Scalable Resource Provisioning in the Cloud Using Business Metrics

Wlodzimierz Funika (1,2)
(1) AGH - University of Science and Technology
Department of Computer Science
al. Mickiewicza 30, 30-059 Kraków, Poland
(2) ACK CYFRONET AGH, ul. Nawojki 11, 30-950 Kraków, Poland
phone: *(+4812) 6174466,* fax: *(+4812) 6339406*
email: funika@agh.edu.pl

Pawel Koperek
AGH - University of Science and Technology
Department of Computer Science
al. Mickiewicza 30, 30-059 Kraków, Poland
phone: *(+4812) 6174466,* fax: *(+4812) 6339406*
email: pkoperek@gmail.com

***Abstract*—Currently cloud infrastructures are in the spotlight of computer science. Through offering on-demand resource provisioning capabilities and high flexibility of management cloud-based systems can seamlessly adjust to the constantly changing environment of the Internet. They can automatically scale according to a chosen policy. Despite the usefulness of the currently available tools in this area there is still much space for improvements. In this paper we introduce a novel approach to automatic infrastructure scaling, based on the observation of business-related metrics. We present details on a tool based on this concept, which uses a semantic-based monitoring and management system, called SAMM. At the end we discuss the capabilities of the new mechanisms.**

***Keywords*-cloud computing; monitoring; scaling; resource provisioning; business metrics**

I. Introduction

While becoming a very promising trend for IT [1], cloud computing platforms offer great flexibility and pricing options which are very interesting especially from the end user point of view. The service consumer pays only for the actually used resources and need not worry about providing/maintaining them. All the required computing power, memory and disk space can be used on demand [2]. Along with easy allocation and de-allocation, many cloud environments offer the ability to automatically add and remove new resources based on the actual usage. These *automatic scaling* capabilities can provide a great value. With such a tool it is possible to seamlessly deal with peak load situations and to reduce the costs of handling a stream of service requests which form predictable patterns.

Usually the rules behind a scaling mechanism are based on the observation of *generic* metrics, e.g. the CPU load. This approach does not require spending additional time to develop customized tools and can be easily applied to many different systems. Nevertheless, it is far from perfect. The decision on what to do with the system is based mainly on low level data which indicate that the system is already undergoing high load or some resources are not being used. One has also to keep in mind that it always takes some time to execute a particular action. Launching a VM instance, connecting it to load balancing facilities and redirecting requests to a new node may take a while. Therefore the action should be executed at such a moment that it will actually improve the situation, instead of generating undesirable overload of the system.

It is common for monitoring tools to provide not only generic, low level information, but also describe the state of resources with metrics tailored to a specific technology or even a particular instance of the system. In many situations such information indicate how much resources will be required in the near future. For example, if the length of the requests queue for computational intensive tasks is rapidly growing, we may be sure that new virtual machines should be deployed. On the other hand, when request buffers on virtual machines are getting empty, the number of running virtual machines may be reduced.

Based on such observations we developed a different approach to automatic scaling. We propose to use higher level data, including customized metrics relevant only to a single system, as decision-making criteria for an auto-scaling mechanism. For example, a resources pool could be extended based on the requests queue length. In this approach we assume that it is far more easier for the user to define triggers for dynamic resource provisioning, when they use concepts directly bound to the application to be scaled.

In this paper we present a modified version of Semantic-based Autonomic Monitoring and Management - *SAMM* ([3], [4]) system, which is using the new paradigm. SAMM is a result of our previous research which was aimed to help administrators meet SLA conditions. To be able to handle this task, the tool monitors a set of customized, high level metrics. They describe the state of a system under observation in the context of business objectives defined in a Service Level Agreement ((SLA)). SAMM was designed to independently modify the application behaviour to prevent it from breaking the contract. Since this level of autonomy is not always desired, we decided to enhance the system with support for custom rules which trigger specified actions.

The rest of paper is organized as follows: Section 2 presents the already existing approaches in the area of

Copyright (c) IARIA, 2011.
ISBN: 978-1-61208-172-4

automatic scaling. Next, in Section 3, we provide more details on the enhancements to SAMM and define (Section 4) an environment which was used to test it. In Section 5 we discuss the results obtained. Finally, Section 6 concludes the paper with a short summary and outlines plans for future work.

