SOCA (2015) 9:193–209
DOI 10.1007/s11761-014-0160-z

ORIGINAL RESEARCH PAPER

Mechanism and architecture for the migration of service implementation during traffic peaks

Pekka Pääkkönen · Daniel Pakkala

Received: 27 August 2013 / Revised: 8 January 2014 / Accepted: 10 June 2014 / Published online: 21 June 2014

Abstract Service-Oriented Architecture has been widely applied in enterprise computing systems for software-enabled services. However, cost efficiency and scalability requirements have moved the execution environment towards the cloud domain. Hybrid approaches have emerged, which utilise both enterprise and cloud domains in order to balance between the cost of service execution and the provided Quality of Service (QoS) for end users. This paper presents a migration, monitoring and load-balancing mechanism and architecture for scaling services between the enterprise and cloud domains during traffic peaks. The argued benefit of the proposal is the automation of the service-migration process and improvement of the QoS. A prototype system is presented as a proof of the conceptual architecture. The performance results in a hybrid cloud environment indicate that service implementation can be migrated and load can be balanced within 200 ms. Furthermore, the mechanism can improve the QoS for end users during traffic peaks. Our approach differs from existing proposals by focusing on the migration of service implementation, instead of the migration of service as part of a virtual machine.

Keywords Service · Migration · Load balancing · Monitoring · QoS · Traffic peaks · Hybrid cloud

P. Pääkkönen (✉) · D. Pakkala
VTT Technical Research Centre of Finland, Kaitoväylä 1, 90570 Oulu, Finland
e-mail: pekka.paakkonen@vtt.fi

D. Pakkala
e-mail: daniel.pakkala@vtt.fi

1 Introduction

The paradigms of Service-Oriented Architecture (SOA) and Service-Oriented Computing (SOC) has been widely applied in enterprise computing systems to enhance the efficiency, agility and productivity of enterprises. However, in recent years, the adoption of SOA and SOC has also increased in other fields of computing. This allows the functional interoperability of technologically heterogeneous distributed computing systems when developing new services. These services are implemented with software that is deployed and executed on one or more networked host computers. The emergence of cloud computing, and especially the Software-as-a-Service (SaaS) deployment paradigm with programmatically accessible interfaces [1], combined with the digitalisation of information, creates a path towards the Internet of services. In this vision, services are provided and used over the Internet by human clients (e.g. via Web browsers or mobile applications), or by the client's computing systems via programmatically accessible interfaces (e.g. via Web services or REST). From the business viewpoint, the cost efficiency of hosting services in a scalable fashion has become a relevant concern. Organisations that provide software-based services over the Internet are looking for ways to minimise costs while providing adequate QoS.

Due to cost efficiency and scalability requirements, the mainstream execution environment for software-based services has changed from enterprise IT systems towards cloud computing paradigm-based deployment environments. Some services can be developed from the beginning to the cloud. In addition, existing services may be migrated to the cloud domain, whereas some services have constraints for cloud migration. The constraints include, for example, privacy concerns, concerns of intellectual property and the business sensitivity of processed data. However, some use cases exist, which may be of benefit to service migration. One example may be a personalised news website, which requires complex processing. QoS may be low during a big news event due to increased load at the server. Another use case may require the high-speed capturing and saving of network packets, which may be limited by database at the server. Migration of load between domains may increase the performance of network monitoring.

The contribution of this paper is a service migration, monitoring and load-balancing mechanism between enterprise computing and cloud computing environments during traffic peaks. The main idea is usage of Infrastructure-as–a-Service (IaaS) cloud instances during traffic peaks to ensure QoS for end users. In addition, the automation of service migration and load balancing is focused on with user-defined rules. A conceptual architecture has been realised in the described prototype system. The architecture differs from existing proposals by not migrating services as part of a virtual machine, but instead focusing on the migration of service implementation. Performance results indicate that service migration and load balancing are quick operations, and can be used for enhancing QoS for end users.

The structure of the paper is as follows. Related literature is presented in Chapter 2. Research scope, problems and method are presented in Chapter 3. Conceptual architecture is provided in Chapter 4 and proof-of-concept in Chapter 5. Performance test procedures are provided in Chapter 6; results are described in Chapter 7 and analysed in Chapter 8. Finally, the results of the study are discussed in Chapter 9 and concluded in Chapter 10.

2 Literature review

First, standardisation and definition of Internet-based services has been studied extensively. Jones reviews different standardisation efforts related to software-enabled services [2]. Service is defined as a "discrete domain of control that contains a collection of tasks to achieve related goals". In this scope, the service definition requires the specification of several aspects including interface, contract, security and performance. Yahia et al. [3] presented a service definition for next-generation networks, where a service is comprised of one or more reusable service components. In addition, the software architecture definition for cloud computing has been discussed [4]. In the proposed Reservoir architecture, a service is an application to be deployed. A Service Definition Manifest is applied to define a contract between the service provider and infrastructure provider for the correct provisioning of the service.

Often applied cloud-service definitions have been specified by NIST [1]. Additionally, literature on dynamically scaling applications in the cloud has been reviewed by Vaquero et al. [5]. Models for the provisioning of cloud services have been specified as Infrastructure-as-a-Service (IaaS), Network-as-a-Service (NaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS). Network scalability (NaaS) techniques are typically performed with overlay networks (L2/L3) or TCP/IP virtualisation (VLAN) [5], which enables virtualised access to network resources for the user. Cloud systems offering virtual hardware infrastructure and networks, while providing control over operating systems, tools and deployed applications, are referred to as IaaS clouds [1]. PaaS refers to the development and deployment of services to a cloud environment with tools and libraries offered by the cloud provider [1]. Finally, SaaS is a model for using applications running in the cloud environment via thin clients (e.g. Web browsers) or via program interfaces, but the user does not have control of the virtual environment, or even the applications [1].

Load balancing is often needed when traffic is being divided among different service containers, e.g. in a PaaS type of cloud model [5]. Centralised load balancing for Web servers has been proposed [6]. In the architecture, a master server controls load-balancing decisions based on server status. Another approach is to execute client-side load balancing to the cloud [7]. The idea is that a client makes a decision for the applied Web server based on the observed load, which is propagated to the client from a proxy. Transmitted HTTP requests are processed by a proxy and delivered to the chosen Web server. Many tools have been productised for load balancing [8], and functionality has also been integrated to cloud systems (e.g. Amazon's Elastic Load Balancing [9]).

In addition, application migration between enterprise and cloud domains from a practical point of view has been studied. Examples include the migration of an IT system in the oil and gas industry from in-house to Amazon EC2 [10], the migration of an automatic collector of process and product data to Amazon EC2 [11], and the development of CloudGenius for the migration of Web servers to clouds [12]. Based on experience of application migration, a taxonomy has been proposed [13]. Additionally, the costs and benefits of cloud computing, when compared to desktop grids, have been analysed [14].

QoS-aware infrastructure for monitoring and controlling service performance with NoSQL-distributed storage systems has been developed for data-intensive applications [15]. Also, a service-oriented monitoring framework with REST and Nagios has been developed [16].

Service migration, management and monitoring in the cloud domain were also a point of focus. Most of the solutions are based on the Open Virtualisation Format (OVF) for representing applications and associated virtual machines (VM). A design for service management in clouds was presented by Rodero-Merino [17]. Their focus is on a new layer for abstracting management of services executed in different clouds. Chapman et al. [4] follow a similar approach as Rodero-Merino [17] for migration of services, based on elasticity rules. Zhao and Huang [18] presented the live migration of virtual machines in a Red Hat Cluster environment based on a cost function of I/O usage. Vaquero et al. [19] present the most advanced work for migration of services in the cloud domain. Further, they show how load can be balanced between the private and public cloud domain. In addition, commercial service migration is available. Notably, Amazon's Elastic Load Balancing [9] enables automatic load balancing based on user-defined rules. Finally, hybrid enterprise/cloud architectures were analysed by Hajjat et al. [20]. Particularly, a model was presented for evaluating the benefits of a hybrid migration approach, which was analysed with a real enterprise application.

When services are defined as part of a virtual machine with OVF-model, several benefits are gained. Interoperability between cloud vendors, extensibility and the descriptive nature of the syntax is important qualities [19]. Also, it is possible to hide service-specific details when a service can be migrated as part of the virtual machine. However, the encapsulation of services inside a VM may not always be desirable. The performance of virtual machine migration may be low. It may take tens of seconds to migrate a Web server [19] or a VM [17,18]. This may lead to further problems, especially when service migration should be a quick operation, for example migration during traffic peaks.

Finally, Bicer et al. [21] proposed cloud bursting for data-intensive workloads. The proposal performs load balancing for MapReduce jobs between enterprise and cloud domains instead of focusing on service migration.

Based on the literature review, it can be seen that there are publications focusing on the full migration of services from the enterprise domain to the cloud domain. In addition, automatic service migration and load balancing for services with a virtualised cloud domain model have been focused on in many publications. However, service migration between enterprise and cloud domains during traffic peaks has not been focused on, when the migrated service is not specified as part of a virtualised model (e.g. OFV), which is the main contribution of this paper. Notably, service migration during traffic peaks is focused on, and the performance of the proposal has been evaluated as a proof-of-concept.

