University of Science and Technology in Krakow

Faculty of Computer Science, Electronics and Telecommunications
Department of Computer Science

Autonomic cloud computing platform for scaling users' application

DARIUSZ CHRZĄŚCIK
RADOSŁAW MORYTKO

Supervised by MARCIN JARZĄB
Assistant Professor of Computer Science

Krakow 2013

Oświadczamy, świadomi odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonaliśmy osobiście i samodzielnie (w zakresie wyszczególnionym we wstępie) i że nie korzystaliśmy ze źródeł innych niż wymienione w pracy.

. . . .

PODPIS

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki

Framework dla autoskalowalnego środowiska wykonawczego dla aplikacji użytkowników w konfiguracji hybrydowej chmury obliczeniowej

Dariusz Chrząścik
Radosław Morytko

Promotor: dr inż. Marcin Jarząb

Krakow 2013

I would love to express my best wishes

..

Abstract

Cloud computing has become an attractive model for provisioning on demand computing resources as services to end-users. It is based on the assumption that almost anything can be viewed as a service, starting from applications delivered over Internet, through hardware in the data centers and ending on computing power. That model appears to be so attractive as from the user point of view the offered resources are infinite, transparent, robust and ready to consume at any time.

Most of the times end-users do not know in advance what the demand for the service is. This creates a requirement in which their systems are auto-scalable, i.e. they support sudden spikes in demand followed by underutilization at other times. An architecture of a cloud computing system, that meets these requirements, has a characteristic of a multi-hierarchical autonomic system, where different layers corresponds to different levels where cloud operates: starting from an application layer, through the application platform, infrastructure to end on a cloud instance.

Problem complexity raises challenges in a variety of aspects, especially in terms of providing cooperation and mutual sharing of resources that may belong to different kind of cloud providers. Therefore, an architecture that enables seamless cooperation among cloud providers and takes into account various QoS requirements of end-users be developed is absolutely vital. The InterCloud architecture is one of the first attempts that had been made in this direction. Having characteristic of an application platform in mind, we propose a variation of that architecture that supports cooperation among these application platform and fulfils needs of a decentralized environment.

The main contributions of this thesis are as follows: a) reference architecture that enables aforementioned scenarios, b) implementation of that architecture c) simulation and laboratory tests.

Contents

1. Introduction

1.1. Motivation

One of the keys factors that has driven transformation of computing industry in the last years is the perception of computing utilities as an ordinary property[30], which can be easily accessed and adjusted to a specific needs. That point of view resulted in profusion of different services, often collectively referred as a cloud computing [41]. Similarly to services common to traditional markets, customers expect them to be accessible on demand and in easy manner, while paying only for the consumed goods. Furthermore, customers are interested in a given service provider only when it is eligible to guarantee appropriate quality of service.

The particular service providers that are addressed by this paper are the ones that supply users with an application execution platform, what is widely known as providing Platform-as-a-Service. In that case, a customer is an entity that has developed application and is eager to deploy it on an application platform that is able to fulfil his specific requirements, both in terms of quality and cost.

Having customer requirements in mind, it is crucial that service provider is able to adapt itself to meet them. For example, such adaptation can be triggered by a sudden spike in resource demand and may result in provisioning additional application platforms. However, due to the complexity of a system under consideration, there are different levels where adaptation is possible:

– user application

– application platform

– infrastructure

What is more, the fact that single service provider is constrained by his finite amount of resources poses a risk that it may not be able to serve customer all the time. Consequently, it is expected that adaptation at a service provider level is also possible, i.e. provider can offload some traffic to a different provider, as long as it satisfies a customer.

While autonomic computing has a long history [42], it has not been directly applied to a multi-layered problem that exists in a cloud computing environment. Especially, the research area at the last layer, which sizes across different service providers, is new. Although, architecture known as InterCloud [27] investigates problem of co-operation and negotiation at cloud level, it neither has been implemented nor presented in context of autonomic system.

1.1.1. Business potential

The rapid growth of interest in cloud computing in recent years resulted in huge sums of money being invested in the field. Figure 1.1 shows the size of the public cloud services market in 2012 and the forecast of its nearly two times growth in 2016. This data suggests that the subject is attractive for IT industry from the economic point of view. However, higher amounts of money spent on cloud services involve higher expectations of theirs quality from customers. Although the most significant players in cloud computing have been in the field for quite a long time, it is still possible to outline some deficiencies their products have. Additionally, lack of common standards hinder cooperation among different cloud providers. For example it is nearly impossible to create an autoscaling cloud federation with Amazon Web Services (current leader in providing cloud services[38]) and another provider. What is more, Amazon AWS users cannot use more lightweight virtualization methods, such as linux containers. Nevertheless, when compared to other companies especially in terms of autoscaling capabilities, Amazon really shines. OpenShift, RedHat PaaS solution, ensures application scaling but with very limited possibilities of customisation of the process – the user can only choose if their application should scale and the whole algorithm is solely based on the number of concurrent requests to the application. Users of Heroku, another PaaS solution, have no automation tool that would control the number of instances (*dynos* in Heroku nomenclature) their application is running on – they can change it manually.

The proposed solution in this dissertation tries to deal with the aforementioned providers problems by outlining an example architecture that enables seamless cooperation among cloud providers and provides auto-scaling capabilities.

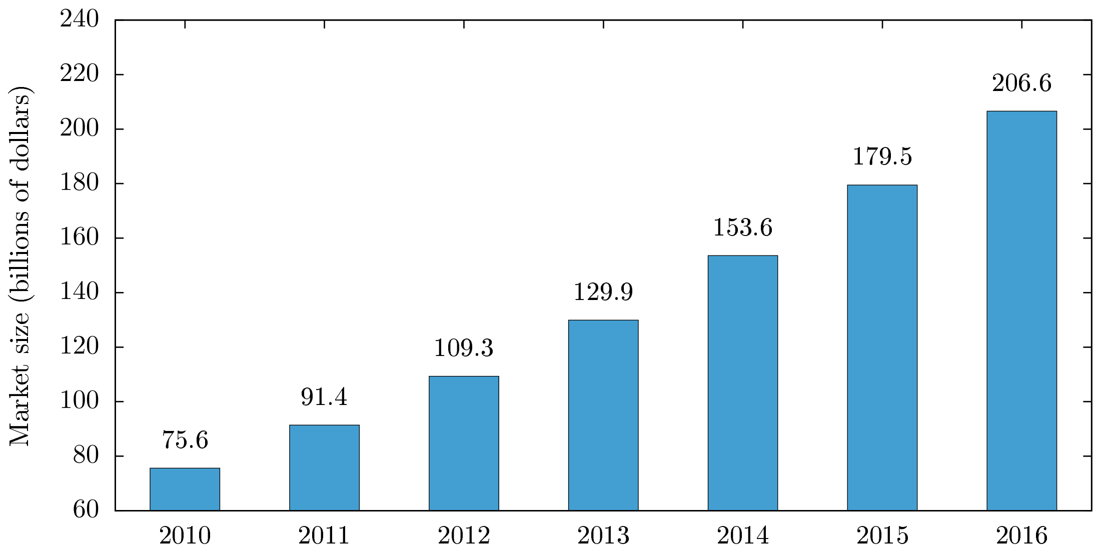

Figure 1.1: Public Cloud Services Market Size, 2010-2016 (forecast). Source: *Gartner, 08/2012*

1.2. Contributions

The main contributions of this dissertation are as follows:

– A proposal of an architecture of federated cloud computing environment, which is based on and can be viewed as a simplified version of *InterCloud*

– The notion of considering each service model as an autonomic system

– The implementation of the proposed architecture using OpenNebula technology stack

1.3. Impact

We hope that the concept of representing each level of an autoscaling subsystem as an autonomic one can be thought-provoking for cloud computing scientists. What is more, we believe that our successful attempt to implement a simplified variant of an InterCloud architecture will cause its gain in interest and popularity. Finally, we consider the ideas contained in this work be beneficial to the OpenNebula ecosystem as they provide insights into the ways Quality of Service can be ensured:

– implementing autoscaling capabilities

– designing *cloud infrastructure* in accordance with InterCloud architecture

1.4. Thesis structure

2. Platform adaptivity

This chapter introduces concepts and mechanisms that enhance a platform with adaptivity capabilities achieved by a fusion of rules, policies and scaling techniques.

2.1. Introduction

What makes a concept of an adaptivity enticing is crucial idea behind guaranteeing Quality-of-Serivce: automatization. Generally, ensuring and enforcing given quality can be seen as continuous monitoring and reacting to events when necessary. Such idea can be expressed as IT process shown in figure 2.1.

While system administrator can be directly responsible for manually performing above-mentioned steps, it is much more favourable to automatize this process leading to a system self-management or an adaptable system. Not only does it lead to efficiency of an IT process but also to its effectiveness [36]. It is possible through:

– *rapid process initiation* - components auto-initiates actions based on information derived from a system

– *reduced time and skill requirements* - automatization of IT processes makes them easier and less troublesome what is especially important for skill-intensive, error-prone and long lasting tasks

In a context of delivering a computing platform, adaptivity adds auto-scaling features to a solution offered by Platform-as-a-Service provider. Self-managing is possible through usage of an Elasticity Controller which gathers probes from resource such as application container and uses that knowledge to execute appropriate action on that resource, indirectly modifying consecutive probes [46]. This concept illustrates diagram 2.2.

The remaining of this chapter describes crucial elements that compose elasticity controller: policies, data analysis, triggered actions and presents a comparison of cloud providers.

2.2. Policies

While auto-scaling is offered by a vast amount of cloud providers (e.g AWS, OpenShift, OpenNebula) it often lacks a sophisticated mechanisms allowing for specific scaling policies, being limited to only one predefined rule as it is in case of OpenShift for example.

Policy denotes a condition which, when satisfied, triggers an action that is supposed to harness cloud instance in a way that future evaluations of condition will be unsuccessful. Typically condition itself is accompanied by a minimal and maximal number of node instances, allowing for ensuring minimal QoS and controlling maximal costs. Currently, industry leaders supports two main kind of policies [4]:

– *expression based* - allows to define how you to scale application in response to changing conditions, which include factors such as memory, CPU usage, cost or some indirect, calculated metrics

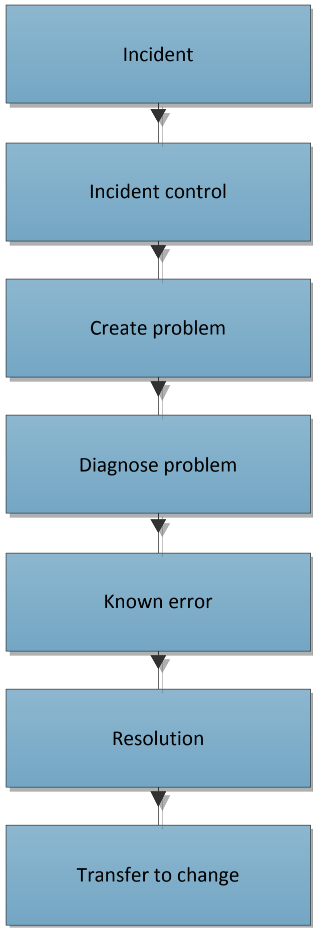

Figure 2.1: Exemplary IT process

– *scheduled* - allows to scale an application in response to predictable load changes. For example, traffic increases during the weekends and decreases on working days. Hence, that predictable traffic patterns is used to scale application based on current time.

Technically, policies are expressed in some human-readable format such as JSON, XML as it is in case of AWS EC2 or custom expression used for example by Carina. Appendix A.2 presents example configuration used by AWS E2 Auto-Scaling.

2.3. Data analysis

Having policies defined, their conditions are evaluated against data acquired from sensors. In a simplest case this evaluation can be based on a Threshold Model [39], which defines a valid range. In cases when given metric violates that condition (i.e. value is either smaller than minimal or bigger than maximal acceptable) corresponding resource is properly adjusted - figure 2.3 illustrates that idea.

While trivial in its form, cases of AWS, OpenShift, Carina, OneCloud proves it is useful in a real-world scenarios. Having that said, more sophisticated algorithms also exists:

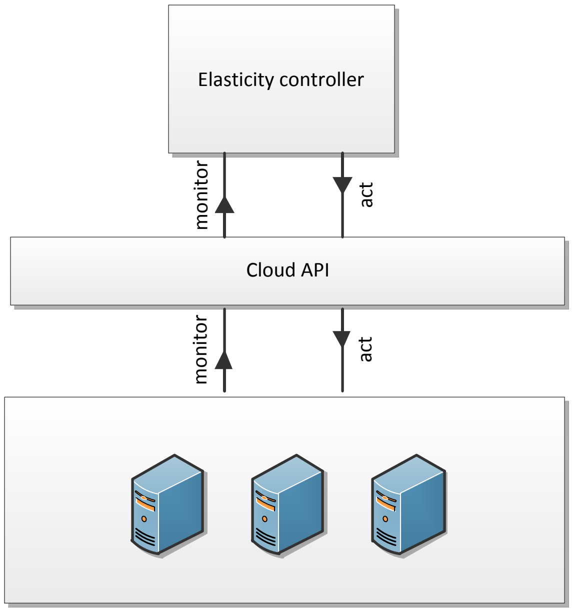

Figure 2.2: Elasticity controller

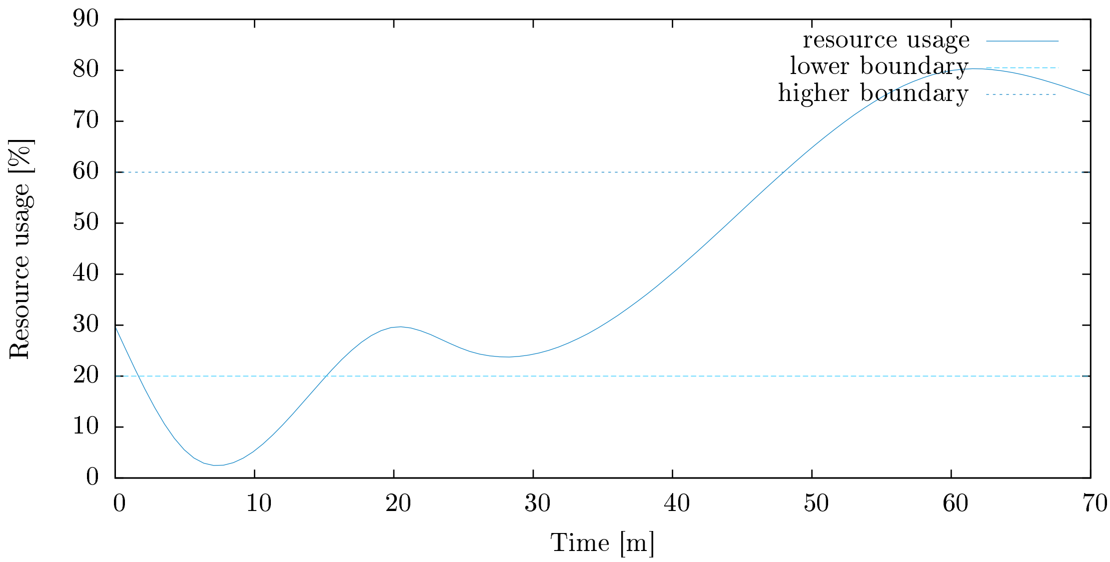

Figure 2.3: Threshold model

– *integer programming* - auto scaling is reduced to server integer programming problems, which aims to minimize the cost or maximize the computing power with either computing power constraints or budget constraints [40]

– *burst based padding* - employs a signal processing technique based on fast Fourier transform, burst pattern is extracted and used to calculate a padding value. Coefficients that represents the amplitude of each frequency component are used to calculate burst density. Depending of that value (i.e. is higher than 50%) appropriate percentile of the burst values are used [44]

– *remedial padding* - padding errors are being recorded and used in successive padding evaluations. In other words, let $e_1, e_2, ...e_k$ denote the recent prediction errors, next the weighted moving average is calculated. Actual applied padding is either padding itself or weighted average, depending which one is greater [44]

– *linearised dynamic control* - linearised correction model is based on control equations [24]:

$$x(k+1) = Ax(k) + Bu(k) \tag{2.1}$$

$$y(k) = Cx(k) + Du(k) + Ez(k) \tag{2.2}$$

where x denotes the state variable vector and coefficient matrices: A B C D E are fitted to historical data as a regression model - [32]

– *markov decision process model* - computes optimal reaction to state changes by using observation on the system with assumptions about the rate of changes expected in the future [23]

2.4. Triggered actions

Actions that are being triggered by a elasticity controller are focused on application scaling. Previous chapter described that problem in detail.

3. Scaling applications

This chapter is devoted to the concept of scaling users' application from the perspective of a cloud platform provider. To achieve that, it presents attainments of research groups working in that area as well as it considers mechanisms used in products currently available on the market.

3.1. Introduction

The reason why scaling application lies in our area of interest is the fact that it is widely accepted measure for improving application performance, consequently increasing offered Quality-of-Service. Enriching system with capability to scale entails avoiding additional costs that are related to coping with excessive traffic. In some cases, these costs may be caused by not handling extra traffic at all and may involve aspects such as: increased response time, processing overhead, space, memory, or money [22].

While scalability is a widely used term, it still lacks a clear and concise definition. Over the time, there were a few attempts to define it, yet not all of them were claimed as successful [35] [33]. Hence, it is necessary to clarify this term before going into further discussion. Instinctively, scalability is perceived as ability of a system to accommodate an increasing number of elements or objects to process. In particular, we can point out different types of scalability that are affected by increased number of requests: [22]:

– *load scalability* - ability to work without delays and unproductive resource consumption at light, moderate, or heavy loads while making good use of available resources. Factors that may hinder load scalability include: scheduling shared resource, self-expansion, inadequate exploitation of parallelism

– *space scalability* - memory requirements do not grow to intolerable levels as the number of items system supports increases

– *space-time scalability* - system continues to function gracefully as the number of objects it encompasses increases by orders of magnitude

– *structural scalability* - implementation or standards do not impede the growth of the number of objects system encompasses

Although, all of the aforementioned aspects are vital for any application, our work focuses solely on the first type of scalability. The reasoning behind this statement is that, while all of these scalability types lies in direct responsibility of an application developer, the load scalability can be additionally improved by adding additional resources to a system. This brings us to a question what kind of resources are used by an application or more appropriately in context of this dissertation: *what kind of resources can we add to improve application performance?* Required resources varies from an application to an application. However, among the most common ones we can distinguish:

– CPU

– memory

– storage

– network bandwidth

It is commonly agreed that there are two main possible ways the resource can be added:

– *horizontal scaling (scaling out)* - adding more nodes to a system, such as servers in a context of distributed application

– *vertical scaling (scaling up)* - increasing capacity of a single node in a system, i.e. adding additional memory, CPU, storage, etc.

What makes scaling application particularly interesting are the benefits offered by a cloud computing, especially the illusion of a virtually infinite computing infrastructure [46]. Making use of virtualization technologies, which often underpins cloud computing platform, allows for resource manipulation in a dynamic, on-demand manner. Although, cloud computing offers additional scaling capabilities, it increases solution complexity since they operate in different layers: server, platform, network as stated in [46]. However, since platform containers are often represented either as virtual machines or another isolated environment (e.g. OpenShift leverages SELinux and cgroups) they are similar in nature to server scaling and supports both scaling up and out. Therefore, the remaining of this chapter is focused solely on server scaling, omitting network scaling as it lays outside of scope of this dissertation.

Having that said, common sense dictates that adding resources is only a part of the success - it should be accompanied by tuning application platform configuration. For example, adding supplementary CPUs without increasing thread pool size that handle requests makes a little sense. Similarly, in context of a Java application, we have to increase heap size, to make a good use of extra memory. While importance of application tuning cannot be underestimated, its detailed analysis lies outside of the scope of this dissertation. Figure 3.1 presents different scalability layers and actions that can be taken at each level to improve application performance.

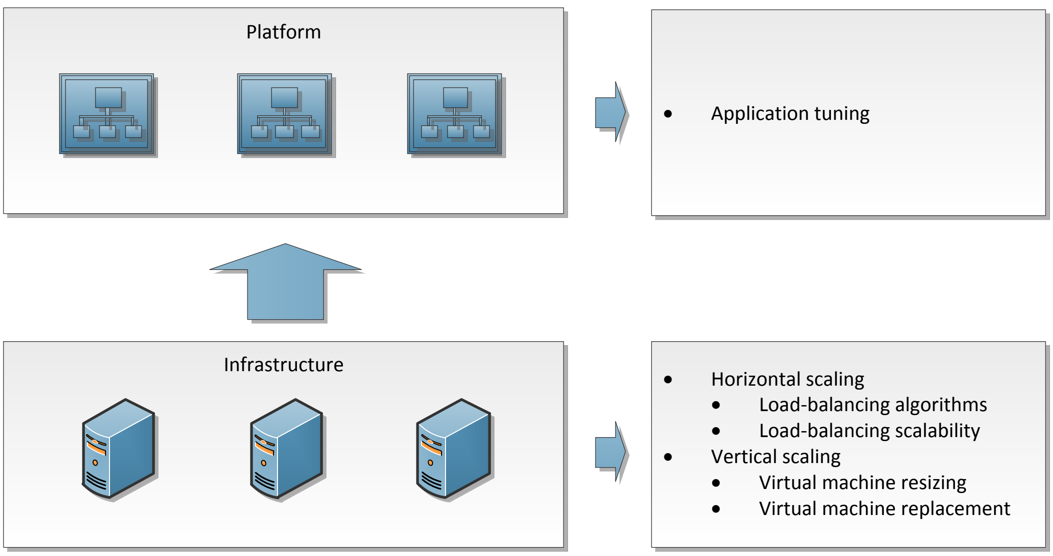

Figure 3.1: Scalability layers

With all that said, there is no silver bullet - not matter what underlying mechanism platform provider decides to use, the application developer is still responsible for creating an application with scaling in-mind. This statement has been already proven in 1967 by Amdahl law, which in short states that sequential component of a parallel algorithm impacts efficiency for a sufficiently large number processors [34] as shown in Figure 3.2. In other words, adding supplementary resources to a poorly written application (i.e. having a lot of sequential or synchronized components) can be beneficial only to a certain degree.

