Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling

Stelios Sotiriadis, Nik Bessis, Cristiana Amza, Rajkumar Buyya

***Abstract*—Today, cloud computing applications are rapidly constructed by services belonging to different cloud providers and service owners. This work presents the inter-cloud elasticity framework, which focuses on cloud load balancing based on dynamic virtual machine reconfiguration when variations on load or on user requests volume are observed. We design a dynamic reconfiguration system, called inter-cloud load balancer (ICLB), that allows scaling up or down the virtual resources (thus providing automatized elasticity), by eliminating service downtimes and communication failures. It includes an inter-cloud load balancer for distributing incoming user HTTP traffic across multiple instances of inter-cloud applications and services and we perform dynamic reconfiguration of resources according to the real time requirements. The experimental analysis includes different topologies by showing how real-time traffic variation (using real world workloads) affects resource utilization and by achieving better resource usage in inter-cloud.**

***Index Terms*—Cloud Computing, Cloud Elasticity, Horizontal Scalability, Vertical Scalability, Cloud Load Balancing**

I. Introduction

CLOUD computing presents new business opportunities as an environment for deploying applications and services. Fundamentally, it provides an elastic infrastructure for utilizing virtual servers that are available anytime from everywhere. It includes three models, namely as Infrastructure as a Service (IaaS) that includes virtualized resources, Platform as a Service (PaaS) that includes an environment for developing applications and services and Software as a Service (SaaS) that includes on demand and pay as you go software [1]. Also, it includes platforms that offer resources such as hardware (CPU, memory, hard-disk), software and network on a bespoke manner. Today, cloud elasticity seems to be a vital cloud asset as it allows users to increase or decrease capacity of virtualized resources on demand, so pay for only what they use. Traditional businesses deploy applications or services in a cloud platform in the form of a unique virtual machine (VM) and make it available to users through a virtualized network of the cloud platform provider. Lately, another similar technology has been emerged, the so called metacloud that is a framework for providing clouds on demand as well as services called microservices i.e. the IBM microservices [2]. The metacloud is a cloud based platform (for example the system offered by Cisco[1]) for deploying and operating private clouds for global organizations.

Similar example is the future Internet concept that allow development of novel cloud applications combining different cloud services that might belong to different cloud providers (i.e. the FIWARE platform[2]). This is generally known as inter-cloud, that refers to multiple cloud providers [3] and is heterogeneous in terms of cloud platform architectures. Metacloud follows the inter-cloud concept thus it is comprised by a network of clouds that are hosted in different places. Here customers can easily host their own private clouds in the metacloud infrastructure for example Cisco metacloud is base exclusively on OpenStack[3].

Inter-cloud services also refer to cloud enablers that are part of a generic service oriented architecture (SOA), in which providers develop applications or services by selecting functionalities from different cloud platforms (i.e. authentication, data storage, data analysis etc.). There are many providers that offer such services and claim to offer an ”infinite” view of virtual resources. This refers to scalability as the ability of the system to accommodate larger loads and to elasticity as the ability to scale with loads dynamically. Cloud platforms offer both, however it does not allow automated VM reconfiguration (that refers to scaling up or down VM resources) based on real time usage. Consequently, as cloud VMs resource usage requirements change dynamically, the initial configuration could lead to service performance degradation, for instance if demand increases significantly.

To deal with these, this work focuses on the dynamic VM reconfiguration in terms or horizontal and vertical elasticity. We define the terms horizontal elasticity that refers to the creation of new VMs and vertical elasticity that refers to the resizing of existing VMs. To address such issues, cloud providers offer elastic load balancers that manage incoming traffic without disrupting the flow of information. In particular, platforms such as OpenStack, VMWare VCloud[4] etc. provide load balancers to achieve stability, adaptability and optimal usage of client workloads. However, in this work we are motivated by the inter-cloud concept that makes elasticity a challenging task. In particular, the elasticity factor, which targets dynamic adaptation to workload changes, is a crucial aspect to be explored. The problem that this work is focusing is timely and comes from the transition from monolithic web applications to distributed microservices where elasticity

[1]http://www.cisco.com/c/en/us/products/cloud-systems-management/

[2]https://www.fiware.org/

[3]https://www.openstack.org

[4]http://www.vmware.com/products/vcloud-suite.html

becomes a hurdle (i.e. how to manage a microservice, scale it up or down automatically and reconfigure it according to real time usage).

We are motivated by the works of (a) [4] that describes the Google Borg as a cloud scheduler, (b) [5] that describes the Kubernetes that is an environment for building distributed applications from containers [6], (c) [2] that presents the IBM microservices that break up large applications to easily to manage, maintain and operate modules, (d) [7] that describe Elastack that provides automated monitoring and adaptation in OpenStack platforms and (e) [8] that describe challenges towards automated cloud application elasticity. The innovation of this work is based on the dynamic elasticity involving running already deployed VMs rather containers that are executed in different cloud platforms, thus we focus on exploring the best method for VM reconfiguration. This includes the dynamic VM reconfiguration that is affected by the variation of service usage in real-time. Current approaches usually refer to services from the same cloud provider or perform elasticity by launching new instances in the provider-side. Having said that, the contributions of the inter-cloud load balancer (ICLBICLB) framework include the following:

(C.I) To provide an elastic inter-cloud load balancer for applications or services composed by microservices. That allows cloud applications or services to scale up/down in accordance to real time resource usage and traffic.

(C.II) To allow dynamic and automated VM capacity reconfiguration when increased workloads (i.e. HTTP traffic) or resource utilization levels (i.e. VM disk usage) are observed.

(C.III) To compare horizontal versus vertical reconfiguration for different scenarios in order to conclude to optimal usage for each of which. We use real world systems to demonstrate our solution such as the OpenStack cloud platform and the Apache Cassandra [9] as a deployed system and the the Yahoo! Cloud System Benchmark (YCSB) [10] as the real world workload for stressing the system and support our research argument for two cases including (a) for increasing traffic and (b) for increasing resource usage.

Regarding C.I, the work proposes a new load balancing layer positioned on top of the cloud platforms. The aim is to move existing solutions (e.g.OpenStack Elastic Load Balancer) a step forward, by allowing management of application and cross-utilization of services belonging to different providers and service owners by decoupling load balancers from the providers. To achieve this we present an experimental study that includes real world workload testing for an Cassandra VM cluster. In particular, we setup a cluster and we test it using YCSB workload to stress system performance. Thus, we used two ways of triggering autoscaling, a) based on the HTTP traffic coming from the YCSB (that is deployed in an external VM) and b) based on the monitoring of the Cassandra resources that is an enterprise-grade search engine [11]. Having said that, we tested the system in different configurations including Cassandra cluster that is deployed in OpenStack and VMWare VCloud.

Regarding C.II, VMs are monitored constantly and capacity is increased or decreased in relation to a VM reconfiguration strategy. The aim is to eliminate downtimes without affecting the overall flow of information. This will allow cloud VMs either to be resized or to be re-instantiated in the same or a different cloud provider with zero loss in communication. The experimental analysis presents various cases in which ICLB could be applied with success. Next, Section II presents a detailed discussion of the literature and tools along with the dependencies and the contribution of our work on and to existing approaches and tools. In Section III we present a discussion of the design issues of ICLB modules, in Section IV the performance evaluation and in Section VI the conclusion and future research directions.

II. Motivation and Innovation

In this work we focus on solving the problem of VM autoscaling by employing load balancers to add VM clones for sharing load or to resize VM size.

Cloud providers claim to offer high availability services with an ”infinite” view of resources. This includes (a) scalability as the ability of the system to accommodate larger loads and (b) elasticity as the ability to cope with loads dynamically. However, VMs resource usage requirements change dynamically, so the initial VM configuration is static and could lead to service performance degradation, for instance if demand increases significantly. Here, we focus on the cloud elasticity that is the degree in which a system can provision/deprovision resources automatically to cope with high demands. Our goal is to reconfigure a VM size according to the resource usage and/or incoming traffic in an automatized way. To achieve fault tolerance and keep VM connection alive, we run experiments in a load balancing environment where VMs are placed under a load balancer VM (the inter-cloud load balancer) that distributes HTTP traffic fairly according to different algorithms (i.e. the round robin). We analyze real time resource utilization levels and we trigger horizontal (increase/decrease VM size) or vertical (cloning VM through replication) elasticity to avoid overloads. Our motivation is based on Figure 1 that experimentally demonstrates a real world system of an Cassandra head node performance that is deployed in an OpenStack system and is under stress.

To achieve a realistic scenario, we used the real world workload of YCSB[5] that is a framework with a common set of workloads for evaluating the performance of different ”key-value” and ”cloud” serving stores. Based on that, we run the YCSB workload in order to explore the performance of the Cassandra system. In detail, we executed YCSB workload to stress system performance with (a) 250 thousand records (normal execution) and 10 million records (extremely heave execution) by running in 5 threads in a small size VM (2 GB RAM, 20 HD disk, 1 CPU Core) and we observed that the VM disk usage is constantly increased meaning at some point the VM will be overloaded and will fail execution of tasks. We

[5]https://github.com/brianfrankcooper/YCSB

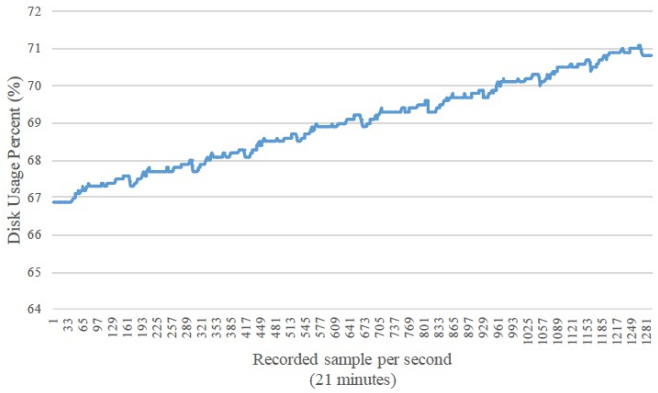

Fig. 1. YCSB Cassandra workload overload execution in small size VM for 250 thousand insert and update records

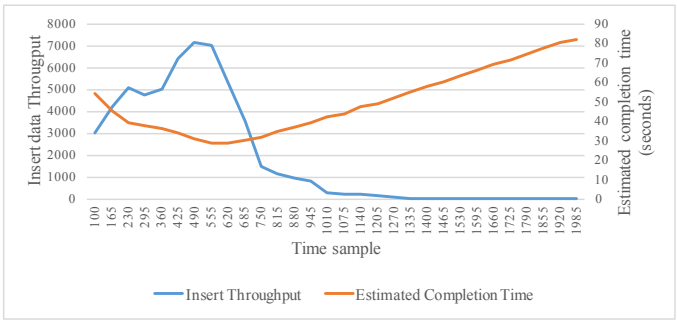

Fig. 2. YCSB Cassandra workload insert throughput data failure and estimation completion time for small size VM (10 million tasks for insert and update records)

can observe that the disk usage percentage is increased from 67% to 71% percent. Based on this trend (and its repeatable pattern that is visually apparent) we can expect (for a heavy workload) that at a certain point the disk usage will reach 100% and the workload execution will fail. So, we aim to avoid the Cassandra cluster overload point in order to trigger VM reconfiguration and achieve load relief either in a clone or in a resized VM.

