [3]. They proved the validity of TBRAM through the reduction of used server-hours as well as improving the resources utilization. However, the authors deployed their solution into a private cloud which can only mimic a public cloud environment. They tested cases with incremental and peak workload of the system. In this paper we wanted to check whether the TBRAM is really a valuable system. Examining the TBRAM system in a public and commercial cloud environment could deliver the answer to that question. Therefore, the main aim of

---

[2]   Architecture Stfrategies for Catching the Long Tail: 2006.
        http://msdn.microsoft.com/en-us/library/aa479069.aspx. Accessed: 2014-09-07

the paper is the further validation of TBRAM approach, as it was proposed in the future research part of the base work [3]. If the results of the study confirm improvement of the model performance, then it could be considered as the solution to the previously mentioned provisioning problems.

2   System Design

2.1   Base System

In this section the base tenant-unaware resource allocation SaaS system (Base System) is described. It conforms to a traditional approach to scaling resources in a cloud and is based on the number of users of the system. It is substituted by *Elastic Load Balancer* service. According to AWS Developer Forum[3] the *Elastic Load Balancer (ELB)* sends special requests to balancing domain's instances to check their statuses (health check) it then round-robins among the healthy instances, having fewer outstanding requests pending. Although the name of this system suggests lack of awareness of tenants it concerns only resource allocation. The system was built according to *Service Oriented Architecture (SOA)* and native multi-tenancy pattern. First, it was implemented as a set of J2EE web applications using *Spring* and *Struts* frameworks. Several parts of the system were later transformed to web services using *WSO2* and *Axis2*. Deploying application as a web service makes it independent of the running platform. It also gives more flexibility when accessing the application.
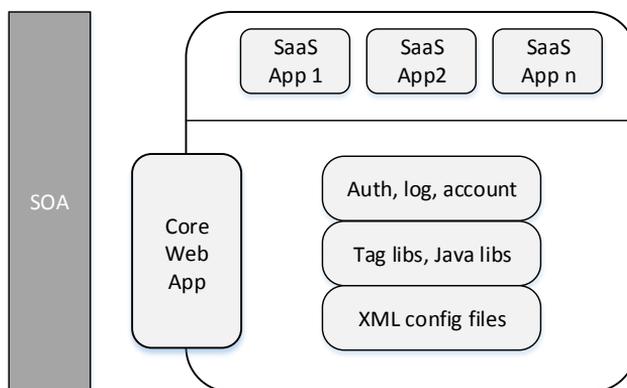

Figure 1

SaaS platform system architecture

A general, high level, overview of the test-bed architecture is shown in Fig. 3, one can see there a group of *Amazon EC2* (*Tomcat*) instances. The number of the instances varies and it depends on the number of simulated users. Each of the instances consists of a Virtual Machine (VM) with one Tomcat web container. In each Tomcat container authorial a SaaS platform is deployed. The platform is the main part of the system as it makes a basis for SaaS applications. It also includes web services and common libraries.

The SaaS platform (Fig. 1) was the main part of the system. The task of the platform was to support deployment of plain web applications as a SaaS service. The idea behind the design of this part was inspired by this work [3]. Their SaaS platform was developed as a part of the *Rapid Product Realization for Developing Markets Using Emerging Technologies* research at *Tecnológico de Monterrey University, Mexico*.

Since the authors had limited means and limited time, The SaaS platform was simplified and focused only on the usability for SaaS system.

The entry point to the platform was the *Core Web App (CWA)*. As the name suggests, it was a web application that worked as a gate. All interactions between outside environment and the parts of the platform were done through this element. In the background, there were applications responsible for users' authorization, account management and logging. The platform contained also common Java libraries used by deployed applications. Configuration was made by the XML or plain text files. The platform exhibited web service interfaces to be consumed by outside applications. One example of such an interface was the interface for metering services. It allows monitoring the usage of resources by the platform. That behavior is depicted by the SOA element in Fig. 1. On top of this, we can see the SaaS applications. These were developed as normal Java web applications, but when deployed on the platform, they gain extra SaaS functionality. Two were implemented, a Sales application and a Contacts Application. It is assumed that two applications are enough to present the platform's functionality as well as the interactions between the deployed applications. It is worth mentioning that there is one more feature, which was not depicted in Fig. 1, the communication channel between the platform and an external database server.