3 Research scope, problems and method

In this paper, we use the term 'service', when referring to a functionality set that is enabled by a software implementation executed in a networked host. The related cloud service model is IaaS [2], where virtual cloud resources are obtained from Amazon EC2, and the service administrator has control over services, operating systems and environments. Also, SaaS [1] may characterise usage, when the service administrator offers access to the service implementation via a programmable interface. We are not focusing on OVF-based service modelling, which defines services as part of a virtual machine(s). Instead, we focus on the migration of a service implementation, for example encapsulated in a JAR/WAR-file(s). This should enable improved performance for the service migration process.

The scope of research covers services, which have already been deployed in the enterprise domain, and continued deployment in the enterprise domain, may be preferred. In these cases, an investment has already been performed to cover costs of database servers and network equipment in the enterprise domain. When processing load requirements increase, some of the load may be preferred to be moved to the cloud domain to ensure QoS caused by an increasing amount of end users or traffic peaks.

The following use cases can be considered as potential applications of the proposed migration mechanism:

Service requires complex processing including reading from a database (e.g. a personalised news website)—an increasing amount of end users may lead to low QoS due to a higher processing load at the server. If requests would be balanced between hosts in different domains, QoS may increase.

Service requires heavy writing into a database (e.g. network monitoring equipment saves traces into a database)—high-speed data collection may be limited by the database at the server, if the amount of collected data is increased. If writes could be distributed between hosts in different domains, performance may improve.

Services and related data need to be transferred between business organisations—in this case, a service is initially executed in an enterprise domain, which is controlled by company A. A business transaction between company A and company B creates a need to transfer services and related data to the enterprise or cloud domain, which is controlled by company B. An automated solution would significantly reduce effort in service migration.

The initial goal of the work has been to develop a mechanism that would increase QoS for end users during traffic peaks (e.g. in a news website use case) and automate management of services. Traffic peaks are encountered frequently with network services, for example during a specific time of the day [22], which may often lead to decreased QoS for end users. The following motivating factors can be considered for the research:

1. Increased QoS for end users, when the resources or computing power of the existing servers in the enterprise domain is not enough for handling the load.
2. Automation of service migration and load-balancing tasks may reduce the effort for managing services on multiple domains.

3. The possibility to use cloud services based on real demand. This may eventually lead to a lower overall cost of service deployment when compared to the full migration of services to the cloud domain. Cost savings would be achieved by avoiding investment in additional cloud instances and network traffic, which could be handled by the existing enterprise nodes.

Based on this rationale, the following research questions are posed:

1. Can an automated service migration, monitoring and load-balancing mechanism be implemented as a software system and what kind of architecture would it have?
    a. How fast is the developed mechanism?
    b. What is the perceived performance improvement from the end-user point of view?

A constructive research method has been applied to answer to the first research question; conceptual and concrete architectures are presented and validated with a laboratory prototype system. Quantitative research methods have been applied for answering the subsequent questions regarding the performance of the mechanism. Database services and database synchronisation are out of the focus of this work, and an item for future research. Also, aspects of possible cost savings were not part of the focus.

4 Conceptual architecture

The architecture of the concept has been described in Fig. 1. The concept is illustrated from the point of view of a service provider/administrator. The administrator is executing the service initially in the enterprise domain and scales the service to the cloud domain during traffic peaks. The services are administrated via the developed admin interface of the Coordinator, and a proprietary admin interface of the cloud domain is applied for getting IaaS cloud instances based on demand.

The Coordinator contains the main logic and functionality in the architecture. It is applied to deploy services, and for the configuration of rules for service migration and load balancing. Additionally, the Coordinator receives monitoring data from a QoS monitor. For example the CPU load, service latency and disk usage can be monitored. The performance data received from the QoS monitor is applied for making decisions for service migration and load balancing.

The other main component of the architecture is the Worker. It is applied for to execute service deployment and load-balancing operations. The Worker communicates with a Service container to execute deployment operations via the admin interface of the Service container. The Worker also controls load balancing and executes operations ordered by the Coordinator.

One main issue of the architecture is the independency of an algorithm for service migration and load balancing. Thus, the service administrator is responsible defining the

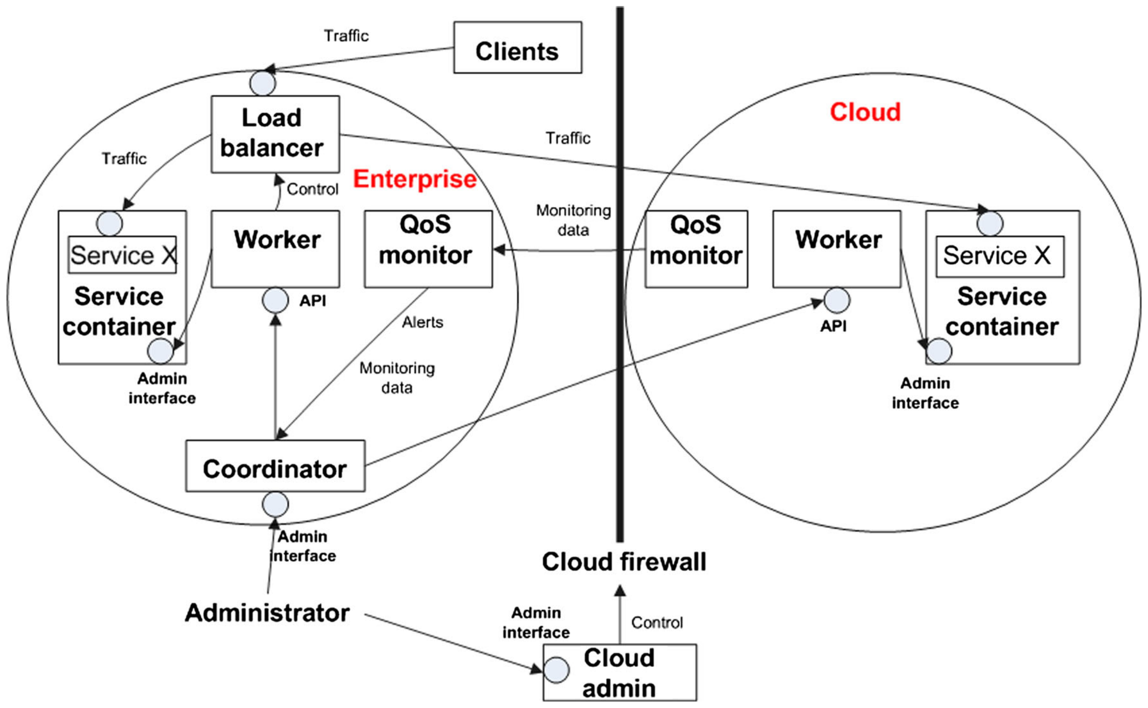


Fig. 1 Conceptual architecture—conceptual architecture is comprised of the coordinator, worker, QoS monitor, service container and load balancer

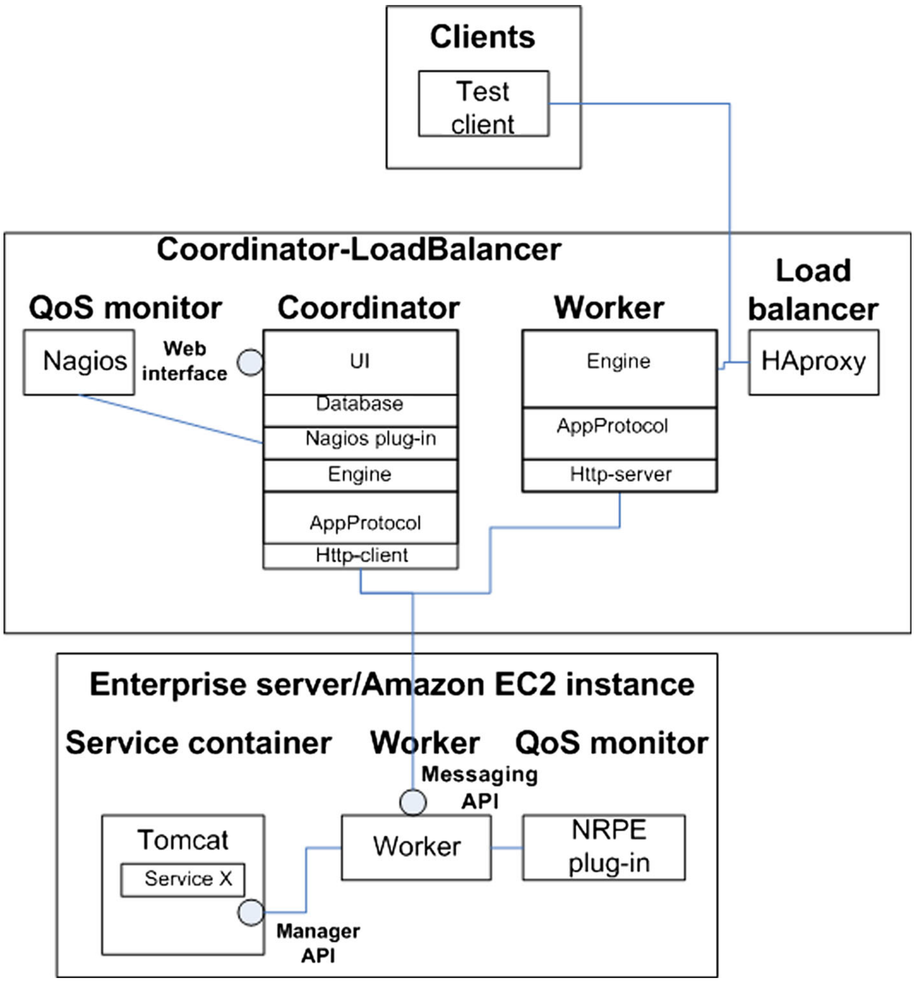


Fig. 2 Architecture of the proof-of-concept—conceptual architecture was implemented as a proof-of-concept laboratory prototype on three nodes and one Amazon EC2 instance

algorithms for service migration, which are specified with migration rules, and configured via the Admin interface. The migration rules are further translated into alerts received from the QoS monitor to execute service migration.