II. Related work

Depending on a user's cloud usage model [5], automatic scaling may be understood in different ways. If the user consumes a ready-to-use software application (*Software as a Service model* [6]) the service provider is the party which is responsible for providing a proper amount of resources. There has to be enough computing power and network bandwidth provisioned to meet the conditions of agreements with clients. In this situation, automatic scaling from the end user perspective is only a feature of the system, which allows to easily satisfy the business needs when they arise. For example, a company which uses an on-line office software suite may need to provide such software to ten new employees. Instead of buying expensive licenses and installing the software on their workstations, the IT department requests for additional ten accounts in the online service.

In the *Platform-as-a-Service* model ([7], [8]) the situation is similar. The service provider is also responsible for the automatic scaling of application. However, usually the user has to explicitly request the provisioning of such resources. Providers are able to influence the applications by defining technical constraints for the platform. This way they may ensure that the architecture of the software deployed allows to add and remove some resources dynamically without disrupting normal operation. The algorithm used in decision making may be fine tuned to the underlying hardware and internal resource handling policies of the provider. Usually the user can influence the automatic scaling behavior by setting the upper and lower boundaries of automatic scaling. This prevents from unlimited consumption of resources and therefore from exceeding an assumed budget.

The last model - *Infrastructure-as-a-Service* (e.g. [9]) relies on virtualizing the infrastructure elements like machines and network connections between them. The user has to install and configure everything from scratch and on its top develop their own applications. On the other hand the environment can be customized in many ways, beginning with virtual hardware resources (e.g. CPU power, storage size) and ending with their own server software. Automatic scaling in this model is understood as provisioning on-demand more resources (e.g. virtual machines, virtual storage devices). The user may delegate this task to the service provider ([10]). Adding and removing certain elements is then usually triggered by user-defined rules specifying what action should be taken when a particular threshold is exceeded. These thresholds are limited to a set of metrics predefined by the provider (e.g. CPU usage, storage usage). Many *IaaS* providers also share an API for operations related to managing acquired resources. With such a manner of interaction with infrastructure, the user may create own management tools which can implement custom automatic scaling policies.

In [11] the authors showed that automatic scaling algorithms working with application-specific knowledge can improve the cost-effectiveness ratio of application deployed in cloud environments. Choosing metrics from a set of traditional system usage indicators as CPU usage, disk operation, and bandwidth usage can be not helpful enough. The authors decided that the deadline for jobs executed by system will be used as a key factor for triggering the autoscaling of the system.

These examples show that currently existing automatic scalability mechanisms can be improved. The presented tools focus on maximizing resources usage, which does not have to be the most important factor from the user point of view, e.g. it may be more important to have a system with very short request response time instead of high CPU usage. There are attempts to change this situation, but there is no generic tool which would be oriented towards easy adaptation to specific systems.

III. Modifications to the SAMM system

Our approach to automatic scaling is based on the assumption that for each application it is possible to choose a set of metrics, which can be used to estimate how much resources will be required in the nearest future. On the most basic level it should be sufficient to be able to predict whether more resources will be required, or some of those currently running may be stopped. With this point of view, the user is able to determine simple thresholds related to triggering some actions influencing the system in a desired way.

The set of metrics under discussion is tightly coupled with an application for which it is to be chosen. Since the metrics with the same names are often used in different contexts, they may have completely different semantics, e.g. the memory usage may be defined as physical RAM usage or virtual memory usage. To handle so much different information, it is required to use a data representation capable of describing all the used concepts. Furthermore, the monitoring system used in this situation has to provide easy ways to create extensions. There should be a possibility to provide support for new data acquisition techniques. In modern complex software systems, measurements will have to be gathered with help of several very different technologies.

Therefore we decided as a starting point for this research to use the result of our previous work on automatic scaling - the SAMM ([3], [4], [12]) system. SAMM is a monitoring and management software, flexible enough to meet the above requirements. By using ontologies to describe resources and metrics available for observation, it has capabilities to express different system architectures and monitoring

facilities. Owing to its module-based architecture based on OSGi bundles and services, it can be extended by support for new technologies without much effort. Reimplementing and replacing the existing components is also feasible.

To meet the requirement of being able to define rules in a convenient way, we came to a new decision-making module. For this purpose we used the Esper event processing engine ([13]). The internal architecture of SAMM with the enhancements introduced is depicted in Figure 1.

