Computer Science • 13 (3) 2012 http://dx.doi.org/10.7494/csci.2012.13.3.45

Jarosław Dąbrowski
Sebastian Feduniak
Bartosz Baliś
Tomasz Bartyński
Włodzimierz Funika

AUTOMATIC PROXY GENERATION AND LOAD-BALANCING-BASED DYNAMIC CHOICE OF SERVICES

Abstract *The paper addresses the issues of invoking services from within workflows which are becoming an increasingly popular paradigm of distributed programming. The main idea of our research is to develop a facility which enables load balancing between the available services and their instances. The system consists of three main modules: a proxy generator for a specific service according to its interface type, a proxy that redirects requests to a concrete instance of the service and load-balancer (LB) to choose the least loaded virtual machine (VM) which hosts a single service instance.*

The proxy generator was implemented as a bean (in compliance to EJB standard) which generates proxy according to the WSDL service interface description using XSLT engine and then deploys it on a GlassFish application server using GlassFish API, the proxy is a BPEL module and load-balancer is a stateful Web Service.

Keywords proxy, load-balancer, load-balancing, automatic generation

1. Introduction

Workflow paradigm is becoming now a more and more popular model for solving business and scientific problems. To provide a complex functionality, there is a need to use different functional modules which can be organized into services. Such modules can have many instances working in a distributed environment, which implies that to achieve the best performance, load balancing needs to be used. This requires the capability of locating an indispensable service, choosing the best instance of the service and invoking it to have some function carried out.

Moreover adding a further module to ensure a new behaviour should be easy. It would also be preferable to separate the design and development of such modules, which means that someone who wants to add a new functionality does not have to know how to place it into a working environment. Also extending the working system with existing modules should be easy and need no implementation changes. The system under discussion is aimed to solve the issues described above.

The paper which addresses the above issues faced by the EU UrbanFlood project [1] is organized as follows. Related work is followed by the general system architecture and workflow management. Section 4 explains the concept and architecture of the proxy generator we exploit. Then the proxy BPEL module is discussed in Section 5. In Section 6, we present the architecture of a load-balancer and then details about the implemented load-balancing strategies. The tests performed on the proxy are presented in Section 7 and followed by conclusions and ideas for the future work.

2. Related work

The issues of building and executing workflows is addressed in a vast number of papers [7], as well as load balancing in parallel and distributed programming [8].

The concept of load balancing with agent systems is addressed in a number of papers. Among them, [5] proposes that a multi-agent system may be used by the operating system to ensure that execution of all tasks on processors could be completed in the shortest possible time. In the realized multi-agent system each processor has an assigned agent which can communicate with agents on neighbouring nodes to exchange tasks. An exchange is performed whenever the agent has an excess as well as a lack of task. However, it is possible that neighbours are in the same situation, i.e., they have a lack or an excess of tasks, then the resident agent creates another type of agent which is aimed to find some tasks for a processor that "suffers" from a lack of tasks.

The author of the HAProxy balancer describes in [11] various strategies of load balancing for applications accessible over network. Since the power of any server is finite, a web application must be able to run on multiple servers to accept an ever increasing number of users. The most popular one but also very effective algorithm is the DNS round robin [4]. The idea is that, if a DNS server has several entries for a given hostname, it will return all of them in a rotating order. Another strategy is to

send a request to "the first server to respond". It is also possible to choose always the least loaded server. The latter concept was enhanced in [10] where authors present a way how to monitor Web Services load and find out how big impact a single request has on the service load. The load balancer can adapt its behavior to a current service state, which means that, e.g., CPU or memory will have a greater impact on load indicator. The authors have decided to use Pearson's Correlation Coefficient to find a relationship between request duration and CPU load.

All of the above approaches were considered when developing our load balancer. Some, like Pearson's Correlation Coefficient were introduced and tested in a real environment.

3. General architecture and workflow management

The big picture of our research can be illustrated with the system architecture of the UrbanFlood project as shown in Fig. 1.