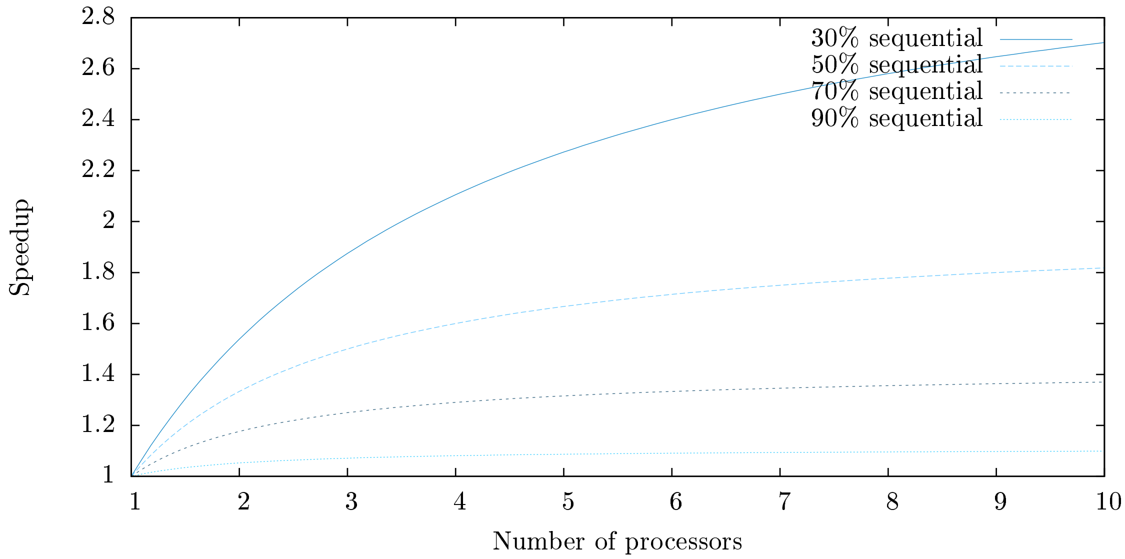

Figure 3.2: Amdahl's law

The rest of this chapter elaborates in detail about horizontal and vertical scaling taking into account mechanisms used in Platform-as-a-Service solutions that are available on the market.

3.2. Horizontal scaling

As outlined in previous section, horizontal scaling is about adding supplementary nodes to a system. As it is common to cloud computing, nodes are represented as virtual machines and this assumption is used in further discussion. Consequently, adding server comes down to cloning a new virtual machine from a template and possibly installing additional software and reconfiguring it later. While mechanism of creating new virtual machine from a template is offered literally in every IaaS platform currently available (OpenStack [17], OpenNebula [14], CloudStack [5] or Eucalyptus [6] to name a few) and is similar in manner, the underlying hardware and virtualization mechanism determines how fast provisioning is done.

Provisioning new server is only a first step in scaling an application, it is required to configure load balancing mechanism to make use of additional node. The two important aspects that have to be consider are: load-balancing algorithms and scalability.

3.2.1. Load-balancing algorithms

Generally, there are two types of load-balancers: hardware and software based. Due to the dynamic nature of system under consideration, we focus only on the latter as it offers a greater deal of flexibility. Among the most common algorithms we can distinguish [9]:

– *round-robin scheduling* - request are sent to successive nodes, according to their weights. This algorithm is fairest when the server's processing time remains equally distributed [9]

– *least connection* - the server with the lowest number of connections receives the connection

– *source routing* - source IP address is hashed, the same client IP address always reaches the same server

– *URI hashing* - URI that designates resource is hashed and divided by the total weight of the running servers. Such hash designates which server that receives the request. In practice, this algorithm is commonly used with proxy caches and anti-virus proxies in order to maximize the cache hit rate.

– *request counting algorithm* - load is distributed the requests among the various workers, ensuring that each gets their configured share of the number of requests

– *weighted traffic counting algorithm* - variation of above-mentioned algorithm with a difference that it is focused on bytes rather than number of request

– *pending request counting algorithm* - scheduler keeps track of how many requests each worker is assigned at present. A new request is automatically assigned to the worker with the lowest number of active requests

Situation gets further complicated when considering real-world web application that sends user information using cookies, what imposes requirement on load-balancer for session stickiness [45].

3.2.2. Load-balancing scalability

Although, it may seem that balancing workloads eliminates problem of a single point of failure (SPOF) among different servers, it is in fact shifted to load-balancing layer. In other words, load-balancer becomes a new SPOF. Therefore, in cases where high availability is required, multi-tiered load balancing architecture should be considered. This, however, seems not to be a case among IaaS or PaaS providers - none of them unequivocally specifies whether their provide redundancy at load-balancer level.

3.2.3. Load-balancer comparision

While there are many load-balancers available on the market, following are credited to be most popular:

– **HAProxy** [8] - load-balancer initially written by Willy Tarreau. Noticeably, it's used by OpenShift [15] to distribute load among gears [16]

– **BIG-IP Local Traffic Manager (LTM)** - solution offered by F5 [7]. Although LTM is a hardware solution, omitted in this section, it also has also its virtualized counterpart.

– **Apache HTTPD** [1] - popular HTTP server. When enhanced with additional modules, it can behave like a proxy or load-balancer. Over the time, there were several attempts to develop such modules: mod_jk [2], mod_proxy_balancer [3], to name a few. While the former is purely AJP13 oriented, the latter supports different protocols: HTTP, FTP and AJP13. As a consequence, only mod_proxy_balancer was taken into account during comparision.

Table 3.1 presents they key performance features and algorithm used to schedule requests.

	Performance features	Scheduling algorithms
HAProxy	– a single-process, event-driven model reduces the cost of context switch and the memory usage – O(1) event checker – single-buffering without copying data between reads and writes – zero-copy forwarding – optimized HTTP header analysis: headers are parsed an interpreted on the fly	– round-robin scheduling – least connection – source routing – URI hashing
BIG-IP Local Traffic Manager	– managing at application services level rather than at individual devices and objects – scripting language that allows administrator to intercept, inspect, transform, and direct application traffic – built-in firewall protection, application security, and access control – real-time protocol and traffic management decisions	
Apache HTTPD	– support for session stickiness by using cookies and URL encoding. This approach [3] avoids unequal load distribution if clients are hidden behind proxies and stickyness errors when a client uses a dynamic IP address that changes during a session	– request counting algorithm – weighted traffic counting algorithm – pending request counting algorithm

Table 3.1: Comparison of load balancers

3.3. Vertical scaling

Essentially, vertical scaling is concentrated upon increasing capacity of single node. Again, when considering technical advancements that comes with cloud computing and virtualization, we can differ two categories of scaling: virtual machine resizing and virtual machines replacement. This distinction is dictated by limitation hypervisors - not all of them are able to resize virtual machine without shutting it down.

3.3.1. Virtual machine resizing

	Memory	**CPU**	**Disk**
KVM 1.2.0		– dynamic pinning CPU to a specific virtual machine (depending on underlying hardware)	– adding a disk to a LVM group
Xen 4.3	– changing the amount of host physical memory assigned to virtual machine without rebooting it – start additional virtual machines on a host whose physical memory is currently full, by automatically reducing the memory allocations of existing virtual machines in order to make space	– dynamic pinning CPU to a specific virtual machine (depending on underlying hardware)	– dynamic block attaching, adding a disk to a LVM group
VMware ESX 5.1	– hot-plugging memory, ex. using VMware vSphere	– hot-plugging CPU, ex. using VMware vSphere	– adding additional disks to existing virtual machine
OpenVZ (kernel: 042)	– configurable via user beancounters	– configurable via user beancounters	– configurable via user beancounters

Table 3.2: Comparison of hypervisors resizing capabilities

3.3.2. Virtual machine replacement

As it was highlighted in previous section, reasoning behind virtual machine replacement is that, in case when dynamic resizing is not possible, a new virtual machine with a desired configuration can be provisioned and replace the old one. Since this is a basic operation, all above-mentioned hypervisors supports this scenario as long as required resources are available.

4. Interoperability of clouds

This chapter introduces the notion of a hybrid cloud and explains its role in IT industry. On top of this deployment model, the concept of InterCloud is presented and elaborated with the emphasis on its application in ensuring scalability of users' services.

4.1. Introduction

From the perspective of a user of PaaS services it is vital that they are able to deploy seamlessly their applications using libraries, tools and services supported by the cloud provider[41]. Judging by such factors as the popularity of Heroku – currently one of the most popular PaaS providers which does not offer more advanced features which would enable management of the infrastructure underpinning the deployment platform, the fact that Microsoft added auto-scaling to its Azure platform as late as in June 2013, it is perfectly possible most PaaS users are satisfied with the current offers of their providers and do need another, more sophisticated functionalities. However, there are more complex applications and systems whose requirements regarding technology stack, availability and scalability are considerably more demanding. For such services there ought to be designed slightly specialized features that would require cooperation among different cloud providers.

4.2. Hybrid cloud

One can imagine scenarios in which customers of cloud services know their applications are vulnerable to sudden variations in demand and their responsiveness must be kept at the same level all the time. In such cases, they want them to scale dynamically according to current load or other predefined or manually specified metrics. What is more, in order to ensure high availability of their services, customers do not want to confine themselves to only one provider – in the best scenario they want their applications (or their logical parts, such as persistence layer) to be spanned across different providers and be able to cooperate with one another at the same time. Additionally, due to privacy concerns of the sensible data, companies are reluctant to put it in the public cloud storage. All these factors lead to the concept of a *hybrid cloud*[41] – the case in which the cloud is a composition of two or more distinct infrastructures which are unique entities, but there are technological means that make it possible to port data and applications among them.

4.2.1. Deployment models

The informal introduction to the concept of a *hybrid cloud* in the previous section requires a strict definition, but it is virtually impossible without defining other deployment models:

– Private Cloud – The provisioned cloud infrastructure is used exclusively by a single organization (that may consist of many business units) and may be owned, managed and operated by the organization or a third party.

– Public Cloud – The provisioned cloud infrastructure is used by general public and may be owned, managed and operated by a business, academic or government organization or some combination of them. It exists on the premises of the cloud provider.

– Community Cloud – The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy , and compliance considerations). It may be owned, managed , and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

Having defined those models, we can see that *hybrid cloud* can be placed among them and be defined as a model in which the provisioned infrastructure is a composition of two or more other infrastructures - *private*,*community* or *public*.

4.2.2. Current usage and trends

Cloud – clients' view

Before digging into the details of current usage and popularity of the hybrid model, it is worth discussing the general attitude of clients towards cloud computing. As the recent survey [20] shows, the major factor that prevents companies from adopting cloud solutions is their concern over security – in 2012 as much as 52% responders considered it as a main concern with the regard to cloud in general. However, the tendency is that more and more enterprises do not find it a major issue as in 2012 the number declined to 46%. Complexity related to the management of cloud components, Vendor lock-in, interoperability and reliability were among the most frequent obstacles to adoption in 2013 for they constituted 46%, 35%, 27% and 22.3% of responders' votes respectively. Total results are shown in the figure 4.1.

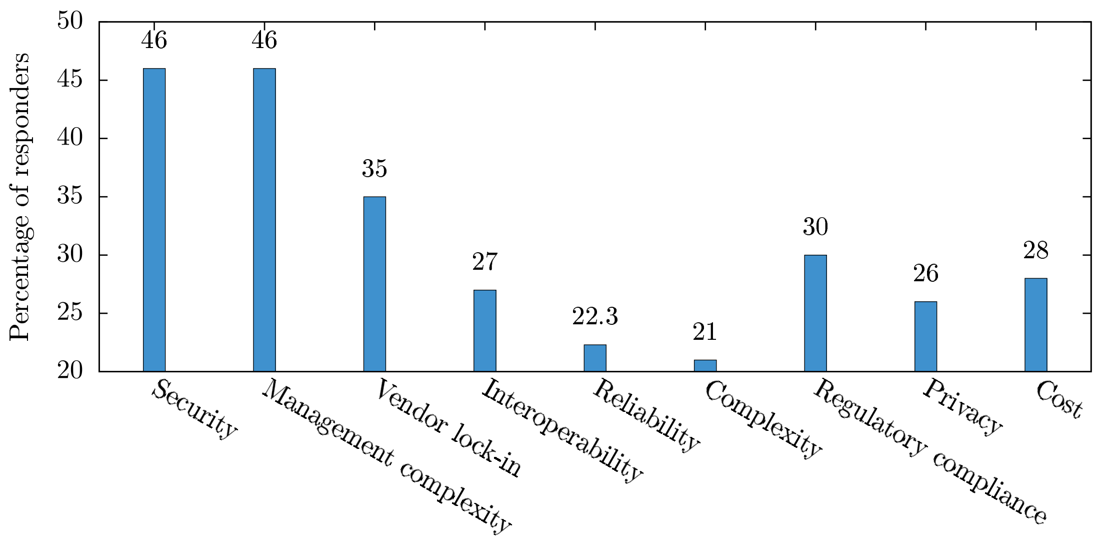

Figure 4.1: Major obstacles to cloud adoption in 2013 according to [20]

The same survey shows that the cloud adoption growth rate is high – 75 percent of responders stated usage of some sort of a cloud platform. This means 8 percent growth when compared to the results obtained in 2012. The expectations for the total worldwide addressable market for cloud computing are to reach $158.8B by 2014 – an increase of 126.5 percent from 2011.

View on hybrid cloud

When it comes to the application of a hybrid model in industry, in most cases the definition introduced in the previous chapter now becomes a 'public-private' composition. And this is how the term should be understood when discussing the results of the surveys which aimed to provide insights onto the view on a hybrid cloud from the customers' perspective. The study [20] forecasts 16 percentage growth in the hybrid cloud adoption in 5 years, from 27 to 43 percent. At the same time, the usage of a public model will decline from 39 to 32 percent. The other survey, conducted by Rackspace [21], provides more detailed data about current usage and popularity of a hybrid model. The first interesting finding is that as much as 60% responders, which included 1300 companies in the UK and US, have moved or are planning to move certain applications either partially (41%) or completely (19%) off the public cloud because of its limitations or the potential benefits of other models, e.g. the hybrid one. The second one is about the pros of adopting the hybrid cloud – potential users find more control (59%) and better security (54%) top benefits of using this deployment model. The other most responded benefits are shown in the figure 4.2.

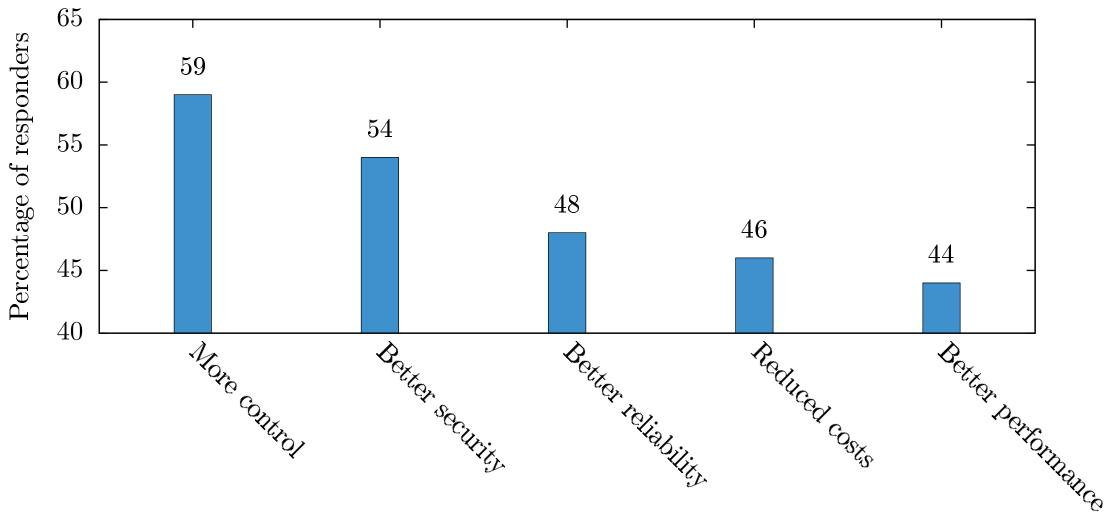

Figure 4.2: Top benefits of using the hybrid model according to [21]

4.3. Federation of clouds – InterCloud

As stated in the previous sections, one of the major obstacles that prevents consumers from adopting cloud solutions is reliability. It appears that these statements are not only imaginary worries of entrepreneurs, but real problems – there are cases, where some providers temporarily run short of capacity (e.g. because of provisioning too many virtual machines) in the face of hight demand [37]. What is more, the more demanding clients require specific QoS to be satisfied by their providers as negotiated in Service Level Agreements. In order to meet these challenges there is a need of a completely new approach to the problem of effective management of resources. The new solution should take into account such factors as:

– users' requests priority

– users' QoS requirements (e.g. the deadline by which some jobs have to be executed)

– price that the clients pay for the usage of resources

Computer scientists in the field of cloud computing devised a model [29] in which resources are managed in a market-oriented fashion that enables dynamic regulation of the supply and demand of resources and promotes the mechanisms for their allocation that would take into account their priorities and levels of utilization. The extension of this model is a vision of creating the federated cloud computing environment, so called *InterCloud*, that "facilitates just-in-time, opportunistic, and scalable provisioning of application services, consistently achieving QoS targets under variable workload, resource and network conditions" [28]. The elements of the proposed architecture are as follows:

– Cloud Exchange – acts as a market maker for bringing together both producers and consumers of services. It allows Cloud Brokers and Cloud Coordinators to match consumers with the fitting offers from providers. Such a market is a step forward towards creating a dynamic infrastructure for trading based on Service Level Agreements.

– Cloud Coordinator – manages the instance of a cloud and its membership in the overall federation; provides an environment (programming, deployment) for applications

– Cloud Broker – acts on behalf of the client; communicates with the Cloud Exchange to find the best cloud instances for the application

4.3.1. Usage in industry

The depicted model has not yet been adopted in the industry, yet some simulations were carried out on a *CloudSim* platform and the obtained results showed that this concept has "immense potential" [28].

5. Related work

This chapter presents recent achievements in a field of platform-as-a-service model by examining: Carina, OneFlow, CloudFoundry and OpenShift. Especially, it focuses on aspects of adaptivity, scalability and cloud federation awareness.

5.1. Requirements

One can notice that elements that yields a solution to a problem stated in the first chapter, which is ensuring that users' application provide appropriate Quality-of-Service for its customers in a most-cost effective manner, were gradually introduced in previous chapters:

– *adaptivity* - ability to adapt (i.e. scale) appropriately to a current usage pattern

– *scalability* - ability to improve application performance by enriching resources

– *cloud-federation awareness* - ability to compose an application deployment using different cloud providers; cooperation with different cloud provider to supply application with extra resources while performing application scaling

Next section states the general overview of the proposed solution, while the consecutive sections details its elements and finally the last section summarises the design choices in a context of system requirements.

5.2. Carina

5.2.1. Overview

Carina is an open source project, released under Apache License 2.0, built on top of OpenNebula, which aims to "(...) standardize the process for automating multi-VM deployments and setting auto-scaling and availability management policies in the cloud." [13]. The project is used by the authors at their work at RIM in an OpenNebula-based private cloud. As it is stated in the requirements of the solution, *Carina* should support variety of features which can be considered worth scrutinizing carefully as they are closely related to notions of adaptivity and scalability. To name the most relevant: [13]

– Collect and aggregate OS or app-specific metrics across a cluster

– Drive elastic scaling of clusters based on workload or events

– Support deployment and handling of failover of services across multiple datacenters

Before delving into more detailed description it is vital to introduce most important terms used in *Carina* documentation to depict this product:

– Environment – a collection of VMs in a master-slave configuration,

– Service – a consumer of cloud resources. Each service can have its own environment configurations and create environments and control them independently of other services,

– Pools – various clusters or virtual data centers in OpenNebula that can be targets for creating an environment

5.2.2. Scaling

Chapter 3 introduced and discussed various types of application scaling. *Carina* offers only one type of scaling and this is *horizontal* scaling – user's capabilities are limited only to add/remove a virtual machine to/from a given environment. Unfortunately there are no means to modify the parameters of a given or a set of virtual machines.