Another exampe is Figure 2 that demonstrates an overutilized Cassandra cluster. In particular, we stress the system by selecting a small size VM in OpenStack and we measure its performance when we run 10 million records. We observed that the VM reaches an overload point of 100% (around after 40 minutes of execution time) and continues at the same levels. At that time we can assume that the overload could cause errors in the workload execution since there is no free disk space. To validate this assumption, we measured the Cassandra insert data throughput and estimated completion time and we demonstrate results in Figure 2. We can observe that around time instance 600 the throughput started to decrease and until time instance 1200 the throughput drops almost to zero. We compared these values with the disk usage and we concluded that this is the time moment when the Cassandra VM disk usage started to become overloaded (the time instance that it reaches 100%). Similarly, estimated completion time started to increase significantly, meaning that the workload execution will be delayed and eventually could not be executed at all (for example after 1235 time instance the estimation completion time is increased dramatically in a steady increasing trend).

So we analyzed resource usage features (i.e. cpu percentage, memory percentage, disk usage percentage etc.) and traffic values in order to trigger autoscaling and avoid overloading. The proposed system is not the first to focus on these issues, but to the best of our knowledge is one of the few that addresses the comparison of horizontal and vertical scaling in inter-clouds in order to conclude to the best solution for different scenarios. In addition, we demonstrate that based on a real world case (of the previous experiment) even state of the art cloud applications (like Cassandra) suffer from automated scaling and automated monitoring. Thus, ”outsourcing” user demands to clone VMs based on inter-cloud load balancers could offer a sophisticated and performance efficient solution. We perform an experimental study of OpenStack and VMWare VCloud platforms to illustrate the effectiveness of our system.

III. Related Work

This section presents a discussion of the related works referring to cloud service elasticity and scalability. We classify elasticity into vertical, which refers to the changing of VM size, and horizontal, which refers to the creation of new VMs. In [12] the authors present a horizontal elasticity solution by allowing VM allocation (by adding or removing VMs) in the cloud. They present a simulation analysis that demonstrates a minimization of SLA violation, by focusing on horizontal elasticity. In [13] the authors monitor VM resources and develop an architecture for reducing the provision overhead. According to [14], the approach does not offer scale-down mechanisms.

In [14], the authors focus on the use of a private cloud environment in order to improve the High Performance Computing (HPC) research infrastructure. Specifically, they have implemented an HPC job scheduler to improve the utilization of the cloud resources. This work, based on OpenStack, focuses on launching new nodes and not on migration or resizing cloud VMs. In [15] the work focuses on the need for integration of QoS and SLA requirements with the cloud and automated dynamic elasticity of the cloud for SLA management and it is a theoretical discussion of research directions. The work presented in [16] focuses on elasticity as the ability of a cloud to add and remove instances in an automated way. The solution is called Elastack, which is a monitoring and adaptive system. The authors claim that the solution is generic and could be applied to existing IaaS frameworks. It is a promising work that is OpenStack oriented.

In [8] the authors present a study on dynamic scaling of applications in the cloud. The work shows efforts at the edge of state-of-the-art technology on cloud elasticity. They present new challenges in the areas of server, network and platform scalability. In [17] the authors demonstrate an automated system for elastic resource scaling of multi-tenant clouds. It is called CloudScale and achieves adaptive resource allocation with no need to know a priori. According to [16], CloudScale focuses on vertical instance scaling (that is it is said to act on the instance itself rather launching new ones). The authors

in [18] present a system called Kaleidoscope that allows cloning of VM instances when demand is increased by copying the complete or partial state of the original instance. Kaleidoscope does not launch new instances. According to [16], this approach requires adapting and integrating within the cloud infrastructure. Also, to be effective it requires installation on all instances.

In [19] the authors present an architecture for an IaaS cloud to allow dynamic resource allocation. They develop a system called Kingfisher that contains components for replication and migration using an integer linear program in order to optimize cost, and implement an OpenNebula extension for load balancing when load is changing. In [20] the authors present an architecture for an IaaS cloud to allow dynamic resource allocation. They develop components for VM scheduling with management objectives for replication and migration and implement an OpenStack extension for load balancing. In [21] the Amazon EC2 Auto Scaling is designed to launch or terminate EC2 instances automatically according to user-defined policies, schedules, and health checks. The solution allows management of VM resources based on predictable and anticipated load changes. Further, the approach uses the Amazon CloudWatch (for notifications and alarms) and the Elastic Load Balancing (for distribution of traffic among various instances) in the autoscaling groups.

Further, [22] provides the Amazon Elastic Load Balancing (ELB) to automate the process of incoming web traffic between different Amazon EC2 instances. By using ELB, the users add and remove instances according to the need of the traffic without disrupting the flow of information. In the case that a VM fails the ELB sends the request to other instances that have been previously configured in the ELB. In [23] the authors investigate the feasibility of dynamic cloud scaling, by focusing on Cloudify telco services. They focus on the migration of processes among peer servers in a transparent way for pro-active resource provisioning based on call load forecasting. The work in [24] presents vertical elasticity for applications with dynamic memory requirements when running on a generalized virtualized environment. The solution offers the ability to scale the VM memory dynamically using memory ballooning provided by the KVM hypervisor.

In [25], the authors show a data centre architecture based on virtual machine monitors to reduce provisioning overheads. They also employ a combination of predictive and reactive methods to determine when to provision resources. [26] discusses the OpenStack load balancing solution of VM traffic. It offers an API to allow distributing requests between VMs similar to Amazon ELB. In [27], the authors present the soCloud that is a PaaS component in multiple clouds that uses load balancers to "switch from one application instance to another" if there are failures. The work of [28] presents the MODAClouds that offers a system for migrations among multiple clouds that reacts to performance reconfigurations. MODAClouds provide innovative features such as avoiding vendor lock-in problems supporting the development of Cloud enabled Future Internet applications and provide quality assurance during the application life-cycle and support migration from Cloud to Cloud when needed. In [5], authors present the Kubernetes that is an "open-source system for automating deployment and scaling" using containerized applications to allow scaling applications on the fly and optimization of resources when needed. It is particularly focused on launching containers that can be horizontally scale-able and composed by microservices. Other works as in [29], that focus in interoperation and job executions in terms of large scale systems does not consider scaling, thus are out of the scope of our study.

We are motivated by the solutions of [15], [16], [27], [5] and [21] that aim to contribute from the perspective of an inter-cloud application. Specifically, we focused on real-time workload analysis such as of the cloud applications traffic when increased usage is observed, as in [22] and [26]. To the best of our knowledge, our work is different from the literature in the aspects of inter-cloud load balancing (placed on top of clouds) and focuses in already running VMs rather than in containers.

IV. Inter-Cloud Elasticity Framework Design

Developers build innovative cloud applications using services from different owners that are deployed in different cloud platforms. Figure 3 shows an example of an application that utilizes inter-cloud services, where Figure 3(a) is the traditional deployment (based on a single provider) and Figure 3(b) is the inter-cloud deployment where an application (App) is composed by services of different providers. In the second case elasticity refers to both VMs of application and services, however the complexity is increased due to the interoperability aspects of these modules. This refers to the ability to build systems from reusable components that offer out of the box functionalities.

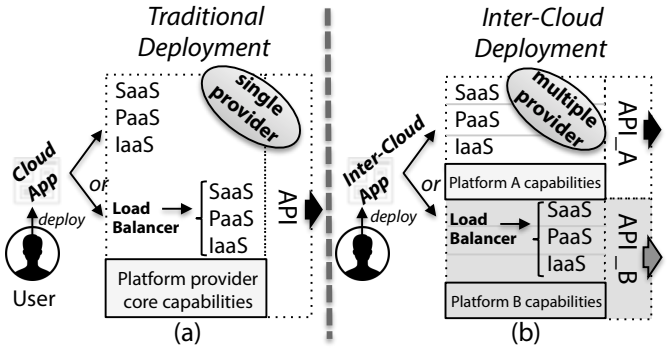

Fig. 3. Inter-Cloud distributed service utilization: (a) Cloud application deployment in a single provider and (b) Inter-Cloud application deployment in multiple providers

The following sections discuss (A) the ICLB framework, (B) its architecture, (C) the analysis of the system configuration and its processes.

A. Inter-Cloud Elasticity Framework

ICLB approaches *cloud and inter-cloud elasticity* from the perspective of distributed and interoperable services [30], in order to allow management of the elastic reconfiguration of VMs without affecting application uptime and by improving

performance. Its key layers are (i) inter-cloud layer, (ii) elasticity layer, (iii) load balancing layer, (iv) monitoring layer, (v) elastic threshold management, (vi) security and (vi) portability, which are presented below.