5 Proof-of-concept

In this chapter, proof of the conceptual architecture is presented. First, the architecture of the implementation is presented in Sect. 5.1. Then, the main data flows of the system have been described in Sect. 5.2. Subsequently, the main class definitions of the Coordinator are presented in Sect. 5.3. Finally, sequence diagrams illustrate the functionality of service deployment and the configuration of migration rules (Sect. 5.4), and service migration and load balancing (Sect. 5.5).

5.1 Architecture of implementation

The presented architecture was implemented and tested with three nodes and one virtual Amazon EC2 instance (Client, Coordinator-LoadBalancer, Enterprise node and Amazon EC2 instance in Fig. 2).

The test client simulated a large number of end users of a service. The Coordinator, QoS monitor, Worker and Load Balancer were executed on one node (Coordinator-LoadBalancer). A Nagios network monitoring tool was chosen as the QoS monitor, which was integrated with the Coordinator. A plug-in was developed for the Coordinator for interacting with Nagios. A Web interface was developed as a user interface for interacting with the Coordinator, e.g. for the deployment of services and configuration of the rules for service migration and load balancing. A simple database contained the previously mentioned rules and state information of the services. The Engine contained the main logic of the Coordinator.

The Coordinator communicated with a Worker for load-balancing operations. HAproxy was chosen as the technology for balancing the load, and the Worker was integrated with it.

The Enterprise Node and Amazon EC2 instance contained a Worker to deploy services. Tomcat was applied as a service container. Tomcat's Manager API was used as an adminis-

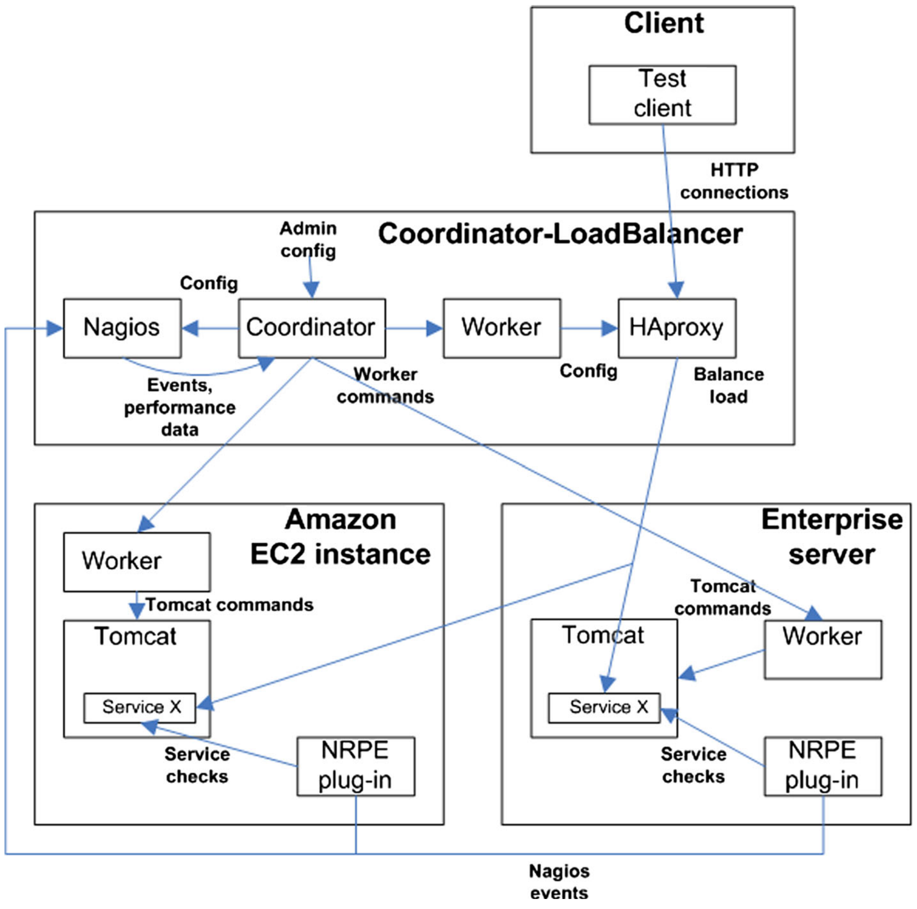


Fig. 3 Data flows of the implementation—important data flows between system components

tration interface for deployment operations. NRPE Nagios Plug-in [23] provided remote-monitoring information from the Enterprise Node and Amazon EC2 instance to the QoS Monitor at the Coordinator node.

Communication between the Coordinator and Worker was implemented over HTTP. Notably, a message-oriented protocol was implemented on top of HTTP.

5.2 Data flows of implementation

Figure 3 presents main data flows of the implementation. First, the data flow from clients is balanced at the HAproxy towards service implementations, which are executed in different domains. Second, service-administration commands and service-rule configurations executed by the administrator are caught by the Coordinator. One Worker executes reconfiguration commands with the HAproxy based on service-rule configurations. Another Worker communicates service-deployment commands to Tomcat. Additionally, the Coordinator initialises Nagios based on configuration of the service-migration rules. Third, Nagios NRPE Plug-ins communicate monitoring data to Nagios. When an important event is triggered, the Coordinator gets the event from Nagios and reads the performance data from Nagios's output performance files for decision-making regarding the execution of service migration.

5.3 Main class definitions of the coordinator

The main definitions of the Coordinator implementation have been described in Fig. 4. The main logic of the Coordinator is in the Engine, which uses the presented class definitions.

The service package contains definitions regarding the configuration of services. The service rules package contains high-level definitions for service migration and load-balancing rules. The Nagios package (part of the Nagios plug-in) contains low-level definitions for monitoring events received from the Nagios monitoring tool. In the following explains the main classes of the packages.

Service class is a high-level concept of a service, which may contain one-to-many ServiceComponents. A ServiceComponent is meant for encapsulating information of an executable service, e.g. a Jar-/War-file in a service container. It may contain one-to-many ServiceInstances, which refer to the execution of a service component in a particular service container (e.g. Tomcat/Glassfish). Additionally, ServiceDataConfig is defined for encapsulating the load-balancing configuration of a ServiceComponent.

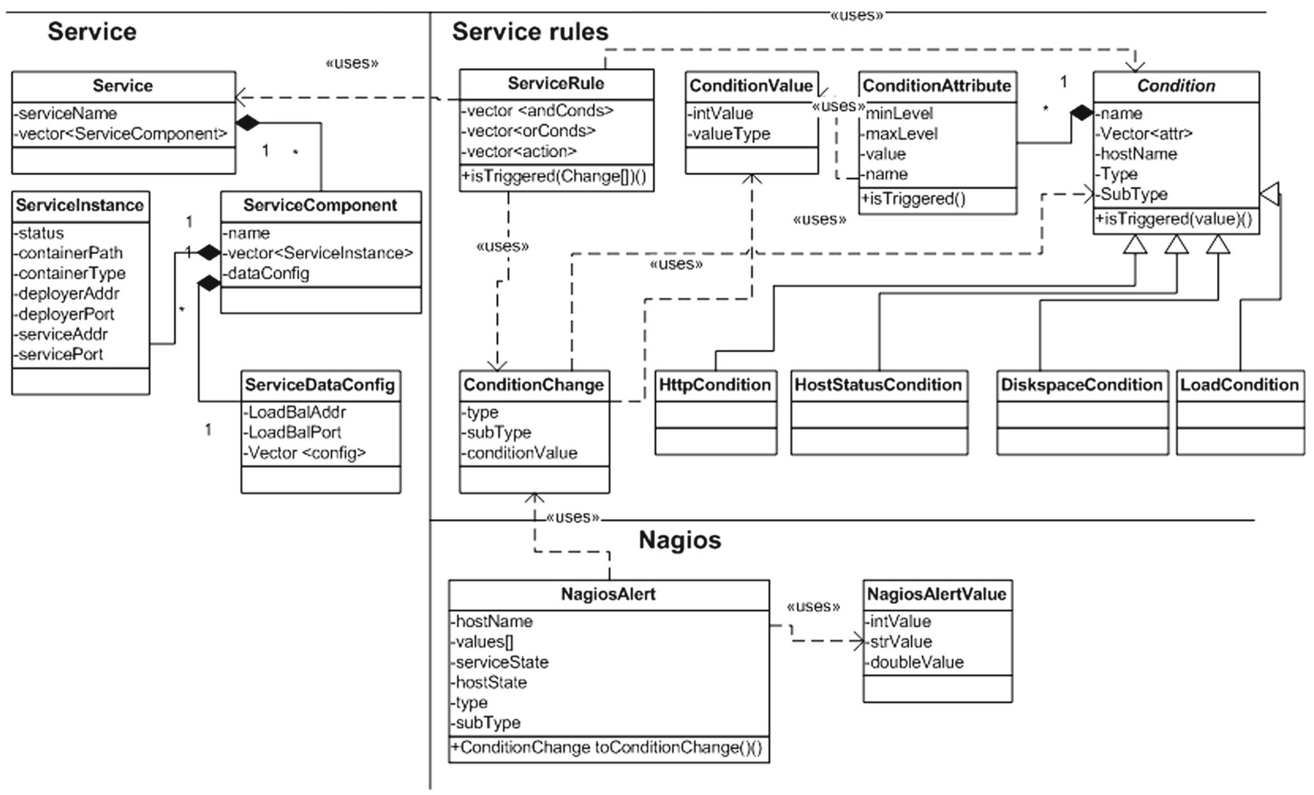


Fig. 4 Main classes of the coordinator implementation—the classes of the coordinator are divided to service, service rules and Nagios packages

Condition class refers to any condition that is applied for decision- making regarding migration and load balancing of services. It may contain one-to-many ConditionAttributes, each associated with a value, with a minimum or maximum level allowed. Currently, four conditions have been defined (load, disk space, host status and Http). ServiceRule-class may contain one-to-many Conditions, which have to be fulfilled to trigger service migration or load balancing. ServiceRule also contains the actions (association with a Service) to be executed in case a rule is triggered.