In order to measure over- and under- utilization certain metrics from running VMs were needed. Some of them were available directly through *Amazon CloudWatch* metering service (CPU usage, network in/out and number of requests per second in case of the ELB). The latter metric can be used to calculate the throughput of the entire system. Other source of data for this metric came from *JMeter* tool. However, monitoring of RAM consumption and number of running threads was not provided by the *CloudWatch*. That is why authors developed a monitoring solution called *Resource Consumption Monitor (RCM)* – a service that sat between the monitoring domain and the *CloudWatch*.

There are two main approaches to the monitoring problem. The first is a distributed approach. It is similar to *Observer Pattern* [6] known from object oriented programming patterns and best practices. In this case monitored VMs register themselves to the monitoring service and then publish their measurements. The main strength here, is build simplicity. It has, however, one big disadvantage – each worker VM needs to be aware of the monitoring process. It is also hard to quickly notice a VM termination due to unexpected events or errors. That is why the second, centralized approach was chosen. In this case, the VMs with SaaS platforms are unaware of being monitored as it is beyond their consideration. The RCM is constantly monitoring the state of VMs by polling AWS cloud (the performance hit was negligible, it was not included in the considerations). After each interval (Polling interval) it collects the measurements from the monitored domain. After another interval (Publishing interval) it publishes collected data to the *CloudWatch* service. Thanks to that, all of the VMs measurements were available in one place.

The *RCM* is a web application, but it can also be used as a standalone console Java application packed into an archive file (jar). It used *the Java Management Extension (JMX)* RMI-based protocol which allows to request information about running Java Virtual Machine. Generally, it is recommended to use an authorial web services to fulfil the same tasks, but since the entire SaaS system is too overly complex, the flexibility offered by web services seems not to be really needed. Especially, that such flexibility comes with a price. First of all, the JMX packets are much smaller than competitive SOAP protocol ones. Therefore, it reduces network traffic and time necessary to decode the packet. The next reason is the requirement for management of web services like Axis2. The *JMX* is built in *Java Runtime Environment (JRE 1.7)*, which is used by authors. Finally, the *JMX* technology is far more robust and advanced. It would be difficult to build a better web service within such a short time frame. It is also transparent for applications running on *JVM*. All what is necessary to do, is to add extra running parameters when starting the JVM.

The *RCM* requires a set of parameters to run. One of them is the running mode which tells the monitor whether to run in test mode (very frequent data collecting, but without publishing them to the *CloudWatch*) or in normal mode (with synchronization to the *CloudWatch*). The chosen mode affected both (polling and publishing) intervals. In test mode the data were gathered every 5 seconds. In normal mode polling was set to 10 seconds and publishing to 1 minute. These settings matched the settings of *CloudWatch* service working in detailed mode. Using RCM one could also manually start or stop the monitoring of certain VM. To tell the RCM which instances to monitor a special tag to VMs was added. The tagging is a feature in AWS cloud that helps to organize running instances. The most common usage of tags is for giving names to the instances which are often more meaningful than their IDs.

Because owned metrics are sent to the *CloudWatch* it was crucial that all the measurements (direct and own) for given VM are taken in the same time. They could be published asynchronously, because they contained a timestamp tag. However, the measurement data itself needed to be synchronized in time. Otherwise they would be invalid. To avoid that synchronization mechanism was implemented in the RCM. It was assured that own measurements data are collected at the same moment as the direct *CloudWatch* data.

The RCM was deployed on a dedicated *t1.micro EC2* instance, because it *should* not affect the work of virtual machines it monitored. Thanks to its web interface, it can be managed from any computer via a web browser.