(i) Inter-Cloud Layer: It contains the inter-cloud communication layer that allows interactions among different cloud platforms. It allows mediation for connecting clouds utilizing APIs and platform interfaces such as OpenStack[6], VMWareVCloud and others presented in [31]. Essential mechanisms include inter-cloud authentication, collection of data related to resource usage, instantiation and deployment of services (in a remote manner) and others. In [32] and [31] we developed a prototype composed not only from heterogeneous cloud platforms but also from independent (in terms of functionality) cloud services. These are developed by different cloud service providers and offered as open source Software-as-a-Service (SaaS). A vital component of this layer is the Inter-Cloud Mediation Service, specified in [33], that uses SaaS based microservices. Examples are the publish subscribe context broker for registrations and subscriptions to services and a complex event processing engine for event management. A detailed presentation of this service is presented in [31] along with the inter-cloud VM migration mechanism.

(ii) Elasticity Layer: It features the dynamic properties for elastic reconfiguration of cloud service VMs. The layer includes two different modes of elasticity: vertical and horizontal. Todays cloud platform providers offer these in the form of launching new or resizing existing VMs instances. The layer focuses on elasticity as the ability of resources to scale out (either vertical or horizontally) so to cope with loads dynamically.

(iii) Load balancing Layer: Most cloud providers offer sophisticated load balancing mechanisms, for instance the OpenStack Load Balancing-as-a-Service [26], the Amazon ELB [22] and others. These are the default mechanisms that users could easily deploy and configure using the platforms dashboard system. The load-balancers could handle traffic by distributing requests to different clone instances. A widely used solution is the open-source proxy servers (i.e the NGINX[7]) for implementing complex load balancing in terms of different algorithms for traffic distribution (i.e round-robin) and dynamic automation of the load balancer based on workload reconfigurations. The inter-cloud load balancing layer involves utilization of multiload balancers. This is because the elastic reconfiguration should be independent of the local load balancers that a user could setup. The objective of the load balancer is to split the HTTP traffic among VMs that are belonging to same or different datacenters. In particular, we perform experiments for cases where there are two levels of balancers, the inter-cloud load balancer that is responsible for balancing HTTP load distribution among different clouds and the local load balancer that could be provided by the cloud it self, for example Amazon EC2 uses the Amazon Elastic load balancing.

(iv) Monitoring Layer: It is a vital requirement to observe real-time resource usage so to allow adaptive decision-making during run-time. This layer is responsible for data collection directly from the running instances and comprises high performance real-time observation of VM usage for the purpose of triggering elastic reconfiguration. Different monitoring server solutions like Nagios[8] and Zabbix[9] offer flexible REST APIs. The assumption is that each VM is monitored constantly and properties like CPU, memory, etc. are evaluated at run-time. We utilize a real time monitoring system that collects resource usage based on an interval. Data is collected and analyzed during runtime (such as CPU, memory and disk usage statistics). More details are presented in the experimentation section.

(v) Elastic Threshold Management Layer: It implements the dynamic and real time cost management function that defines the elastic reconfiguration thresholds for services. The layer provides functions to calculate profits or overheads of a service owner or provider by analyzing HTTP traffic.

(vi) Security Layer: Cloud services are usually deployed as web applications: prevention of attacks that could increase traffic between application and services is of vital importance. The layer establishes an external security layer to increase security and discover attacks.

(vii) Portability Layer: It defines the process for systematizing application and service deployment among different cloud platforms. The layer defines solutions for portability by automating the deployment of self-sufficient containers (i.e Docker[10]). In [32] and [31] we presented inter-cloud IaaS portability solutions that are used here.

B. Architecture of inter-cloud based on ICLB framework

ICLB framework, demonstrated in Figure 4, targets the automation of the inter-cloud elasticity and performs dynamic VM reconfiguration based on the variation of HTTP traffic.

In detail, Figure 4 demonstrates the interactions among the various modules, the flow of traffic initiated by a user to the ICLB component, and the configuration requirements of the developer that deploys the service. ICLB differentiates three actors: the application/service owner, named as App owner, the user that forwards traffic to the service (i.e by making HTTP requests to the App) and the 3rd party service owners that are independent of the cloud platforms. The next section presents the structure of the ICLB internal processes regarding initialization and during traffic.

1) Initialization of the ICLB architecture: The ICLB initialization process defines processes prior to the publication of the service. The following describes the configuration of ICLB modules including initialization and management of elasticity.

[6]OpenStack API: http://developer.openstack.org
[7]NGINX: http://nginx.org
[8]Nagios: www.nagios.org
[9]Zabbix: www.zabbix.com
[10]Docker: https://www.docker.com

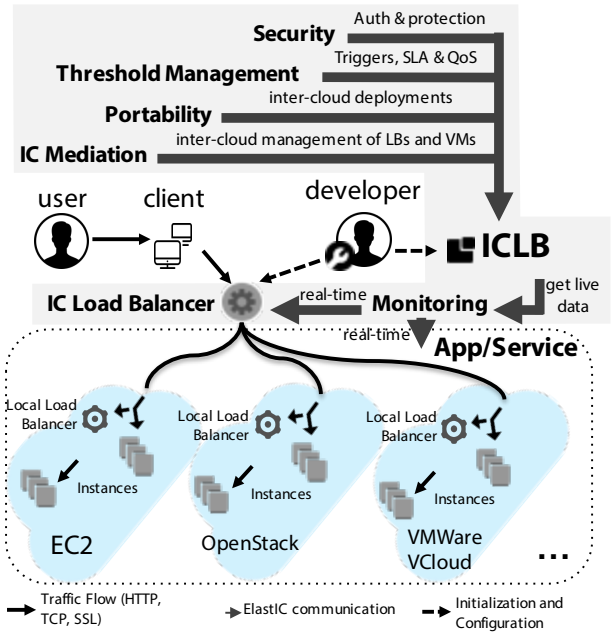

Fig. 4. Inter-Cloud Elastic Service Architecture

(i) The assumption is that the service owner actor has already implemented an App/service using available inter-cloud platform services. By using the *Portability module* the actor creates App/services in proprietary containers that are portable, thus independent of cloud providers and infrastructure constraints. These are also ready for deployment (i.e available in the Docker) to any cloud provider. The owner deploys the App/service to the preferred cloud locations (i.e different clouds). During deployment the service makes the initial configuration of VM resources (CPU, memory, HD) and virtual network settings. At this point, the App is composed by other services that are all identified by a unique URI (that is the service endpoint IP). The owner uses ICLB to define two categories of service as follows:

- The brokers are general purpose services and do not store permanent data. These are in case of service downtime (i.e while cloned, resized and re-launched) and will not affect the App/service functionality (i.e missing data because of a failure).
- The storage services for permanent data storage that constitute data redundancy mechanisms.

(ii) The IC load balancer is configured by the ICLB components in order to allow traffic distribution in the deployed App. At this point the assumption is that the owner instantiates at least two VMs that host the App in the same or different clouds. During initialization, the owner configures any local load balancers of the local providers by creating, if needed, replicated instances of the local services. The owner configures the IC load balancer to distribute traffic among service endpoints based on HTTP load balancer algorithms such as round-robin, least-connected or IP-hash.

(iii) Using ICLB, the service owner configures the monitoring aspects of the App/service. In particular, the monitoring module allows live data capturing based on an interval (i.e data collection every 5 seconds). The monitoring component requires installation and configuration within the VMs and will offer a centralized control point for storing data including configuration and performance data. The real-time time monitoring will allow automated actions of the ICLB components, i.e restart a VM using the IC mediation component.

(iv) The IC mediation component is based on a service-centric architecture as in [31]. This allows remote connection to the various clouds interacting through REST-Ful cloud services. Key functionalities include identity management and cloud registry (secure authentication to connect to clouds), subscription service (data retrieval regarding services virtual resources and context management (subscribe, unsubscribe, create, update and register context), complex event management (interfaces to the inter-cloud mediation for defining rules and patterns to react on certain event flows) and others described in [31]. The mediation works independently of the ICLB component, yet it uses its interfaces to get and post events to the clouds. For example, the ICLB module, through the mediation service will send a request for restarting a VM. Major aspects of elasticity including horizontal and vertical elasticity are defined in this component.

(v) The cost management component provides generalized functions and algorithms for real-time data handling. ICLB uses it to define triggers for inter-cloud elasticity. The owner initializes this component by selecting one of the preferred real-time resource usage assessment algorithms. These are simple rules: for instance, when traffic increases 50% a new instance will be re-launched or resized and ICLB will notify the IC load balancer.

(vi) The security component includes the application and service authentication point that is implemented within the IC mediation. This also includes an external security layer that increases security for detecting attacks in order to prevent increasing traffic. Since ICLB features cloud elasticity in terms of HTTP traffic monitoring solutions like ModSecurity[11], an Apache module for real-time analysis with few changes to existing infrastructure. The owner defines the security protection rule.

(vii) The ICLB component is initialized according to the requirements of the owner. The component could perform horizontal and/or vertical elasticity based on rules and real-time data captured by the monitoring and security within the inter-cloud system.

2) Elasticity processes of the ICLB service: ICLB targets real-time traffic management of cloud applications and services. The following demonstrates the interaction among processes during incoming HTTP traffic. Firstly, the users send HTTP requests to the IC load balancer, which in turn forwards the traffic to the local load balancers or the VMs (depending on the initial deployment). The IC load balancer is deployed as a cloud service within the inter-cloud system and monitored

[11]ModSecurity: www.modsecurity.org

in real time. Also, it utilizes an HTTP load balancer algorithm (pre-defined by the owner) for traffic distribution. The volume of the traffic is monitored by the ICLB component every interval, while the IC load balancer resource usage is captured by the monitoring component. ICLB is based on real-time monitoring; it defines the automation mechanism according to the configuration parameters of the owner. In case of increased or decreased traffic, the cost management rules trigger actions in ICLB, which sends events to interacted components. These are as follows.