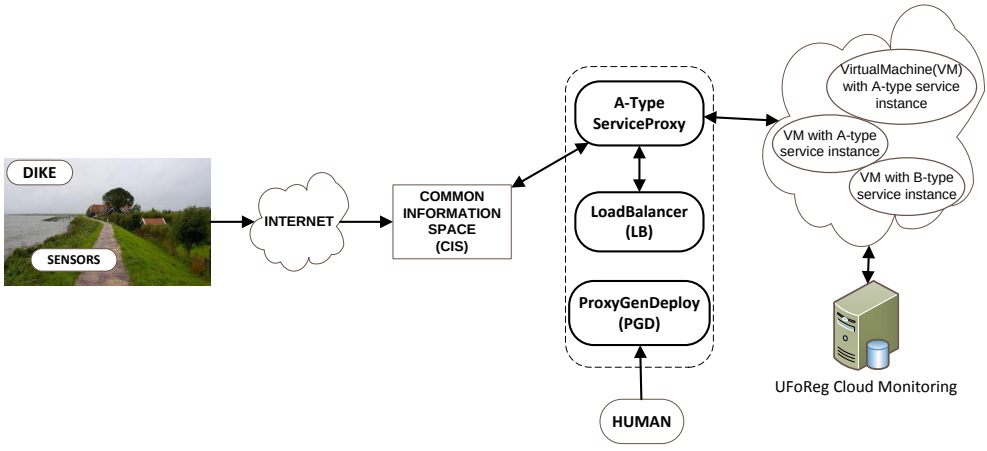

Figure 1. General idea of the system architecture of UrbanFlood.

UrbanFlood is an Early Warning System (EWS) which can play a crucial role in mitigating flood risk by detecting conditions and predicting the onset of a catastrophe before the event occurs, and by providing real time information during an event. The components of EWS are organized into a workflow. The term workflow denotes a graph of tasks connected by control and/or data-flow dependencies. In our case, the source of the data are sensors located in a dike. The only one requirement to add a new sensor is connect it to the Internet which is used to send the collected data. Although a package from a sensor is received by the Common Information Space (CIS) – a platform for hosting EWS's, it is not responsible for computing the data. CIS can send the request to a well known endpoint – proxy which redirects the request for execution on a concrete service instance.

To increase its performance a service should have many instances but from the CIS point of view there is only one component responsible for request execution. Please note that a proxy is created for one type of service (more technically: for one interface) like a service is created for a specified purpose, e.g., one service is responsible for computing flood risk and another one for the visualization of water level. Proxy has two activities: first it should contact LoadBalancer (LB) to get a concrete service instance address and then it should redirect the request received from the CIS to this instance. A service can be the last node of the workflow, but it is not necessary. A cloud used to host services (one service per Virtual Machine (VM)) is monitored by the UFoReg system. LB queries UFoReg to get VM-related measurements. The data collected can be used to select the least loaded VM which hosts a specified service instance.

4. Automatic proxy generation and deployment

At the beginning of our research we used the GlassFish ESB platform to generate proxy for services. Preparing a package ready for deployment on the application server is a complex process which requires a sequence of steps described coarse-grained below.

1. Create an empty BPEL module and copy to it files needed in each proxy module e.g. an LB WSDL file.
2. Copy a service WSDL and XSD files.
3. Create a BPEL [3] process sequence which generally consists of: receiving input data and assigning it to local variables, contacting LB to get service instance address, changing service endpoint address in the partner link, invoking the request and assigning its result to local variables, sending execution time to the LB and returning the result.
4. Create a Composite Application to deploy the proxy on the GlassFish server.
5. Deploy the prepared package.

All these steps are the same for any type of service so we aim to automate this process. This automation was realized as an EJB service called ProxyGenDeploy (PGD).

To easily understand the whole process, the PGD architecture (please see Fig. 2) can be helpful.

It presents the whole flow responsible for generating and deploying proxy. PGD becomes a four arguments on input: service type, WSDL URL, input XSD URL, and output XSD URL. The steps that follow are: to download files from given URLs, then WSDL and XSD files are transformed (using the XSLT engine) to a BPEL module. After successful generation a JAR containing the BPEL module is created, which is next used to create a ready-to-deploy composite application packed into a ZIP archive. Nowadays PGD uses a dedicated GlassFish server to deploy proxies. The old proxy is stopped and undeployed (if exists) and the new proxy is deployed and started to