2.2   Tenant Aware System

The base SaaS system was implemented as a reference tenant-unaware resource allocation system. The main flaw of its design was rigid management of VM instances in the cloud. Thus, it could lead to serious over- and underutilization problems. In Chapter III, we show this based on test results. One of the ways to tackle aforementioned issues was proposed in [3] as a tenant-based resource allocation model (TBRAM) for scaling SaaS applications over a cloud infrastructure. By minimizing utilization problems, it should decrease the final cost of running the system in the cloud.
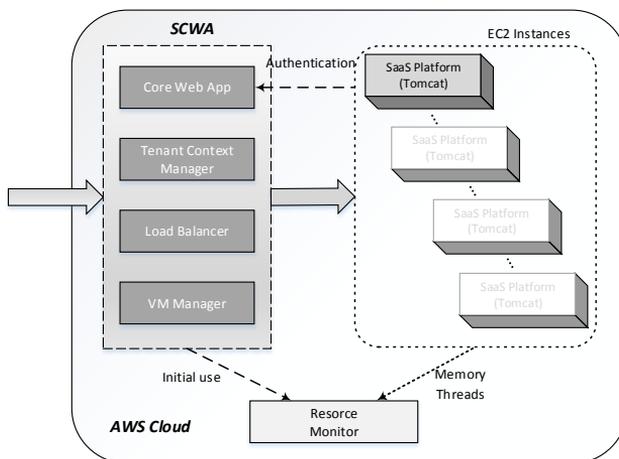

Figure 2
TBRAM system architecture

The TBRAM consists of three approaches that use multi-tenancy to achieve its goals. The first approach is tenant-based isolation [11], [12], which separates contexts for different tenants. It was implemented with tenant-based authentication and data persistence as a part of the SaaS platform (Tomcat instances). The second, is to use tenant-based VM allocation [11], [12]. With this

approach we are able to calculate the actual number of the VMs needed by each tenant in a given moment. The third approach is the tenant-based load balancer that allows to distribute the virtual machines' load with respect to certain tenant. An overview of the architecture is presented in Fig. 2. The dashed line in the picture denotes communication with web services. One can notice that the *SaaS Core Web App (SCWA)* element in the Fig. 2 is the only change made to the original test-bed [3].

A simple load balancer based on round-robin IP address algorithm, is not the best solution to isolate each tenant. Since users from the same tenant share certain tenant-related data it would be a good idea to dispatch their requests to the same VM (if possible). That could reduce the amount of tenant data kept by each VM since some of them would be serving only a few tenants. That could also lead to better usage of servers' cache mechanisms by concentrating on data that are really shared by number of tenant's users. That in turn could for example, reduce a number of requests to database engine.

The key task of the load balancer was to isolate requests from different tenants. The tenant-based load balancer worked in 7th layer of the OSI model. It used information stored in the session context as well as local applications data to assign the load efficiently. The idea behind request scheduling is that requests from one tenant should be processed on the same VMs. If that is impossible, then the number requests should be limited, so they were not scattered along the whole balancing domain. That can also allow reducing context switching and using previously cached data. The traditional scheduling process uses only current status data, so it does not belong to dynamic load balancers family, but TBRAM-based solution is based on adaptive models of load balancing.

As suggested in other work [3], we made the load balancer a part of *SaaS Core Web App (SCWA)*. It was a natural choice to put that element there, since all the requests came through it anyway (because of the centralized authorization service). Design of the load balancer is similar to the one proposed in current work [3]. It consisted of five elements: *Request Processor*, *Server Preparer*, *Cookie Manager*, *Response Parser* and *Tenant Request Scheduler*. Each of them was responsible for specific function in the processing pipeline sequence. The most important was the last part of processing assigned to *Tenant Request Scheduler*. The scheduling policy enforces that the subsequent requests from the same tenant should be dispatched to the same VM. If a given VM was saturated, then the scheduler dispatched the request to the next available VM of that tenant. Finally, if no other VM was available, the scheduler requested a new VM from the VM Manager.

The HTTP as the Internet protocol was designed to be stateless. It means that every request is independent. It starts from handshaking in order to establish a connection. Then data exchange appears for one or possibly more server's resources. After that the connection is closed. When user requests another

resource the whole procedure repeats. However there was a need to keep a track of users action for example to make functioning of shopping chart possible in online shops. Because of private IP addresses it was not feasible to recognize all users just based on their IP. This is where the session mechanism comes with help. In general, it allows storing user related data at the server side and therefore distinguish each unique user. It works fine when there is only one server dealing with a given user, because of the limited session scope. If there are more servers this has to be handled differently. One of the solutions for that problem is clustering of Tomcat servers. But even better solution is to dispatch given user's requests in a unique server as it eliminates the need of session sharing. For that purpose many available load balancers offer so called session stickiness or session affinity. This feature allows grouping requests of a given user within a session scope and sending these requests to the same server. When it comes to tenant-based load balancer it could be called tenant stickiness or affinity. It can be imagined as yet another layer above the session layer which groups requests from a given tenant.