(i) ICLB monitors resource usage and HTTP traffic using the monitoring component (based on interval measurements). In particular, we set the monitoring threshold to every second for the whole set of the experimental study. Data is collected and analyzed on the fly.
(ii) ICLB is configured for triggering events (according to a decision-making process) based on rules and/or patterns coming from the cost management component. If a rule is met, a sequence of events is triggered. The security component is utilized as a web firewall to classify incoming traffic to healthy and malicious requests.
(iii) ICLB, through the IC mediation, sends a request for performing autoscaling (vertical or horizontal elasticity to increase or decrease resources). In the horizontal case it also sends the URI of the new deployed instance that has been generated using the portability module.
(iv) Events coming to the IC mediation assigned as actions for clouds (i.e launch a new instance). In case of horizontal elasticity a response is generated to the ICLB to update the IP of the new instance.
(v) For horizontal elasticity, ICLB gets the IP of the new instance and performs a sanity check to know when the service is up; at this point it updates the IC load balancer list of IPs without dropping the connection. For vertical elasticity the IC load balancer sends requests to the instance automatically when it is up.
(vi) The IC load balancer continues to distribute the traffic according to a load balancer algorithm (that is predefined in the initialization stage), and from this moment it forwards HTTP requests to the new clone instance (horizontal) or the resized instance (vertical).
(vii) The ICLB continues monitoring. In case of decreasing traffic it repeats the same process to drop/shutoff an instance; however, in this case it first updates the system to delete the instance IP, if required, by the owner (i.e for cases where a cloud client would like to release IPs to decrease costs). After, it sends the request to the IC mediation component for removing the instance IP from the list. In any other case, the IC load balancer keeps the IP in the list; however, it does not send traffic until they become active again.

To demonstrate the above interactions, we present a simple example that involves the monitoring of the traffic for increased workloads by 50% for more than 1 hour. Foe example, in this case, the ICLB framework will resize an instance of the inter-cloud from a small to medium size (i.e from 1 CPU, 2 GB RAM, 20 GB HD to 2 CPUs, 4 GB RAM, 40 GB HD). A more complex case is the generation of a new clone instance of the selected medium size. In this case the IP of the new VM will be registered into the IC load balancer and the traffic will be spread among the pool of cloned VMs.

C. Analysis of the ICLB configurations

The App/services owner configures the load balancer by including the addresses of the VMs (IPs and ports). The default setup for the load balancer is set to the round robin algorithm that distributes the HTTP traffic fairly. With regards to the service configuration, the App/services owner/developer configures VMs (that belong to the inter-cloud system) and installs components (such as security, monitoring etc.) using the portability deployment module, then selects whether to use a local load balancer (Local LB) or not. The final service endpoints are forwarded to the ICLB that configures the inter-cloud load balancer (IC-LB), the monitoring (to get real-time data), the IC mediation (i.e to get authentication tokens from clouds) and the cost management functions. Finally, ICLB is set to serve incoming user traffic. We further detail the vertical and horizontal autoscaling configuration as follows:

- Vertical autoscaling refers to the reconfiguration of the hardware resources of a running VM including virtual CPU number, disk and RAM size. Various cloud providers include different size flavors, thus the threshold for horizontal autoscaling triggers an action for changing the cloud platform flavor that in turn upscales or downscales according to dynamic user needs. Vertical autoscaling creates App/service downtime, however it keeps the same endpoint URI.
- Horizontal autoscaling refers to the creation of new VMs that are clones of the selected instance, utilizing a process of creating a new image by assigning the same resources (flavor). After the creation, the cloned VM is executed independently of the initial VM. The Horizontal autoscaling does not create any downtime of the App/service as the image creation process does not stop VM execution, yet the new cloned VM gets a new endpoint URI.

The load balancers are organized by the ICLB service, so in the horizontal case the load balancer forwards all the traffic to the running instance(s) while in the vertical it is configured with the new endpoint.

V. Performance Evaluation

This section presents the performance analysis of the ICLB framework. It includes (a) the experimental setup, (b) the benchmark analysis, (c) the comparison of horizontal and vertical autoscaling, (d) the inter-cloud load balancing benchmarks, the inter-cloud load balancing scenarios based on (e) HTTP traffic volume and (f) resource usage and (g) the inter-cloud load balancing based on various service layers (for example layers of microservices that integrate a cloud application).

A. Experimental setup

We developed the experiments using two infrastructures based on (a) an OpenStack and (b) a VMWare VCloud platforms. The OpenStack system is comprised by 11 nodes (1 head and 10 compute) with total 128 Cores, 284 GB RAM and 12 TB HDD and has been deployed as an experimental infrastructure while the VMWare is from a commercial provider. We deployed different VMs following the default sizes of OpenStack that we duplicated in VMWare. Using both systems we aim to demonstrate the inter-cloud notion.

The experimental setup includes the utilization of a load balancing solution deployed in OpenStack, while the VMs are deployed in OpenStack and VMWare. We perform horizontal and vertical autoscaling in both platforms using their RESTnAPIs. The fundamental objective is to keep connection of VMs alive during each ICLB process. To demonstrate a real world scenario, we deployed Cassandra and we run different experiments to explore the performance of ICLB framework. Cassandra is an open source search engine using schema free JSON documents and provides an HTTP web interface. We used the the Yahoo! Cloud System Benchmark (YCSB) workload in order to stress the CPU utilization for different workloads and we set threshold values to trigger VM reconfiguration (i.e. when CPU utilzation is higher than a specific CPU usage percentage). For the whole set of the experiments we use the YCSB core workload, that has a mix of 50/50 reads and writes.

The YCBS workload allows us to test our system using a real world workload benchmark that increases CPU utilization. Figure 5 demonstrates a simple example of this scenario where it is shown that the HTTP traffic is forwarding from YCSB node (that in our case simulates a user) to the Cassandra cluster. Here the client forwards requests to the Cassandra node using the inter-cloud load balancer (ICLB), by sending traffic to N_1 while N_2 is offline for the moment. A monitoring service that is preinstalled in all nodes and monitors different features including percentage of CPU usage, Memory, HD, IOs etc.

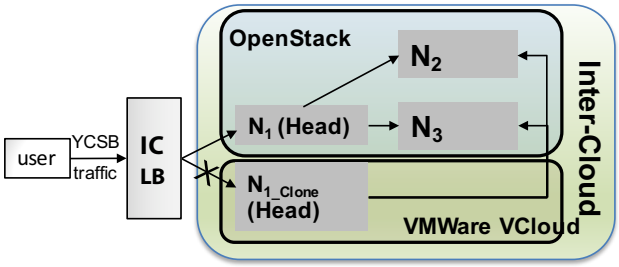

Fig. 5. HTTP traffic forwarding from user that sends YCSB workload to the Cassandra cluster

In detail, the traffic is forwarding to the N_1 that is the head node of the Elastisearch cluster consisting of three nodes (N_1, N_2, N_3). The assumption is that when the traffic overcomes a threshold a new clone VM ($N_{1_{Clone}}$) will be created (for example from size small to medium) to handle the increasing demands and replace N_1. In our case, we set the default parameter for autoscaling the HTTP requests, however this can be extended to include CPU utilzation and other. In more detail, the monitoring service uses the psutil library[12] and is responsible for collecting monitoring data per second, and sends it to the ICLB along with the identification number of the node.

Based on this setup, we perform the following experiments.

(A) Firstly, we perform a benchmark analysis of horizontal and vertical autoscaling including various tests of cloud VMs sizes (named as flavors). For each experiment we run 10 tests and we measure the average of these executions. The experimental setup includes load balancing in the Cassandra cluster.

(B) The autoscaling of a single VM that utilizes other services belonging to different clouds. The aim is to explore if there are any downtimes in terms of horizontal autoscaling and also to measure the times needed for vertical autoscaling (that is the time to create new instances).

(C) The benchmark analysis of the ICLB that distributes traffic into the Cassandra cluster (comprised by two VMs) in order to compare the total execution time, requests served and transfer rates. In addition, we compare round robin, IP hashing and least connected load balancing algorithms to explore their performance for various levels of concurrent requests.

(D) The autoscaling of the Cassandra in a topology in which the ICLB component distributes requests among two identical VMs for a large number of requests that simulate incoming traffic using the YCSB workload. The aim is to compare horizontal and vertical autoscaling and to explore an optimization scheme for further gains.

(E) The ICLB scenario in which the ICLB component monitors resource utilization levels and accordingly triggers autoscaling based on CPU usage.

(F) A complex inter-cloud heterogenous topology in which the monitoring of CPU levels triggers autoscaling.

B. Benchmark Analysis of Horizontal and Vertical Autoscaling

This section presents the analysis of autoscaling of the Cassandra (that in our example demonstrates the cloud application/system) based on the YCSB workload. In particular, Table I shows the measurements of the horizontal autoscaling that includes three different VM configurations (known also as flavors). We deploy Cassandra and we configure flavors, where flavor *f1* is 1 CPU, 1 GB memory and 1 TB HD, *f2* is 1 CPU, 2 GB memory and 20 TB disk and *f3* is 2 CPU, 4 GB memory and 40 GB HD. We divided the VM creation phase into two stages; the *Response* and the *Active server*. Specifically, the *Response* is the time needed for the request to send a VM creation to the OpenStack API and get the response (shown as status 202 Accepted) information from the virtual server.

It should be mentioned that the ICLB sends API calls using the REST interfaces of the platforms, so the response time is the time needed for a call to be executed (i.e. send an HTTP GET/POST) and return the successive HTTP response. The response data includes VM identification, endpoint IP etc. The

[12]https://pythonhosted.org/psutil/

Active server is the time needed for the server to acquire status *Active* in the OpenStack system. We execute a sequence of 10 requests for each of the three flavors, thus the total experiment includes 30 executions with 60 measurements executed in a total of 50 minutes. The total creation time is the sum of the two aforementioned times. With regards to the network aspects we follow the default network configurations of the cloud platforms (for example OpenStack uses the Neutron service[13] that provide networking as a service and are responsible for creating the virtual interfaces for the OpenStack system). According to Table I we calculate that the average total time for Cassandra VM creation for *f1* is 51.44 seconds, for *f2* 40.47 seconds and for *f3* is 38.36 seconds.