In this paper we strongly advocate spanning cloud resources across multiple vendors/providers. Despite having in its requirements such a position, *Carina* **does not** support deployment or scaling of services across different providers. In the environment configuration we define only **one** endpoint – this is can be thought of a reference to a cloud provider. As it can be only one per environment, it is not possible to deploy or scale some artifacts across many providers.

5.2.3. Adaptivity

In *Carina* there are mechanisms which can perform application scaling, both in manual and automatic fashion. It is possible for a system administrator to directly modify the existing application and its environment by changing its parameters manually. Automatic management of an environment is done by defining *scaling policies* which can be in two flavours: *time-based* and *load-based*. Those are called by the authors *elasticy policies*.

Time-based policies defines how the system should react in the given window frame(s). In each frame we can specify the minimum number of virtual machines that comprises an environment. On the other hand, *load-based* policies define the way the system reacts to **average cpu usage** by the environment. This is the simplest *threshold-model* as introduced in chapter 2. The user enters predicates which are evaluated against data gathered by OpenNebula and execute scaling actions (*scaleup* and *scaledown*) if they are true. What is more, the minimal and maximal number of virtual machines must be specified. At present the only parameter that is taken into account while performing actions triggered by *load-based* policies is cpu usage.

To increase the application availability, *Carina* introduces the notion of *availability policies*. Their main function is to take a recovery action in response to a deletion or errors during deployment of a virtual machine. Recreation of a virtual machine is the most fundamental and atomic operation under this model.

5.2.4. Design

To complete the discussion about *Carina* it is essential to include its architecture overview. This is shown in figure 5.1.

There are number of things that cause concern:

Persistency No one knows for what reason, but *Carina* uses two databases. What makes things even worse, they completely differ in types, as *MySQL* is a relational and *redis* is a nosql database. This places additional burden on technical requirements of the platform.

Usage of Apache To work properly, *Carina* needs Apache http server. In the server cgi-bin directory there are placed bash scripts which are invoked by contextualization scripts executed in different stages of virtual machines

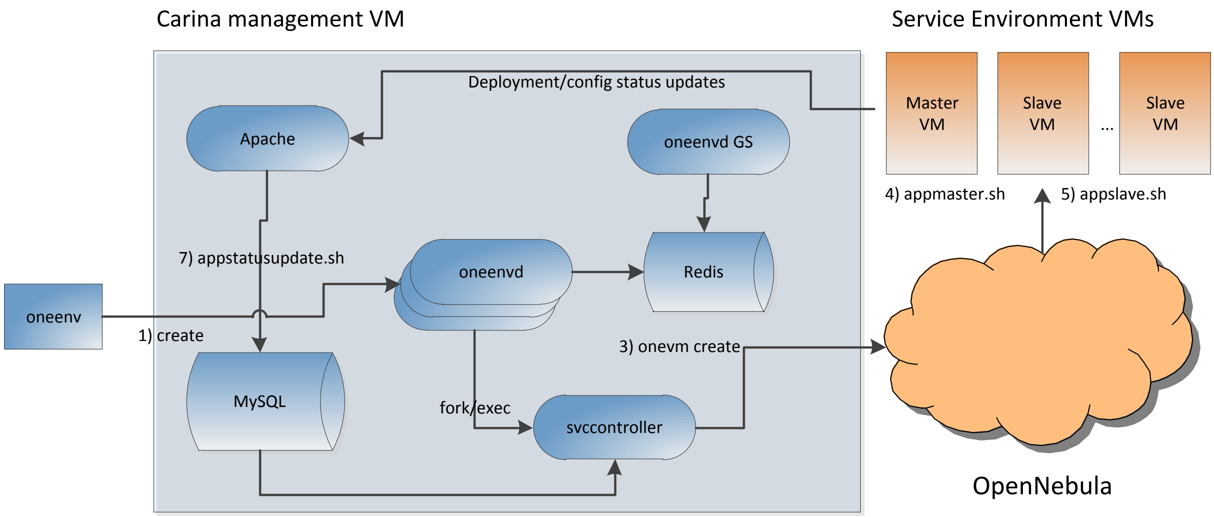

Figure 5.1: Carina – components interaction

life cycle. Instead, we recommend setting proper web services, environment-agnostic, in a central node which can be run next to global scheduler.

User accounts *Carina* forces the creation of a system-wide user for each service. These concepts should be totally independent.

Provisioning User has to use in the environment specification previously defined OpenNebula images. Their further adjusting happens in the contextualization phase. This forces the user to modify the state of a virtual machine when there is a need to perform any changes, for example in the software stack. Instead we suggest using some provisioning software such as puppet or chef – this would enable users to change software stack only in configuration files (recipes) not by altering the virtual machine state.

5.2.5. Summary

Carina emerged as a promising platform built on top of OpenNebula, which lacked most of its functionalities at that time. The good point is the fact that this product is used in a business context and judging by the authors' description it actually works. However, due to deficiencies mentioned in the previous sections, *Carina* cannot be considered a mature and generic solution for more demanding cloud providers.

5.3. OneFlow

5.3.1. Introduction

OneFlow [11] is a management system integrated into *OpenNebula* that aspires to enhance it with multi-tiered applications (service) deployment and auto-scaling capabilities. It is available as a core part of *OpenNebula* since 4.2 version and released under *Apache License, Version 2.0*.

5.3.2. Features

In a context of this thesis, *OneFlow* enriches *OpenNebula* with only one, yet powerful feature: application scaling based on policies. Besides this, benefits of using *OneFlow* are as follows:

– defining multi-tiered applications (services) as a collection of applications

– providing configurable services from a catalogue and self-service portal

– enabling tight, efficient administrative control

– fine-grained access control for the secure sharing of services with other users

5.3.3. Key technical concepts

Adaptivity

Adaptivity is achieved by specifying policies, that violated, trigger events impacting service environment. There are two types of policies available:

– *Auto-scaling based on metrics* - policy defines an expression that triggers scaling adjustments. Expressions can use performance data sent from virtual machine using *OneGate* as well as virtual machine data such as CPU, memory, network bandwidth

– *Auto-scaling based on schedule* - policy specifies a time or a time recurrence and correlates that information to appropriate service adjustments

Apart from scaling policies, scaling is controlled by a minimal and maximal number of virtual machines that can compose a service.

Scaling

OneFlow is capable of **only** horizontal scaling: it adds or removes virtual machine using *OpenNebula API*, when appropriate event is triggered. Violating given policy results in virtual machine's state transition into `SCALING`. After proper action is finished, virtual machines converges to a `COOLDOWN` state. What this simple state transition does is allows an environment to accommodate changes and adjust to a recent changes, disabling scaling for a given period of time.

Cloud federation awareness

OneFlow itself **does not** cooperate with different cloud instance. Hence, one is limited to *OpenNebula* hybrid or public cloud capabilities that employs oZones and *AWS*, respectively. Although, they expand single *OpenNebula* instance capacity it is limited in a way that discriminates it as possible solution. Firstly, the problem with oZones is that they require centralised management, while cloud federation is by definition distributed and independent. Secondly, cloud bursting into *AWS* limits instance monitoring capabilities. Consequently, it is not possible to supervise provisioned instances.

5.3.4. Summary

Taking everything into consideration, OneFlow is a valuable addition to OpenNebula ecosystem. Undoubtedly, its auto-scaling feature with elastic policy mechanism cannot be underestimated. However, actions taken by scaling mechanism are coarse-grained by being limited to a horizontal scaling. What makes things worse, OneFlow can not leverage resources of multiple providers associated within a cloud federation.

5.4. OpenShift

5.4.1. Introduction

OpenShift is developed by *RedHat* Platform-as-a-Service solution. It is available in two different flavours:

– *OpenShift Enterprise* (private cloud)

– *OpenShift Online / FreeShift* hosted by RedHat (public cloud)

The latter is freely distributed, but it is limited to only 3 gears (a 'gear' is a resource maintained by *OpenShift*, such as application server or database). Beside this, source code is hosted on github and licensed under *Apache License 2.0*.

5.4.2. Features

High level features that distinguishes *OpenShift* are as follows:

– accelerated application service delivery by on-demand and self-provision application stack access

– minimised vendor lock-in by portability (there are no proprietary APIs, technologies)

– automatic application scaling

Beside this, key qualities include:

– polyglot stacks: there is a number of built in application stacks (Java, Ruby, Python, databases); user is allowed to defined one himself through a concept of 'cartridge'

– one click deployment: application are managed by *OpenShift* through git repository. Deploying new version of application comes down to pushing new version application's source code.

– SELinux-based secure containers for multi-tenancy

– automatic application stack provisioning, it is only necessary to specify required cartridge (application stack) and all the dependencies are provided by platform

5.4.3. Key technical concepts

Adaptivity

OpenShift's adaptivity capabilities are constrained to a simple case: while creating a new application, it is possible to specify whether application should scale. As a consequence, additional gear is allocated to serve *HAProxy* [8]. There is only **one built-in** scaling policy that is based on a relatively straightforward algorithm [16]:

> *The algorithm for scaling up and scaling down is based on the number of concurrent requests to your application. OpenShift allocates 10 connections per gear - if HAProxy sees that you're sustaining 90% of your peak capacity, it adds another gear. If your demand falls to 50% of your peak capacity for several minutes, HAProxy removes that gear.*

Scalability

As previous point implies, OpenShift **solely** uses horizontal scaling: it adds additional gears if number of requests increases beyond certain level.

Cloud federation awareness

OpenShift does not use any notion of a cloud provider or datacenter. However, since cartridges are in fact Linux processes running on OpenShift Nodes, it is possible to deploy OpenShift Enterprise on any infrastructure running RHEL. It can consist of physical nodes as well as virtual machines, managed, for example, by OpenStack. With all that said, is still requires a centralised, OpenShift controlled environment. Hence, OpenShift **is not** capable to work in a cloud federation.

Multi-tenancy

In OpenShift, each application is represented as a set of gears. For example, gear is a database or application server. Gears are hosted on OpenShift Nodes. There is a many-to-one relationship between gear and OpenShift Nodes. What is leveraged to achieve this multi-tenancy is:

– SELinux, isolation between applications running at the same node

– control groups (cgroups), fine-grained control over the memory / CPU / IO utilisation / networking on per process basis

– kernel namespaces, groups of processes are separated, so that they cannot see resources in other groups

5.4.4. Summary

Summing up, key advantage of the OpenShift seems to be it simplicity: creating and deploying applications using OpenShift is as simple as it gets. Beside this, built in support for most popular technologies and simple API makes OpenShift attractive platform. Having that said, oversimplified auto-scaling and being constrained to a single cloud provider may be not sufficient in complicated scenarios.

5.5. CloudFoundry

5.5.1. Overview

Cloud Foundry is a platform-as-a-service solution which can be installed on local or off-premises infrastructure, such as Amazon Web Services, OpenStack or vSphere. It is available in two versions as

– an open-source project

– a commercial solution, *Pivotal CF*

In this section we want to elaborate only on an open-source project.

5.5.2. History

The project was initially developed by VMWare and its first release was in 2011, It was then that VMware decided to make it available to the general public and released it under Apache License 2.0. For the next consecutive two years, there were maintained two versions of *Cloud Foundry* – as a hosted solution owned by VMWare and an open-source project. However, in December 2012 VMware and EMC shared information on the *Pivotal Initiative* [19] – a virtual organization of people from both companies with background in *big data* and *cloud application platforms*. With the advent of the this new entity, *Pivotal* [18], it was announced the launch of the new product – Pivotal One, which is powered by *Pivotal CF*, the enterprise version of *Cloud Foundry*. Now users can choose between the open-source and commercial solutions.

5.5.3. Features

Below there are listed some of the main functionalities offered by *Cloud Foundry*:

- Support for multi-provider ecosystem (*Multi-Cloud*)
- Reduced the management burden of the life-cycle of an application by a simplified CLI that enables, e.g. faster and instant deployment,
- Dynamic routing which enables and enhances horizontal scaling and updating applications,
- Usage of own-developed lightweight containers (*Warden*), which ensures the proper isolation level and improves the speed of movement of application instances between virtual machines (utilizing *aufs* – advanced multi layered unification filesystem)
- Health management – continually monitoring application instances and taking according recovery actions if needed
- Standards-based authorization and authentication system which supports various standards such as LDAP, SAML or OAuth 2.0

5.5.4. Key technical concepts

Cloud Foundry-specific

Warden The main goal of *Warden* is to manage isolated and resource-controller environments called containers. It is responsible for management of the whole lifecycle of an environment and provides API for actually performing any operation on a container. At first, *Warden* used LXC technology under the hood, but now containers are an own product of *Cloud Foundry* community.

Droplet Execution Agent *DEA* manages *Warden* containers, stages user's application and runs droplets – wrappers around already deployed applications.

BOSH *BOSH* is a tool chain for release, deployment and lifecycle management of distributed services. Its functionalities can be compared to those offered by *Chef*.

Adaptivity

Cloud foundry does not support auto-scaling – it is only possible for the cloud consumer to scale the application manually. However, there can be additional tools which can add this functionality, such as *RightScale*.

In terms of monitoring the state of applications, the platform uses *Health Manager* to collect relevant data and then it notifies *Cloud Controller*, which in turn takes appropriate actions. This enables, for example, to spawn a new node if an existing one needs a replacement.

Scalability

Cloud foundry supports only horizontal scaling not only within one provider, but also across many. This feature is called *Multi-Cloud* in Cloud Foundry nomenclature.

Cloud-federation awareness

As it was stated in previous sections, Cloud Foundry is *cloud-federation aware*, i.e. it allows users to span their applications across many providers, which results in a hybrid solution that comes in many flavours: *public-public*, *public-private* or *private-private*.

5.5.5. Summary

Owing to the variety of useful features, great support from community and significant IT enterprises such as IBM, VMware to name a few, *Cloud Foundry* can be considered a mature product which can compete with current leaders in the fields, such as Amazon AWS. Additionally, it is highly probable that acquiring the hosted solution by *Pivotal* may be of a great benefit to the whole cloud community as the market will be more competitive.

5.6. Providers comparison

In this section we would like to compare all depicted providers as well as other solution in terms of *adaptivity* and *scalability*.

Table 5.1 presents a summary of cloud providers auto-scaling capabilities. Interestingly, all of them are focused solely on horizontal scaling, ignoring advantages offered by a fine-grained approach to scaling that leverage scaling up and application tuning.

	Horizontal scaling	**Vertical scaling**	**Application tuning**
Infrastructure provider			
Carina	✓	×	×
OneFlow	✓	×	×
AWS EC2	✓	×	×
Platform provider			
CloudFoundry	×	×	×
OpenShift	✓	×	×
AppEngine	✓	×	×
Azure	✓	×	×
Heroku	×	×	×

Table 5.1: Comparison of cloud providers scaling capabilities

On the other hand, table 5.2 is a summary with approaches to adaptivity taken by different cloud providers.

	Policies	Data analysis
Infrastructure provider		
Carina	– time frame based – expression based (only for CPU)	– threshold model that takes into account minimal and maximal permitted instances of an application as well as application priority
OneFlow 4.2	– time frame based with customizable padding – expression based build on custom language, where all vm's metrics are supported – customizable adjustment padding, cooldown time	– threshold model
AWS EC2	– time frame based – expression based, where expressions corresponds to a AutoScalingGroup – actions are triggered by a CloudWatch alarms – customizable adjustments paddings, types, cooldown time	– threshold model, takes into account minimal and maximal permitted instances of an application as well as application priority
Platform provider		
CloudFoundry	×	×
OpenShift	– single built-in policy	– single built-in threshold model that scales an application when CPU load is greater than 50% for a given period
AppEngine	– built-in policy based on request queue length – adjustable minimal, maximal number of application instances, pending latency	– queue-based, new instance is provisioned if queue length got too long
Azure	– time frame based – expression based, where expression can involve either CPU usage or Queue length – customizable adjustments paddings, types, cooldown time	– threshold model, that takes into account minimal and maximal allowed instances
Heroku	×	×

Table 5.2: Comparison of cloud providers approach to adaptivity

6. Cloud-SAP

This chapter introduces the reference architecture known as Cloud-SAP, highlights its core concepts and indicates possible implementation ideas.

6.1. Motivation

As one can see, latest advances in providing platform-as-a-service still do not yield a comprehensive solution that embraces a range of fine-grained actions relevant to a given situation in an environment, as it was summarised in previous chapter. All of the presented solutions solely engage horizontal scaling, while other IT processes and corresponding actions such as vertical scaling, cloud bursting or application platform restart remains neglected. What is more, none of the investigated systems is a self-managing one, taking passive approach in enforcing given QoS. To make matter worse, some platforms such as Carina or OneFlow are provider-specific, hence, requiring vast knowledge and time of entities responsible for its supervision.

What system has all of the above-mentioned features is an autonomic one [36], characterised by a self-management ability and driven by four crucial attributes:

– *self-configuration* - dynamic adaptation to changing environment such as provisioning new application instances, application tuning, traffic delegation to an external provider

– *self-healing* - discovering, diagnosing and reacting to environment disruption such as container outage

– *self-optimising* - monitoring and tuning resources: migrating containers, offloading traffic, for example

– *self-protecting* - detecting and protecting against threats such as Deny-of-Service by provisioning extra resources

Beside this, key customer values such as IT process execution cost or time needed to complete it are enhanced through rapid process initiation and reduced time and skill requirements [36]. As specified in blueprint, applying autonomic system to manage application environment is possible because following conditions are met:

These four system capabilities combined together yield a promising architecture from QoS ensuring perspective. Beside this, key customer values such as IT process execution cost or time needed to complete it are enhanced through rapid process initiation and reduced time and skill requirements [36]. As specified in blueprint, applying autonomic system to manage application environment is possible because following conditions are met:

– tasks involved in associated IT process needs to be automated

– it is possible to initiate processes based on observable and detectable situations

– autonomic manager posses sufficient knowledge

Cloud Self Adaptive Platform (Cloud-SAP) is a reference architecture aspiring to supply application providers with an autonomic computing environment. Figure 6.1 shows Cloud-SAP place in an exemplary application provisioning request.

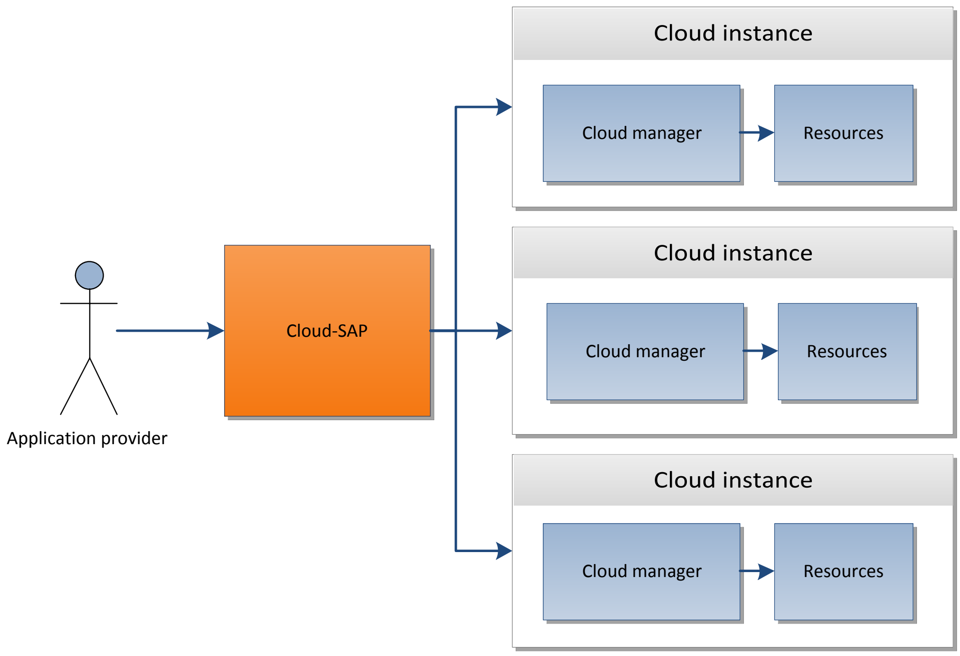

Figure 6.1: Cloud-SAP place in an exemplary application provisioning request

Application providers can be both end-users or other system interfaces, Cloud-SAP is suitable for both cases. Cloud-SAP is entirely cloud providers and resources agnostic as long as they externalise management interfaces.

Successive sections define core layers of a Cloud-SAP and give insight into possible implementation decisions, while the next chapter presents a proof-of-concept implementation.

6.2. Overview

The previous section dealt with all premises that led to a conclusion that the proposed architecture can rely heavily on the concept of an autonomic system. In this section the further elaboration on the solution is presented. Figure 6.1 showed the context which the architecture applies to. As one can see, Cloud-SAP can be seen as a proxy between the user and cloud instances and as a coordinator of the latter. Although cloud instance itself consists of a cloud manager and additional resources, in general, it is perfectly legitimate to consider it a resource as well.