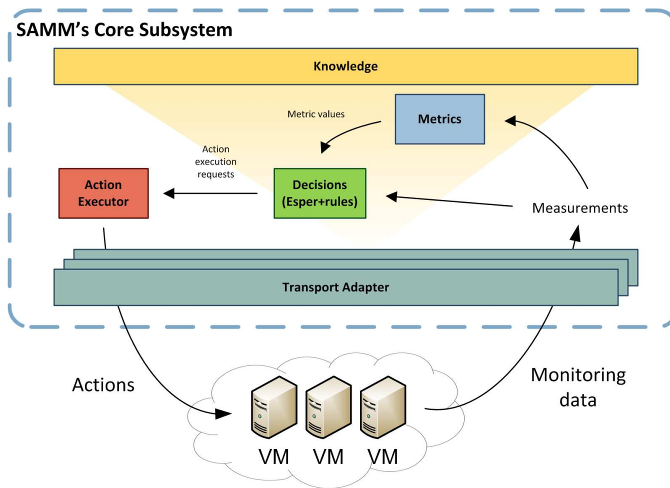

Figure 1. SAMM's architecture after introducing described changes

The flow of measurement activities is as follows:

1) Measurements are being collected by the *Transport Adapters*. *Transport Adapters* are a an abstraction layer over different technologies used for data collection, e.g. there are separate adapters for Java Management Extensions (JMX)([14]) and Eucalyptus ([15]). They translate internal SAMM measurement requests to language specific constructs for a particular technology, e.g. to Java method invocations.
2) *Metrics* module processes measurements according to formulas defined in form of Java classes. The values obtained in this way are sent further to the *Decisions* module.
3) Values coming from *Transport Adapters* and *Metrics* modules are processed directly by the *Decisions* module. The Esper event processing engine captures specific events and based on them can trigger an action execution. Rules that describe which events should trigger which actions are defined by the user. It is optional to trigger an action.
4) Whenever the *Decisions* module detects exceeding a threshold, an action execution request is sent to *Action Executor*. This component tries to modify the infrastructure by using a specific *Transport Adapter*, since actions can be executed only with help of particular communication protocols. The exact steps executed by an action are provided as a Java code.

Once SAMM has been enhanced, measurement and metrics values are processed by Esper as events. Owing to this, conditions specified in rules may be very flexible and may include e.g. aggregation functions or consider values only from within a particular interval of time. Since custom queries can be used, the only constraint is the flexibility of the Esper query language.

IV. Evaluation of the Approach

To evaluate our approach to automatic resources provisioning we applied the business-metric based scaling policy to a sample application - a simple service for numerical integration. To be able to easily scale the number of nodes used for computations, the *Master - Slave* model was applied. In the application's architecture (presented in Fig. 2) there is a main *master* node, responsible for dispatching the requests and one or more *slave* nodes performing numerical integration. We do not discuss the exact algorithm of numeric integration or its parameters since it is out of scope of interest of the paper.

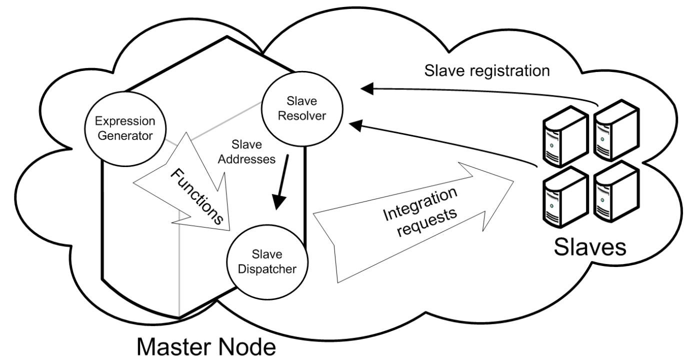

Figure 2. Test application architecture

The *Master* node is built from three components:

- **Slave Dispatcher** - handles the queue of incoming numerical integration requests. If there are any requests in the queue, **Slave Dispatcher** passes them to one of the registered slaves. The *slave* is chosen by performing the following steps:
  1) Retrieve a proxy for another *slave* from **Slave Resolver**
  2) If the *slave* is capable of handling a next request, send it to this node, else go to point 1.
  3) If there are no slaves capable of handling the request - wait e.g. for 5 seconds and start looking for an appropriate node from the beginning.

  A slave node is capable of handling a next function if it does not overflow the buffer of numerical integration requests. The size of buffer was set to 25 requests.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-172-4

The **Slave Dispatcher** is the main component of *Master* node.
- **Slave Resolver** - maintains the set of proxies to *slave* nodes. In this component *slave* nodes register at the beginning of their work. The information about the addresses of nodes are later shared with **Slave Dispatcher**.
- **Expression Generator** - generates the functions for which integration operations are further performed on *slave* nodes. The functions may be generated infinitely or read from a file in chunks of a specified size.