TABLE I
HORIZONTAL AUTOSCALING BENCHMARK ANALYSIS

Request Number	1	2	3	4	5	6	7	8	9	10
f1: Response	29.92	29.24	13.85	17.085	23.14	20.36	21.79	30.39	13.57	10.81
f1: Active server	28.91	44.19	24.53	36.16	23.47	30.63	29.91	26.92	25.19	34.3
f2: Response	8.67	25.09	9.69	16.85	23.83	15.86	20.36	19.89	21.85	12.24
f2: Active server	19.74	23.86	22.13	32.1	28.08	33.44	16.96	18.87	14.86	20.4
f3: Response	10.2	19.17	15.81	14.33	10.91	13.78	9.99	15.56	17.31	18.56
f3: Active server	31.31	20.5	23.97	16.7	26.29	14.77	23.66	20.66	32.146	27.93

Based on these we conclude the following findings.

- The average time for the VM creation for all cases is less than 44 seconds.
- As the size of the requested VM is increased, the time for VM creation is decreased slightly, thus bigger VM sizes does not affect performance significantly.
- The average time for a VM creation, independent of its size, is 43 seconds.
- The VM is not yet accessible as the system has not yet assigned a floating IP to it. Based on a basic experiment in an OpenStack platform we concluded that the IP allocation and assignment process would increase the average creation time by an average of 6 seconds, so the average VM creation time becomes 48.55 seconds.

In Figure 6 we visualize the horizontal autoscaling values of Table I in a clustered column diagram to highlight the above findings.

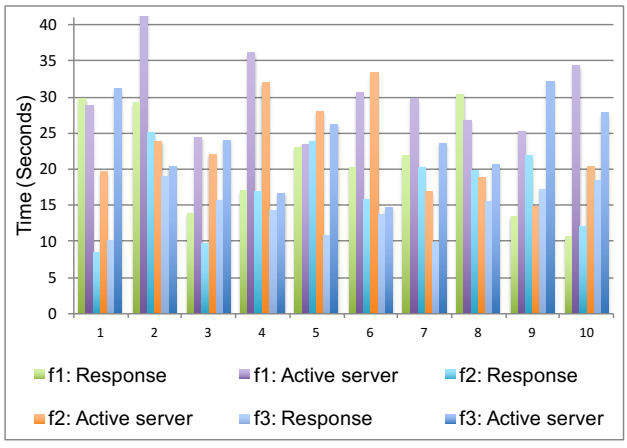

Fig. 6. Benchmarking: Horizontal autoscaling of a cloud VM

[13]https://wiki.openstack.org/wiki/Neutron

Table II demonstrates the vertical autoscaling values (resizing of a VM). In this case we execute a simple upscaling resizing in an OpenStack system using the API and by sending HTTP requests. It includes the upscaling from flavor *f1* to *f2*, *f2* to *f3* and *f3* to *f4*. The new flavor, which is not in the previous horizontal case, is *f4* and includes 4 CPUs, 8 GB RAM and 80 GB HD. We executed 10 requests (as previously) and we measured two parameters that divide the overall upscaling process into the altering of *flavor* to *flavor Uptime* and *flavor* to *flavor Downtime*. The first reflects the resizing *preparation time* where the instance is online and the second the *execution time* where the instance is offline.

TABLE II
VERTICAL AUTOSCALING BENCHMARK ANALYSIS

Request Number	1	2	3	4	5	6	7	8	9	10
f1-f2: Uptime	26.74	24.31	28.57	28.24	29.26	26.97	23.20	26.32	28.59	25.27
f1-f2: Downtime	41.52	27.27	25.35	23.09	23.64	27.19	41.22	37.50	34.58	31.56
f2-f3: Uptime	21.45	29.22	22.97	22.49	28.38	30.23	27.37	13.26	13.16	18.20
f2-f3: Downtime	16.07	33.06	29.75	28.42	36.02	31.82	29.13	22.54	21.10	26.96
f3-f4: Uptime	17.76	19.10	16.79	19.90	22.37	20.95	21.03	22.58	14.95	25.50
f3-f4: Downtime	36.01	19.24	22.85	31.97	26.09	28.63	23.88	29.45	22.95	18.72

The sum of the values of the two parameters defines the total upscaling time. The same configuration is valid for the downscale process. Based on the measurements in Table II we observe that the total time of upscaling from *f1* to *f2* is 68.26 seconds, from *f2* to *f3* is 37.52 seconds and from *f3* to *f4* is 53.77 seconds. We further define a connection looseness factor that is the division of uptime to downtime value. The factor is calculated as 1.17 for resizing *f1* to *f2*, 1.21 from *f2* to *f3* and 1.29 from *f3* to *f4*. Based on the table values we conclude the following.

- The overall time for VM resizing is always less than 53.5 seconds.
- As the size of the requested VM is increased, the time for VM creation does not change linearly and is related to the size of the required flavor and the bandwidth.
- The VM is accessible as the system keeps the same endpoint (IP).
- The connection looseness factor shows an increasing trend each time the VM upscales, thus VMs tend to loose connection when size is increased.

In Figure 7 we visualize the vertical autoscaling values of Table II in a clustered column diagram in order to highlight above findings.

C. Comparison of horizontal and vertical autoscaling

This section demonstrates the comparison between horizontal and vertical autoscaling that it is triggered by ICLB. We base out experiment in Figure 5, however we activated the $N_{1_{Clone}}$ so users send requests to the ICLB component that in turn triggers the autoscaling mechanism. In this case load balancing is performed in a round-robin fashion, meaning that half of the re. For this experiment autoscaling is executed every 100 YCSB requests in order to measure the downtimes (vertical) and VM creation times (horizontal) of identical flavors (*f2* and *f3*). Figure 8 demonstrates the comparison between the horizontal and vertical autoscaling of a cloud App for flavor *f2* along with the linear trend lines.

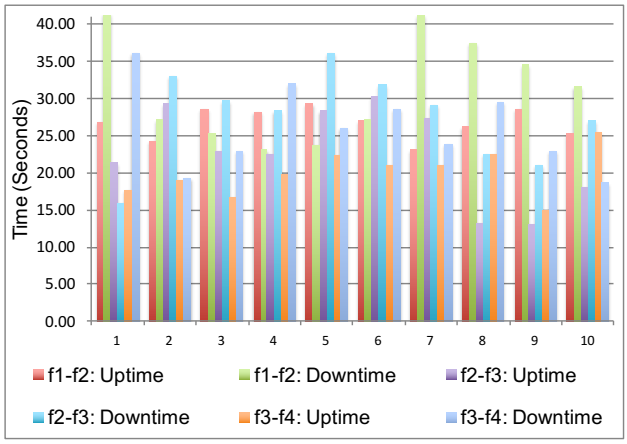

Fig. 7. Benchmarking: Vertical autoscaling of a cloud VM

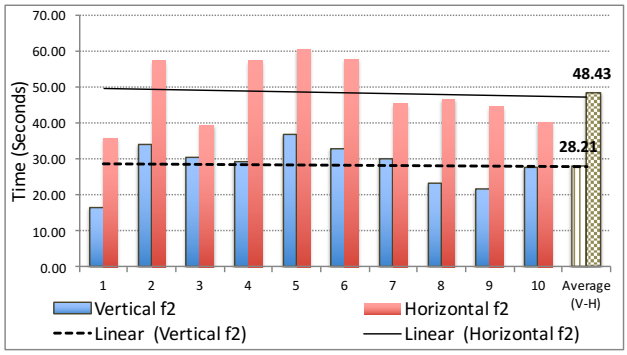

Fig. 8. Horizontal versus vertical autoscaling of a cloud App for flavor *f2*

Similar to Figure 8, Figure 9 demonstrates the comparison between the horizontal and vertical autoscaling for flavor *f3*.

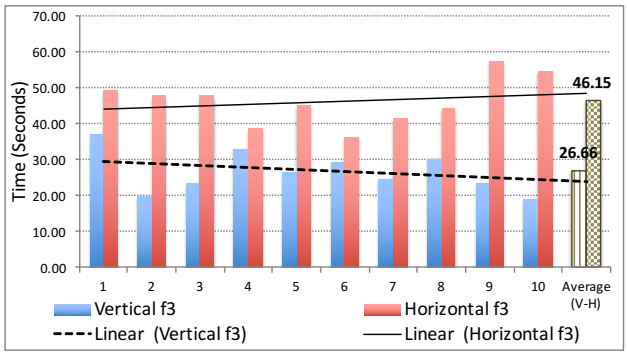

Fig. 9. Horizontal versus vertical autoscaling of a cloud App for flavor *f3*

Based on the previous figures we conclude to the following findings.

- The horizontal autoscaling outperforms the vertical one, since the downtime average is lower. The average values of vertical autoscaling (28.21 and 26.66 seconds) compared to the horizontal ones (46.15 and 48.13 seconds) demonstrate that, based on times, the preferred solution is the horizontal.
- The trend lines show that the vertical autoscaling downtimes for both *f2* and *f3* decrease as the ICLB component executes the mechanism based on the specific number of YCSB runs. In contrast, the horizontal autoscaling case demonstrates a marginally decreasing trend for f2 and an increasing one for *f3*.

It should be noted that during this experiment the assumption was that the $\text{App}_{\text{Clone}}$ VM has been created at a previous stage.

D. Inter-Cloud Load Balancing Benchmarks

This section presents the fundamental benchmark study of the ICLB that is the key component of the ICLB framework. The topology of the App/services is similar to Figure 5 where users send HTTP requests to the ICLB that in turn it forwards each into the N_1 or the $N_{1_{Clone}}$ VMs, however in this case N_1 is deployed in OpenStack and $N_{1_{Clone}}$ is deployed in VMWare VCloud. The experimental configuration involves YCSB executions of *workloada*[14] that includes 1000 record counts and operation counts. To compare performance, we present two scenarios in which (a) all the traffic is forwarded to one VM that executes all HTTP requests (here, ICLB is acting as a proxy) and (b) the traffic is distributed among two identical VMs (here, ICLB is acting as a load balancer in a round-robin fashion). Figure 10 demonstrates the results of the execution of 1000 requests (with total transfer size of 21.6 KB) where the percentages demonstrate the requests served within a certain time (ms).