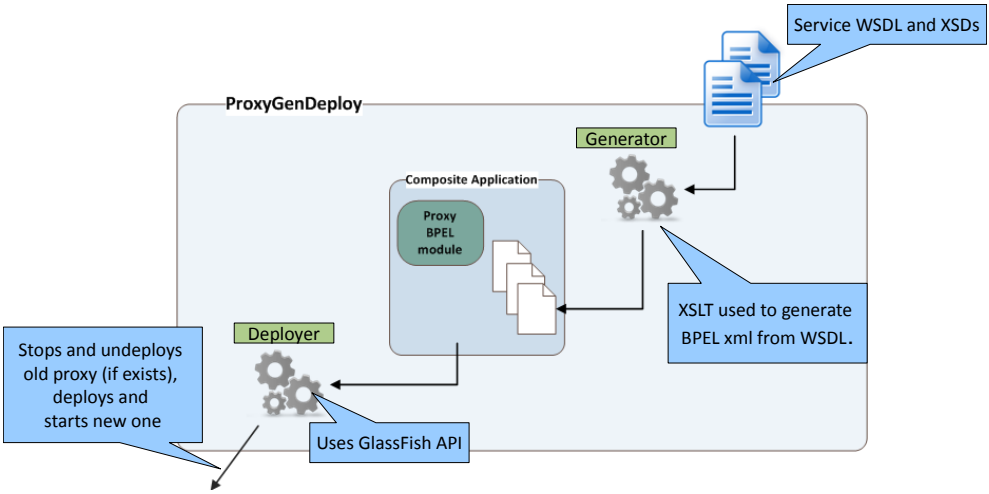

Figure 2. Architecture of ProxyGenDeploy.

do this, GlassFish API [2] was used. After these steps, the newly generated proxy is ready to use and is listening to incoming requests. No more steps are required.

5. Proxy

Proxy provides a single well known endpoint, which acts to the underlying set of (dynamically provisioned) services, hidden below it. Note that a proxy is not created manually. It is generated by PGD and it is automatically deployed on the GlassFish server. It is required to have sun-bpel-engine installed on the target GlassFish server.

Fig. 3 depicts a sequence diagram of request execution, which involves the following steps:

1. Receive a request from the CIS [1] of environmental phenomena through wireless sensors.

1.1 Query LB for a service url address.

1.1.1 LB chooses one of the registered services based on a specified strategy.

1.1.2 Proxy receives service's address.

1.2 Proxy uses Partner Link for contacting services and here a service endpoint is dynamically assigned to it.

1.3 Proxy gets a current time just before passing the request to the service.

1.4 Send request to the service.

1.5 Proxy receives operation results.

[1]The CIS (Common Information Space), part of the Urban Flood project, is a generic framework for hosting Early Warning Systems based on the monitoring

1.6 Proxy gets the current time just after receiving a response from the service.
1.7 Sending time statistics to the LB.
1.8 Return the operation results.

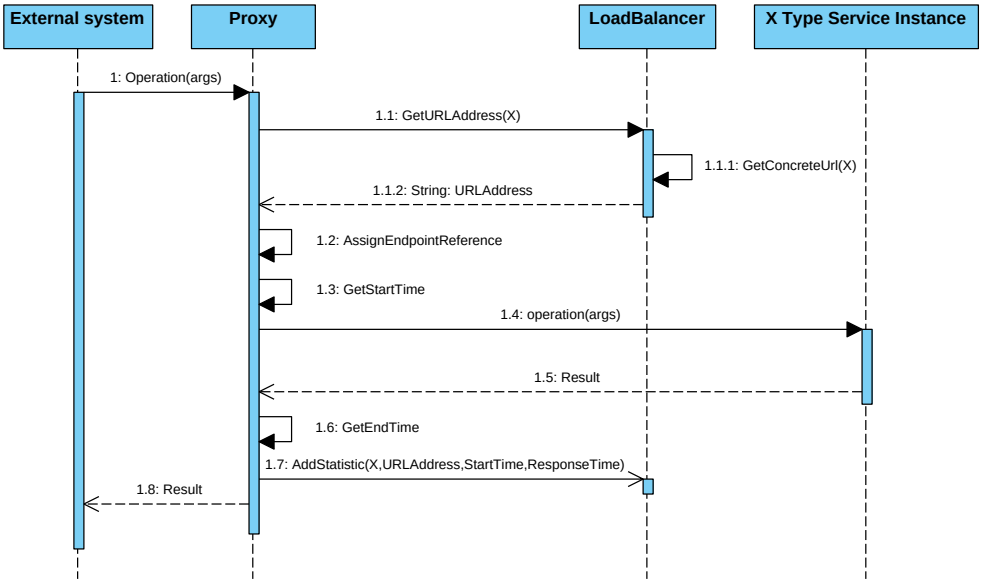

Figure 3. Proxy instrumentation.