3   Test Results and Analysis

We tested our TBRAM-based SaaS system and compared it with the non-TBRAM version. The comparison was made in terms of overutilization, underutilization and financial cost. To calculate cost of running certain SaaS system in the cloud, billing statement delivered by Amazon was used. To measure over- and under-utilization of resources, measurements data collected by *Amazon CloudWatch* monitoring service were used. Before this was done the entire test bed was deployed in Amazon cloud environment (AWS). Then the system was stressed with the workload of HTTP requests. We used a cluster of *JMeter* machines performing test plans to achieve that. There was one test bed. The main difference in the architecture was in the entry point to the SaaS system. In the case of TBRAM it was the SCWA element including a load balancer, tenant context manager and virtual machines manager. In the case of the standard model it was just the *Elastic Load Balancer (ELB)* service from Amazon.

3.1   Over- and Under- utilization Results

Test results were collected by Amazon CloudWatch monitoring service. This tool allows to view some basic statistics of data in form of charts. However, in order to perform more advance analysis it was needed to download the raw data for further processing. The results are presented in following tables (Table 1, Table 2).

Table 1 presents results of the tests performed over the Base System that used traditional resource scaling. The results come from memory and CPU monitoring of the system. The first column contains the months of simulated year (24 hours of

tests). The second column presents the number of virtual machines running in each simulated month.
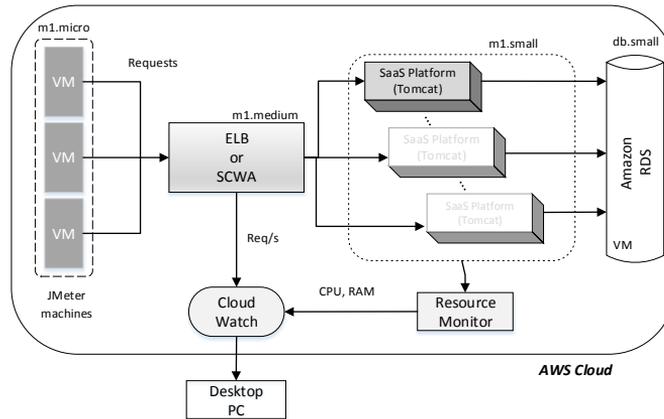

Figure 3

Test bed architecture

This number is valid only for the incremental workload tests. The peak-based tests were conducted in slightly different way. Instead of time constraints they were set to perform certain number of test plan iterations. We can notice the difference between the test types in server hours provided by the VMs each simulated month.

Table 1

Results of the Base System tests

Simulated month	VMs	Server-hours		Combined-incremental		Combined-peak	
		incremental	Peak-based	UU (%)	OU (%)	UU (%)	OU (%)
January	2	1440	2460	38.75	0.00	31.94	0.00
February	2	1440	2580	20.83	0.83	34.17	0.00
March	2	1440	2460	0.00	0.00	35.42	0.00
April	2	1440	2580	0.00	9.15	37.22	0.00
May	4	2880	4200	19.38	6.66	43.36	0.98
June	4	2880	5160	9.79	0.00	26.39	0.00
July	4	2880	4920	10.63	0.00	19.03	0.00
August	4	2880	8040	10.83	0.00	64.31	0.49
September	8	5760	9840	31.61	0.00	7.43	0.49
October	8	5760	10320	34.48	0.00	0.21	1.46
November	8	5760	9840	19.90	0.00	1.56	0.49
December	8	5760	7440	21.39	0.00	56.04	0.00
Total		**40320**	**69840**				
			Avg.	**18.13**	**1.39**	**29.76**	**0.33**