Before the more detailed diagram of the solution is presented, it is essential to introduce another indispensable function that lies in the heart of any autonomic entity – its ability to detect an undesirable state and take appropriate actions. This is achieved by the presence of a *control loop* which collects data about the environment, analyzes it and takes all needed steps to recover the system. Not only can the attributes introduced in the previous section be bound to an autonomic system, but they can also be thought of main and broad categories of a control loop [36].

Having said so, we are ready to give a layered overview of a proposed solution, which is shown in figure 6.2.

The building blocks of the architecture are:

– Managed resources

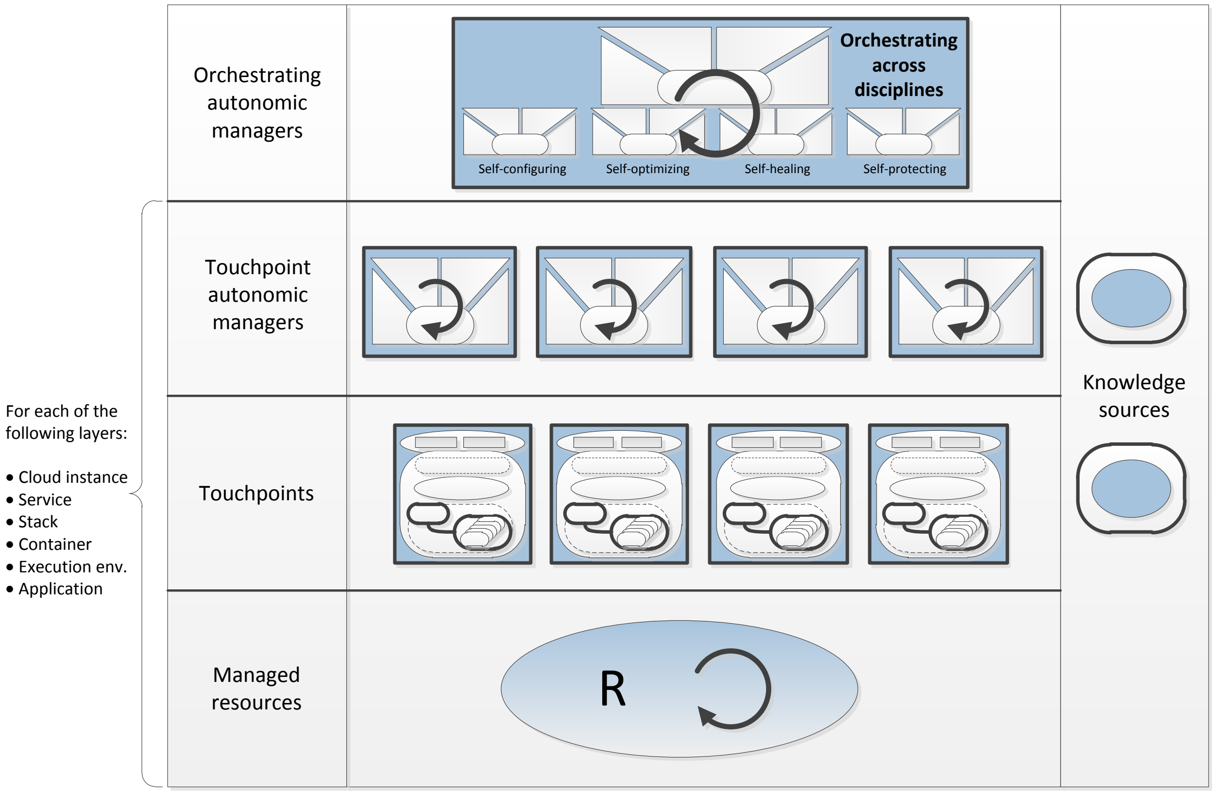

Figure 6.2: Cloud-SAP components

- Touchpoints
- Touchpoint autonomic managers
- Orchestrating autonomic managers
- Knowledge source

As resources were identified on various levels of abstraction, starting from the application that is to be deployed and ending on a cloud instance that aggregates different applications and services, we can think of the last three layers (managed resources, touchpoints and touchpoint autonomic managers) as a one logical component which can aggregate and manage other components. This recursive definition applies to identified managed resources described in the next section, apart from application as we consider it the most fundamental and indivisible resource.

6.2.1. Managed resources

Managed resources form the lowest layer of a stack. When it comes to application provisioning in a cloud environment we can think of the following resources:

- *Application* – application itself can be considered resources which can be managed by an external entity, e.g. by restarting/killing it,
- *Execution environment of an application* forms an inherent part of an application lifecycle and its proper tuning can greatly influence its performance,
- *Container* – an isolated and controlled environment in which applications are deployed,

– *Stack* – every application has a technology stack associated with it; what is more, if we confine ourselves to map one instance of a container with only one application, we can think of stack as a composition of application (and its execution environment) and a container,

– *Service* – more complex solutions/applications require interactions with various components that are not necessarily implemented using the same technology stack; service denotes a logical entity that consists of a list of stacks that should form one, indivisible component,

– *Cloud instance* – an entity able to deploy *services*

What is more, the listed resources are nested within one another and one's ability to work/run smoothly highly depends on the condition and state of the resource it forms part of. This is shown in figure 6.3.

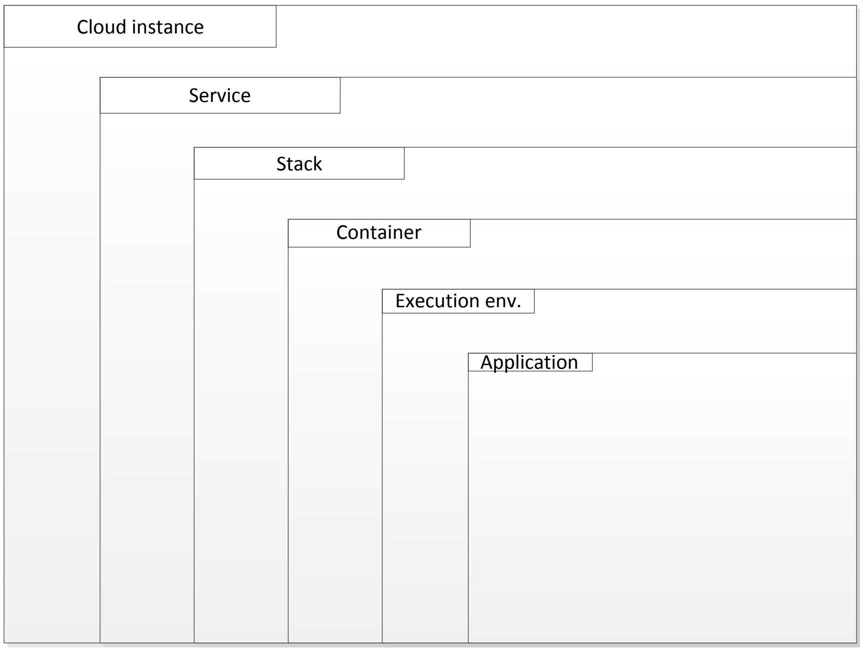

Figure 6.3: *Managed resources* identified by Cloud-SAP and their aggregation

As it is shown in figure 6.2, each *managed resource* can have embedded self-management control loop. The control loop may or may not be externally visible by the manageability interface. For example, an application can have *self-configuring* control loop which performs appropriate runtime tuning of its configuration files, while not being visible (by not providing any API) and manageable by any entity.

6.2.2. Touchpoints

Touchpoint forms the layer above managed resources. It has two main functions [36]: 1) provides manageability interface and 2) implements *sensor* and *effector* behaviour.

By means of manageability interface, external entities can control managed resources. It can be done by, e.g. leveraging API, configuration files or logs. Additionally, it is possible to obtain information via touchpoint about the state of the resource. *Sensor* and *effector* behaviours refers to mechanisms that allows collecting data and change the behaviour of the resource, respectively.

6.2.3. Touchpoint Autonomic Manager

Touchpoint Autonomic Manager is a component that implements the control loop behaviour and, in effect, manages the managed resources through exposed touchpoints. As it was stated in the previous sections, only four types of control loops are of an interest in the proposed architecture, i.e. *self-configuring*, *self-healing*, *self-optimising* and *self-protecting*.

The actions executed by each control loop on the assigned managed resource are defined in a *policy* that is associated with the given autonomic manager. Thus, there is a one-to-one mapping between managed resource and touchpoint, and touchpoint and autonomic manager.

Each autonomic manager implements the control loop which can be split into four consecutive blocks, which share knowledge among one another. The output from one block forms an input to another. However, it is perfectly possible for a block to perform an action for its side effects, e.g. an action can result in producing relevant information contained in log files, which, in turn, comprises a piece of shared knowledge. Each block implements different behaviour based on the abstraction level and resource they operate on (see figure 6.2). Nevertheless it is possible to define their generic behaviour, which embraces the following phases that correspond to a given block:

– *monitor* – this function of autonomic manager is responsible for collecting and aggregating data about the resource it manages,

– *analyse* – this part takes the data from *monitor* phase and perform in-depth analysis of it, which can result in, e.g. prediction plans. It is possible to incorporate more complex mechanisms, such as machine-learning, to fully use the obtained information for better prediction models,

– *plan* – this block is responsible for planning the appropriate action based on the output of the previous block and policies,

– *execute* – the goal of this function of autonomic manager is to execute the planned action

What is more, an autonomic manager exposes *sensor* and *effector* manageability interfaces which enables other entities (including other autonomic managers) to manage it in a similar manner as the component itself manages a resource. The image of an autonomic manager with its control loop and provided interface is shown in figure 6.4.

6.2.4. Orchestrating autonomic manager

While an autonomic manager which manages a single resource may be satisfactory enough in many business cases, there are situations when additional coordination among managers is essential to attain the certain goal. This can result in incorporating autonomic behaviour within an entity in a system-wide scope.

Such a coordination can be achieved by introducing another autonomic manager whose sole purpose is to harmonize the work of dependent ones by extensive use of their *sensor* and *effector* interfaces. These managers are called *orchestrating managers*.

When it comes to the complex task of ensuring the Quality of Service requirements for a client of a cloud computing environment, it is vital to ensure proper mechanisms that would allow seamless cooperation among different cloud providers. Such cooperation could result in a migration of an application between different clouds, market-driven choice of the cloud providers and so on.

As the exemplary actions can be regarded complex ones, they cannot be associated with actions taken by autonomic managers of only one type. Thus, an orchestrating autonomic manager should coordinate other managers of any type – an arbitrary mixture of self-healing, self-configuring, self-optimizing and self-protecting managers. Employment of these concepts in Cloud-SAP will be discussed later in detail.

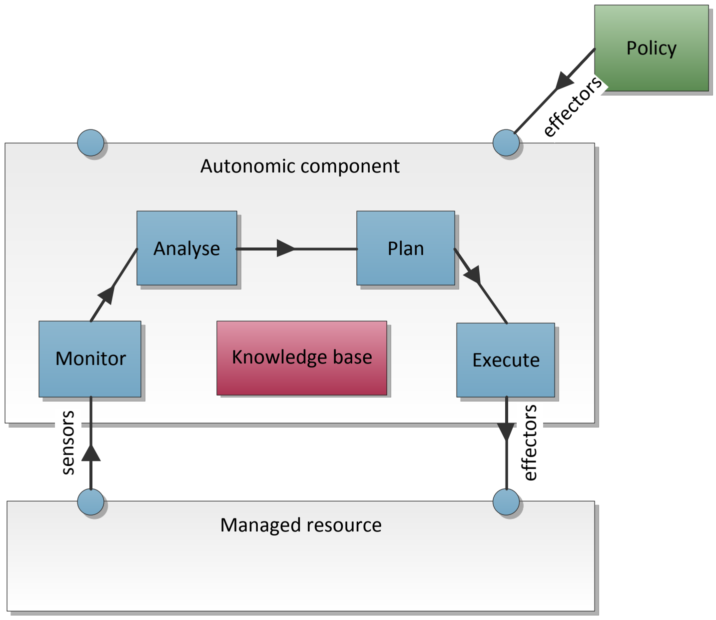

Figure 6.4: Autonomic manager

6.2.5. Knowledge source

Knowledge source provides access to knowledge stored, for example, in a registry, dictionary or database. It is recommended that knowledge source share knowledge among multiple managers, consequently extending their capabilities by performing additional tasks covering recognising particular symptoms or applying specified policies, for instance.

Whilst knowledge can be expressed as symptoms, policies, change requests or history logs, Cloud-SAP differentiates its two main types:

– *policy* - set of constraints that, when evaluated, influence system behaviour and possibly trigger further actions. Cloud-SAP does not specify policy format, however, it is vital to employ only one format in whole system so it can be freely exchanged

– *problem determination knowledge* - correlation of data, symptoms and decision trees determining actions that can be taken to eliminate symptoms

An autonomic manager can obtain knowledge in one of three ways:

– *receive it dynamically* - a common way for passing a policy to a manager.

– *retrieve it from external knowledge source* - an approach for obtaining symptom definitions or historical knowledge. For example, detailed history of a resource, its problems and actions taken can be retrieved from a log file.

– *create it dynamically* - monitoring and execution phase are strictly correlated and consequently data received from sensors, actions taken by effectors and their result can be stored as a knowledge and provide valuable information during future problem investigations

6.3. Autonomic application manager

6.4. Autonomic application platform manager

6.5. Autonomic container manager

It is expected that autonomic container manager supervise lifecycle of a container, modifying its properties accordingly to a given condition.

6.5.1. Managed resource

To recap, *container* is an entity that provides an execution environment for an application platform. There is a variety of technologies that intend to provide an isolated, secure execution environment. Table 6.1 groups them into three main categories: full virtualization, os-level virtualization, operating system process.

	Full virtualization	**OS-level virtualization**	**OS process**
Features	– complete simulation of machine's hardware – full isolation – host system is not shared among guests	– isolation is based on user-spaces – shared kernel – lesser overhead than full virutalization – effective i/o operations	– custom solution that uses processes, cgroups, SELinux, for example – lesser overhead possible
Examples	– KVM – Xen – VMware	– LXC – OpenVZ	– OpenShift containers

Table 6.1: Containerisation technologies

While choosing containerisation technique is entirely left to an implementation, Cloud-SAP recommends using os-level virtualization or operating system processes as they involve lesser resources and are more flexible, especially in terms of dynamic adjustment. Moreover, lightweight containers have been proved more effective than full virtualization in some scenarios: [43].

Not only can container be a managed resource but also it can have embedded control loop, depending on chosen mechanism. For example, Xen uses mechanism called 'memory ballooning' to intelligently distribute memory resources among guest systems what is in fact an example of self-optimising control loop. Though, embedded container loops are interesting they does not lie in Cloud-SAP area of interests and as a consequence are not further discussed.

Container uses variety of underlying resources such as storage or network connection. However, for simplicity, Cloud-SAP does not take them into consideration and therefore they are represented merely as a container properties not first-class resources.

Touchpoint

Container's touchpoint externalises hypervisor and operating system APIs. Similarly to all touchpoints, sensors for gathering data and effectors for influencing it can be distinguished. It is highly recommended for them to be linked, forming together manageability capabilities. In other words, property (i.e. CPU) can be read only when it is possible to set it as well. For instance, free / used CPU, free / used memory, network bandwidth, disk usage can be measured. Cloud-SAP requires free CPU to be at least supported.

6.5.2. Autonomic controller

Autonomic container controller aims to automate container's management function and externalise its configuration through its interfaces. In order to do achieve that, it implements a control loop that fully covers container life cycle: gathering information about it, analysing it, planning and executing actions on top of that. These four functions along with knowledge necessary to perform them designate modular structure of a controller as shown in figure 6.4.

Monitoring

As it was already mentioned, monitoring module aims to gather data from container's sensors. While Cloud-SAP does not have any specific requirements regarding underlying monitoring mechanism, there are a few aspects that should be taken into consideration:

Metrics rozne opcje: cpu, memory, co wybrac przynajmniej cpu

Data filters co robic z danymi, agregowac, jesli tak to jak, trzeba niwelowac zaburzenia (art? o monitoringu)

Persistence czy przechowywac dane, jesli tak to jakie, potrzebne do analizy, wyciagania wnioskow (dane, reakcja -> odpowiedz)

Standard compatibility Whilst there is no single, industry accepted standard of virtual machine monitoring, there were initiatives such as OCCI [10] that aims to close that gap. Having scaling across multiple cloud instances in mind, it is vital to provide compatibility at a container monitoring level. However, in some cases, standard-defined API may no be sufficient for a specific needs

Analysis

Planning

Execution

Knowledge

6.6. Autonomic stack manager

6.6.1. Managed resource

6.7. Autonomic cloud instance manager

6.8. Autonomic cloud federation manager

Autonomic cloud federation manager is a building block of the proposed architecture that forms the highest layer of its model (see figure 6.2). Additionally, it is an element which directly cooperates with a client by capturing and handling their requests.

This manager has two main responsibilities, which include 1) handling deployment of a service requests from the clients and subsequent requests directly applied to that service (e.g. queries about its state), 2) scaling the previously-deployed service across different cloud vendors. The former involves capturing the request from a client, determining the best cloud vendor or vendors for the described service and its deployment. This behaviour is strictly related with *plan* and *execute* functions of an autonomic manager. The latter, on the other hand, involves *monitoring* the dependant autonomic managers that manage in an autonomic fashion cloud instances, *analysing* obtained knowledge and deciding if taking any auto-scaling action is actually necessary to ensure satisfying Quality-of-Service requirements. If such an action is needed, it must be *planned* and *executed*.

The use case of scaling a service withing a federation of cloud vendors can be considered an action which coordinates cloud instance managers. It is perfectly legitimate to view the cloud federation manager as an *orchestrating* one. What is more, such a coordination requires cooperation with not only autonomic managers of a single type, but with the ones that are a mixture of self-healing, self-configuring, self-optimizing and self-protecting. Taking this into account, the orchestration can be classified as an *orchestration across disciplines* [36].

6.8.1. Structure

It is clear from the description of requirements of this autonomic manager in the previous section that *plan* and *execution* functions are common for both of them. Therefore the *cloud federation manager* is composed of two internal autonomic managers, which perform complete different functions, the first one – *monitor* and *analyse* and the second – *plan* and *execute*, yet they cooperate with each other to realize the full control loop.

To be able to handle incoming requests, the manager exposes API a client communicates with. The complete structure of this component is shown in figure 6.5. In the next sections a detailed description of each internal autonomic manager will be presented.

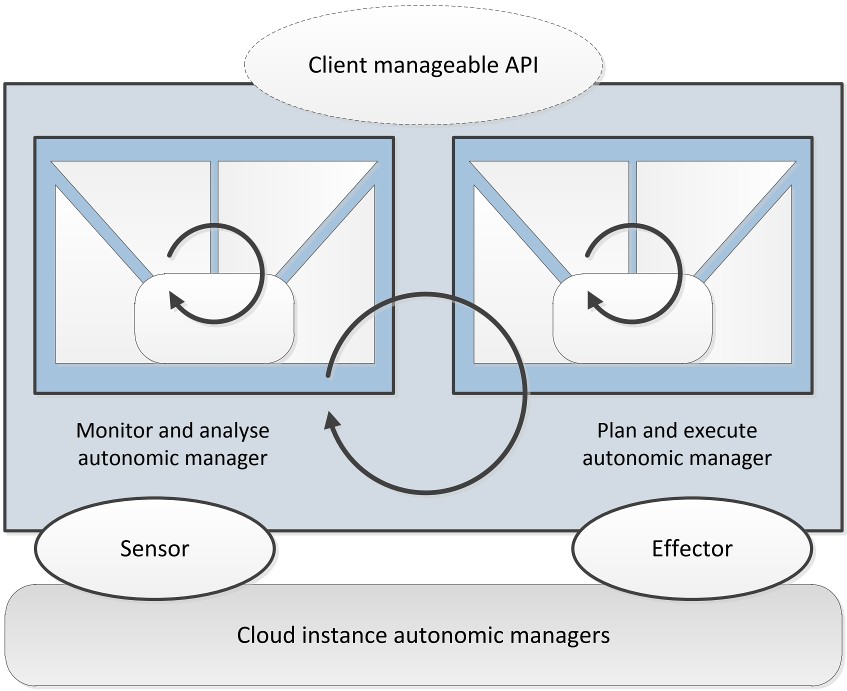

Figure 6.5: Design of an orchestrating autonomic manager

6.8.2. Managed resource

As it was stated in the previous sections and can be seen in figure 6.5, *cloud instance autonomic managers* are managed resources of this manager. Each cloud instance is a deployment environment for a service or a stack, i.e. it is possible to span the service across different clouds by deploying stacks which make up the given service to different providers. Of course it is legitimate to deploy the whole service to only one provider.