Each unexpected failure (e.g. LB is not available, a service of a given type doesn't exist, etc.) will create an appropriate SOAP error message generated by the Proxy BPEL module.

6. Load balancer

Fig. 4 presents the general architecture of LoadBalancer. The functionalities of the LB are listed and described following the figure.

1. *Register/Unregister service instance.* It is used to inform LB about new services. While adding a new instance, required are: service type name, VM id (unique for each one) and service address.
2. *Add statistics.* Used to collect data about execution times for each service. Such measurements are sent by the Proxy.
3. *Get service address.* Returns a concrete instance address using load balancing strategies. In order to choose the least loaded VM, LB uses a filtering mechanism. The Strategy pattern together with the Composite pattern were combined to ensure the flexibility and exchangeability of the load balancing algorithm. When "get service address" is invoked, the whole list of VMs of a given type goes on the

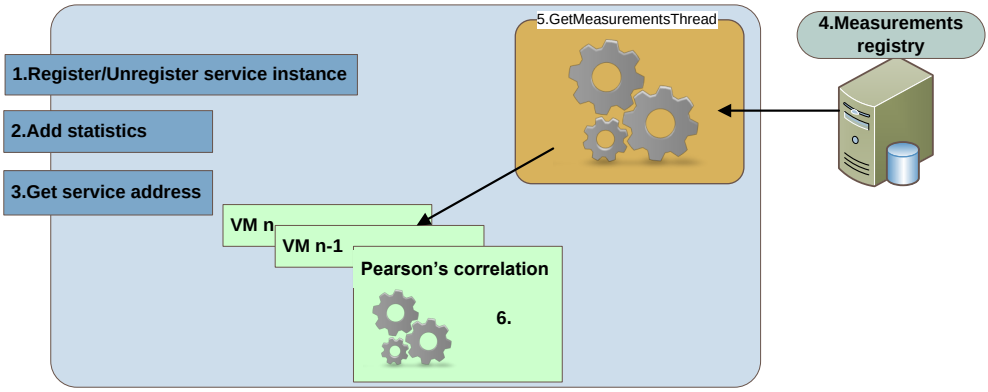

Figure 4. Architecture of LB.

input of the chain of filters. After which step of filtering, the list of VMs shrinks and as a result only one VM is chosen, finally the address of this VM is returned. To facilitate understanding the above, an example process is illustrated in Fig. 5.

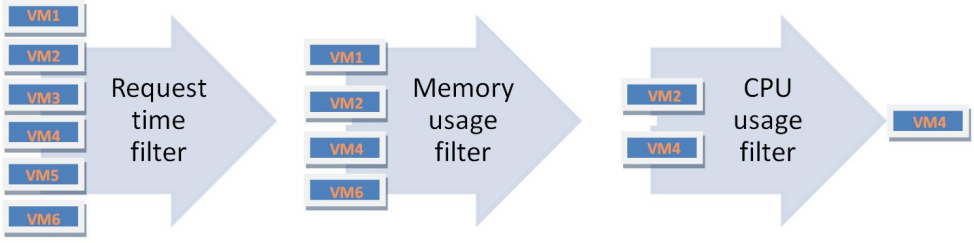

Figure 5. Chain of filters.

We have implemented three filters. The first one filters machines according to the count of running requests. The second one filters machines according to the average request time execution. The third one filters VMs according to their load indicator, which is a weighted average of CPU and memory usage. LB holds a reference to a single filter. Using Composite Filter other filters can be aggregated into the chain. Adding a new filter is very simple, it requires implementing a Filter interface. In the future it would be possible to create an XML configuration where a list of the used filters and their order could be stored.

4. *UFoReg measurements.* UFoReg(UrbanFlood Registry) is a database which collects VM-related measurements and publishes it using HTTP GET method.
5. *GetMeasurements Thread.* Because collecting VM-related measurements per request can be too expensive, we introduced a TimerTask which is called periodically (being configurable). It holds a list of registered VMs, through which it iterates during every execution to collect VM-related measurements. For each

VM it connects to UFoReg and collects measurements generated since the last execution up to now. The information gathered is then used to compute an average CPU usage, average memory usage, CPU usage weight and memory usage weight, the last two values are computed using Pearson's correlation.