ConditionChange-class refers to a change in a condition, which may trigger a ServiceRule. NagiosAlert-class refers to low-level alert received from Nagios and is related to an event regarding status of a Nagios service or a host. NagiosAlerts are converted to ConditionChanges for decision-making regarding service migration.

5.4 Service deployment and configuration of migration rules

Figure 5 presents the deployment of a service and the configuration of migration rules. In the use case, a user deploys a service with the Coordinator on the Enterprise Node. Additionally, load balancing is configured via HAproxy. Finally, migration rules are set, which leads to performance monitoring with Nagios.

In step 1, the user provides a War-file of the service, the name of the service and service component, path of deployment, address and port of the Worker, and configuration for load balancing via the Web interface. In step 2, the Coordinator sends the received information to the Worker at the Enterprise Node. Subsequently, the Worker deploys the service locally in Tomcat via the manager API (step 3).

After deployment of the service, the load-balancing configuration is sent to the Worker at the Coordinator node (step 4). The Worker performs the required changes to an HAproxy configuration file (step 5) and restarts it using the hot configuration capability of HAproxy [24] (step 6). The Coordinator saves the configuration of the service locally to a simple database (step 7).

Finally, the user sets rules for service migration and load balancing (step 8), which are saved to the database by the Coordinator (step 9). Then, the received configuration is saved to the Nagios configuration files (step 10), and Nagios is reloaded (step 11). This causes Nagios to start balancing the load towards the deployed service on the Enterprise node. Thus, the definition of service rules leads to monitoring the service state with Nagios.

5.5 Service migration and load balancing

Figure 6 presents the migration of a service from the Enterprise Node to the Amazon EC2 instance based on a change of

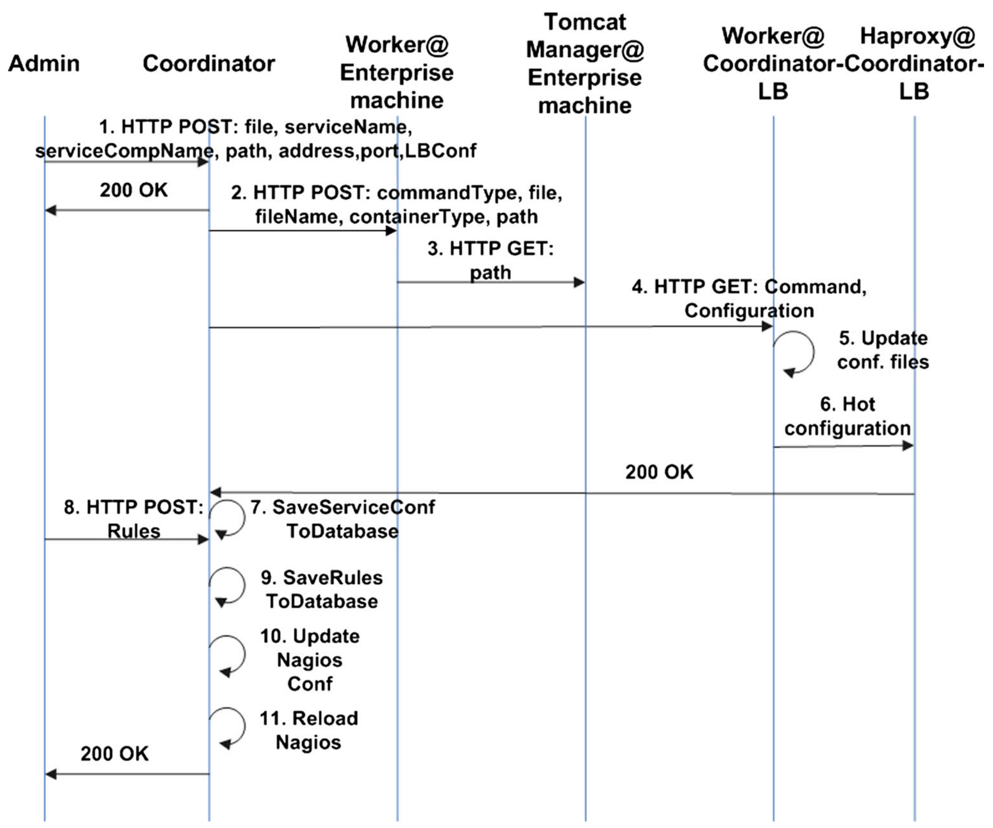
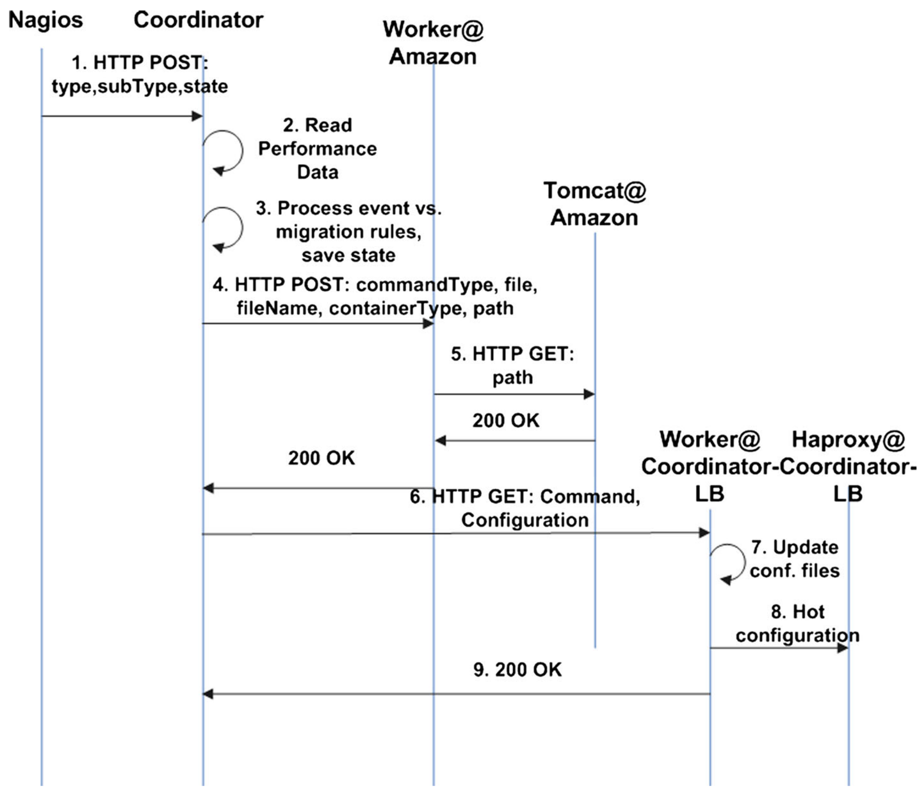

Admin
Coordinator
Worker@ Enterprise machine
Tomcat Manager@ Enterprise machine
Worker@ Coordinator-LB
Haproxy@ Coordinator-LB
1. HTTP POST: file, serviceName, serviceCompName, path, address,port,LBConf
200 OK
2. HTTP POST: commandType, file, fileName, containerType, path
3. HTTP GET: path
4. HTTP GET: Command, Configuration
5. Update conf. files
6. Hot configuration
200 OK
8. HTTP POST: Rules
7. SaveServiceConf ToDatabase
9. SaveRules ToDatabase
10. Update Nagios Conf
11. Reload Nagios
200 OK

Fig. 5 Sequence diagram for service deployment and configuration of migration rules—first, a service is deployed to the Enterprise Node (steps 1–6). Then, load balancing is configured based on user-defined rules (steps 8–11)

Fig. 6 Sequence diagram for migration of a service—a service is migrated from the Enterprise Node to the Amazon (steps 4–9) instance based on a trigger received from Nagios (steps 1–3)



condition triggered by Nagios. Additionally, operations for load balancing are presented.

First, Nagios detects an event based on the earlier configuration, and sends the event (NagiosAlert in Fig. 4) to the Coordinator (step 1). For example, the load on the Enterprise Node may be monitored, which was started based on the definition of service rules. If the load increases above the configured threshold, an alert will be triggered by Nagios. Next, the Coordinator reads the monitoring data from the Nagios performance files (step 2). Subsequently, the event (ConditionChange in Fig. 4) is processed and compared to migration rules (ServiceRule in Fig. 4) (step 3). In addition, the state of the condition is saved at the Coordinator. If the migration rule is triggered, the associated action will be executed. The action contains an end state for different services and may also contain information about load balancing (ServiceDataConfig in Fig. 4), as in this particular case.