Touchpoints

As the cloud federation manager has to gain information about the state of a given cloud instance and be able to order the deployment of a stack or a service, the cloud instance exposes its manageability interface and its two types:

Sensors – set of attributes which forms the state of a given cloud instance. Attributes should include inter alia overall resources usage, pricing, geographical location. The more thorough elaboration will be held in the section devoted to *plan and execution manager*.

Effectors – set of stack/service management operations, such as deploy, migrate or delete.

6.8.3. Plan and Execution Autonomic Manager

The main purpose of this component is to select, based on knowledge provided directly to this manager from *monitor and analyze manager* or a user (i.e. the specification of a service), cloud providers for a given service and carry out the planned action. In the following sections a detailed description of each function is given.

Plan

The aim of this function is to give a detailed plan for the action to be performed. In the context of this dissertation the first output produced by this component, regarding the use cases of deploying a service and dynamically scaling it, is a *service deployment plan*, which is a mapping between available resources (cloud instances in this case) and concrete stacks making up the service. The second, definitely less complex, is a queries scheme about the state of the already-deployed services which is based on user's requests. In this case the component only propagates the request to the execute function. As this is only a mediation, there will be no further elaboration on this aspect.

Problem description The problem of resource management in cloud computing environments is hard and complex. This is because of several aspects. The first one is the characteristic of resources available in a cloud – not only are they geographically distributed, but they can be under different administrative domains and can vary in accessibility. This places a heavy burden on cooperation possibilities across cloud providers, both in a technical and non-technical way as providers need to maintain the high level of trust in order to effectively deliver its products. Additionally, different entities forms the resources – starting from individual devices, through virtual machines and whole workstations, ending with clusters and data centers. The second one are the increasing Quality-of-Service requirements from the clients. They want resources they use to be reliable, flexible (able to scale according to workload), fault-tolerant, energy-efficient and, of course, cheap. It is clear that clients and cloud vendors have different objectives and supply-and-demand patterns.
There are two main approaches to tackle the problem of effective resource management and scheduling:

- **conventional style**, in which the mapping decision is determined by some *cost function*. The downside of this method is the fact that in many cases the decision is a function of system-centric parameters [26] and because of it the outcome may lead to better utilization of cloud resources, not necessarily the user's application. What is more, this model treats all resources as if their cost was the same and applications are considered equally important what not always is the case.

  The positive side of this method is the fact that it involves only one entity that governs the scheduling and allocating the resources. What is more, it is up to the platform how many factors should be incorporated in such a function. Depending on the needs, a more sophisticated or simpler matching algorithm be applied, e.g. an algorithm that takes into account only available budget of a client.

  Legion [31] can be an example of a solution which uses this method.

– **economics-paradigm based** [25], in which the decision is not made statically by one entity, but is driven by the users' requirements. The model tries to adopt the real-world economic solution that involves markets and brokers on the computational ground. In this paradigm, clients (representing *demand*) want to solve their problems at the lowest possible price while assuring required Quality-of-Service requirements and constraints, such as time frame. Resource providers (representing *supply*) want to maximise the utilization of their resources while keeping their prices at the level that is attractive for other customers.

  Here are the most common economic models which can be applied to resource scheduling and management [26]:

  – the commodity market model;
  – the posted price model;
  – the bargaining model;
  – the tendering/contract-net model;
  – the auction model;
  – the bid-based proportional resource sharing model;
  – the community/coalition/bartering model;
  – the monopoly and oligopoly

Function input Depending on the actual request, this function can have access to different knowledge that makes its input:

Service deployment When the user wants to deploy a service on a cloud infrastructure, they have to prepare its detailed description containing technical (stacks) and Quality-of-Service requirements.

Service scaling When an application needs to be scaled, there is available additional knowledge that this component should use of – detailed information about service's performance up to that point, e.g. service resource utilization level (CPU, I/O), response time, its users profile including their geographical location and so on. This knowledge in a form of a prediction model should be passed to the component by the *analyse* function.

Function output As it was stated in the introductory section, this function produces *service deployment plan*. In the same section we conducted a quick survey of the most common approaches to the resource allocation and mapping problem. The proposed architecture places no restrictions on the chosen approach, as both paradigms can be fitted into it. In the case of *conventional approach*, this component should be equipped with a matching function that takes as an input service description (and the information of its so far execution in the case of an autoscaling action), offers from cloud providers and make a final decision based on these arguments. In the other case, this component can be thought of a cloud exchange [28], that brings together service producers and consumers. Autonomic cloud instance managers could be considered bid-makers and would compete with one another for the client. What is more, if the model is bid-based, the client can have implemented behaviour that would also allow them to participate in auctions. This could be done by introducing another component that would act in user's name.

Execute

This function takes either a query about the state of a service or a deployment plan of the given service and passes execution requests to the proper cloud instance autonomic managers.

6.8.4. Monitor and Analyse Autonomic Manager

Introduction

In many cases designers of applications cannot in advance foresee the detailed requirements of resources of their products, e.g. the number of virtual machines comprising it. This is especially true in case of *start-ups* when it is not known whether the product will catch public attention and who exactly matches the target audience. In such circumstances the cloud provider should be able to quickly adapt to new conditions to prevent the service from being unable to work properly, e.g. take into account the geographical location of the application's users and deploy a service in close proximity to them so as to minimize the latency.

As it was discussed in earlier chapters, the above-mentioned use case refers to *scaling* capabilities of a cloud provider. However, even if a cloud provider has *auto-scaling* mechanisms, there can be unacceptable delays in application response time because of the *reactive* nature of the feature – the system decides to scale the application based on the *current* resource consumption and utilization rate. Since it involves instantiating and provisioning virtual machines there is always a time overhead associated with it. Hence, there is a need that a system could *predict* demands and behaviours of a hosted service in a *proactive* way.

The goal of this component is to gather data about current performance of a deployed service and based on it perform an in-depth analysis which results in a resource usage prediction model and auto-scaling schemes. This is the objective of the *monitor* and *analyse* functions, respectively.

Prediction models

Monitor

Analyse

6.9. Summary

7. Implementation

In this chapter we outline implementation details about each component of the proposed solution.

7.1. Requirements

7.1.1. Functional

One can notice that elements that yields a solution for a problem stated in the first chapter, which is ensuring that users' application provide appropriate Quality-of-Service for its customers, were introduced in previous chapters:

- scalability – ability to improve application performance by enriching
- adaptivity – ability to adapt (i.e. scale) appropriately to current usage pattern
- inter-cloud awareness – ability to cooperate with different cloud provider to supply application with extra resources

Having those in mind, we can make the list of functional requirements more formal:

1. The user of the platform is able to:
   (a) deploy a service,
   (b) cancel the service,
   (c) check the status of the previously ordered-to-deploy service at any time. *Status* means a) whether or not the deployment succeeded, b) current uptime of the service, c) current cost
2. One of the elements of the platform is a client application that is used by the user of the platform to communicate with it,
3. During the deployment process, the platform takes as an input a description of the service (application) that consists of:
   - service name,
   - software stacks (e.g. *java*, *ruby*),
   - auto-scaling policies (per each stack) which define i) minimal and maximal number of VMs that are needed for the stack, ii) name of the policy (algorithm) which is used for scaling, iii) parameters of the policy
4. Deployment of a service is done in a way which minimizes the cost from the client's perspective with ensuring Quality-of-Service requirements at the same time,

5. It is assumed that the application which is going to be deployed is properly and fully tuned so that it is not possible to improve its performance by changing its or any of its components configuration(s),

6. The platform monitors the state of the deployed services and based on the results of this process takes appropriate steps in order to meet the auto-scaling requirements. These include a) altering VM's parameters and configuration, b) vertical scaling, c) horizontal scaling, d) scaling stacks among different cloud providers

TODO Alternative scenario – the client has a predefined budget that they cannot exceed – it can be mentioned in the overall discussion of the solution

7.1.2. Non-functional

– The platform uses *OpenVZ* as a hypervisor

– The platform uses *OpenNebula* and *AppFlow* as data-center management tools

– The platform does not confine itself to one provider, but to a *ecosystem of various cloud providers* that offers deployment capabilities which vary in terms of quality of service, cost, etc.

– All communication between the user and the platform and among platform components should be encrypted

8. Evaluation

This chapter contains results and discussions on the figuation of tests run on the proposed solution.

8.1. Introduction

We carried out a number of tests which aim to prove the solution be better than currently available, especially in terms of:

1. deployment cost
2. cost of providing given Quality-of-Service
3. deployment time

Hardware and virtual machines configuration

Many of the carried out tests were run on the same hardware and/or used the same virtual machines so for clarity of presentation their parameters are presented not in a description of every test, but here in table 8.1.

8.2. Cost of service deployment

Description

The aim of this test case is to test the primary use case of the proposed solution – deployment of a service with the emphasis of client's **cost**. It should show that the platform chooses the best mapping between the stacks and cloud providers so that the client's pays the **lowest** possible price.

Name	**VM**	**CPU**	**RAM**	**HD**
Desktop		Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz	8GB	1TB
Node1		AMD Athlon^{TM}64 X2 Dual Core, 2000MHz	3GB	160GB
Node3		AMD Sempron(tm) Processor 3000+	2.5GB	160GB
Node4		AMD Duron(tm) Processor 1GHz	2.5GB	60GB
Frontend{1,2}	✓	1 CPU	512MB	10GB

Table 8.1: Configuration of hardware/virtual machines used during tests

	CP-1	CP-2	CP-3
java	150	120	180
ruby	220	290	250
postgres	320	240	290
python	200	260	180
amqp	330	390	285

Table 8.2: Price for a stack in the given cloud provider

Preconditions

Service specification (A.2) forms an input to the application. Its elements are different software stacks that are parts of the whole service. Each cloud provider has its own price for a given software stack which is shown in table 8.2. Diagram 8.1 illustrates the simplified environment setup.

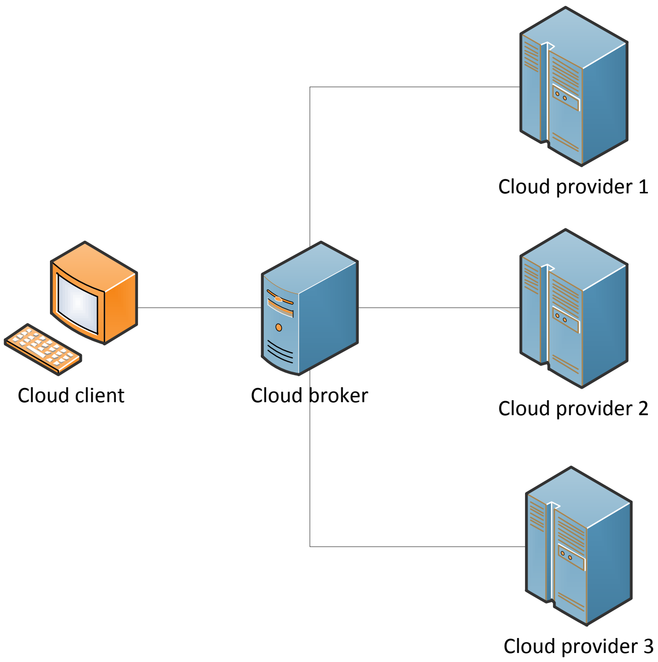

Figure 8.1: Deployment cost: environment configuration

Results

Table 8.3 shows obtained mapping between stacks and cloud providers. Taking into account this result, figure 8.2 shows comparison of cost the client would have to pay with and without such a mapping.

	CP-1	CP-2	CP-3
java		x	
ruby	x		
postgres		x	
python			x
amqp			x

Table 8.3: Chosen cloud providers for the given stack

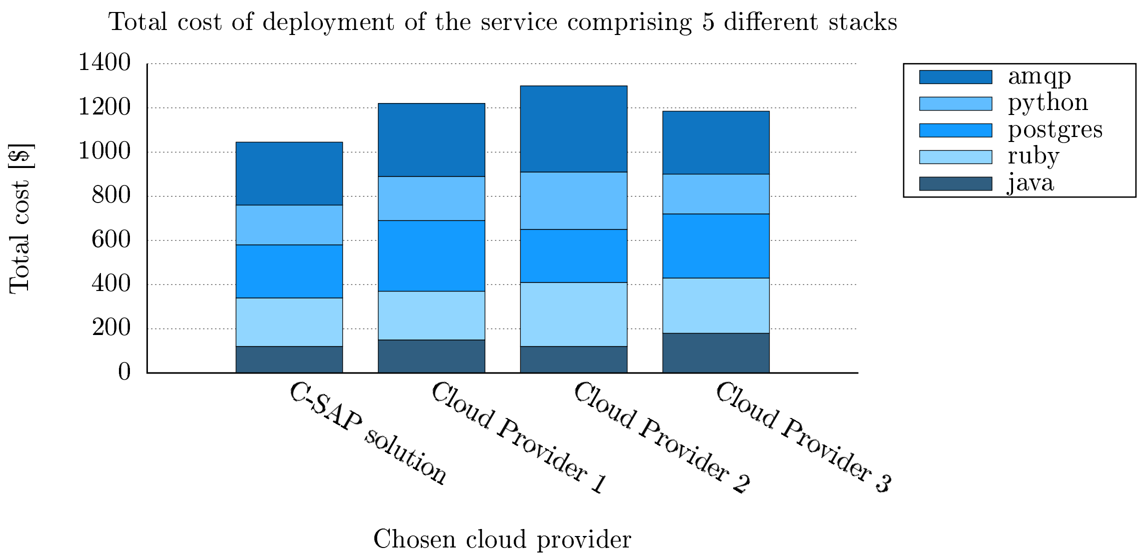

Figure 8.2: Comparison of the deployment cost when the service is deployed only on a selected cloud provider or a combination of cloud providers selected by Cloud-SAP

Conclusion

Obtained results clearly show that our proof-of-concept product met expectations of this test case as the client was offered the cheapest deployment scheme among various cloud providers for a given software stack. The mapping mechanism is simple yet considerably powerful – in this simple scenario the savings were significant since they constituted nearly 20 percent of the price offered by *Cloud Provider 2*. This shows that implementing similar solutions in the real world could be of great benefit to cloud consumers.

8.3. Auto-scaling – single-provider based

Description

Motivation The aim of this test-case is to show that introducing a multi-layered auto-scaling platform is of a great benefit in terms of cost and resource usage for cloud consumers and cloud providers respectively. What is more, this test should prove that once there are many levels on which scaling operations can be performed, there are significant reduction in costs paid by consumers. We want to prove it by comparing our proof-of-concept product to *Carina* [12]. *Carina* embraces a whole range of various scaling policies for users' environments (environment, in this context, means an application with all its software dependencies that are to be deployed on the cloud), but there is only one way in which they are executed – by managing the number of virtual machines (i.e. horizontal scaling). Quite on the contrary, *Cloud-SAP* has mechanisms that allow to scale the environment vertically in the first place and if that turns out to be not sufficient, horizontally. This test shows the influence of lack/presence of this feature on price and resource consumption.

Scenario The test scenario involves a) deployment of a sample environment on the cloud, b) substitute the real module responsible for collecting CPU usage date for a mock one, c) monitoring the scaling actions performed by each solution, d) figuation of the cost the client has to pay for the service. Deployment of a service is done in a product-specific manner (up to the point where the vm deployment request is passed to OpenNebula). Mocking the CPU usage was possible by replacing the part in *InformationManager* responsible the collecting CPU data in a host for a request to a web service which generated a time-based mocked-values. Choosing the appropriate function is another issue and is discussed in the next paragraph.

CPU usage function

We wanted to ensure that both solutions, *Carina* and *Cloud-SAP*, collected the same data regarding the CPU usage for the given time. What is more, the curve should resemble CPU usage in the real business scenarios as much as possible. Thus, we took into account the following factors:

- every peak in CPU usage should be followed by a gradual descent which would mimic the real auto-scaling actions executed by each solution,
- (boundary conditions) CPU usage should be between 0 and 100 and, additionally, should exceed the previously set scaling threshold value of each solutions so that it would trigger the scaling policies figuation
- once the scaling operations have completed, CPU usage should be constant at a rate which would not introduce any changes in the environment settings (such as the number of virtual machines and parameters of any virtual machine)

This resulted in the function whose formula is in (8.1) and plot in figure 8.3.

$$\begin{aligned} f(x) &= -\frac{31}{264}x^3 + \frac{3}{8}x^2 + \frac{1751}{132}x \\ CPU\_usage(t) &= \begin{cases} f(t) & 0 \quad \le t < 11.583 \\ f(t-10) & 11.583 \quad \le t < 21.441 \\ 25 & 21.441 \quad \le t \end{cases} \end{aligned} \tag{8.1}$$

Because *Carina* is capable of only horizontal scaling and *Cloud-SAP* of both horizontal and vertical, one can question whether the descent in CPU usage can be actually the same in both cases. To answer this question we want to look into the description of the virtual machines comprising the deployed environment. It states that each VM can use up to 30% power of the CPU of a host. Thus, we can be fairly sure that adding another virtual machine

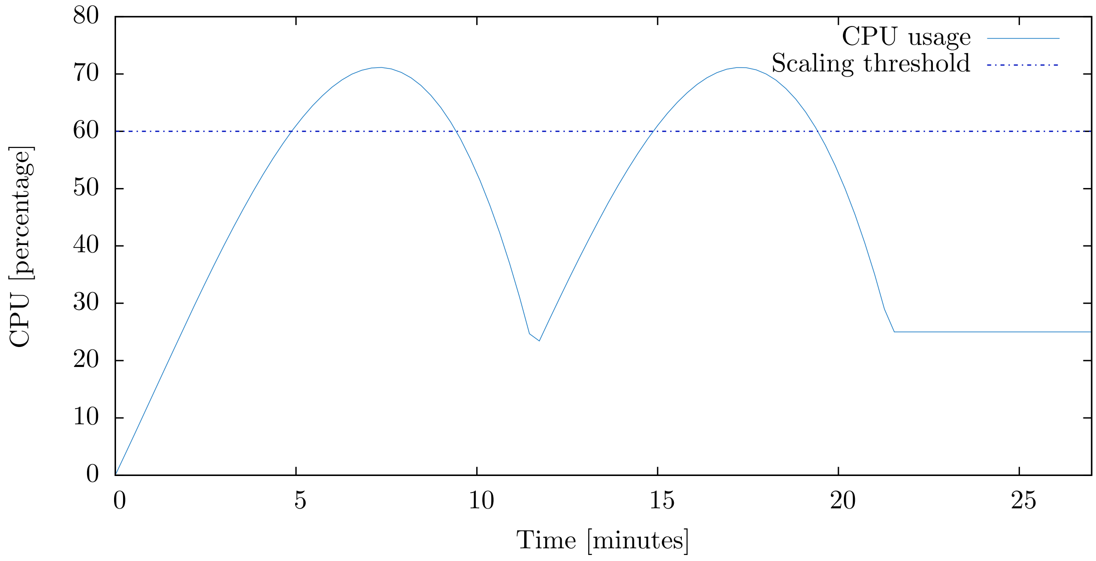

Figure 8.3: Auto-scaling - single-provider based: CPU usage function

that uses 0.3 CPU capacity of its host is equivalent with changing the CPU usage settings of already deployed VM from 30 to 60%.

Preconditions

Environment specification

Auto-scaling policy specifications As it is shown in figure 8.3, we set in both products the threshold values of CPU usage that trigger performing auto-scaling actions to 20 and 60 percent. To apply this setting, the auto-scaling part of service specification looks as follows: in Carina the user has to add the parameters of an auto-scaling policy in a hash that describes the environment under key *:elasticy_policy*. It is possible to specify the minimal and the maximal number of virtual machines that forms the environment and, what is most important, expressions which figuation results in scaling the application. The part responsible for this is shown in listing 8.1.

Listing 8.1: Carina service specification used for testing auto-scaling with 1 cloud provider

```
ENVIRONMENT = {
  'testenv' => {
    ...
    :elasticity_policy   => {
      :mode => 'auto',
      :min => 2,
      :max => 4,
      :priority => 10,
      :period => 2,
      :scaleup_expr   => 'avgcpu > 60',
      :scaledown_expr => 'avgcpu < 20'
    }
  }
}
```

In Cloud-SAP it is a matter of setting appropriate arguments to a specific policy. In this case the policy is *threshold_model* and values are 20 and 60 for lower and upper bound respectively. The auto-scaling part from service specification is shown in listing 8.2.

Listing 8.2: Cloud-SAP service specification used for testing auto-scaling with 1 cloud provider

```
{
  ...
  "policies":[
  {
    "name":"threshold_model",
      "arguments": {
        "min":"20",
        "max":"60"
      }
  }
  ]
  ...
}
```

OpenNebula/Carina/Cloud-SAP configuration Since Carina and Cloud-SAP used two different instances of OpenNebula installed on separate virtual machines, it is essential the configuration be the same on each of them. Listing 8.3 shows the most important OpenNebula excerpt from settings file used in this test case – polling interval specification, which was set to 30 seconds. To ensure that both products can actually use up-to-date data, they should figuate their policies every 30 seconds or longer. In Carina the user cannot specify this value, because it is hard-coded in source code to 60 seconds. In Cloud-SAP, however, this value was set in a configuration file and was equal to 60 seconds as well as in Carina.