6. *Pearson's correlation.* As mentioned, one of our filters used by LB assumes that the load indicator is calculated for each VM. We calculate the load indicator (li) as follows:

$$li = CPU_{weight} * CPU_{load} + MEMORY_{weight} * MEMORY_{load} \tag{1}$$

The problem is how to determine the appropriate weights. We decided to find out how big impact each request has on CPU and memory load. Having collected the execution time for each request and corresponding CPU and memory average load, we can calculate Pearson's correlation. The correlation coefficient range is from -1 to 1. A value of 1 implies that a linear equation describes the relationship between X (request execution time) and Y (CPU or memory load) perfectly, with all data points lying on a line for which Y increases as X increases. A value of -1 implies that all data points lie on a line for which Y decreases as X increases. A value of 0 implies that there is no linear correlation between the variables.
Having two vectors, where X are request execution times and Y are average cpu/memory loads:

$$X = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_n \end{bmatrix} \; Y = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_n \end{bmatrix} \tag{2}$$

we compute Pearson's correlation coefficient (r) as follows:

$$r = \frac{\sum_{i=1}^{n} (X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n} \left(X_i - \overline{X}\right)^2} \sqrt{\sum_{i=1}^{n} \left(Y_i - \overline{Y}\right)^2}} \tag{3}$$

After that, the weights for CPU and memory are computed as follows:

$$\begin{aligned} CPU_{weight} &= \frac{r_{cpu}}{r_{cpu} + r_{memory}}, \\ MEMORY_{weight} &= \frac{r_{memory}}{r_{cpu} + r_{memory}} \end{aligned} \tag{4}$$

Please note that owing to Pearson's correlation, load balancing will be better suited to the running system because weights are computed periodically.

7. Tests

The LB mechanism was tested with one type of service. It has only one method which takes one argument (*number of iterations*) and performs many operations on floating-point numbers in each iteration. We had five instances of such services deployed at the ACK CYFRONET AGH computer.

Physical machine parameters:
Server name: HP ProLiant BL2×220c G5
CPU: 2×Intel Xeon L5420 (4 cores, 12M L2 Cache, 2.5 GHz, FSB 1333 MHz)
Memory: 16 GB (4×4096 MB 667 MHz)
Virtual machine parameters (one machine per service, each with the same configuration):
1 core
Memory: 1 GB

LB was configured as follows: the first filter was filtering the machines according to the count of requests running on it and it was leaving three machines on the list, the second one was filtering the machines according to the least response times and it was leaving two machines on the list, third one was filtering the machines according to a load indicator and it was leaving only one machine on the list. During the tests ten threads were sending requests to the proxy over five minutes.

The first tests were performed with the number of iterations equal to 5000. In such a situation the average response time on the unloaded machine is about 37 seconds (Fig. 6).

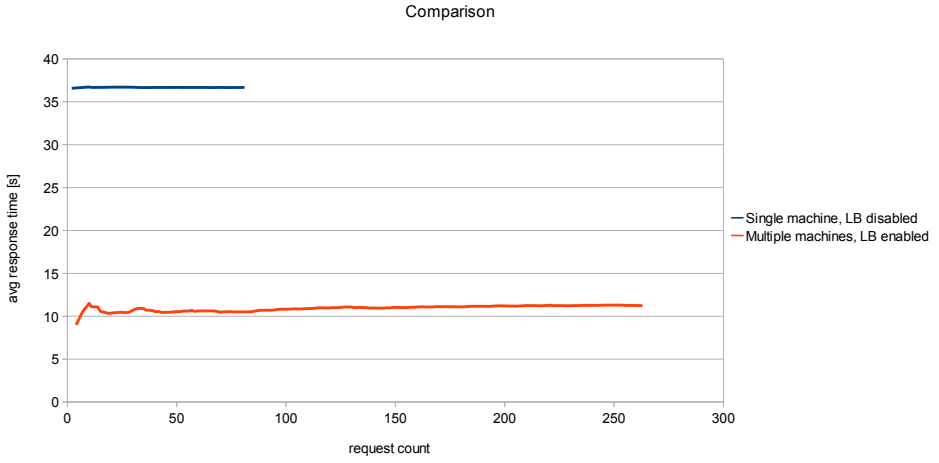

Figure 6. Average request duration with LB and without LB for number of iterations equal to 5000.