In the case of the incremental test the total value can be simply calculated by multiplying the number of VMs by the number of hours in the month (number of VMs * 24h * 30 days). In the case of the peak-based test such calculation is not straightforward. This is because peak-based tests were little longer than the original time frame of 24 hours (simulated year) per each test. The last four columns of the table contain *combined utilization*. This term describes a situation during the tests when a given VM was saturated or underutilized with respect to the both measures (CPU and memory). The combined utilization percentages were calculated based on formulas [3]:

$$\%UU(Underutilization) = \frac{\left(\frac{Combined\ UU}{Measurements\ per\ hour}\right)}{server\ hours} \qquad (1)$$

Formula (1) calculates the combined underutilization of a given VM per each time period. It yields a percentage of wasted VMs out of all available in that time. We could also divide the number of measurements when a VM was underutilized by the number of all measurements to get the information how often the UU occurred. This number oscillated around 50% for both test types. However, the measure defined by the formula (1) is more informative since it shows the size of the problem, not just the occurrence frequency. In case of overutilization the following formula was used:

$$\%OU(Overutilization) = \frac{Combined\ OU}{(Measurements\ per\ month * VMs\ number)} \qquad (2)$$

Overutilization informs us about a percentage of VMs that were saturated each month. It is calculated by dividing the number of measurements that had inflection points by all measurements performed during the measured time period (measurements per month * VMs number). As it can be noticed OU hardly appear in the tests. This is because the VMs workload was chosen with overutilization in mind. We did not want to saturate the VMs too much, but during the final tests the system behaved even better than expected. Therefore saturation of the machines was lower. Nevertheless, the tests of TBRAM system were conducted under exactly the same conditions, so it shouldn't bias the results. One can also observe the total number of server-hours provided by the SaaS platform VMs. It was 40320 and 69840 for the incremental and peak-based tests respectively.

Table 2 shows results of the tests of the TBRAM system which uses tenant-based load balancing and resource scaling. Since the VMs fleet size was adjusted to the current needs dynamically there is no corresponding VMs number column with fixed size for each month. The first observation is that the total server-hours were reduced by 19.94% and 30.21%, for the incremental and peak-based tests, respectively. The %OU was marginally smaller as in the case of the Base System. There was however a difference in %UU between the systems. First of all, we can notice that underutilization for incremental workload was smaller at the beginning of the simulated year as compared with the Base System. This is at least partially caused by the dynamic scaling method of TBRAM system. Whereas, the Base

System started with 2 VMs the second system could increase this number starting from only one machine. As it can be seen in Table 2 TBRAM system did not perform so well in the second part of the year.

Table 2

Results of TBRAM system tests

Simulated month	Server-hours		Combined-incremental		Combined-peak	
	incremental	Peak-based	UU (%)	OU (%)	UU (%)	OU (%)
January	720	1200	0.00	0.00	0.00	0.00
February	768	2040	0.00	0.81	13.52	0.65
March	1440	1920	0.00	2.44	0.83	0.65
April	1440	1440	0.00	1.63	26.25	3.26
May	1800	4440	2.78	0.00	0.00	0.00
June	2160	3960	10.56	6.50	2.08	2.60
July	2880	2280	15.83	0.81	16.89	2.60
August	3240	6000	1.19	0.00	31.54	0.65
September	3600	7680	29.17	0.81	1.62	0.65
October	3960	6720	40.02	1.63	20.94	0.00
November	4680	6000	54.45	0.81	25.00	0.65
December	5586	5064	51.59	1.63	35.42	0.00
Total	**32274**	**48744**				
		Avg.	**17.13**	**1.42**	**14.51**	**0.98**

It is important to notice that both averages for %UU are generally lower than in the case of traditional scaling system. In order to check if the TBRAM system leads to the significant improvement over the Base system, we used the t-test to compare UU% in peak-based test:

Table 3

Parameters of the t-student test

N	Degrees	Accuracy	α	t_α
12	22	97.50%	0.025	2.07

where, N is the number of samples (months). Degrees stands for degrees of freedom of the t-test and it is equal to ($N_1 + N_2 - 2 = 12 + 12 - 2 = 22$). Unlike the authors of the base paper the accuracy was chosen to be set to 97.5% rather than 99.5% because it was thought that test conducted in real public cloud environment is less predictable than a private cloud cluster. The t_α is a base parameter value from the t-student distribution table for the significance α = 0.025. If the t-student value for given columns in both tables is greater than the base parameter (2.074) then one can say with 97.5% certainty, that the column's averages are significantly different. We can see that the t-student value for the %UU in peak-based test is equal to 2.1854 (>2.074). Therefore the TBRAM system statistically improved that characteristic. However, %OU averages were

not improved according to the t-student test. The t-student values are given in TABLE3. They were calculated using the following formula:

$$t = \frac{x_1 - x_2}{\sqrt{\left(\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}\right)}} \tag{3}$$

where x_1 is an average and s_1 is a standard deviation of a column in Table1, where x_2 and s_2 are respective values for a column in Table 2.