One of the main assumptions about the test application is that always at least one *slave* node has to be running. It was also decided that at most ten instances could get started automatically. The stream of integration requests in the test scenario was tailored to these parameters. The system running all ten *slave* nodes at once could handle without problems the load used in the test scenario.

A. Test Environment

The development work and tests were carried out using the FutureGrid project environment [16]. The **India** Eucalyptus ([15]) cluster was used. The following virtual machine types were provided:

- `m1.small` - 1 CPU, 512 MB of RAM, 5 GB of storage space
- `c1.medium` - 1 CPU, 1024 MB of RAM, 5 GB of storage space
- `m1.large` - 2 CPUs, 6000 MB of RAM, 10 GB of storage space
- `m1.xlarge` - 2 CPUs, 12000 MB of RAM, 10 GB of storage space
- `c1.xlarge` - 8 CPUs, 20000 MB of RAM, 10 GB of storage space

The cluster is built up from 50 nodes and each node is able to run up to 8 `m1.small` instances. *Slave* nodes in our application do not use much storage space and memory. To have got a fine-grained level of the management of the computing power, we decided to use `m1.small` instances for them. The *Master* node application had higher memory requirements, thus we deployed it on a `c1.medium` instance.

B. Case Study

To evaluate the quality of our approach we compared two strategies of automatic scaling. The first one exploits a generic metric - the CPU usage. The second strategy uses a business metric - average time spent by computation requests while waiting in *Slave Dispatcher's* queue for processing. Upper or lower limits for such a metric could be explicitly included in a *Service Level Agreement*, e.g. the service provider might be obligated to ensure that a request won't wait for processing for longer than one minute.

The triggering rules used in the first approach are as follows:

- Start another *slave* node virtual machine: the average CPU usage of *slave* nodes from the last 300 seconds was higher than 90%
- Stop a random *slave* node virtual machine: the average CPU usage of *slave* nodes from the last 300 seconds was less than 50%

The triggering rules used in the second approach are:

- Start another *slave* node virtual machine: the average wait time of request from within the last 300 seconds was higher than 35 seconds
- Stop a random *slave* node virtual machine: the average wait time of request from within the last 300 seconds was less than 10 seconds

The presented parameters were tuned up specifically to the infrastructure on which we carried out the tests and the current load of the cluster. The infrastructure was used in parallel with other users, thus, e.g., the startup time of virtual machines differed over time.

For the evaluation we prepared a test scenario consisting of 120 steps. Each step included adding new requests of numerical integration to *Slave Dispatcher's* queue and waiting for 60 seconds. We had to ensure that two auto-scaling strategies will have to handle exactly the same situation. It was therefore decided to generate the exact functions to process before the actual test and store them in files. Later, when the test was run, *Expression Generator* was reading functions from these files and passing further for processing.

The number of requests added to the queue was equal to

$$ExprNum(n) = 100 + 5 * (n \ mod \ 80)$$

(where `n` is the iteration number) of each step of workload. The exact values of formula's constants were chosen in such a way, that a setup of ten virtual machines running constantly during the test scenario could handle the workload. The workload simulates a basic situation requiring automatic scalability capabilities: a constant increase in load in a period of time with a rapid drop right after it.

V. Test results

To compare the automatic scaling exploiting the selected strategies, we gathered data about the average wait time (Figure 3), *Slave Dispatcher's* input queue length (Figure 4) and the number of running instances (Figure 5) when executing the test scenario. To sum up the differences we computed the average values of those metrics. Table I presents the results.

At the beginning (the first half of an hour), according to the growing demand for computing power, SAMM started some slave nodes. They were kept busy, because the number of integration requests was growing.