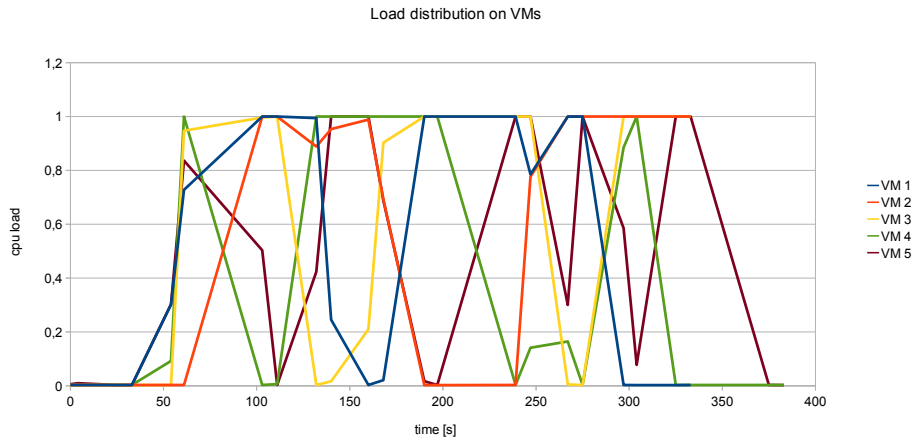

Figure 7. Load of VMs during tests for number of iterations equal to 5000.

Fig. 7 presents how VMs were loaded during tests.

The second series of tests was performed with the number of iterations equal to 500. In this situation the average response time on the unloaded machine is about 3.7 s (Fig. 8).

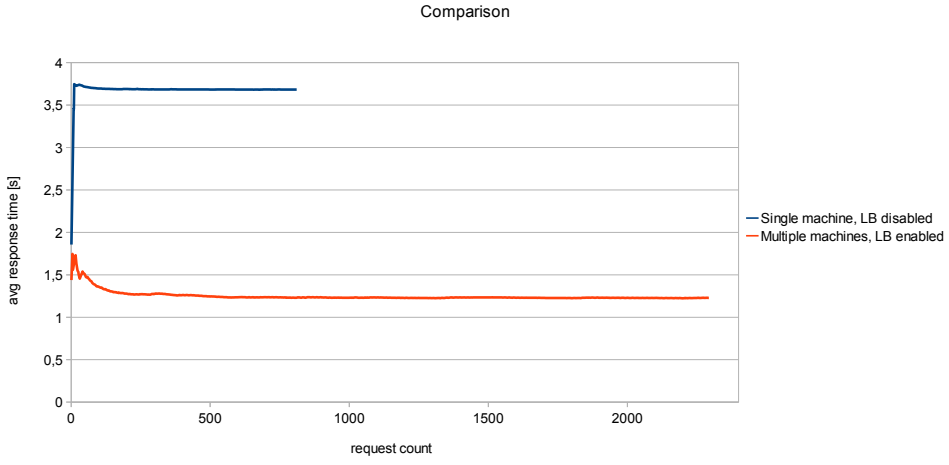

Figure 8. Average request duration with LB and without LB for number of iterations equal to 500.

Fig. 9 presents how VMs were loaded during tests.

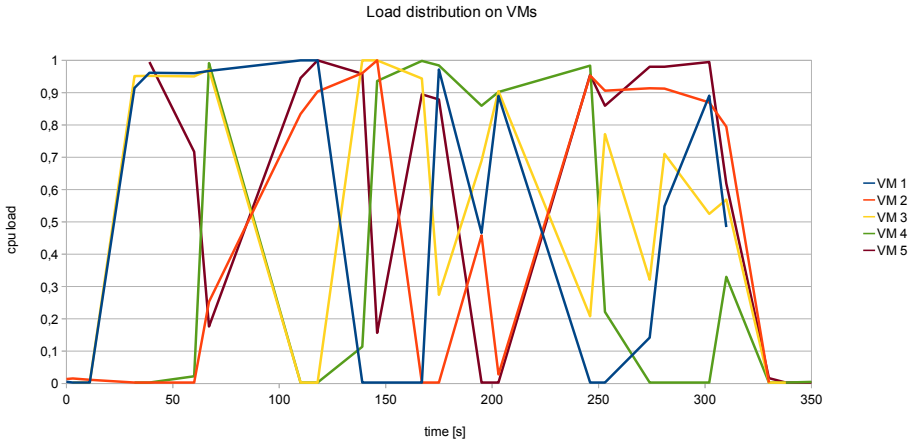

Figure 9. Load of VMs during tests for number of iterations equal to 500.