3.2   Cost Analysis

Before we explain the cost analysis, a brief description of AWS pricing model is needed. Even despite this vendor specific model, the general idea of pay-per-use is common with cloud computing providers.

One of the main reasons for using the cloud infrastructure is its flexibility. AWS model is also flexible and is based on either pay-as-you-go and pay-per-use. The first one means there is no need for long term contracts neither for any minimal commitment. The latter one is strongly related to utility computing roots of the cloud computing. It means that we pay for what we used. Abovementioned fact is generally true, but with certain granularity, for example: each started running hour of EC2 instance. In case of the AWS there is also no need to pay up-front for any resource. We are also free to over-utilize or under-utilize our resources without any additional fees. There are three fundamental characteristic for which one pays in the AWS. These are: *CPU, Storage* and *Transfer OUT*.

In case of *computing,* we pay for each partial hour of our resources from start to stop of the instance. If it comes to *storage* we pay per each GB of stored data. There are many usage ranges with different prices. The more data we store the less per GB we pay. The *transfer OUT* is generally considered as data transferred out of the AWS resources through the Internet. Transfers between our AWS resources do not count in the same way. Communication between the resources within the same *Availability Zone* (distinct physical location belonging to certain region) is free of charge. What is also important is that we do not pay for any inbound traffic to our cloud resources. It does not matter whether the *transfer IN* comes from our other resources or from the Internet.

Table 4

EC2 SaaS platform instances cost comparison

	Incremental	Peak-based	**Total**
Base System	10.20 USD	16.49 USD	**26.69 USD**
TBRAM	7.57 USD	12.24 USD	**19.81 USD**
Total	**17.77 USD**	**28.73 USD**	**46.50 USD**

We used the *AWS* part to compare our system with the *EC2* computing service. This included the instances running the SaaS platform, *ELB* and *CloudWatch* monitoring. It also included the *AutoScaling*[4] service, but it was free to use as a tool. One pays just for the outputs of that tool, like increased number of running EC2 instances. Except the server-hours mentioned before, the EC2 cost depends on: instances type (m1.small, m1.medium and ELB on-demand instances in this case) and of course the number of instances. It is worth mentioning, that *Amazon* charges additionally for the amount of data processed by ELB load balancer. The last EC2 service that was included in the cost calculation was running *CloudWatch* in detailed monitoring mode. We were charged only one time per use for both tested systems. Therefore, it was excluded from the cost comparison.
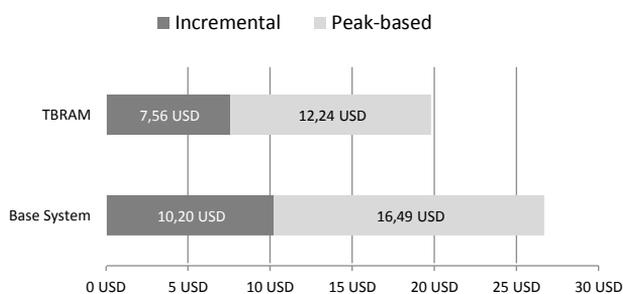

Figure 4

The SaaS platforms costs comparison

Table 4 shows the cost of both systems excluding the load-balancing cost (ELB or SCWA). That means that only costs of running the SaaS platform VMs were included. Fig. 4 we visualize the costs distribution for each test (simulated year = 24hour of real tests). The TBRAM cost is again, lower than the Base System's with 25.8% improvement. This holds even in case of incremental workload test when the TBRAM system did not statistically improve the underutilization.