While observing the system from the CPU usage point of view, more resources should still be allocated - there was

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-172-4

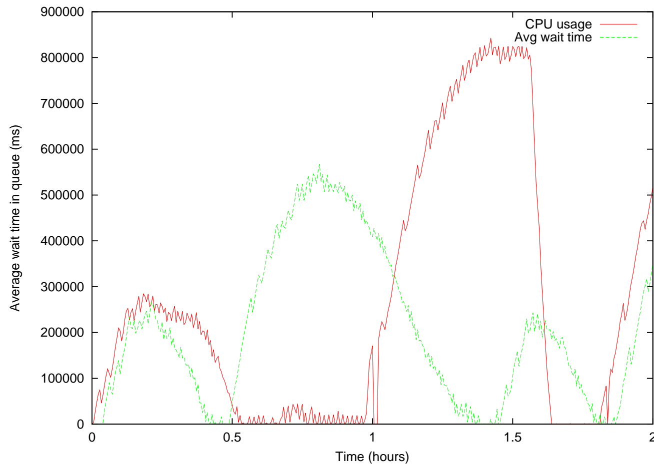

Figure 3. Average wait time during test scenario execution for both strategies (*CPU usage* and *Avg wait time*)

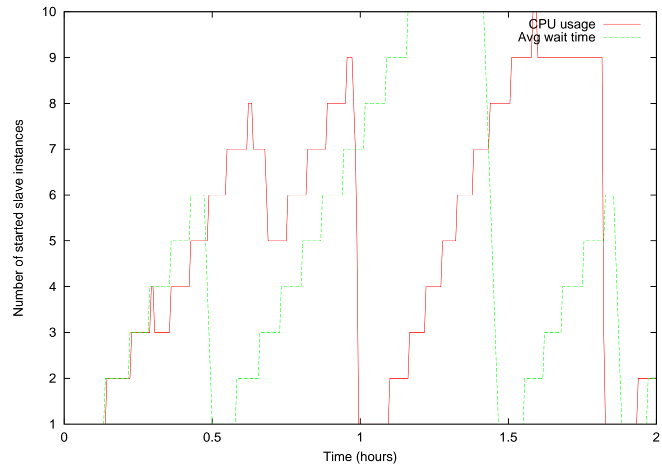

Figure 5. Number of running instances during test scenario execution for both strategies (*CPU usage* and *Avg wait time*)

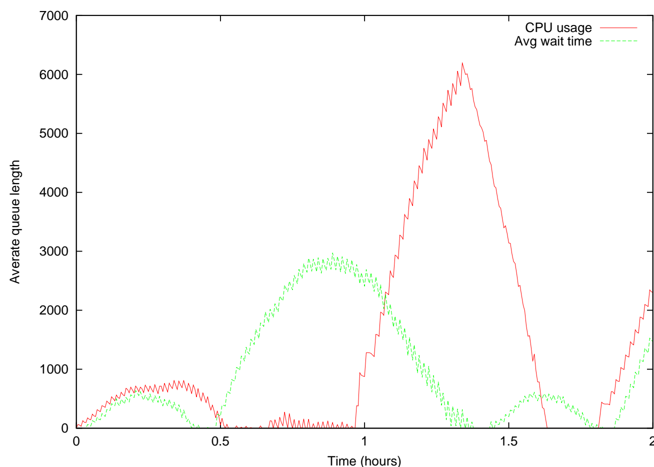

Figure 4. Queue length during test scenario execution for both strategies (*CPU usage* and *Avg wait time*)

not enough computing power to process incoming requests. The number of *slaves* was increased until the requests were handled faster than the new ones were added. Once the queue got completely empty, CPU usage dropped and SAMM terminated the unused virtual machines. However, some minutes later, the queue got refilled. Again, more resources were acquired. At the end, after eighty minutes (when the workload rapidly drops), the number of requests in the queue was still high (over 6000).

In the second approach, the rapid drop of average wait time at the beginning also indicated that some resources should get released. The virtual machines were fully used, but the wait time for requests was acceptable. When more and more requests kept joining the queue, the average time required to start the numerical integration got extended. Even if the machines were not utilized 100%, the quality of the service was dropping according to the *average wait time* metric. SAMM therefore acquired more resources to improve the situation. In a longer perspective such a behavior resulted in a shorter response time in comparison with the CPU usage-based scaling strategy.

Table I
AVERAGE METRIC VALUE FOR AVGWAITTIME AND CPU STRATEGIES

Metric	AvgWaitTime	CPU
Average instances number	4.53	4.96
Average wait time (ms)	203987.08	266362.25
Average queue length	934.06	1392.72

VI. CONCLUSIONS AND FUTURE WORK

Using the average wait time metric has a positive impact on the system when considering its operation from the business point of view. Since the end users are mostly interested in making the time required to wait as short as possible, the amount of the resources involved should be increased according to this demand. By improving this factor, the potential business value of the presented service grows. The system was automatically scaled by SAMM not only from the technical point of view but also from the business value perspective. On the other hand, since the CPU usage was not the main concern, the system may be used not in the most efficient way. If there would be an SLA which does not cover the request wait time, it would be more reasonable to use the CPU usage as a trigger for automatic scaling. Choosing the most appropriate metric to use is up to the user - they have to decide which one is the most important from their perspective.