The third series of tests was performed with the number of iterations equal to 50. In this situation the average response time on the unloaded machine is about 500 ms (Fig. 10).

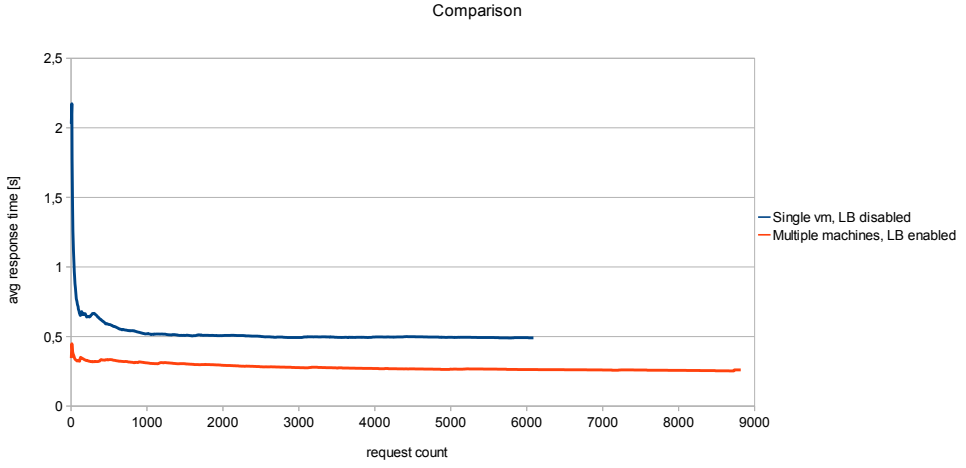

Figure 10. Average request duration with LB and without LB for number of iterations equal to 50.

Fig. 11 presents how VMs were loaded during tests.

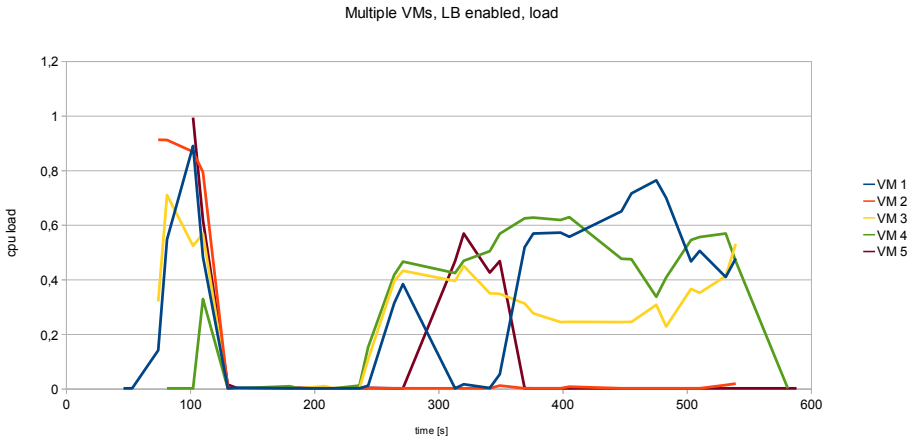

Figure 11. Load of VMs during tests for number of iterations equal to 50.

As we can see, in each test better performance was achieved while using the implemented system. Figures 6, 8 and 10 present decrease of average service response time while using load balancer. As it was mentioned, each test was stopped after 5 minutes. During this time, more requests were processed using load balancer so line for the single VM is shorter. Values are lower for the lower number of requests because the machines were less loaded.

Figures 7,9 and 11 present CPU load of each machine. The requests were distributed between machines which means that load balancer was working good.

In Tables 1, 2 the experimental results are compared.