Table 5

EC2 load balancing cost comparison

	Incremental		Peak-based		**Total**	
	cost	%system cost	cost	%system cost	**cost**	**%system cost**
ELB	3.20 USD	23.88%	5.25 USD	24.14%	**8.45 USD**	**24.04%**
m1.medium	4.25 USD	35.97%	5.61 USD	31.43%	**9.86 USD**	**33.24%**
Total	**7.45 USD**	**29.55%**	**10.86 USD**	**27.43%**	**18.31 USD**	**28.25%**

---

[4] Amazon Auto Scaling: http://aws.amazon.com/autoscaling/

However, the biggest cost reduction is for the peak-based workload tests. It needs to be remembered that the SCWA contained also other parts of the system. Therefore the TBRAM system could not possibly work without that component.

The next step was to compare the separated costs of load-balancing with the results presented in Table 5. The base system used ELB as a load balancer and the authorial TBRAM system used m1.medium instance with the SCWA (that contained load balancer). The column named *%system cost* shows what part of the total system cost constitute the load-balancing part. The ELB made 24.04% of the Base System cost, when the *m1.medium* instance made 33.24% of the TBRAM system cost. The ELB was also 14.2% cheaper in terms of USD price. This was because the cost of *m1.medium* instance per hour is over 6 times more than the ELB. We can see that, even despite additional data processing and transfer cost that the ELB introduces, the Amazon's load-balancing service was cheaper. The main difference was in the ELBs scaling up ability. One could notice the moment when the ELB scaled up during the preliminary tests. This motivated us to the built up the authorial load balancer deployed into more powerful EC2 instance than the standard m1.small. That was clearly an over-provisioning for the time when the system load was low. The lack of enterprise scalability of the SCWA (for small (<10 tenants) scale implementation is equally good as ELB) was the main reason for the higher load-balancing cost of the TBRAM system. It is valuable to notice, that in spite of the fact, that ELB use is cheaper it is not so "resource wise" as the proposed load balancer used in SCWA approach.

Conclusions

This work was inspired and based on the base paper [3]. We wanted to check in practice if the model proposed in the base paper can really influence cost-effectiveness of SaaS systems running in a public cloud. As opposed to just testing the model in the private Eucalyptus cloud. Comparing results from [3] with ours, great similarities were shown. In the base paper [3] the authors achieved 32% server-hours reduction compared to traditional resource scaling. In this work we achieved about 20% and 30% reduction in case of incremental and peak-based tests, respectively. Better result for the peak-based test is caused mainly by the TBRAM underutilization improvement achieved for this type of workload. In the base work the model statistically improved also only the underutilization, but for both types of workload. Thus, we think that this work confirms the TBRAM benefits making it worthwhile, in practice.

Development and deployment of the SaaS systems into AWS cloud made us to draw some other conclusions, too. First, this research showed that TBRAM can improve cost-effectiveness. On the other hand, conformance to that model introduces non-negligible development overhead. This is because we need to write the code for the proposed load balancer, a VM manager (scaling) and a resource monitor. These are not trivial elements to implement and have a great influence on the overall system performance. They are also not easy to test. Because most of

the system's components are independent and distributed, practically the only place they can be fully tested is within the cloud environment, in which, they are implemented. Thus, they make our system more complex and error-prone. Without using the TBRAM one could simply utilize the robust and flexible services delivered by a cloud provider like *Elastic Load Balancer* and *CloudWatch* monitoring for the case of Amazon, which are not so resource efficient, but noticeably cheaper. So, the model introduces additional costs for the system development and deployment. It is up to us to calculate whether the one-time cost will be returned by the possible savings from a decreased, system running costs.

References

[1]   M. Armbrust, et al.: Above the Clouds - A Berkeley View of Cloud Computing. EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2009-28 February 10, 2009

[2]   Jie Guo Chang et al. (2007) A Framework for Native Multi-Tenancy Application Development and Management. 2007 9th IEEE International Conference on e-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-Commerce, and e-Services, 23-26 July 2007 (Piscataway, NJ, USA, 2007) pp. 470-477

[3]   J. Espadas, A. Molina, G. Jimenez, M. Molina, R. Ramirez, and D. Concha: A Tenant-based Resource Allocation Model for Scaling Software-as-a-Service Applications over Cloud Computing Infrastructures, Future Generation Computer Systems Vol. 29, Issue 1, January 2013, pp. 273-286