Next, the service will be migrated to Amazon EC2 based on a message exchanged between the Coordinator and the Worker executed at the Amazon instance (steps 4–5). Finally, load balancing towards Amazon EC2 will be configured (step 6). The HAproxy configuration file will be updated, and a new entry will be added for balancing traffic towards Amazon (steps 7–8).

6 Performance testing

In this Chapter, test set-up and test procedures are presented for performance analysis of the proof-of-concept.

6.1 Latency of service migration and load balancing

6.1.1 Purpose

The test was performed to measure latency regarding service migration and load-balancing procedures (see Fig. 6). Both procedures should be fast enough to enable quick migration of services and configuration of load balancing.

6.1.2 Test procedure

In the test, an event received from Nagios was simulated, and it was propagated to the Coordinator. The event was processed, which triggered a service rule. The pre-configured action was deployment of a service in Amazon EC2 and configuration of load balancing at the HAproxy towards the service. After service deployment and load balancing, the initial set-up was established again by undeploying the service and removal of the load-balancing configuration. Five thousand test iterations were executed. The deployed service was a small War-file (~5kB).

6.1.3 Test metric

Latency was measured in each step of the process with a latency-measurement tool [25]. The steps of the procedures under measurement at the Coordinator are similar to the steps in Fig. 6:

Step 1: Transmission of a test event from a simulated Nagios instance to the Coordinator.
Step 2: Reading of performance data from the Nagios performance file.
Step 3: Processing of the event in the Coordinator.
Steps 4–5: Deployment of the service to Amazon EC2.
Step 6: Transmission of the load-balancing configuration for service deployed at Amazon EC2.
Step 7: Updating of the HAproxy configuration files.
Step 8: Hot reconfiguration of HAproxy.
Step 9: Response message from the Worker to the Coordinator.

Additionally, latency in the deployment of a service instance in Tomcat was measured at the Amazon EC2 instance as follows:

Step 1: Send deploy message (HTTP GET) via Tomcat Manager API.
Step 2: Receive response message (200 OK) from Tomcat Manager API.

Delay components were calculated for each step as follows:

$$delayAvg_{k,k-1} = \frac{\sum_{k=1}^{tests} t_{step(k)} - t_{step(k-1)}}{tests} \quad (1)$$

6.2 Performance improvement during a traffic peak

6.2.1 Purpose

The test was performed to validate that service migration during traffic peaks leads to increased performance for the end users of a service.

6.2.2 Test procedure

In the test, a Test client executed a varying number of HTTP requests towards the server (Fig. 3). Initially, load balancing was configured to route traffic towards the Enterprise server. At the Enterprise server, a service was deployed, which simulated the execution of a CPU-intensive process. For each received message at the server, 100,000 bytes of text were encrypted and decrypted with the Advanced Encryption Standard (AES) algorithm. In addition, the CPU usage at the

Table 1 HW/SW configuration of the test bed—HW/SW configurations in the different nodes of the prototype system are presented

	Test client	Coordinator	Enterprise server	Amazon EC2 instance
HW	Dell Latitude D400 laptop, Intel Pentium M, 1.6 GHz, Ubuntu Karmic, RAM = 1GB	Dell Latitude D400 laptop, Intel Pentium M, 1.4 GHz, RAM=1GB	Dell Optiplex 990, 4 CPU* 3.1 GHz, RAM=8 GB	Reserved small instance in Ireland (1 vCPU, 1.7 GB, low network perf)
SW	Java v1.6.20	Java v1.6.24, HAproxy v1.4.20, Nagios Core v3.2.3	Java v1.6.24, Tomcat v7.0.26	Java v1.6.24, Tomcat v7.0.26

Enterprise server was limited to 25 % with the cpulimit Unix-tool [26]. It was applied for configuration of performance between the Enterprise Node and the Amazon EC2 instance closer to each other (see the configuration in Table 1).

The load at the Enterprise server and Amazon EC2 instance was monitored with Nagios and NRPE plug-ins. Service migration rules were configured as follows:

Migration rule 1: If the load is above 60 % of CPU time, deploy the service to Amazon EC2, and balance the load between the Enterprise Node and the Amazon EC2 instances by using the round-robin algorithm of HAproxy.

Migration rule 2: If the load is lower than 30 %, undeploy the service in Amazon EC2, and balance the load back to the Enterprise Node.

Migration rule 1 was aimed at balancing the load during a traffic peak towards Amazon EC2 to increase QoS perceived at the Test client. Migration rule 2 was aimed at balancing the load back to the Enterprise Node, when the traffic peak is over.

CPU load as a percentage (%) was converted to a Nagios numerical load value by doubling the percentage value. Thus, 60 % load on the application level translated into a 2*0.6 = 1.2 value on the Nagios load configuration [24].

The test client had 15 Java threads, which tried to send as many HTTP GET messages as possible to reach the target message rate. The target message rate was first increased from 5 to 30 msg/s, and then decreased back to 5 msg/s. The change of message rate was discrete (2 msg/s) with a 30 s interval.

The test was executed three times, with and without (reference) the migration mechanism.

6.2.3 Test metric

Measurements were performed at the test client to analyse QoS as follows:

$$latencyAvg = \frac{\sum_{k=1}^{tests} t_{response(k)} - t_{transmission(k)}}{tests} \quad (2)$$

$$throughput = \frac{20}{t_{20th\,transmission} - t_{1st\,transmission} + latency_{20th\,transmission}} \quad (3)$$

Throughput is actually real-time application-level goodput, which is calculated at every 20th completed message transaction. Average, median, minimum and maximum values were calculated for each test run. Additionally, the time at the triggering of migration rules was captured, and sent to the test client for latency measurement and throughput during the detected traffic peak.

6.3 Set-up of the test bed

Table 1 presents the HW and SW configuration of the test bed. The Test client, Coordinator and Enterprise Node were connected to each other via a 100 Mb bridge. An Internet connection was established via a 1 Gb test network in Oulu, Finland. The Amazon EC2 instance was executed in Ireland.

7 Test results

7.1 Latency of service migration and load balancing

Figure 7 describes test results in service migration and load balancing. Notably, the components of total latency have been described.

7.2 Performance improvement during a traffic peak

Figure 8 presents latency at the Test client without service migration during a traffic peak. Figure 9 presents latency at the Test client with service migration and load balancing during a traffic peak. Additionally, triggering events of migration rules have been presented. Figure 10 presents throughput at the Test client without service migration and load balancing. Figure 11 presents throughput with service migration and load balancing during a traffic peak.

Table 2 presents the minimum and maximum range (in the three test cases) for latency and throughput regarding different statistical measures.

8 Analysis

In Sect. 8.1, the architecture and design of the prototype system is analysed. Section 8.2 analyses the results of perfor-

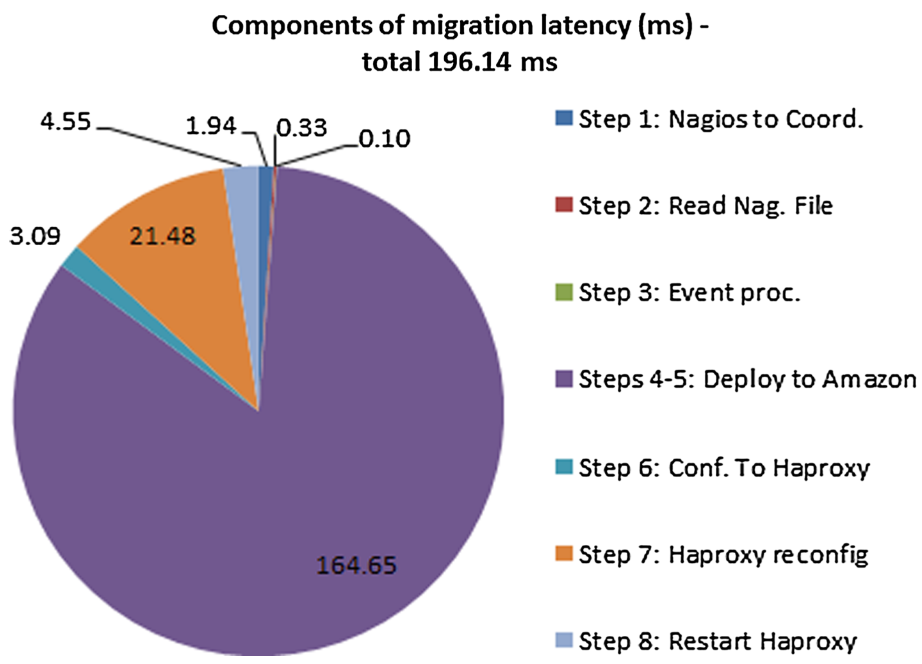


Fig. 7 Components of latency in service migration and load balancing

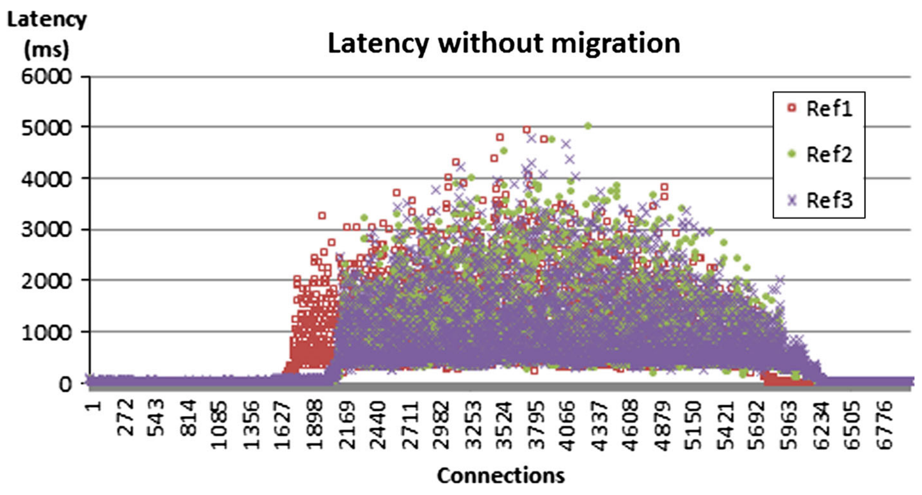


Fig. 8 Latency during a traffic peak without service migration and load balancing