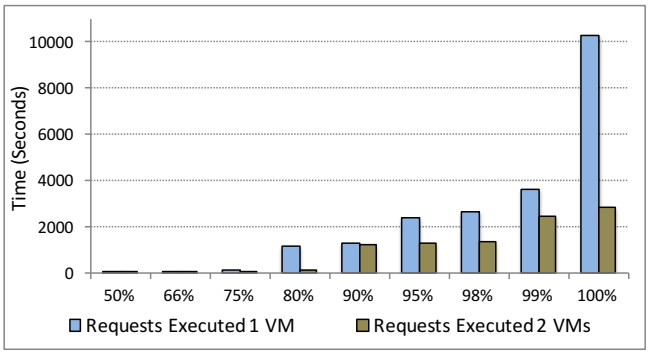

Fig. 10. Benchmarking: Percentage of requests served within a certain time (ms)

Figure 11 shows metrics including the mean time across all concurrent requests, the mean time per request, and the total time of execution for both cases. We can observe that the ICLB improves times for each metric. Further to the aforementioned scenarios, another useful analysis is the comparison of the load balancing algorithms that could assist with the next sections experimental developments. Figure 12 shows the comparison between the round robin, IP hash and the least connected algorithms.

In particular, Figure 12 (a) demonstrates the total time required to execute 5,000 and 10,000 requests with concurrency level of 100 users in both cases and Figure 12 (b) the comparison of the requests per second for the same configuration.

It is apparent that in the low volume requests scenario (C:100/R:5000) the least connected algorithm offers improved

[14]https://github.com/brianfrankcooper/YCSB/blob/master/workloads

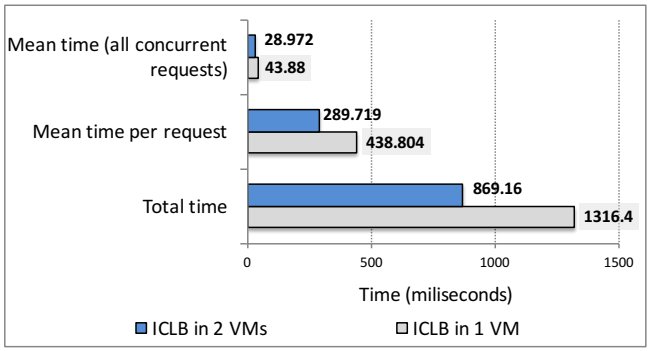

Fig. 11. Benchmarking: Mean time across all concurrent requests, the mean time per request, and the total time of execution

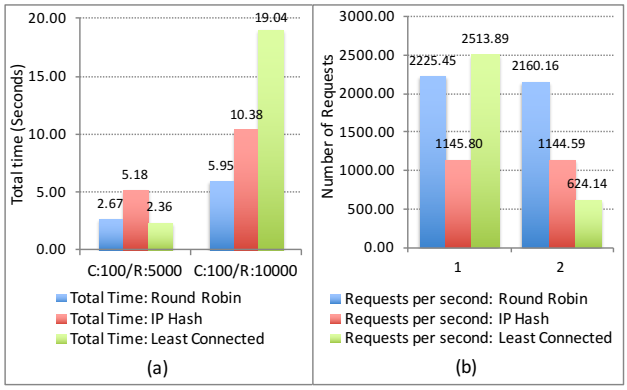

Fig. 12. Benchmarking: Comparison between round robin, IP hash and least connected load balancing algorithms

results (lowest total time and highest serving volume of request per second). In contrast, when the number of the requests increased significantly (C:100/R:10000) the round robin algorithm outperforms all three. Based on Figures 10, 11 and 12 we make the following observations.

- The percentage of the requests served demonstrates that in the case of 1 VM, 50% of the requests are completed in at most 91 milliseconds while 90% of the requests are completed in at most 1.196 milliseconds. All requests are completed within 9.524 milliseconds, which was the longest request time noted in the test.
- During the execution of the benchmark almost 90% of the requests are served in similar times, while above this amount, the serving times differ radically. Eventually, 100% of requests are executed in 9.524 ms for 1 VM and in 2.663 ms for 2 VMs.
- The transfer rate (KBs per second) for 1 VM is 9.61 while for 2 VMs it is 14.56. The transfer rate factor (that is measured as the division of the transfer rate of two to one VM) is calculated as 1.5, thus it shows significantly increased performance.
- The requests per second for 1 VM is 22.79 while for 2 VMs it is 34.52, showing that in the second case the serving volume has been increased significantly.
- The mean time (that is the time between one HTTP request and another) is well improved in the load balancing case, since we observe around 60% performance gain.
- Based on the experimental tests, the least connected algorithm is most suitable for 5,000 requests with the round robin for 10,0000 requests.

E. Inter-Cloud Load Balancing based on the volume of the HTTP Traffic

This experiment shows ICLB autoscaling performance analysis of the topology of Figure 13. We experiment at the App and LLB level, while the SLBs are monitored in order to ensure that there are no failures in communication. In this case, during the execution of a number of requests, we trigger vertical autoscaling based on HTTP traffic reconfiguration.

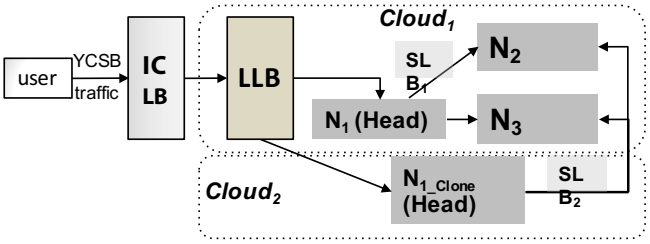

Fig. 13. HTTP interactions among ICLB users and App/services and components

The experimental configuration is as follows.

- We executed an extensive number of 30,000 HTTP requests using the round robin algorithm. The traffic is directed from the ICLB to the LLB component that in turn distributes it to the App and App_{Clone}.
- We set a rule for vertical and horizontal autoscaling to the 3,000 HTTP requests.
- The ICLB and LLB VMs includes 2 CPUs, 4GB RAM, 40GB HD (flavor *f3*) while the App and App_{Clone} includes 1 CPU, 2GB RAM, 20GB HD (flavor *f2*).
- The vertical autoscaling mechanism sends a resizing request for App_{Clone} from flavor *f2* to *f3*.
- The horizontal autoscaling mechanism requests a flavor *f3*.

Figure 14 shows the comparison between vertical (ICLB Vert.) and horizontal (ICLB Hor.) autoscaling performance during run-time execution of 30,000 HTTP requests. In particular, Figure 14 (a) demonstrates the time taken for tests that is measured in seconds (primary vertical axis) and the requests per second (secondary vertical axis), while Figure 14 (b) demonstrates the mean time per request measured in milliseconds (primary vertical axis) and the transfer rate (secondary vertical axis). Observing the figures, we conclude that the horizontal outperforms the vertical autoscaling.

According to Figure 14 (a) and (b), we conclude to following findings.

- The horizontal outperforms the vertical autoscaling for requests executed per second, mean time per request and transfer rate. The performance factor for horizontal (that is calculated as the division of transfer rate between horizontal and vertical autoscaling) is calculated at 1.68.
- The total time (time taken for tests) for the vertical case is higher. This comes in contrast to Figure 9 benchmark

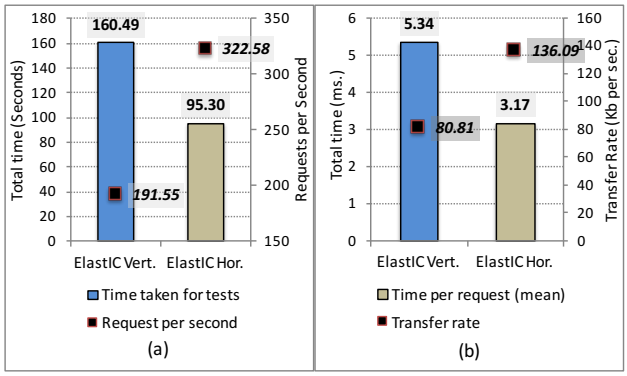

Fig. 14. Comparison between vertical and horizontal autoscaling performance during run-time execution of 30,000 HTTP requests

results, that demonstrate that the time needed for vertical autoscaling is less than the horizontal. Yet, we have also observed that during the experiment the App VM (that is the VM that did not upscale and continued to serve HTTP requests as normal) started to show a degradation in the volume of requests served because of delays caused by the increasing load in the ICLB component.

To minimize that issue, caused when the VM status is resizing or migrating, we developed an optimization scheme that allows direct interactions with the load balancer in order to change its configuration parameters on the fly during run-time and before the execution of the vertical autoscaling request. We noticed that when the ICLB configuration is changed before (i.e remove an instance from the list before its status changes to offline) the distribution of the load balancer is well optimized. Figure 15 demonstrates the optimized performance results.

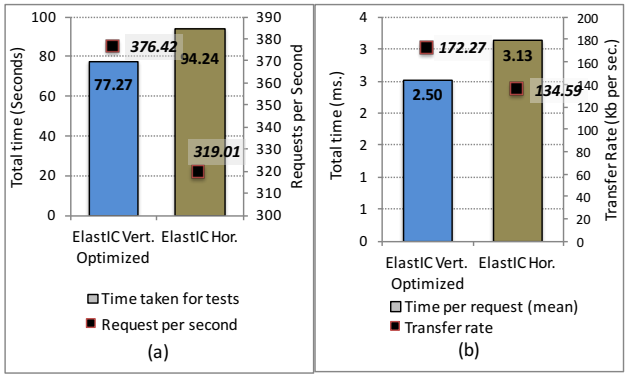

Fig. 15. Comparison between optimized vertical and horizontal autoscaling performance during run-time execution of 30,000 HTTP requests

According to the optimization scheme, the performance of the vertical to horizontal autoscaling is increased by a factor of 1.27. Similarly, results for other metrics (Figure 15) are well improved: for instance, the vertical autoscaling serves 376 requests per second while the horizontal serves around 319.