Listing 8.3: OpenNebula configuration excerpt – information manager

```
HOST_MONITORING_INTERVAL = 300
HOST_PER_INTERVAL        = 15
VM_POLLING_INTERVAL      = 30
VM_PER_INTERVAL          = 10
```

. Vertical scaling mechanism in *Cloud-SAP* is implemented in a way that causes exponential growth of CPU resources consumption. When there is a need to perform a scaling action on CPU usage, current value is taken and increased by 30%.

Deployment diagrams Deployment diagrams show the placement of each component in both installations. They are shown in figures 8.4 and 8.5.

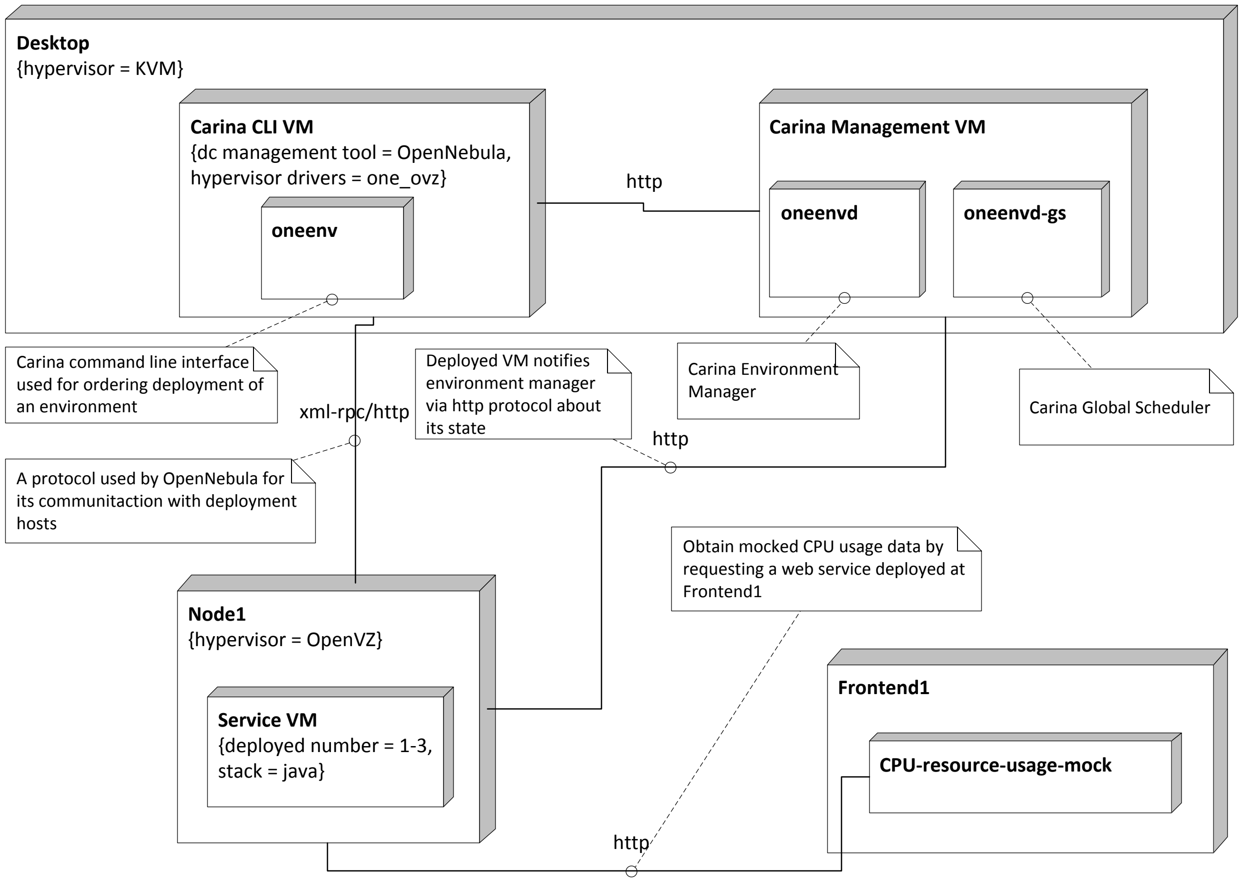

Figure 8.4: Auto-scaling - single-provider based: deployment diagram of Carina

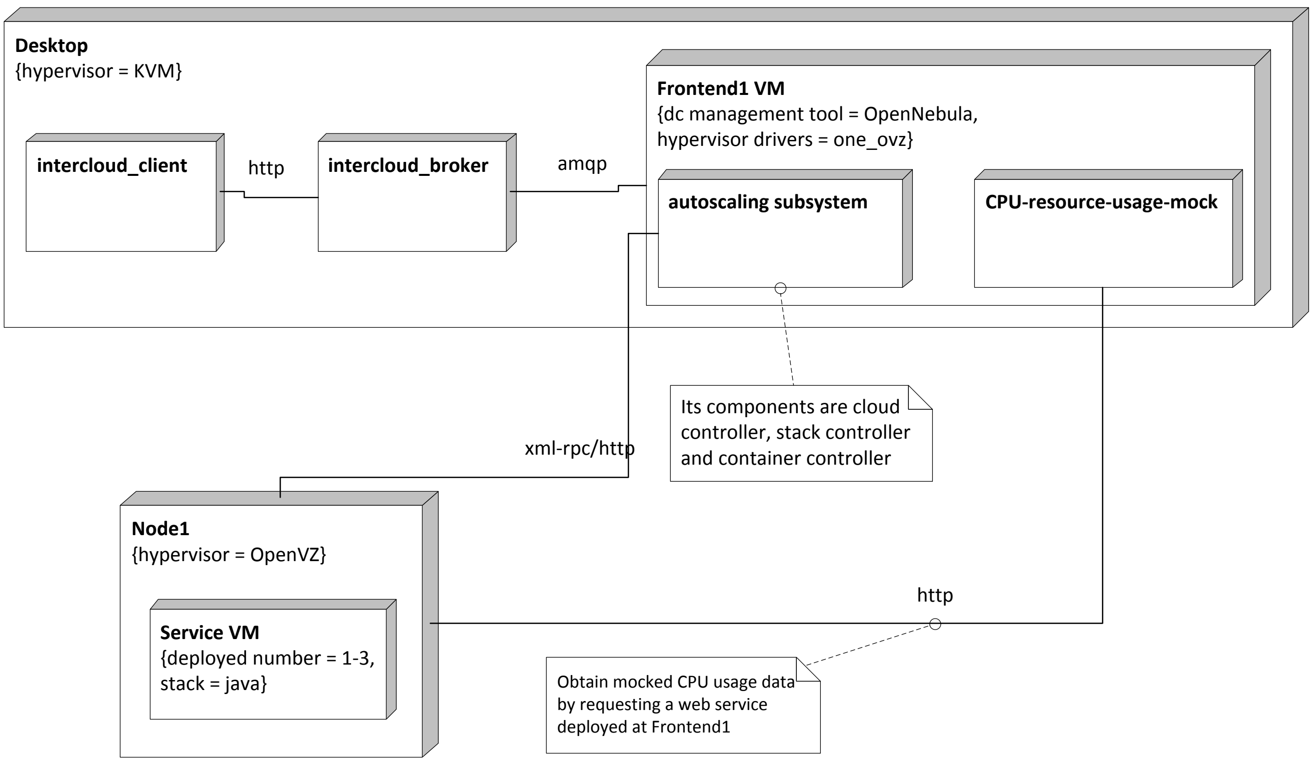

Figure 8.5: Auto-scaling - single-provider based: deployment diagram of Cloud-SAP

Name	CPU	RAM	HD
(virtual machine) carina-frontend	1 CPU	1GB	6GB
(virtual machine) carina-management-vm	1 CPU	1GB	9GB

Table 8.4: Configuration of hardware/virtual machines used during tests

Hardware/VM configuration

All virtual machines were deployed onto *Node1*, which uses *OpenVZ* as a virtualization technology and whose hardware configuration can be found in table 8.1. *OpenNebula* was installed on a *Frontend1* virtual machine and its configuration is in the same table. *Carina* required the usage of 2 virtual machines which were deployed on *Desktop* with KVM as a hypervisor and whose configuration is in table 8.4. Diagram 8.6 illustrates the physical setup of the test-case.

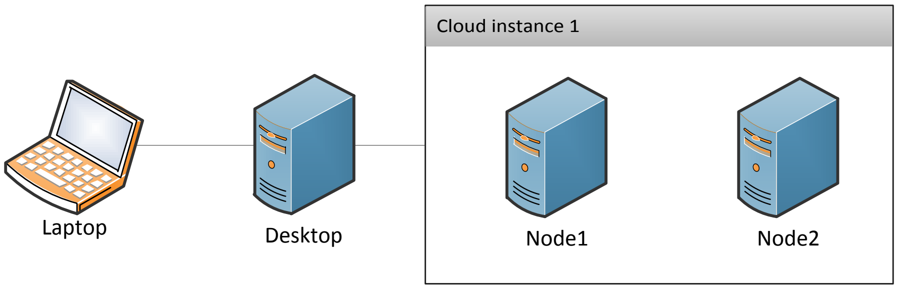

Figure 8.6: Auto-scaling - single-provider based: environment configuration

Expectations

As auto-scaling behaviour of both products is determined by CPU usage of the deployed service, we can predict the outcome of the test by thorough analysis of the plot in figure 8.3.

The first conclusion is that all auto-scaling actions are performed after exceeding the upper threshold value of CPU usage, this is 60 percent in our case what happens about 5 minutes after the deployment of an environment is completed. CPU consumption remains high for about 2 minutes what enables both products to take appropriate, auto-scaling steps. We expect *Carina* to deploy another virtual machine and *Cloud-SAP* to scale vertically. Then it gradually goes down, which simulates that all taken actions have successfully completed. Then, for another several minutes, the cycle recurs, but this time we expect the proposed solution to scale horizontally. *Carina* is expected to behave as in the previous cycle. Once the cycle has completed, after roughly 21 minutes, the value of CPU consumption remains constant at 25 percent and the test is completed.

Results

In the first place it is worth discussing the results of *Carina* and *Cloud-SAP* separately and then compare resource consumption of each solution and its influence on cost paid by the end-user.

Cloud-SAP Our proof-of-concept solution behaved exactly as expected. To make the discussion more clear, there is an excerpt from a log file of this test run in listing 8.4.

Listing 8.4: Cloud-SAP logs excerpt regarding auto-scaling actions

```
...
[2013-12-31T15:25:45.05] DEBUG : Notifying process 2736 about the deployed service
...
[2013-12-31T15:31:45.98] DEBUG : CONTAINER Concluded that currently {"id":1,"correlation_id
    ":875,"ip":"192.168.0.100","type":"master","probed":"1388500305","requirements":{"cpu
    ":0.3,"memory":512},"stack_id":1} is insufficient (by key: CPU)
[2013-12-31T15:31:45.98] DEBUG : CONTAINER Attempt to scale CPU up for a container: {"id
    ":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed":"1388500305","
    requirements":{"cpu":0.3,"memory":512},"stack_id":1}
D, [2013-12-31T15:31:46.78] DEBUG : Prepared payload for CPU increase: {"cpulimit"=>39} for
     {"id":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed
    ":"1388500305","requirements":{"cpu":0.39,"memory":512},"stack_id":1} at node1
...
[2013-12-31T15:32:46.15] DEBUG : CONTAINER Attempt to scale CPU up for a container: {"id
    ":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed":"1388500365","
    requirements":{"cpu":0.39,"memory":512},"stack_id":1}
[2013-12-31T15:32:46.94] DEBUG : Prepared payload for CPU increase: {"cpulimit"=>50} for {"
    id":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed":"1388500365","
    requirements":{"cpu":0.507,"memory":512},"stack_id":1} at node1
...
[2013-12-31T15:33:46.32] DEBUG : CONTAINER Concluded that currently {"id":1,"correlation_id
    ":875,"ip":"192.168.0.100","type":"master","probed":"1388500425","requirements":{"cpu
    ":0.507,"memory":512},"stack_id":1} is insufficient (by key: CPU)
D, [2013-12-31T15:33:47.12] DEBUG : Prepared payload for CPU increase: {"cpulimit"=>65} for
     {"id":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed
    ":"1388500425","requirements":{"cpu":0.6591,"memory":512},"stack_id":1} at node1
...
[2013-12-31T15:43:50.27] INFO : STACK Delegating execution to a cloud-controller
[2013-12-31T15:43:50.28] INFO : Received request of insufficient_slaves to be performed on
    a stack #<AutoScaling::Stack @id=1 @correlation_id =628 @type=:java @state=:deployed
    @data=nil @service_name="Auto-scaling test">
...
```

Carina As *Carina* is capable only of horizontal scaling, in discussion of its resource usage we can confine ourselves to counting the number of deployed virtual machines in the given time intervals. In the log files we can trace all actions that were triggered during the test case. As listing 8.5 shows, they completely met the expectations expressed in the previous section – there were two "SCALEUP" jobs which resulted in increase of the total number of virtual machines comprising the environment.

Listing 8.5: Carina environment manager logs with taken actions (*jobs*)

```
ID   ENVID     CONFIG_NAME      TYPE      SUBMIT_TIME                  STATUS
--   -----     -----------      ----      -----------                  ------
206     12         testenv      CREATE    Tue Dec 31 14:38:23 +0000 2013 DONE
207     12         testenv     SCALEUP    Tue Dec 31 14:48:07 +0000 2013 DONE
208     12         testenv     SCALEUP    Tue Dec 31 14:58:07 +0000 2013 DONE
```

. Since the base configuration assumed that the environment consists of 2 virtual machines, we can say that about 10 minutes after deployment of an environment, the number of VMs increased to 3, and after another 10 minutes, to 4.

Deployment finished at 15:25:45 and at this time we started the mock-cpu-usage web service. When the platform concluded that there are insufficient resources, it tried for three times changing parameters (virtual CPU used by the vm) of the virtual machines forming the service. Once scaling vertically was not possible such an information was passed to cloud controller which performed scaling across different cloud providers.

Cost

In our scenario cost is roughly equivalent with the CPU usage, so the plot of a cost virtually presents the cpu usage. Figure 8.7 shows cpu usage by environment when deployed and configured by two competing products. If

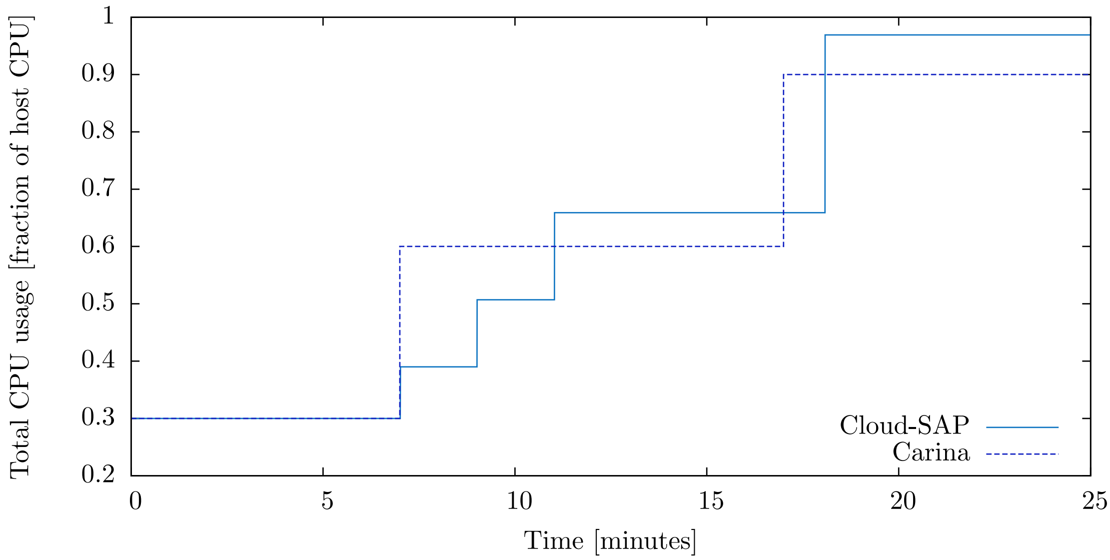

Figure 8.7: Auto-scaling - single-provider based: comparison of cost/CPU usage

we were to roughly estimate the cost in both solutions it would be equal in Carina:

$$7 \cdot 0.3 + 10 \cdot 0.6 + 8 \cdot 0.9 = 15.3 \quad \text{[currency unit]} \tag{8.2}$$

and in Cloud-SAP:

$$7.0166 \cdot 0.3 + 2 \cdot 0.39 + 2.01667 \cdot 0.507 + 7.05 \cdot 0.659 + 6.91667 \cdot 0.959 \approx 15.186483 \tag{8.3}$$

Conclusion

The obtained results shows that the proposed solution has enormous potential in terms of better utilization of available resources of the cloud and reducing cost paid by cloud consumers.

Judging by the plot of cost (figure 8.7), we can be fairly sure that it is possible to obtain better results by proper tuning of the scaling vertical mechanism. In particular, one can consider changing the default CPU growth from 1.3 to other or apply a different strategy, for example a growth by a constant rate. What is more, in order to get the best results, the policy scaling interval must be chosen with a great care with the consideration of specific parameters of a given environment.

8.4. Auto-scaling – multiple-provider based

Description

Motivation

This test case aims to prove that scaling across multiple cloud providers is vitally important while ensuring appropriate Quality-of-Service, especially in cases of increased number of service requests. Such scaling scenario, known also as cloud-bursting, leverage benefits arising from offloading application load to an external provider. In order to verify our concept we compare number of transactions per second guaranteed by *Cloud-SAP* and *Carina*. While our proof of concept solution features multi-layer scaling and is cloud federation aware, *Carina* adopts only horizontal scaling. It is expected that test case prove benefits emerging from enriching application platform provider with an cloud federation awareness.

Scenario

Testing requires following steps to be done:

1. deploy exemplary service
2. observe system behaviour under load, simulated by a resource usage mock. It is vital for a resource usage to exceed a single provider capabilities
3. assess system characteristics, including number of handled transactions per second

Load simulation

Similarly to a previous test case, we leveraged a resource mock that allows us to precisely control monitoring information returned to an system. Besides, in order to simplify test case, we were solely focused on a CPU usage. CPU usage function has to exceed upper threshold limit at some point and stay at that level, triggering successive auto-scaling events. However, at some point single cloud provider resources will be surpassed. Equation (8.4) denotes a resource usage function that is expected to fulfil above-mentioned scenario and is illustrated in figure 8.8.

$$f(x) = \frac{29}{1500}x^3 - \frac{21}{20}x^2 + \frac{533}{30}x \tag{8.4}$$

Preconditions

Environment specification

Auto-scaling policy specifications Policies used in this test are based on a threshold model with the following properties:

– value below 20 triggers scaling down event

– value grater than 60 triggers scaling up event

However, considering the fact that we are entirely focused on scaling up, only the upper limit is relevant in our case. Listings 8.6 and 8.7 presents scaling policies for *Carina* and *Cloud-SAP* respectively.

Listing 8.6: *Carina* service specification used for testing auto-scaling with 2 cloud providers

```
ENVIRONMENT = {
  'testenv' => {
    ...
```

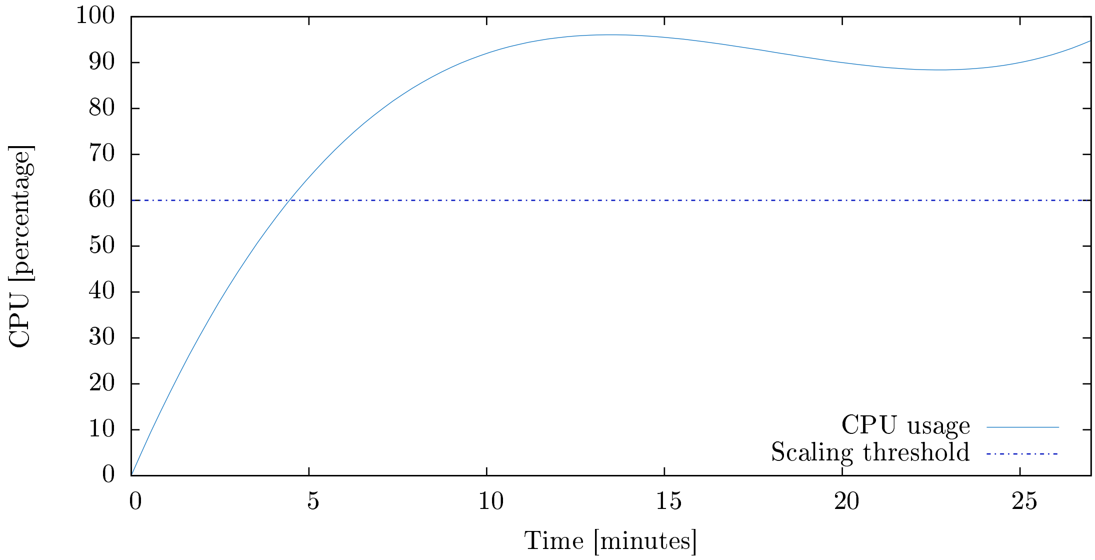

Figure 8.8: Auto-scaling - multiple-provider based: CPU usage function

```
      :elasticity_policy   => {
        :mode => 'auto',
        :min => 2,
        :max => 10,
        :priority => 10,
        :period => 2,
        :scaleup_expr   => 'avgcpu > 60',
        :scaledown_expr => 'avgcpu < 20'
      }
    }
  }
```

Listing 8.7: ***Cloud-SAP*** **service specification used for testing auto-scaling with 2 cloud providers**

```
{
  ...
  "policies":[
  {
    "name":"threshold_model",
      "arguments": {
        "min":"20",
        "max":"60"
      }
  }
  ]
  ...
}
```

OpenNebula/Carina/Cloud-SAP configuration Components common to *Carina* and *Cloud-SAP*, namely *OpenNebula* instances and computing nodes, were configured in the same fashion. Listing 8.8 depicts key configuration elements of *OpenNebula* monitoring mechanism.