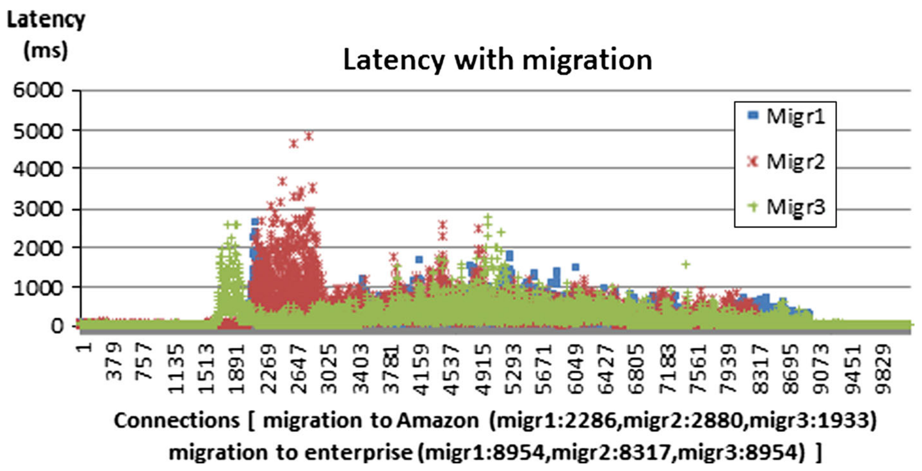


Fig. 9 Latency during a traffic peak when service is migrated to Amazon EC2 and the load is balanced. Triggering events for migration and load balancing are also provided

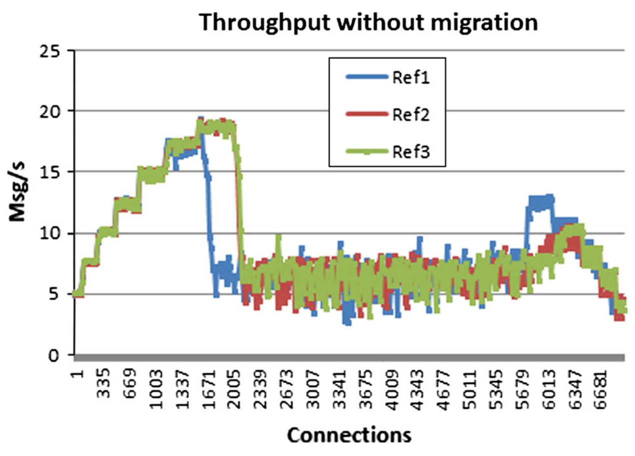


Fig. 10 Throughput during a traffic peak without service migration and load balancing

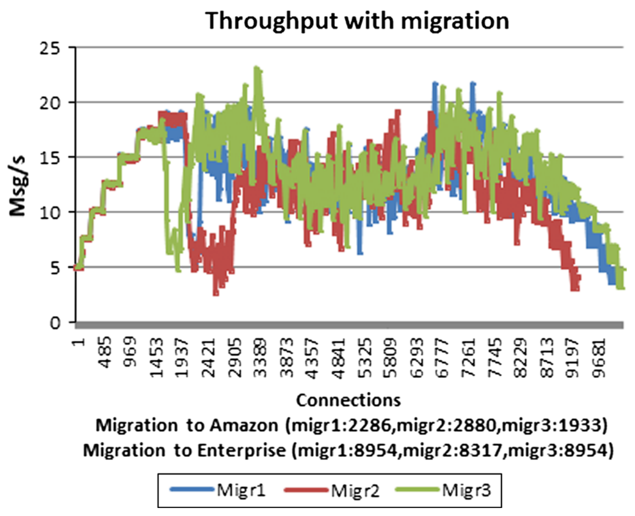


Fig. 11 Throughput during a traffic peak, when service is migrated to Amazon, EC2 and the load is balanced. Triggering events for migration and load balancing are also provided

Table 2 Statistical distribution of latency and throughput with/without service migration and load balancing—range (minimum and maximum) for the reference and the migration mechanism has been provided

	Ref-min	Ref-max	Migr-min	Migr-max
Connections	6933	7030	9043	10199
Avg. lat. (ms)	596.3	614.6	196.6	287.2
Median lat. (ms)	427.9	456.2	106.8	166.5
Min lat. (ms)	16.0	16.2	15.9	15.9
Max lat. (ms)	4804.4	5035.9	2670.3	4830.1
Avg. lat. in peak (ms)			244.3	400.9
Thr.put max (msg/s)	19.2	19.4	19.1	23.2
Thr.put avg. (msg/s)	8.7	9.1	12.4	13.6
Thr.put in peak (msg/s)			12.9	14.7

mance tests. Finally, the solution is compared to related work in Sect. 8.3.

8.1 Architecture and design

8.1.1 Conceptual architecture

We concentrated on the migration of service implementation, which would be executed inside a VM, instead of migration of VMs. The rationale is improved performance in terms of service migration latency, when VM instantiation is performed before the actual migration process. However, relaxing the VM dependence has implications for the service architecture.

When services are defined with the OVF-model, many details of the service implementation can be hidden, because they can be migrated as part of a virtual machine. Our approach exposed some of the details to be handled. For example, the communication parameters of service components have to be handled to enable load balancing at HAproxy.

In our approach, the service administrator is responsible for specifying the service migration rules. Currently, CPU load, disk usage, HTTP and host condition can be specified in the migration rules, which are mapped into events received from Nagios. However, the list of monitored variables can be extended.

The main components of the architecture are the Coordinator and the Worker. The idea is that the Coordinator manages all Workers, which are needed for managing services in containers and load balancing traffic between service instances.

8.1.2 Proof-of-concept

Implementation architecture We chose HAproxy as a load balancer and Nagios as a network monitoring tool, because they are production-level tools for their intended purpose. Other alternative technologies could have been chosen.

The Worker and Coordinator communicated over HTTP protocol. We chose a Jetty Http server [27], because it enables building embedded HTTP servers to the Java implementation, instead of running a separate instance (e.g. Tomcat server). We also implemented a message-oriented protocol on top of HTTP to enable different application layer protocols. For example, load-balancing and service-deployment messages from the Coordinator to the Worker use different messaging protocols, which were specified for the purpose. In particular, application protocols were implemented as a middleware layer between the HTTP-implementation and application layer of the Worker. Each application layer instance (engine) had a unique identifier, which was applied for messaging with the Coordinator. The Worker applied one IP address and port for communication, and messages were relayed to the registered application layer instances (engines). This enabled many application layer instances to be executed in the Worker (e.g. executed in different threads). For example, engines at the Worker for Tomcat and HAproxy (Fig. 2) could be reached at the same IP address and port-pair of the Worker.

In the prototype system, a load balancer was executed on the same node with the Coordinator. However, HAproxy and the associated Worker need not be colocated with a node where the Coordinator is executed. A change in the load-balancing configuration was implemented by modifying the HAproxy configuration files, and by executing hot reconfiguration [8]. This required the execution of a Unix-script from the Java implementation, which restarted HAproxy.

Changes in QoS monitoring are executed by modifying the Nagios configuration files and by reloading Nagios. In practice, a UNIX system call is required from the Java implementation. The developed NRPS Plug-in at the Coordinator read the performance data from the Nagios output files after it received a Nagios event. This required the Coordinator and Nagios to be colocated on the same node (Fig. 3). The NRPS Nagios plug-in (Fig. 3) was applied for remote monitoring on enterprise and Amazon domains, but also any other Nagios remote-monitoring plug-in could have been applied for the same purpose.

Main class definitions The rationale for the Service-package is to encapsulate information of an abstract Service, which may contain a different number of ServiceComponents. An example of a ServiceComponent could be a JAR-file. However, the JAR-file could be executed in multiple service containers. Thus, ServiceInstance corresponds to the actual JAR-file, which is executed in a service container (e.g. Tomcat/Glassfish). Configuration for load balancing (ServiceDataConfig) has been associated with ServiceComponent, because service components can be executed on multiple service containers, which need configuration for load balancing (such as for HTTP proxying).