F. Inter-Cloud Load Balancing based on Resource Usage Monitoring

As discussed in Section III, the monitoring component performs real-time analysis of Apps/services and thus could trigger autoscaling according to resource usage. To demonstrate this, we present an experimental analysis similar to the topology of Figure 13 by triggering vertical autoscaling when CPU level is increased over a certain amount, as per the following configuration.

- The assumption is that at initialization stage the $\text{App}_{\text{Clone}}$ is offline and will be started according to a CPU load threshold of the App VM.
- We configure the ICLB service to trigger horizontal autoscaling for App VM CPU loads higher than 15%.
- We measure real-time CPU load following an experimental time frame.

Figure 16, shows the App CPU load during this time frame. It could be observed that at 16:07:31 the CPU load becomes 16%, a percentage that triggers the autoscaling mechanism. However, it should be mentioned that this is an ideal scenario that we stress the VM to increase the CPU load percentage in order to trigger autoscaling. The AppClone that is offline will be started automatically then the load will be 15% or higher. For example, in the case where the threshold will be set to a higher number i.e. 50% autoscaling will not be triggered until the load will reach such number. So, there the challenge is to define the correct threshold to avoid suboptimal autoscaling. However, this is not the aim of this study and in the future work we expect to increase the number of experiments in order to define a historical record that will be able to train a machine learning algorithm that will define autoscaling triggering according to selected features such as CPU, memory etc.

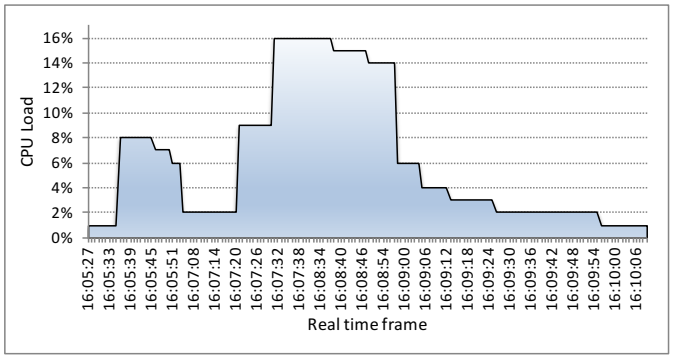

Fig. 16. CPU load of App VM

Figure 17, shows the comparison between normal (execution without adding a new instance) and CPU load balancing triggering and the values of the selected metrics.

In detail, it shows the performance gain in selected metrics for normal and ICLB load balancing when the CPU reaches the 15% limit. According to these, we conclude the following findings.

- The ICLB minimizes the CPU load. This could be observed in Figure 16 at the time that the new instance becomes available.

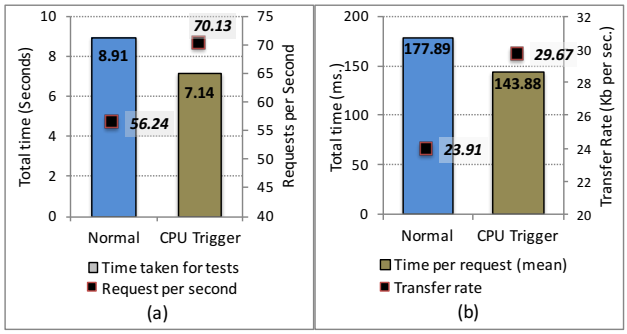

Fig. 17. Comparison between normal and CPU load balancing triggering for total test time, request served per second, time per request and transfer rate

- The performance factor regarding the transfer rate is 1.25 presenting an important optimization gain.

To summarize, the experimental analysis of Section 5, Subsections D and E, present the various cases of VM reconfiguration in an inter-cloud system. Following the flow of the experimentation, we concluded that the horizontal and vertical elasticity could offer significant optimizations and could be triggered either based on increasing traffic volume or resource utilization. The downscaling process supports only flavors with similar hard disk sizes, and for these ICLB offers similar results, thus due to this technical limitation we decided to demonstrate only upscaling cases.

G. Load Balancing based on different Inter-Cloud Layers

Until now, the experimental analysis included tests executed in different cloud providers but on the same platform (OpenStack). This section presents an extensive experiment of a heterogeneous inter-cloud system, where applications and services are deployed in different cloud platforms. The assumption is that the ICLB is deployed in Cloud A, and App and App_{Clone} in Cloud B. Both applications utilize a set of 3rd party services belonging to Cloud C (S_1, S_2) and the experiment will dynamically create a new one that will be the result of a vertical autoscaling (S_3) triggered by the ICLB component. In particular, we have deployed Cloud A in Amazon, Cloud B in VMWare Cloud and Cloud C in OpenStack to demonstrate heterogeneity. Figure 18 shows the topology of the inter-cloud system.

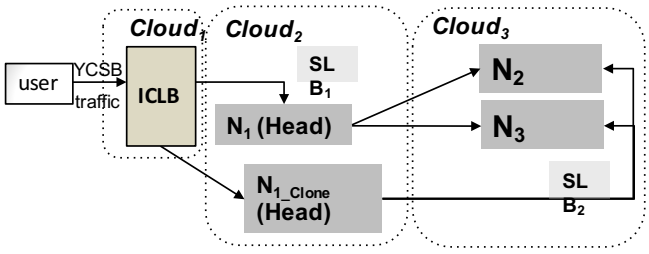

Fig. 18. Topology of HTTP traffic in three heterogeneous clouds

The experimental configuration involves the following setup.

- The time frame of the test is set to 300 seconds in which it executes around 50,000 HTTP requests.
- The total traffic for all cases is 21.6 MB per second.
- We measure the CPU load and we set a vertical VM autoscaling threshold at 15%. This means that a new VM will become active when the CPU of one of S_1 or S_2 is above that amount.
- We execute the experiment based on a round robin load balancing algorithm.

The monitoring component collects the resource CPU load that is also evaluated in real time from the ICLB component. Figure 19 demonstrates the CPU load variation during this time frame. It could be observed that when Service 1 reaches its peak limit (threshold of 15%), ICLB triggers the reconfiguration of resources that denotes the creation of Service 3 (in heterogeneous Cloud 3). Figure 20 further includes two sub-frames for (a) characterizing the first frame when CPU utilization is 16% (Service 1), and (b) the time frame that vertical autoscaling is executed (total time of 94 seconds). At the end of this process a new instance is available for traffic distribution in the ICLB. Lastly, after the creation of the new instance, it can be observed that the CPU loads are reduced (below the threshold of 15%). Similar to Figure 16, the experiment includes a selected threshold that is to overcame (i.e. the 15% threshold). To conclude, the results of the experiment demonstrate that the autoscaling can be executed relatively fast (i.e. one and a half minute) if we consider the number of operations involving in this process (create a new VM, configure network interface, etc.). In addition, we set the threshold to 15% CPU percentage, that clearly is a low load, however we mostly wanted to demonstrated the process of autoscaling rather the conceptualization of selecting the ideal thresholds. As mentioned before these could be the result of an analysis using historical data from real world datacenters in order to train the system define thresholds according to real usage, a direction that we aim to focus in future works.

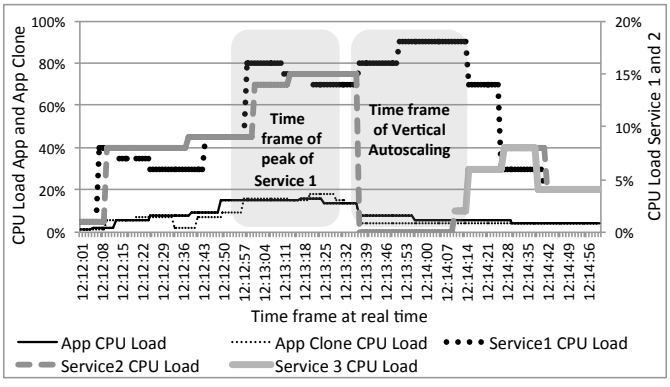

Fig. 19. CPU load of inter-cloud topology for time frame of 300 seconds

To demonstrate the effectiveness of the ICLB autoscaling in the above scenario we compare the next two cases.

(i) We execute the same experiment with the topology of 18 without triggering vertical autoscaling. This case involves two identical 3rd party services.
(ii) We execute the vertical autoscaling to examine how it affects the performance of the HTTP request service

considering any delays that could be included in the ICLB component when it updates the configuration file.

Figure 20 demonstrates comparison between various metrics for cases of vertical and non-vertical autoscaling.

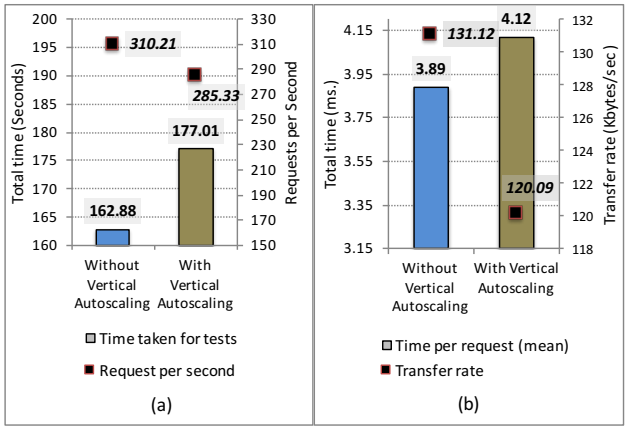

Fig. 20. Comparison between total time of tests, requests per second, time per request and transfer rate for the two cases of vertical and non-vertical autoscaling

Based on the analysis of this section we concluded the following findings.

- The inter-cloud topology demonstrates interactions among heterogeneous cloud platforms and services. The ICLB component performs real-time evaluation of results (from the monitoring component) and the IC mediation service allows communication among the heterogeneous clouds using their APIs.
- The ICLB component reduces the CPU load of the services at the moment that the new service instance is registered in the ICLB component.
- There are no errors and failures during serving the whole set of the 50,000 HTTP requests.
- The non-vertical autoscaling case outperforms the vertical one in terms of requests execution, however the transfer rate factor is measured at 1.08, which is marginally better, caused by the delays of the ICLB component.