The actual improvement introduced by the dynamic resource provisioning highly depends on an actual workload. In our test scenario the system shortened the time spent on

Copyright (c) IARIA, 2011.

waiting by 62 seconds. However there are also situations in which the strategies bound to generic metrics can be as good as the one based on business metrics, but it would require more efforts to choose the most relevant ones and to tune correct limit values.

The possibility to dynamically increase the amount of resources involved in providing a service is a response for the need to efficiently operate in a very quickly changing environment such as the Internet. Adding resources on-demand significantly extends the capabilities of a running system, which is enabled to serve both small and big numbers of users. There is no unused capacity, so the operational costs can be lower.

Making the automatic scaling tools more aware of business rules makes them more useful, especially when they are used in privately held cloud systems. Deep knowledge about the elements running inside the system, provides better insight into scaling rules. They can be fine-tuned to particular hardware and software setup, so the balance between the allocated and spare resources can be properly maintained. For the crucial applications the system could provide all the required computing power and low priority tasks can be automatically maintained. The service provider is able to do more with the same or even smaller amount of computing power.

Our research in automatic scaling area is ongoing. We plan to further extend SAMM with a web interface which would facilitate using its functionality. Another interesting goal of the research is to add support for other cloud stacks, e.g. Open Stack [17] or Open Nebula [18], thus making SAMM interoperable in a heterogeneous environment.

Acknowledgments: This research is partly supported by the AGH University grant. The experiments were carried out using the FutureGrid project resources [1].

References

[1] Buyya, R. et al. *Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility*. FGCS, vol. 25(6), 2009, 599-616

[2] Zhang, Q., Cheng, L., Boutaba, R. *Cloud computing: state-of-the-art and research challenges*. Journal of Internet Services and Applications 1, 7-18 (2010).

[3] Funika, W., Kupisz, M., Koperek, P.: *Towards autonomic semantic-based management of distributed applications*, Computer Science Annual of AGH-UST, vol. 11, 2010, pp. 51–63, AGH Press, Krakow, 2010

[4] SAMM Project website, http://code.google.com/p-/sammanager (accessed: 13.05.2011)

[5] Rimal, B. P., Choi, E., Lumb, I. *A Taxonomy and Survey of Cloud Computing Systems*, pp.44–51, 2009 Fifth International Joint Conference on INC, IMS and IDC, 2009

[6] Google Inc, Google Apps website, http://www.google.com/apps (accessed: 26.04.2011)

[7] Google Inc, App Engine project website, http://code.google.com/appengine (accessed: 26.04.2011)

[8] Amazon Web Services LLC, http://aws.amazon.com/elasticbeanstalk (access: 09.05.2011)

[9] Amazon Web Services LLC, http://aws.amazon.com/ec2 (accessed: 26.04.2011)

[10] Amazon Web Services LLC, http://aws.amazon.com/autoscaling (accessed: 13.05.2011)

[11] Mao M., Li J., Humphrey M., *Cloud Auto-scaling with Deadline and Budget Constraints*, 11th IEEE/ACM International Conference on Grid Computing (Grid 2010)

[12] Funika, W. et al., *A Semantic-Oriented Platform for Performance Monitoring of Distributed Java Applications*, in: M. Bubak, G. D. van Albada and J. Dongarra and P. M.A. Sloot (Eds.), Proc. ICCS 2008, volume III, LNCS 5103, Springer, 2008, pp. 233-242.

[13] Esper Project website, http://esper.codehaus.org (access: 10.05.2011)

[14] Oracle Corporation, website, http://www.oracle.com/technetwork/java/ja-vase/tech/javamanagement-140525.html (accessed: 14.05.2011)

[15] Eucalyptus Systems, Inc. website, http://www.eucalyptus.com (accessed: 13.05.2011)

[16] FutureGrid Project website, https://portal.futuregrid.org (accessed: 13.05.2011)

[17] Open Stack project website, http://www.openstack.org (accessed: 13.05.2011)

[18] Open Nebula project website, http://opennebula.org (accessed: 13.05.2011)

[1] https://portal.futuregrid.org