The ServiceRules-package is aimed at encapsulating a high-level view for state of the service (based on different Conditions), which is visible to the maintainer of the service. Currently, the CPU load (LoadCondition), disk usage (DiskspaceCondition), HTTP (HttpCondition) and host services (HostStatusCondition) can be specified in the migration rules (UI at the Coordinator), which are mapped into events received from Nagios. Service may have multiple associated Conditions, any of which may trigger an action. An action corresponds to the end state of a Service, which is compared to the earlier state of the Service (Fig. 3). Typically, deployment of ServiceComponent(s) and configuration of load-balancing rules (ServiceDataConfig) are compared, and the required action is executed.

The Nagios-package contains a low-level view of the host and services, according to the Nagios Core documentation [24]. Notably, Nagios has different states (Ok, Warning, Critical, etc.) associated with a service or host, which should be mapped to an understandable concept on a higher level. Currently, a mapping between the views has been specified to the NagiosAlert-class, which converts a Nagios event to a ConditionChange. An example is system load, which is a numerical value, and depends on the amount of processors available, and processor idle time. However, mapping to an understandable measure at the high level requires effort (e.g. 0–100 % system load would be less unambiguous).

8.2 Performance results

8.2.1 Latency of service migration and load balancing

When the results of processing latency in service migration and load balancing are analysed (Fig. 7), it can be seen that total latency of the operations is less than 200 ms. The main components of latency (∼95 %) are service migration towards Amazon EC2 (steps 4–5) (83.5 %) and reconfiguration of load balancing at HAproxy (step 7) (11.3 %).

The main latency components were analysed further. Deployment of services over the Tomcat Manager API in the Amazon EC2 instance was also measured. The average delay in service deployment in Amazon was ∼91 ms, which comprises more than half of the service-migration latency. The other half of the service-migration latency is caused by large RRT from Finland to Ireland, where the Amazon EC2 instance is deployed.

Reconfiguration of the load-balancing process consists of modification of HAproxy configuration files, and restarting HAproxy. This process was studied further. New tests with 1,000 test iterations were performed, in which a simple load-balancing configuration was added and removed at the Enterprise node. The results indicate that, when a new configuration is added, ∼89 % of the delay is caused by restarting the HAproxy process. In practice, this is caused by a system call from the Java implementation to the Linux operating system, which restarts HAproxy.

8.2.2 Performance improvement during a traffic peak

When latency is analysed from client point of view (Figs. 8, 9), it can be seen that the migration mechanism can be applied to enhance QoS. When no migration mechanism is applied, latency increases from tens of milliseconds up to 5 s during a traffic peak. When a migration mechanism is applied, latency increases at first, but reduces after service migration and load balancing has been executed. This can also be noticed when migration events are compared to changes of latency (see Fig. 9).

Table 3 Comparison of architectural elements between our work and published literature

Paper	Service definition	Migration rules	Migration condition/ trigger	Coordination	Monitoring	Load balancing
Rodero-Merino [17]	SDF, OVF	SDF, OVF	Based on KPI	Claudia and virtual infrastructure manager	KPI (job queue)	–
Katsaros [16]	REST API to data model	–	–	–	Nagios	–
Zhao and Huang [18]	Red Hat Cluster Conf. files	–	Cost function	DLBA algorithm	Cost function (CPU and I/O load)	VLAN reconf.
Chapman [4]	ADL, OVF	Elasticity rules	Based on KPI	Service manager, grid management service	KPI (job queue in Condor scheduler)	–
Amazon Elastic LB [9]	Auto scaling group of EC2 instances	Scaling policies for auto-scaling groups	Alarms, policies	Amazon	CloudWatch metrics/alarms	Elastic LB
Vaquero [19]	OVF	RIF, Policy language	Based on KPI	Rule manager, rule engine, rule enforcement	Custom monitoring module	App re-tiering use case (Tomcat, Jetty)
Bicer [21]	Generalised reduction API	–	–	Head node	–	Job stealing
Our contribution	Service- package	ServiceRules-package	ServiceRules-package	Coordinator-module	Nagios	HAproxy

When realised throughput is analysed, it can be seen that the enterprise node is able to process ∼20 mgs/s (Fig. 10). However, when more messages are transmitted to the server, throughput decreases significantly and latency increases. With the migration mechanism throughput also decreases at first until service migration and load balancing have been executed. After service migration and load balancing, throughput is much higher and latency much lower when compared to the reference.

When the detailed results are analysed (Table 2), it can be seen that with the service migration mechanism up to 47 % more messages are served in total. It can also be seen that average and median latency is significantly lower without service migration and load balancing. Also, average latency during a traffic peak is less than 500 ms, which is a significant improvement over the reference (Fig. 8).

The average throughput is 3.3–4.9 msg/s higher, when compared to the reference. The increase in throughput was studied further by executing the reference test with Amazon EC2 instance (without Enterprise node). The results indicate that an average throughput of 5.0–5.3 msg/s could be achieved. Thus, load balancing between the Amazon EC2 instance and the Enterprise Node with the presented mechanism enables quite efficient usage of the extra capacity, even though it has not been optimised for maximisation of throughput/latency.

8.3 Comparison to literature

The most closely related literature to our work has been compared in Table 3 from the point of view of different architectural elements. The main difference is that our work is the only one that focuses on the migration of services during traffic peaks between enterprise and cloud domains, which does not associate the migrated service implementation as part of a virtual machine based on an OVF-model. In the following, the works are compared to our contribution in detail.

Rodero-Merino et al. [17] present a service abstraction layer (Claudia) for service management and monitoring. They propose a definition of service and migration rules with Service Description Files (SDF), which is an extension of the OVF-format. In the architecture, Claudia and the virtual infrastructure manager perform coordination, and the migration of services is performed based on changes in Key Performance Indicators (KPI). The biggest differences to our work is that the detailed description of resource monitoring based on KPIs is missing, and load balancing has not been dealt with, nor do they focus on migration of services during traffic peaks.

Katsaros et al. [16] present integration between Nagios and REST for building a framework for monitoring services. They developed the NEB2REST module for mapping information received from Nagios to the database of the monitoring service, which can be compared to the Nagios plug-in we developed for the Coordinator (Fig. 2). Libvirt library was used for monitoring the virtual infrastructure, whereas we used NRPS for monitoring remote hosts. The main difference to our work is that we also present integration with load balancing and user-defined rules for automatic service migration during traffic peaks.

Zhao and Huang [18] have presented a load balancing algorithm for the live migration of virtual machines in a Red Hat cluster environment. Migration of VMs is based on a cost function, which is comprised of CPU and I/O usage. Live migration of VM took 162 s in average. The difference to our approach is that service migration rules are not specifically defined. Further, their concept of load balancing is focused on reconfiguration of VLAN across domains rather than redirection of traffic (with HAproxy).

Chapman et al. [4] follow a similar approach as Rodero-Merino [17]. Services are specified with Application Description Language (ADL), which is based on the OVF-standard. Elasticity rules specify migration of services based on the status of KPIs. Condor Manager monitors the status of the job queue and executes migration of services. The Service Manager/Grid Management Service performs the migration of services. Monitoring of services focuses on KPIs such as job queue, which is different from our approach of using real HW-related information provided by Nagios. Also, they have not focused on a load-balancing solution for handling traffic peaks.

Elastic Load Balancing at Amazon EC2 enables the load-balancing functionality in the cloud domain. The load can be balanced based on health checks of instances, and auto-scaling enables the automatic starting of instances on demand based on user-defined policies [9]. The main difference is that our solution focuses on service migration and load balancing between the enterprise domain and cloud domain.

Vaquero has presented similar work on the migration of services in the cloud domain [19]. Service definition is based on the OVF-standard, and a policy language is applied to specify migration rules. A custom-monitoring module detects a change in KPI status, and a Rule-Enforcement module executes the actions specified in the Rule Engine. An application re-tiering use case presents how load can be balanced, when a Web service is migrated between public and private cloud domains. Differences to our work are lack of details about monitoring (in the Custom-monitoring module), and their focus on virtualised environments. On the other hand, Vaquero et al. [19] have performed a detailed performance evaluation for migration rules (e.g. Rule manager), which is also important.

Finally, Bicer et al. [21] proposed a framework for the cloud bursting of data-intensive computing. The solution is able to distribute MapReduce jobs between enterprise and cloud domains, which significantly reduces processing latency. A new MapReduce type of API (a generalised reduction API) is used for specifying jobs. A head node in the architecture coordinates the assignment of jobs between the domains. Data to be processed is migrated between domains instead of the migration of services. The main differences to our proposal is a lack of monitoring functionality or migration rules, and their focus on the migration of data for MapReduce processing, instead of our focus on the migration of service implementations.

Practical reports [10–13] on cloud migration may also be compared to our work. However, they are related to the full migration of applications and services from the enterprise domain to the cloud domain, not on hybrid solutions such as handling traffic peaks. The benefits of migration between enterprise and cloud domains have been analysed [20]. Notably, component placement and access control lists were focused on during the migration process. This work appears to be complementary to our work. Moreover, Hajjat et al. [20] mention "handling of dynamic variations in workload" as a potential benefit of migration, which was covered in our work.

9 Discussion

In the following research results, limitations and future work are discussed.

The main difference between our work and other proposals is that we did not associate the migration of service implementation to the migration of a virtual machine. When services are migrated, e.g. based on the OVF-model, some details of the migrated service can be hidden inside the migrated virtual machine. In our work, we assumed that a virtual machine was instantiated beforehand, for example by purchasing a reserved instance from Amazon. The biggest benefit should be improved performance in the migration process, when compared to the migration of virtual machines. The delay in the migration of a virtual machine is tens of seconds [17–19], which is high when compared to the migration of the service implementation ($\sim$200 ms). However, this has some implications on the developed service architecture.