This study focuses on the ICLB level, thus portability and security issues have been treated as a black-box. Especially, inter-cloud IaaS level portability is the total time for migrating a VM among clouds. To have a more complete view, we executed a simple example case of an Cassandra instance migration (running on Ubuntu 12.04 LTS-64 of 800 MB). The process includes a) log into the original cloud platform (where the VM is already running), b) create a clone snapshot (i.e. of an already operated Cassandra node), c) download the snapshot, d) log into the target cloud, e) upload the snapshot, f) set keypair, g) launch the cloned instance (that inherits all the configuration of its "master" VM image, and h) set instance IP. The total time for instance migration is 64.93 seconds and could be taken into consideration when horizontal autoscaling includes VM migrations.

VI. Conclusions and Future Work

We proposed the ICLB, a modular framework that allows load balancing of inter-cloud applications and services that belong to heterogeneous providers. We aimed to improve the elasticity in the IaaS level through autoscaling of cloud and inter-cloud VMs, so we highlighted key requirements. We also utilized various load balancing configurations in order to ensure zero downtime. We based our initial hypothesis in a real world system (Cassandra cluster deployed in OpenStack and VMWare) in order to demonstrate the problems and issues on scaling inter-cloud applications. The experimental analysis is positive and shows various topologies in which ICLB framework could be applied along with fundamental benchmarks on horizontal and vertical autoscaling that could serve other studies, as well. The contributions of our work include the proposition of a new inter-cloud load balancer that acts on top of the clouds and allows interactions among heterogeneous cloud platforms. We compared different scenarios for vertical and horizontal elasticity and we demonstrated that in both cases we could executed the experiments without any loss in communication or failures. The future research steps involve different directions of solutions that could be applied as optimization schemes including machine learning algorithms. Also, we expect to increase the number of experiments in order to define a historical record that will be able to train a machine learning algorithm that will define autoscaling triggering according to selected features such as CPU, memory etc.

References

[1] S. Sotiriadis, N. Bessis, F. Xhafa, and N. Antonopoulos, "Cloud virtual machine scheduling: Modelling the cloud virtual machine instantiation," in *Proceedings of the 2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, ser. CISIS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 233–240. [Online]. Available: http://dx.doi.org/10.1109/CISIS.2012.113

[2] IBM, "Microservices from theory to practice: Creating applications in ibm bluemix using the microservices approach, http://www.redbooks.ibm.com, aug. 2016."

[3] D. Petcu, "Consuming resources and services from multiple clouds," *J. Grid Comput.*, vol. 12, no. 2, pp. 321–345, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10723-013-9290-3

[4] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 18:1–18:17. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741964

[5] E. A. Brewer, "Kubernetes and the path to cloud native," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 167–167. [Online]. Available: http://doi.acm.org/10.1145/2806777.2809955

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2890784

[7] L. Beernaert, M. Matos, R. Vilaça, and R. Oliveira, "Automatic elasticity in openstack," in *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, ser. SDMCMM '12. New York, NY, USA: ACM, 2012, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2405186.2405188

[8] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, Jan. 2011. [Online]. Available: http://doi.acm.org/10.1145/1925861.1925869

[9] A. Cassandra, "http://cassandra.apache.org."

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[11] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2015.

[12] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, April 2012, pp. 204–212.
[13] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1:1–1:39, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1342171.1342172
[14] I. Kureshi, C. Pulley, J. Brennan, V. Holmes, S. Bonner, and Y. James, "Advancing research infrastructure using openstack," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 4, pp. 64–70, December 2013. [Online]. Available: http://eprints.hud.ac.uk/19421/
[15] S. Bouchenak, "Automated control for sla-aware elastic clouds," in *Proceedings of the Fifth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, ser. FeBiD '10. New York, NY, USA: ACM, 2010, pp. 27–28. [Online]. Available: http://doi.acm.org/10.1145/1791204.1791210
[16] L. Beernaert, M. Matos, R. Vilaça, and R. Oliveira, "Automatic elasticity in openstack," in *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, ser. SDMCMM '12. New York, NY, USA: ACM, 2012, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2405186.2405188
[17] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038921
[18] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara, "Kaleidoscope: Cloud micro-elasticity via vm state coloring," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 273–286. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966471
[19] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, June 2011, pp. 559–570.
[20] F. Wuhib, R. Stadler, and H. Lindgren, "Dynamic resource allocation with management objectives: Implementation for an openstack cloud," in *Proceedings of the 8th International Conference on Network and Service Management*, ser. CNSM '12. Laxenburg, Austria, Austria: International Federation for Information Processing, 2013, pp. 309–315. [Online]. Available: http://dl.acm.org/citation.cfm?id=2499406.2499456
[21] Amazon auto scaling documentation. [Online]. Available: http://aws.amazon.com/documentation/autoscaling/
[22] Amazon elastic load balancing documentation. [Online]. Available: http://aws.amazon.com/documentation/elastic-load-balancing/
[23] N. Janssens, X. An, K. Daenen, and C. Forlivesi, "Dynamic scaling of call-stateful sip services in the cloud," in *Proceedings of the 11th International IFIP TC 6 Conference on Networking - Volume Part I*. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 175–189.
[24] G. Molto, M. Caballer, E. Romero, and C. de Alfonso, "Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements," *Procedia Computer Science*, vol. 18, no. 0, pp. 159 – 168, 2013.
[25] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1:1–1:39, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1342171.1342172
[26] "Load balancing as a service," 2105. [Online]. Available: https://wiki.openstack.org/wiki/Neutron/LBaaS
[27] F. Paraiso, P. Merle, and L. Seinturier, "socloud: a service-oriented component-based paas for managing portability, provisioning, elasticity, and high availability across multiple clouds," *Computing*, vol. 98, no. 5, pp. 539–565, 2016. [Online]. Available: http://dx.doi.org/10.1007/s00607-014-0421-x
[28] D. Ardagna, E. D. Nitto, P. Mohagheghi, S. Mosser, C. Ballagny, F. D'Andria, G. Casale, P. Matthews, C. S. Nechifor, D. Petcu, A. Gericke, and C. Sheridan, "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds," in *2012 4th International Workshop on Modeling in Software Engineering (MISE)*, June 2012, pp. 50–56.
[29] Y. Huang, N. Bessis, A. Brocco, S. Sotiriadis, M. Courant, P. Kuonen, and B. Hisbrunner, "Towards an integrated vision across inter-cooperative grid virtual organizations," in *Proceedings of the 1st International Conference on Future Generation Information Technology*, ser. FGIT '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 120–128. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10509-8_15
[30] S. Sotiriadis, N. Bessis, A. Anjum, and R. Buyya, "An inter-cloud meta-scheduling (icms) simulation framework: Architecture and evaluation," *IEEE Transactions on Services Computing*, vol. DOI: 10.1109/TSC.2015.2399312, pp. 1–1, 2015.
[31] S. Sotiriadis and N. Bessis, "An inter-cloud bridge system for heterogeneous cloud platforms," *Future Generation Computer Systems*, no. 0, pp. –, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X15000400
[32] L. Vacanas, S. Sotiriadis, and E. Petrakis, "Implementing the cloud software to data approach for openstack environments," 2015.
[33] S. Sotiriadis, N. Bessis, P. Kuonen, and N. Antonopoulos, "The inter-cloud meta-scheduling (icms) framework," in *Proceedings of the 2013 IEEE 27th International Conference on Advanced Information Networking and Applications*, ser. AINA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 64–73. [Online]. Available: http://dx.doi.org/10.1109/AINA.2013.122

Stelios Sotiriadis is currently a research scientist in the Edward Rogers Sr. Department of Electrical and Computer Engineering of the University of Toronto, Canada. His research interests are related to distributed systems and especially Cloud computing systems, Inter-Cloud, Future Internet (FI) applications and Internet of Things (IoT). He has published over 70 papers and he won 2 best paper awards. His personal profile is available in www.sotiriadis.gr.

Nik Bessis is a full Professor of Computer Science and the Head of Department of Computing at Edgehill University, UK. He is a Fellow of HEA, BCS and a Senior Member of IEEE. His research is on social graphs for network and big data analytics as well as on developing data push and resource provisioning services in IoT, FI and inter-clouds. He is involved in and led a number of funded research and commercial projects in these areas. Prof. Bessis has published over 250 papers, won 4 best paper awards and is the editor of several books and the Editor-in-Chief of the International Journal of Distributed Systems and Technologies (IJDST).

Cristiana Amza is an Associate Professor in the department of Electrical and Computer Engineering of the University of Toronto. Cristiana Amza received her B.S. degree in Computer Engineering from Bucharest Polytechnic Institute in 1991, the M.S. and the Ph.D. degrees in Computer Science from Rice University in 1997 and 2003 respectively. Her research interests are in the area of distributed and parallel systems, with an emphasis on designing, prototyping and experimentally evaluating novel algorithms and tools for self-managing, self-adaptive and self-healing behavior in data centers and Clouds. She is actively collaborating with several industry partners, including Intel, NetApp, Bell Canada, and IBM through IBM T.J. Watson, Almaden and IBM Toronto Labs.

Rajkumar Buyya is a Professor of Computer Science and Software Engineering; Future Fellow of the Australian Research Council; and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft Pty Ltd., a spin-off company of the University, commercialising its innovations in Grid and Cloud Computing. He received B.E and M.E in Computer Science and Engineering from Mysore and Bangalore Universities in 1992 and 1995 respectively; and a Doctor of Philosophy (PhD) in Computer Science and Software Engineering from Monash University, Melbourne, Australia in 2002. He served as the foundation Editor-in-Chief of IEEE Transactions on Cloud Computing. He is currently serving as Co-Editor-in-Chief of Journal of Software: Practice and Experience, which was established over 45 years ago.