Listing 8.8: OpenNebula configuration excerpt – virtual machine and information manager

```
HOST_MONITORING_INTERVAL = 300
HOST_PER_INTERVAL        = 15
VM_POLLING_INTERVAL      = 30
VM_PER_INTERVAL          = 10
```

Deployment diagrams Components that took part during testing are show in figures 8.9 and 8.10.

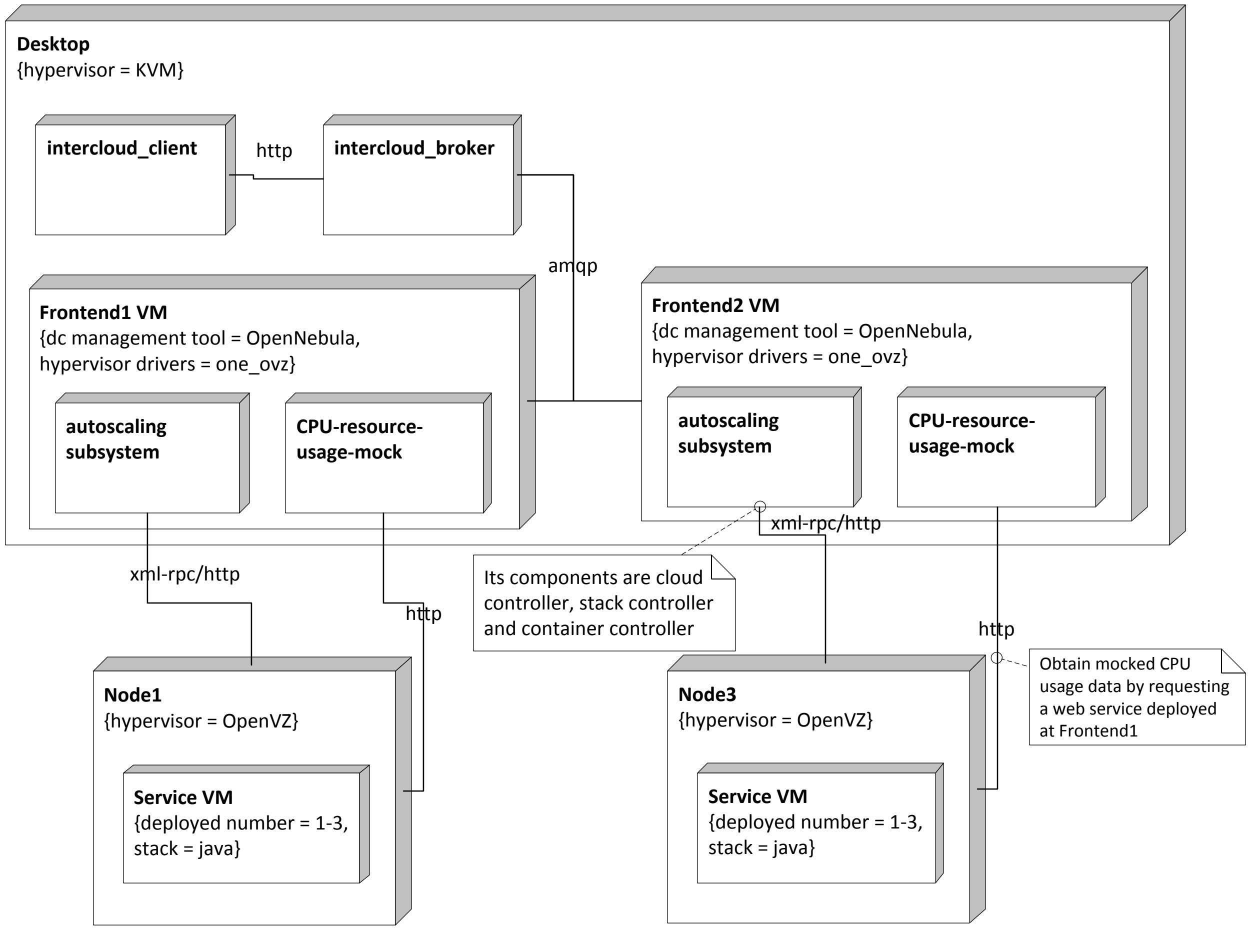

Figure 8.9: Auto-scaling - multiple-provider based: deployment diagram of *Cloud-SAP*

Hardware/VM configuration

Figure 8.11 depicts physical configuration of the environment. In short, setup was as follows: all *Cloud-SAP* components, apart from cloud client deployed on *Laptop*, were provisioned on *Desktop*. *Cloud Proivder 1 (CP-1)* is *OpenNebula* instance known as *Frontend1* which uses *Node1* as a computing node and is deployed on *Desktop*, while *CP2* uses *Frontend2* and *Node2*. Specification of nodes is listed in table: 8.1. Deployment cost of a java stack was 50 and 60 using *CP-1* and *CP-2*, respectively.

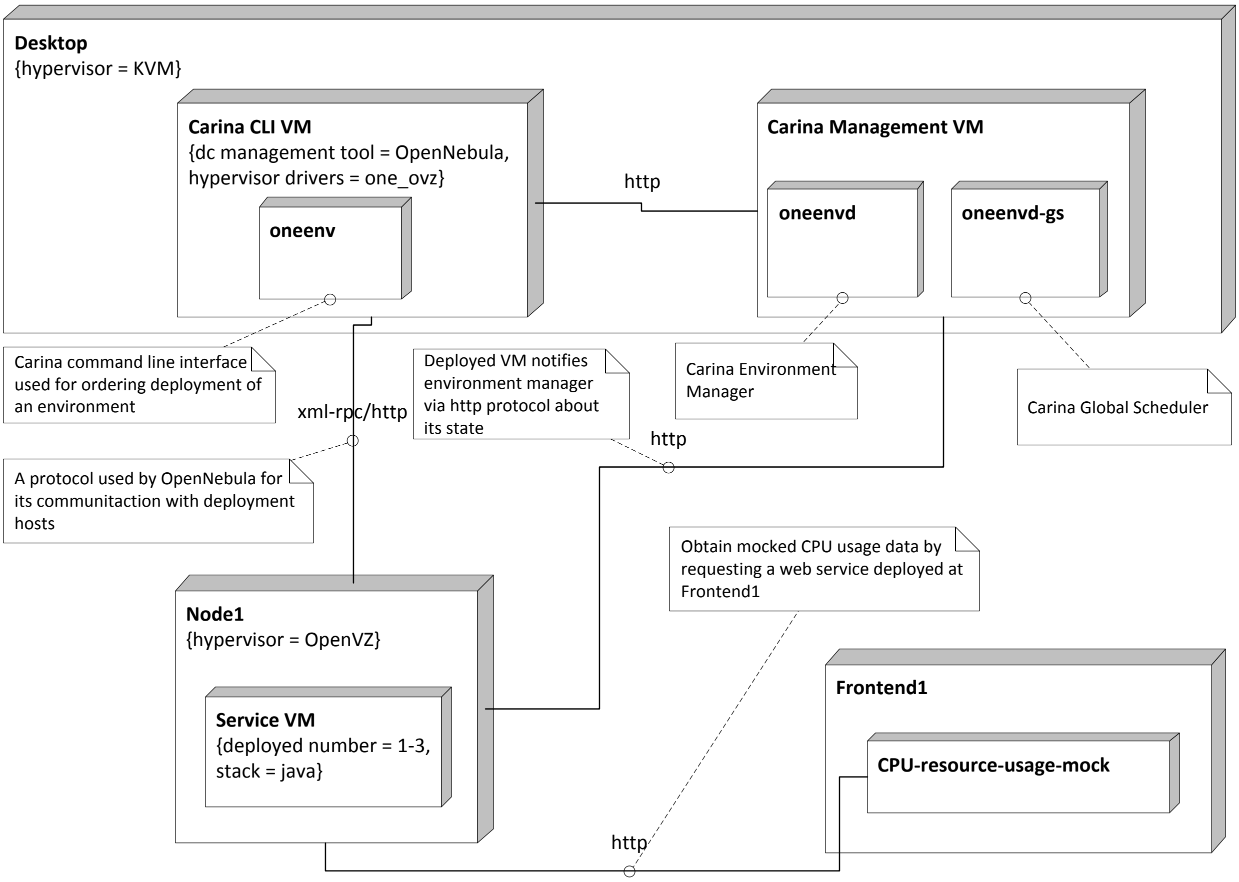

Figure 8.10: Auto-scaling - multiple-provider based: deployment diagram of *Carina*

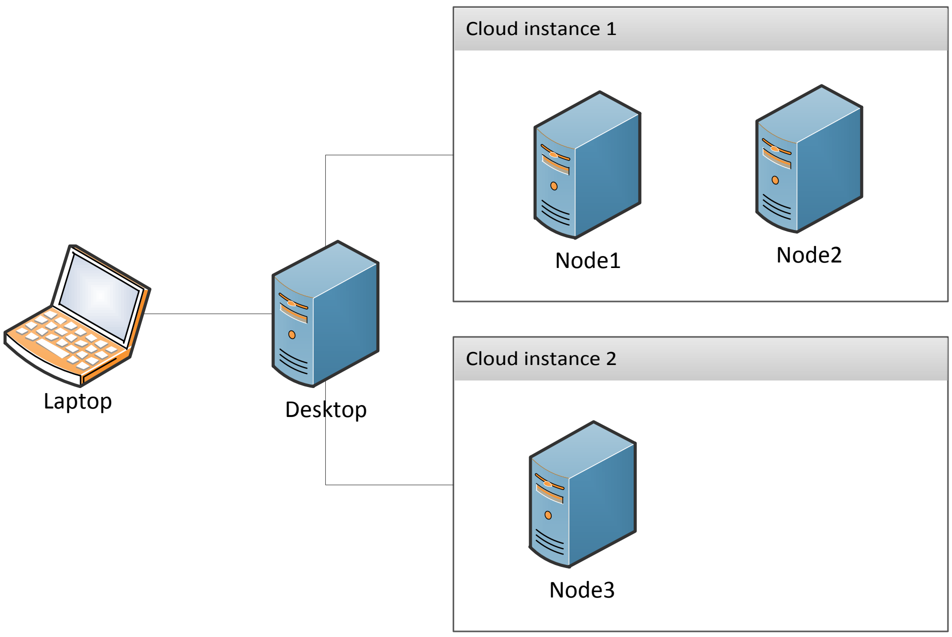

Figure 8.11: Auto-scaling - multiple-provider based: environment configuration

Expectations

Taking CPU usage function into account, we can predict outcome of the test. Scaling policy is violated in 5th minute, hence, it is expected that first scaling event occur after that point. Moreover, since that moment, CPU usage remains beyond threshold implying successive scaling actions.

Due to the fact that *Cloud-SAP* favours fine-grained scaling actions such as vertical scaling, it is expected that these actions occur sooner than horizontal scaling. However, at some point *CP-1* capacity won't be sufficient, hence, another stack instance (one master instance, two slaves) will be deployed on *CP-2*. Subsequent scaling actions will take place solely on a *CP-2* 2 and will involve further vertical scaling.

Carina, on the other hand, supports solely horizontal scaling using single cloud provider. On top of that, it is assumed that succeeding slaves instances will be added to a service up to the point where Cloud Provider won't have enough resources to proceed with further scaling requests.

Results

Cloud-SAP In total, there were 13 vertical (increasing CPU limit) 2 horizontal (adding a container) scaling events. First action took place minute after first violation of scaling policy rule. Since then, slaves' CPU were successively increasing to the point were further scaling wasn't possible - 13th minute of the test. Therefore, *Cloud-SAP* deployed two new slaves on *CP-2*. Listing 8.9 presents logs covering service lifecycle.

Listing 8.9: Cloud-SAP logs excerpt regarding auto-scaling actions on multiple providers

```
D, [2014-01-03T15:43:48.782879 #2740] DEBUG -- : CONTAINER Analyzing data {:container=>#<
    AutoScaling::Container @id=1 @correlation_id=875 @ip=#<IPAddr: IPv4
    :192.168.0.100/255.255.255.255> @type=:master @probed="1388501025" @requirements={"cpu
    "=>1.8824555100000004, "memory"=>512} @stack_id=1>, :metrics=>{"CPU"=>["71", "70"]}}
D, [2014-01-03T15:43:48.783877 #2740] DEBUG -- : CONTAINER Concluded that currently {"id
    ":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed":"1388501025","
    requirements":{"cpu":1.8824555100000004,"memory":512},"stack_id":1} is insufficient (by
     key: CPU)
...
D, [2014-01-03T15:43:48.784777 #2740] DEBUG -- : CONTAINER Attempt to scale CPU up for a
    container: {"id":1,"correlation_id":875,"ip":"192.168.0.100","type":"master","probed
    ":"1388501025","requirements":{"cpu":1.8824555100000004,"memory":512},"stack_id":1}
I, [2014-01-03T15:43:49.512883 #2740]  INFO -- : CONTAINER Cannot reserve:
    2.447192163000001 with {:memory=>358.76171875, :cpu=>1.9480000000000002} at node1 (
    AutoScaling::InsufficientResources)
...
I, [2014-01-03T15:43:49.514541 #2740]  INFO -- : STACK Got unprocessed conclusion:
    insufficient_cpu for {"id":2,"correlation_id":876,"ip":"192.168.0.101","type":"slave","
    probed":"1388501010","requirements":{"cpu":1.8824555100000004,"memory":512},"stack_id
    ":1}
...
I, [2014-01-03T15:43:50.279589 #2740]  INFO -- : STACK Cannot reserve: {:cpu
    =>1.8824555100000004, :memory=>512.0} with {:cpu=>0.3675444899999998, :memory
    =>2673.109375} (AutoScaling::InsufficientResources)
I, [2014-01-03T15:43:50.279955 #2740]  INFO -- : STACK Delegating execution to a cloud-
    controller
I, [2014-01-03T15:43:50.280566 #2740]  INFO -- : Received request of insufficient_slaves to
     be performed on a stack #<AutoScaling::Stack @id=1 @correlation_id=628 @type=:java
    @state=:deployed @data=nil @service_name="Deployment time test service">
```

Carina Similarly to a *Cloud-SAP*, first scaling event (slave addition) was noticed in 6th minute of test. Scaling jobs were continuously invoked every 2 minutes until capacity of a *CP1* hasn been fully exploited. Listing 8.10 summarises jobs performed by *Carina*.

Listing 8.10: Carina environment manager logs with taken actions (*jobs*)

```
ID   ENVID     CONFIG_NAME      TYPE      SUBMIT_TIME                   STATUS
--   -----     -----------      ----      -----------                   ------
266      17        testcase2     CREATE    Fri Jan 3 10:41:15 +0000 2014 DONE
267      17        testcase2    SCALEUP    Fri Jan 3 10:47:44 +0000 2014 DONE
268      17        testcase2    SCALEUP    Fri Jan 3 10:49:31 +0000 2014 DONE
267      17        testcase2    SCALEUP    Fri Jan 3 10:51:02 +0000 2014 DONE
268      17        testcase2    SCALEUP    Fri Jan 3 10:53:07 +0000 2014 DONE
```

Transactions per second

Chart 8.12 illustrates how service capabilities changed over the time. Service capability is expressed as a number of transactions per second that can be handled by a service. Please note, that this measure, while giving good insight into application potential, is hard to simulate, hence, we assumed that 1 VCPU provides enough resources to successfully serve 100 TPS. Knowing that, we roughly estimate total service capacity, which at the end of the test amounts to:

– Cloud-SAP: 283 transactions per second

– Carina: 180 transactions per second

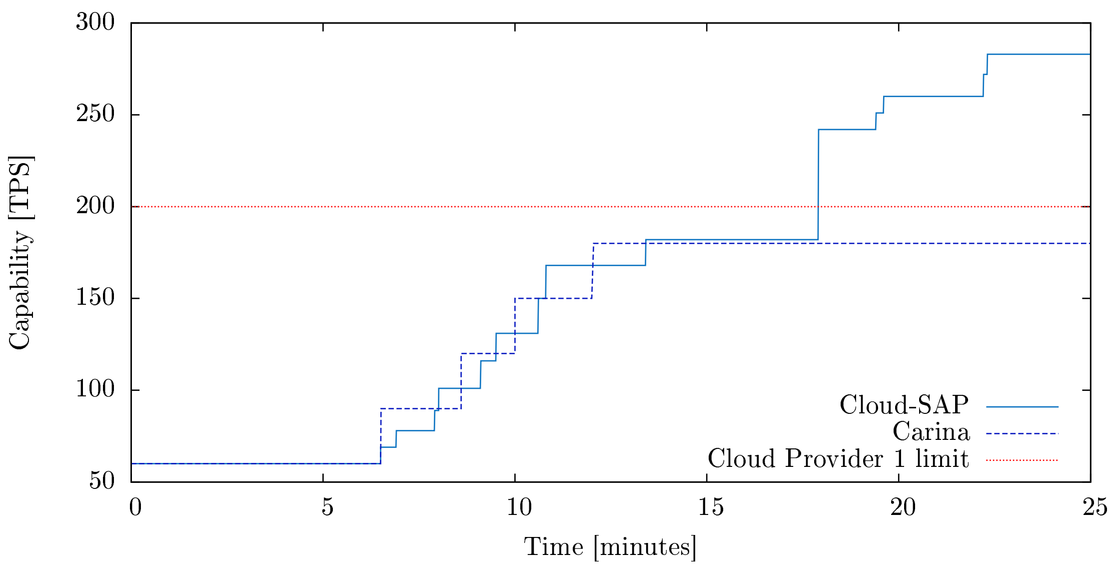

Figure 8.12: Auto-scaling - multiple-provider based: comparison of processed transaction per second

Conclusion

Comparing expectations with test output, one can see that expectations has been fully met. While *Carina* was solely focused on horizontal scaling and bound to a single cloud provider, *Cloud-SAP* first took fine-grained actions to later leverage resources of a second cloud provider.

Cloud-SAP has been proven to be a better solution, offering greater capabilities to a service. This was possible by exploiting resources of two federated cloud providers, crucial concept behind *Cloud-SAP* design.

8.5. Deployment time – solution comparison

Description

In this test we want to compare our solution to one of those available at the market which use *OpenNebula* as an underlying tool for managing resources of a data center and *OpenVZ* as a hypervisor in terms of **deployment time**, one of the most important factors of products whose main purpose is to scale applications. *Carina* [12] can be considered a perfect match of a solution for such a comparison and tests are ran against it.

This test involves the steps of i) instantiating one of the tested product, i.e. Cloud-SAP or Carina, ii) ordering the deployment of a service whose specification is shown in listing A.3, iii) measuring the time needed to set up the environment of the service.

Preconditions

It is assumed that *Cloud-SAP* and *Carina* according with *OpenVZ* as an underlying virtualization technology are correctly installed and configured. Each test case must be run in an isolation so before performing any test all virtual machines present at the deployment node are removed.

OpenNebula configuration

To ensure objectivity in tests, OpenNebula was configured in both products in the same way. One of the key factors that could influence the deployment time is the configuration of scheduler. Its parameters are shown in the listing 8.11.

Listing 8.11: OpenNebula scheduler configuration

```
SCHED_INTERVAL = 30
MAX_VM         = 300
MAX_DISPATCH   = 30
MAX_HOST       = 1
HYPERVISOR_MEM = 0.1
```

Service description

The service comprises a simple java enterprise application, deployed in a master-slave configuration with one VM set as a load balancer and other nodes that serve as workers, which uses Tomcat as a web container.

The description of a service expressed in Carina format can be found in listing A.1.

Hardware configuration

All virtual machines were deployed on a host named *Node1*, whose configuration can be found in table 8.1. Diagram presents the physical configuration of nodes.