For example, many preliminary operations in the current architecture have to be performed before automation is achieved. First, an instance has to be acquired with the Amazon account. Also, the required SW tools have to be installed (Java, Tomcat etc.). Then, the Amazon firewall has to be configured to enable traffic between enterprise and Amazon domains. Finally, Nagios and the NRPS Plug-in, HAproxy, Tomcat, the Controller and the Workers have to be started in different domains. In addition, configuration information from service deployment, load balancing and migration rules create complexity for the Web interface of the Coordinator. However, the previously mentioned operations may be automated further, if more functionality was built inside the Coordinator.

The performance tests focused on testing migration procedures between enterprise and cloud domains for a small service (∼5 KB JAR-file). However, a realistic scenario may consist of large files to be transferred between the domains, which have to be taken into account. Also, multiple service components may be defined as part of the migrated service. If a component exposes a public interface for the client, communication configuration (ServiceDataConfig in Fig. 4) should be defined to enable load balancing.

Further, a slow network connection between the enterprise and cloud domain may reduce the benefit of service migration during a traffic peak. Actually, a big factor in migration latency (Fig. 3) was the connection between the enterprise node (Finland) and the Amazon EC2 instance (Ireland). It should also be noted that HAproxy may drop some incoming packets, when it is restarted/reconfigured.

One limitation regarding the proposal may be applicable use case scenarios. A main determinant is the sensitivity of information, which is associated with the migrated service and related data. There are risks involved in migrating sensitive service data to the cloud domain, which has to be taken into account. Thus, it is argued that the proposal may be more suitable for the migration of services that deal with non-sensitive information.

We experimented with one possible service, which simulated CPU-intensive processing in the server. Another service may include complex reading from a database server (e.g. a personalised news website). Alternatively, DNS load balancing may be applied between the domains to solve the problem, but it does not handle monitoring the load or QoS for the detection of traffic peaks. A service may also include increasing the amount of database writes to be served (e.g. a network monitoring use case), which may have similar related issues without an automated migration mechanism. Finally, a service and its related data may need to be transferred between the enterprise/cloud domains of different companies. If service migration was performed manually, the service would need to be manually instantiated in the new domain, and end-user client application may need to be modified to direct traffic towards the server of the new domain. This may be a laborious process, which may be eased with the proposed approach for automated service migration. In this case, the Coordinator and Load balancer may need to be executed in a third-party domain, which would be controlled based on agreements between the companies.

The main items for future work are related to improvement of the existing components of the architecture, and extensions with new components. The existing components needing further development include development of a simpler user interface for the Coordinator, increased level of automation in the Coordinator, implementation of a database for the storage of service state and service rules, and enhanced mapping between Nagios-based event data and user-defined rules. New development items include the cost-saving aspects of the solution, dynamic scaling of cloud instances, support for database scalability and security aspects.

10 Conclusion

This paper focused on automated migration, monitoring and load-balancing mechanisms for services between enterprise and cloud domains during traffic peaks. The focus was on the migration of service implementation, as opposed to migration of service as part of a virtual machine. The rationale was better expected performance in the migration process. The main research question focused on architecture and the realisation of the concept for service migration. First, a conceptual architecture was presented, which enabled automated migration and load balancing of services based on user-defined migration rules. Second, a prototype system based on open-source SW components was presented as a proof of the conceptual architecture. The answer to the research question is as follows from an architectural point of view: In the implementation, HAproxy enabled load balancing between clients and services. Nagios realised monitoring of services and hosts and Tomcat acted as a service container. Additionally, a Coordinator-entity encapsulated rules and logic for service migration and load balancing and the Worker-entity facilitated load balancing and service-deployment operations. Sequence diagrams illustrated the communication between architecture components, when service migration and load balancing was executed.

The sub-research questions were related to the performance of the developed mechanism. The developed migration mechanism was quite fast as service migration and load balancing between the enterprise domain and cloud domain could be executed in less than 200 ms (sub-research question 1a). The main sources of delay in the procedure were deployment of services at the service container (e.g. Tomcat), and reconfiguration for load balancing. Finally, performance for simulated end users was measured in a use case, where a CPU-intensive process was executed in the server, and a traffic peak was simulated. The average latency decreased from ∼600 to ∼200–300 ms, and the average throughput increased from ∼9 to ∼13 msg/s, when the results produced with the proposed mechanism were compared to the reference case (sub-research question 1b). This proves that the mechanism can be used to increase QoS for end users during traffic peaks.

Acknowledgments Jussi Liikka from VTT provided comments to the initial idea of the prototype system and chosen technologies. This work was partially funded by TEKES (the Finnish Funding Agency for Technology and Innovation) in ITEA2 (Information Technology for European Advancement) ACDC (Adaptive Content Delivery Cluster) and ICARE (Innovative Cloud Architecture for Entertainment) projects

References

1. Mell P, Grance T (2011) The NIST definition of cloud computing. NIST special publication 800–145. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Accessed 21 Feb 2013
2. Jones S (2005) Towards an acceptable definition of service. IEEE Softw 22:87–93. doi:10.1109/MS.2005.80
3. Yahia IGB et al. (2006) Service definition for next generation networks. In: Proceedings of the international conference on networking, international conference on systems and international conference on mobile communications and learning technologies
4. Chapman C et al. (2010) Software architecture definition for on-demand cloud provisioning. In: Proceedings of the ACM international symposium on high performance distributed computing (HPDC'10), pp 87–93
5. Vaquero L, Rodero-Merino L, Buyya R (2011) Dynamically scaling applications in the cloud. ACM SIGCOMM Comput Commun Rev 41:45–52. doi:10.1145/1925861.1925869
6. Bhadani A, Chaudhary S (2010) Performance evaluation of web servers using central load balancing over virtual machines on cloud. In: Proceedings of the 3rd annual ACM bangalore conference
7. Wee S, Liu H (2010) Client-side load balancer using cloud. In: Proceedings of the 25th symposium on applied computing, pp 399–405
8. HAproxy (2013) http://haproxy.1wt.eu/. Accessed 21 March 2013
9. Auto scaling Developer Guide (2013) http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/Welcome.html. Accessed 21 March 2013
10. Khajeh-Hosseini A, Greenwood D, Sommerville I (2010) Cloud migration: a case study of migrating an enterprise IT system to IaaS. In: Proceedings of the 3rd IEEE international conference on cloud computing, pp 450–457
11. Babar MA, Chauhan MA (2011) A tale of migration to cloud computing for sharing experiences and observations. In: Proceedings of the 2nd international workshop on software engineering for cloud computing, pp 50–56
12. Menzel M, Ranjan R (2012) CloudGenius: decision support for web server cloud migration. In: Proceedings of the 21st international conference on world wide web, pp 979–988
13. Tran V et al. (2011) Application migration to cloud: a taxonomy of critical factors. In: Proceedings of the 2nd international workshop on software engineering for cloud computing, pp 22–28
14. Kondo D et al. (2009) Cost-benefit analysis of cloud computing versus desktop grids. In: Proceedings of the IEEE international symposium on parallel and distributed processing, pp 1–12
15. Chalkiadaki M, Magoutis K (2012) Managing service performance in NoSQL distributed storage systems. In: Proceedings of the 7th workshop on middleware for next generation internet computing
16. Katsaros G, Kubert R, Gallizo G (2011) Building a service-oriented monitoring framework with REST and nagios. In: Proceedings of the IEEE international conference on services computing, pp 426–431
17. Rodero-Merino L et al (2010) From infrastructure delivery to service management in clouds. Future Gener Comput Syst 26:1226–1240. doi:10.1016/j.future.2010.02.013
18. Zhao Y, Huang W (2009) Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud. In: Proceedings of the 5th international joint conference on INC, IMS and IDC, pp 170–175
19. Vaquero LM, Daniel M, Galan F, Alcaraz-Calero JM (2012) Towards runtime reconfiguration of application control policies in the cloud. J Netw Syst Manag 20:489–512. doi:10.1007/s10922-012-9251-3
20. Hajjat M et al. (2010) Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In: Proceedings of the ACM SIGCOMM conference, pp 243–254
21. Bicer T, Chiu D, Agrawal G (2011) A framework for data-intensive computing with cloud bursting. In: Proceedings of the IEEE international conference on cluster computing, pp 169–177
22. Sen S, Wang J (2004) Analyzing peer-to-peer traffic across large networks. IEEE/ACM Trans Netw 12:219–232. doi:10.1109/TNET.2004.826277
23. Nagios Remote Plugin Executor (2013) http://exchange.nagios.org/directory/Addons/Monitoring-Agents/NRPE-2D-Nagios-Remote-Plugin-Executor/details. Accessed 21 March 2013
24. Nagios Core Documentation (2013) http://nagios.sourceforge.net/docs/nagioscore/3/en/toc.html. Accessed 21 March 2013
25. Pääkkönen P, Prokkola J, Lattunen A (2011) Instrumentation-based tool for latency measurements. In: Proceedings of the ICPE'11: WOSP/SIPEW international conference on performance engineering, pp 403–412
26. CPU Usage Limiter for Linux (2013) http://cpulimit.sourceforge.net/. Accessed 21 March 2013
27. Jetty Http server (2013) http://jetty.codehaus.org/jetty/. Accessed 21 March 2013