[4]   C. Hong et al.: An end-to-end Methodology and Toolkit for Fine Granularity SaaS-ization. 2009 IEEE International Conference on Cloud Computing (CLOUD) 21-25 Sept. 2009 (Piscataway, NJ, USA) pp. 101-108

[5]   R. Iyer, et al. VM3: Measuring, Modeling and Managing VM-shared Resources. Computer Networks. 53, 17 (Dec. 2009) pp. 2873-2887

[6]   R. C. Martin, M. Martin: Agile Principles, Patterns, and Practices in C#. 2006, Prentice Hall

[7]   G. V. Mc Evoy, B. Schulze (2008) Using Clouds to Address Grid Limitations. 6th International Workshop on Middleware for Grid Computing, MGC'08, held at the ACM/IFIP/USENIX 9th International Middleware Conference, December 1-5, 2008

[8]   X. Meng, et al. (2010) Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. 7th IEEE/ACM International Conference on Autonomic Computing and Communications, ICAC-2010 and Co-located Workshops, June 7-11, 2010 (Washington, 2010) pp. 11-20

[9]   M. Pathirage, A Multi-Tenant Architecture for Business Process Executions 2011 IEEE International Conference on Web Services (ICWS) pp. 121-128 ISBN 978-1-4577-0842-8

[10]   M. Stillwell, et al. Resource Allocation Algorithms for Virtualized Service Hosting Platforms. Journal of Parallel and Distributed Computing. 70, 9 2010, pp. 962-974

[11]   W. Stolarz, M. Woda: Proposal of Cost-Effective Tenant-based Resource Allocation Model for a SaaS System. New Results in Dependability and Computer Systems: Proceedings of the 8th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, September 9-13, 2013, Brunów, Poland / Wojciech Zamojski [et al.] (eds.) Springer, 2013, pp. 409-420

[12]   W. Stolarz, M. Woda: Performance Aspect of SaaS Application Based on Tenant-based Allocation Model in a Public Cloud. Proceedings of the 9th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, June 30-July 4, 2014, Brunów, Poland / Wojciech Zamojski [et al.] (eds.) Springer, 2014

[13]   J. Yang, et al. (2009) A Profile-based Approach to Just-in-Time Scalability for Cloud Applications. CLOUD 2009 - 2009 IEEE International Conference on Cloud Computing, September 21, 2009

[14]   Q. Zhang, et al. Cloud Computing: State-of-the-art and Research Challenges. Journal of Internet Services and Applications. 1, 1 (2010) pp. 7-18

[15]   B-H Li, et al: Cloud Manufacturing: a New Service-oriented Networked Manufacturing Model. Computer Integrated Manufacturing Systems CIMS 2010; 16(1) pp. 1-7

[16]   S. Li, L. Xu, X. Wang, J. Wang, Integration of Hybrid Wireless Networks in Cloud Services Oriented Enterprise Information Systems, Enterprise Information Systems, Vol. 6, No. 2, 2012, pp. 165-187

[17]   Q. Li, Z. Wang, W. Li, J. Li, C. Wang, R. Du, Applications Integration in a Hybrid Cloud Computing Environment: Modelling and Platform, Enterprise Information Systems, Vol. 7, No. 3, 2013, pp. 237-271

[18]   L. Ren, L. Zhang, F. Tao, X. Zhang, Y. Luo, Y. Zhang, A Methodology towards Virtualization-based High Performance Simulation Platform Supporting Multidisciplinary Design of Complex Products, Enterprise Information Systems, Vol. 6, No. 3, 2012, pp. 267-290

[1]   F Tao, L Zhang, YF Hu. Resource Services Management in Manufacturing Grid System. Wiley, Scrivener Publishing, 2012, ISBN 978-1-118-12231-0

[2]   F. Tao, L. Zhang, V. C. Venkatesh, Y. Luo, Y. Cheng: Cloud Manufacturing- a Computing and Service-oriented Manufacturing Model.

Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture, 225(10), 2011, pp. 1969-1976

[3]   X. Xu: From Cloud Computing to Cloud Manufacturing, Robotics and Computer-Integrated Manufacturing, Volume 28, Issue 1, February 2012, pp. 75-86