Table 1

Experimental results with load balancer

number of iterations	requests count processed in 5 minutes without load balancer	requests count processed in 5 minutes with load balancer	Increase in performance %
5000	81	263	224,7 %
500	811	2293	182,7 %
50	6089	8818	44,8 %

We have tested how much faster the requests would be executed when using five VMs with LB vs. the case without LB (every request was delivered to the same instance of the service). It would be ideal if having 5 services we could execute 400%

Table 2
Experimental results without load balancer

number of iterations	average request duration without load balancer [ms] (a)	average request duration with load balancer [ms] (b)	Increase in performance (a/b)
5000	36675	11233	3,26
500	3682	1228	2,99
50	491	260	1,89

more requests over the same time and obtain the average request duration five times lower w.r.t. to the case without LB. But in the real world it is almost unfeasible, because the requests are not equally distributed between VMs. The mechanism of choosing the least loaded machine takes also non-zero time, which extends the request execution duration. This can be observed using LB. The less a request execution time takes, the less requests are executed over the same time and the less is the increase of the average request duration.

8. Concluding remarks

Our research addressing invoking services from within workflows allows to draw the following conclusions:

- Auto-generation and auto-deployment proxies for services is making work in the described environment easier and more comfortable.
- The first tests show that the implemented LB mechanism improves performance.
- We expect that further research should bring even better results.
- We noticed that LB strategies have to be suited to the environment in which it is used so it is necessary to try various LB strategies [9]. It means that before configuring LB the user should more or less know how many services would be used, how much time the average execution time will take and how often proxy would get requests. Having got such information it would be possible to try a couple of configurations to find out which one would provide the best performance.
- As LB is a single point of failure, in the future it may be necessary to introduce an agent-based approach [6]. Using this concept LB would involve many agents distributed across several nodes. The role of agents is to communicate with each other to choose the least loaded service of a given type.

Acknowledgements

Our thanks go to dr. Maciej Malawski for inspiring discussions. This work is partially supported by the EU UrbanFlood project no. 248767 and by the European Union within the European Regional Development Fund program as part of the PLGrid Plus Project POIG.02.03.00-00-096/10 (`www.plgrid.pl`).

References

[1] Urban flood. `http://urbanflood.eu/aboutus.aspx`, 2012.

[2] The openesb wiki. `http://wiki.open-esb.java.net`, July 2012.

[3] Ws bpel, version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`, July 2012.

[4] Bourke T.: *Server Load Balancing*. O'Reilly, 2001.

[5] Cetnarowicz K., Drężewski R.: Maintaining functional integrity in multi-agent systems for resource allocation. *Computing and Informatics*, 29:947–973, 2010.

[6] Cetnarowicz K., Gruer P., Hilaire V., Koukam A.: A formal specification of m-agent architecture. In *From theory to practice in multi-agent systems : second international workshop of Central and Eastern Europe on Multi-Agent Systems, CEEMAS 2001*, pp. 62–72, 2001.

[7] Deelman E., Gannon D., Shields M., Taylor I.: Workflows and e-science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[8] Janhavi B., Surve S., Prabhu S.: Comparison of load balancing algorithms in a grid. In *Proc. of IEEE International Conference on Data Storage and Data Engineering (DSDE), Feb. 2010, Bangalore, India*, pp. 20–23, 2010.

[9] Pinedo M. L.: *Scheduling. Theory, Algorithms and Systems*. Springer, 2008.

[10] Porter G., Katz R. H.: Effective web service load balancing through statistical monitoring. *Commun. ACM*, 49(3):48–54, March 2006.

[11] Tarreau W.: Making applications scalable with load balancing. `http://www.exceliance.fr/sites/default/files/biblio/art-2006-making_applications_scalable_with_lb.pdf`, September 2006.

Affiliations

Jarosław Dąbrowski
AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Computer Science, al. Mickiewicza 30, 30-059 Krakow, Poland, `jaroslaw@student.agh.edu.pl`

Sebastian Feduniak
AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Computer Science, al. Mickiewicza 30, 30-059 Krakow, Poland, `sebastia@student.agh.edu.pl`

Bartosz Baliś
AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Computer Science, al. Mickiewicza 30, 30-059 Krakow, Poland, balis@agh.edu.pl

Tomasz Bartyński
AGH University of Science and Technology, ACC Cyfronet AGH, ul. Nawojki 11, 30-950 Krakow, Poland, t.bartynski@cyfronet.pl

Włodzimierz Funika
AGH University of Science and Technology, Faculty of Electrical Engineering, Automatics, Computer Science and Electronics, Department of Computer Science, al. Mickiewicza 30, 30-059 Krakow, Poland, funika@uci.agh.edu.pl

Received: 9.12.2011
Revised: 3.07.2012
Accepted: 9.07.2012