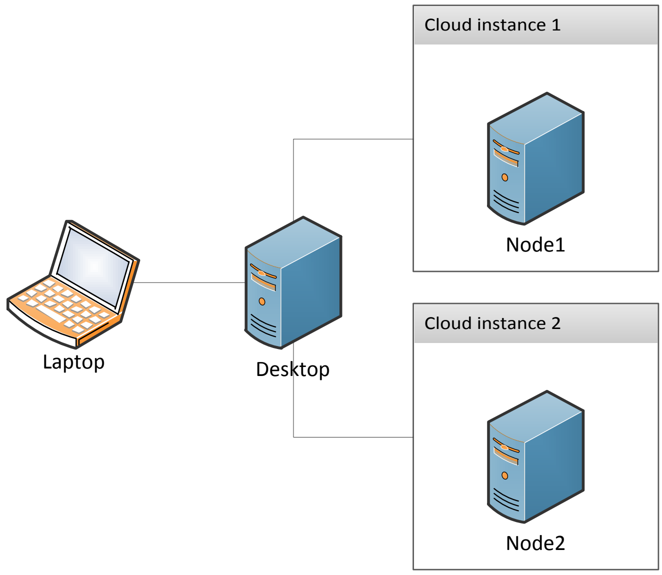

Figure 8.13: Deployment time - solution comparison: physical environment setup

Environment configuration

Deployment diagram for *Carina* implementation is shown in figure 8.14 and for *Cloud-SAP* in figure 8.15.

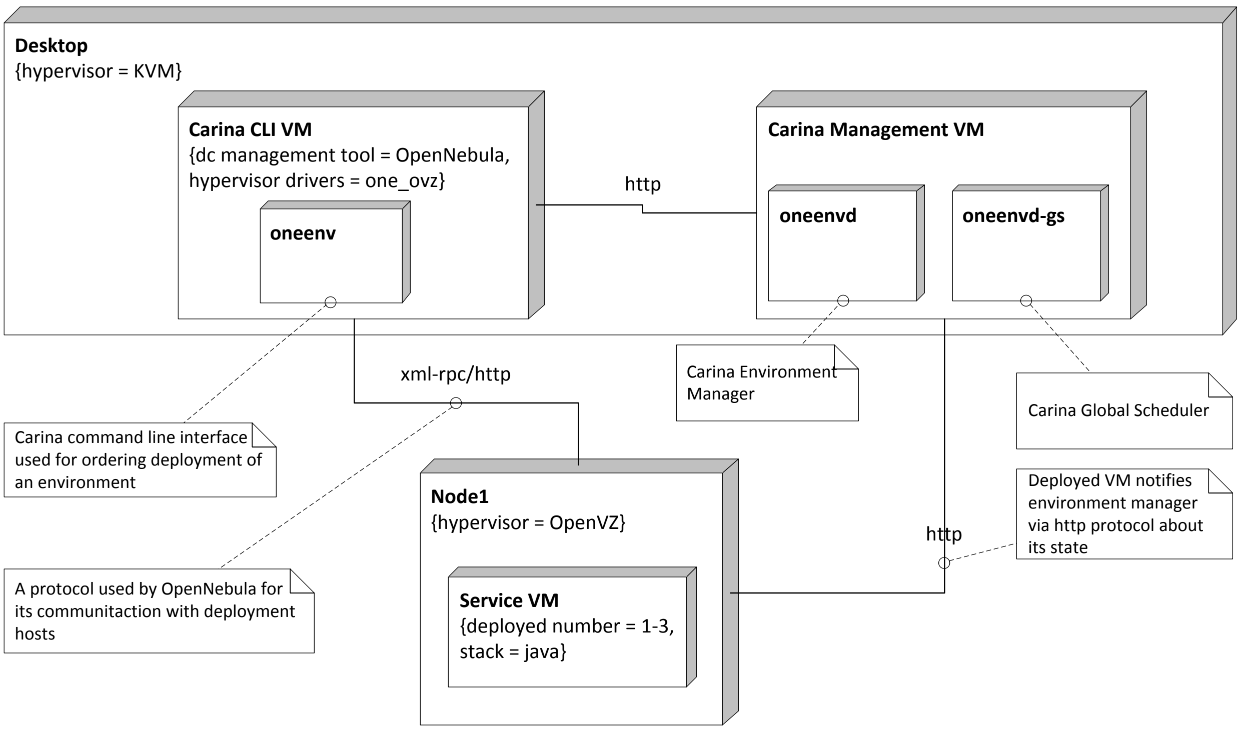

Figure 8.14: Deployment diagram of *Carina*

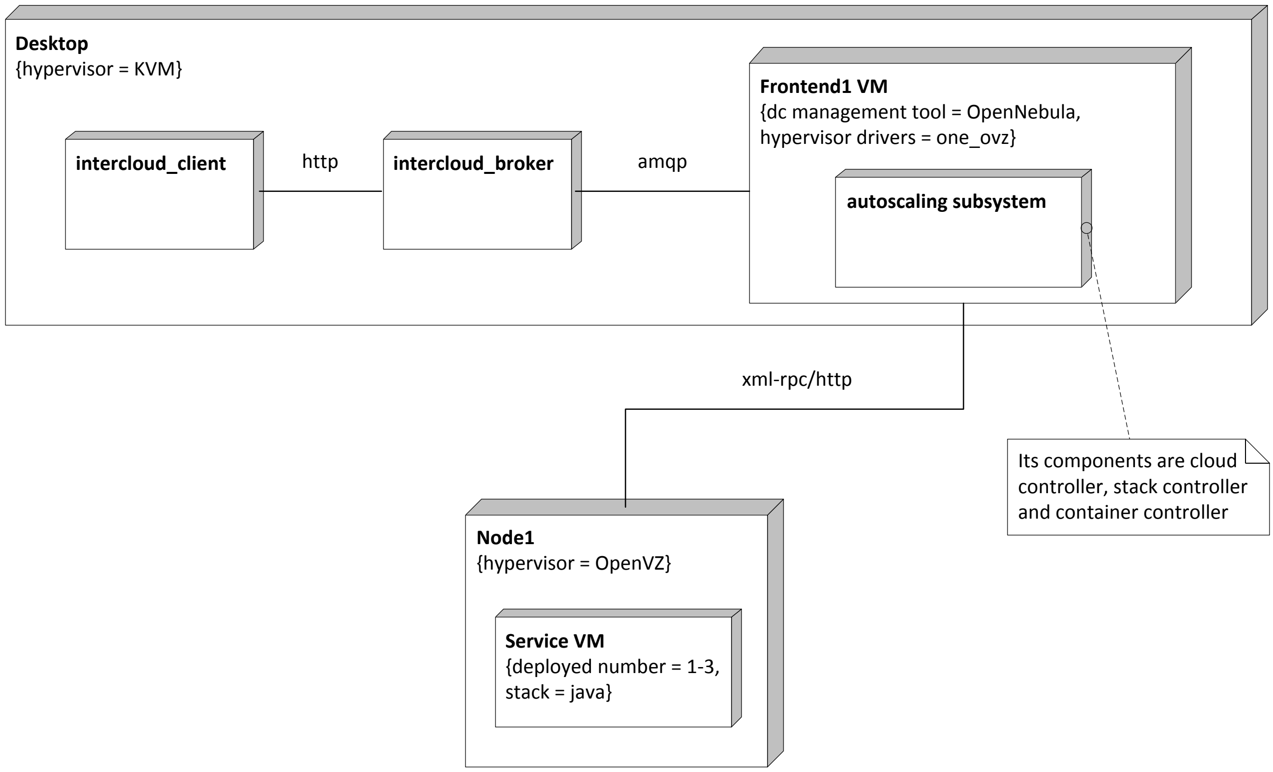

Figure 8.15: Deployment diagram of *Cloud-SAP*

	Solution	
Instance no	Cloud-SAP	Carina
2	198.0	158.7
3	261.72	208.1
4	294.38	230.6

Table 8.5: Average deployment time for the service with the various number of VMs used for the whole environment

Results

Obtained results are shown in table 8.5 and in figure 8.16. For a given number of instances we ordered deploying a service 10 times and the values shown in those figures are an average of these runs.

Conclusion

As one can can notice, deployment of a java stack takes using *Cloud-SAP* lasts longer than when using *Carina* for the same purpose. Moreover, the more instances are in such stack, the greater difference is. This difference in provisioning time is mainly driven by the fact that *Cloud-SAP* uses a great deal of components in comparison to *Carina*, what is a direct consequence of chosen architecture. Specifically, the request from a client is passed to a broker, then the best provider is chosen and finally the deployment using the select provider and its *Applfow* server takes place. Contrary, *Carina* directly operates on *OpenNebula* giving much simpler flow.

Noticeably, deployment time never was a crucial issue for *Cloud-SAP*, hence, this issue is of a little importance. What is actually important for *Cloud-SAP* is deployment cost and service performance. In other words, slightly

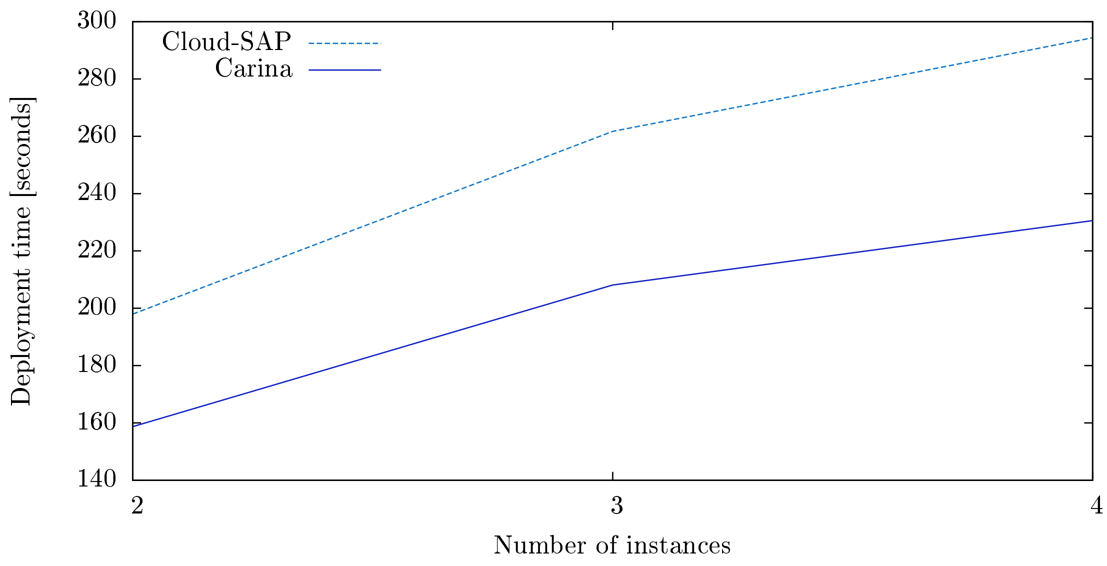

Figure 8.16: Average deployment time for two competing products when the variable is the number of instances of VMs

greater deployment time is a price that is paid to be sure that resources are deployed optimally and with Quality-of-Service guarantee.

9. Summary

A. Code listings

A.1. Service specifications

Listing A.1: Carina Environment Specification which was used during tests of deployment time

```
ENDPOINT = {
  'mm01' => {
    :proxy   => 'http://192.168.0.35:2633/RPC2',
    :oneauth => "svc:xxxxx"
    }
  }


TEMPLATE = {
  'tomcat' => {
    :file       => "~/vm/tomcat.vm",
    :cpu        => "0.3",
    :memory     => 512,
    :network_id => { 'mm01' => 7 },
    :image_id   => { 'mm01' => 10 }
  },
  'haproxy' => {
    :file       => "~/vm/haproxy.vm",
    :cpu        => "0.3",
    :memory     => 512,
    :network_id => { 'mm01' => 7 },
    :image_id   => { 'mm01' => 11 }
  }
}


ENVIRONMENT = {
  'testenv' => {
                :type                  => "compute",
                :endpoint              => "mm01",
                :description           => "Example environment",
                :master_template       => "haproxy",
                :master_context_script => "sample-master-context-script.sh",
                :master_setup_time     => 30,
                :master_context_var    => "BALANCE_PORT=8080",
                :slave_template        => "tomcat",
                :slave_context_script  => "sample-slave-context-script.sh",
                :slave_context_var     => "APP_PACKAGE=gwt-petstore.war",
                :placement_policy      => "pack",
                :num_slaves            => 3,
```

```
                    :slavedata              => "8080",
                    :adminuser              => "root",
                    :app_url                => "http://%MASTER%:8080/testapp"
       }
    }
```

Listing A.2: Cloud-SAP Service Specification (without scaling policies)

```
{
  "name": "my new facebook",
  "stacks": [
  {
    "type":"java",
    "instances":1
  },
  {
    "type":"amqp",
    "instances":1
  },
  {
    "type":"python",
    "instances":1
  },
  {
    "type":"ruby",
    "instances":1
  },
  {
    "type":"postgres",
    "instances":1
  }
  ]
}
```

Listing A.3: Cloud-SAP Service Specification used for testing deployment time

```
{
  "name":"Deployment time test service",
  "stacks":[
  {
    "type":"java",
    "instances":2,
    "policy_set":{
      "min_vms":0,
      "max_vms":2,
      "policies":[
      {
        "name":"threshold_model",
        "parameters":{
          "min":"5",
          "max":"50"
        }
      }
      ]
    }
  }
```

```
  ]
}
```

A.2. Scaling policies

Listings A.4 illustrates XML-based policy that is used by Auto Scaling of the Amazon Web Services EC2.

Listing A.4: Scaling policy - AWS EC2


```
<DescribeAutoScalingGroupsResponse xmlns="http://autoscaling.amazonaws.com/doc
    /2011-01-01/">
  <DescribeAutoScalingGroupsResult>
    <AutoScalingGroups>
      <member>
        <Tags/>
        <SuspendedProcesses/>
        <AutoScalingGroupName>my-test-asg</AutoScalingGroupName>
        <HealthCheckType>EC2</HealthCheckType>
        <CreatedTime>2013-01-22T23:58:48.718Z</CreatedTime>
        <EnabledMetrics/>
        <LaunchConfigurationName>my-test-lc</LaunchConfigurationName>
        <Instances>
          <member>
            <HealthStatus>Healthy</HealthStatus>
            <AvailabilityZone>us-east-1e</AvailabilityZone>
            <InstanceId>i-98e204e8</InstanceId>
            <LaunchConfigurationName>my-test-lc</LaunchConfigurationName>
            <LifecycleState>InService</LifecycleState>
          </member>
        </Instances>
        <DesiredCapacity>1</DesiredCapacity>
        <AvailabilityZones>
          <member>us-east-1e</member>
        </AvailabilityZones>
        <LoadBalancerNames/>
        <MinSize>1</MinSize>
        <VPCZoneIdentifier/>
        <HealthCheckGracePeriod>0</HealthCheckGracePeriod>
        <DefaultCooldown>300</DefaultCooldown>
        <AutoScalingGroupARN>arn:aws:autoscaling:us-east-1:123456789012:autoScalingGroup:66
            be2dec-ee0f-4178-8a3a-e13d91c4eba9:autoScalingGroupName/my-test-asg<
/AutoScalingGroupARN>
        <TerminationPolicies>
          <member>Default</member>
        </TerminationPolicies>
        <MaxSize>5</MaxSize>
      </member>
    </AutoScalingGroups>
  </DescribeAutoScalingGroupsResult>
  <ResponseMetadata>
    <RequestId>cb35382a-64ef-11e2-a7f1-9f203EXAMPLE</RequestId>
  </ResponseMetadata>
</DescribeAutoScalingGroupsResponse>
```

List of Tables

List of Figures

Bibliography

[1] Apache httpd. http://httpd.apache.org/. Accessed: 2013-08-30.

[2] Apache mod_jk. http://tomcat.apache.org/download-connectors.cgi/. Accessed: 2013-08-30.

[3] Apache mod_proxy_balancer. http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html. Accessed: 2013-08-30.

[4] Auto scaling - amazon web services. http://aws.amazon.com/autoscaling/. Accessed: 2013-08-30.

[5] Cloudstack. http://cloudstack.apache.org/. Accessed: 2013-08-30.

[6] Eucalyptus. http://www.eucalyptus.com/. Accessed: 2013-08-30.

[7] F5. http://www.f5.com/. Accessed: 2013-08-30.

[8] Haproxy. http://haproxy.1wt.eu/. Accessed: 2013-08-30.

[9] Haproxydoc. http://haproxy.1wt.eu/download/1.3/doc/configuration.txt. Accessed: 2013-08-30.

[10] Occi. http://occi-wg.org/. Accessed: 2013-08-30.

[11] Oneflow. http://docs.opennebula.org/stable/advanced_administration/application_flow_and_auto-scaling/index.html. Accessed: 2014-01-20.

[12] Opennebula carina. https://github.com/blackberry/OpenNebula-Carina. Accessed: 2013-12-16.

[13] Opennebula carina blog. http://opennebula.org/introduction-to-the-carina-environment-mana Accessed: 2014-01-18.

[14] Opennebula project. http://opennebula.org/. Accessed: 2013-08-30.

[15] Openshift. https://www.openshift.com/. Accessed: 2013-08-30.

[16] Openshift - scaling. https://www.openshift.com/developers/scaling. Accessed: 2013-08-30.

[17] Openstack. http://www.openstack.org/. Accessed: 2013-08-30.

[18] Pivotal. http://www.gopivotal.com/. Accessed: 2014-01-22.

[19] Pivotalinitiative. `http://blogs.vmware.com/vmware/2012/12/the-pivotal-initiative.html`. Accessed: 2014-01-22.

[20] The future of cloud computing 3rd annual survey. Tech. rep., 2013. Report from the survey available at `http://mjskok.com/resource/2013-future-cloud-computing-3rd-annual-survey-results`. Accessed: 2013-09-03.

[21] Rackspace 2013 hybrid cloud survey results. Tech. rep., 2013. Report from the survey available at `http://www.rackspace.com/knowledge_center/article/rackspace-2013-hybrid-cloud-survey-results`. Accessed: 2013-09-03.

[22] A., B. Characteristics of scalability and their impact on performance.

[23] ABDEEN, M., AND WOODSIDE, C. M. Seeking optimal policies for adaptive distributed computer systems with multiple controls. Third International Conference on Parallel and Distributed Computing, Applications and Technologies.

[24] ABDELZAHER, T., SHIN, K., AND BHATTI, N. Performance guarantees for web server end-systems: A control theoretical approach, 2002.

[25] BUYYA, R., ABRAMSON, D., GIDDY, J., ET AL. A case for economy grid architecture for service-oriented grid computing. In *IPDPS* (2001), vol. 1, pp. 20083–1.

[26] BUYYA, R., ABRAMSON, D., GIDDY, J., AND STOCKINGER, H. Economic models for resource management and scheduling in grid computing. *Concurrency and computation: practice and experience 14*, 13-15 (2002), 1507–1542.

[27] BUYYA, R., RANJAN, R., AND CALHEIROS, R. Intercloud: Scaling of applications across multiple cloud computing environments.

[28] BUYYA, R., RANJAN, R., AND RN., C. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services.

[29] BUYYA, R., YEO, C., AND VENUGOPAL, S. Market-oriented cloud computing: Vision, hype, and reality for delivering it service s as computing utilities. *High Performance Computing* (2008).

[30] BUYYAA, R., YEOA, C., VENUGOPALA, S., BROBERGA, J., AND BRANDICC, I. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems 25* (2009).

[31] CHAPIN, S. J., KATRAMATOS, D., KARPOVICH, J., AND GRIMSHAW, A. S. The legion resource management system. In *Job Scheduling Strategies for Parallel Processing* (1999), Springer, pp. 162–178.

[32] DIAO, Y., LUI, X., FROEHLICH, S., HELLERSTEIN, J., AND PAREKH, S. On-line response time optimization of an apache web server. International Workshop on Quality of Service.

[33] DUBOC, L., ROSENBLUM, D., AND WICKS, T. A framework for modelling and analysis of software systems scalability. Proceeding of the 28th international conference on Software engineering.

[34] G., A. Validity of the single processor approach to achieving large scale computing capabilities. vol. 983 of *AFIPS Conference Proceedings*.

[35] HILL, M. What is scalability? ACM SIGARCH Computer Architecture News.

[36] IBM. An architectural blueprint for autonomic computing, 4th edition.

[37] LEAVITT, N. Is cloud computing really ready for prime time? Available at `http://www.leavcom.com/pdf/Cloudcomputing.pdf`. Last accessed: 2013-09-10.

[38] LEONG, L., TOOMBS, D., GILL, B., PETRI, G., AND HAYNES, T. Magic quadrant for cloud infrastructure as a service. Tech. rep., Gartner, Gaithersburg, Aug. 2013.

[39] LITOIU, M., WOODSIDE, M., AND ZHENG, T. Hierarchical model-based autonomic control of software systems. Design and evolution of autonomic application software.

[40] MAO, M., LI, J., AND HUMPHREY, M. Cloud auto-scaling with deadline and budget constraints. IEEE/ACM International Conference on Grid Computing.

[41] MELL, P., AND GRANCE, T. The nist definition of cloud computing.

[42] R., M. Autonomic computing.

[43] RATHORE, M., HIDELL, M., AND SJODIN, P. Kvm vs. lxc: comparing performance and isolation of hardware-assisted virtual routers, 2013.

[44] SHEN, Z., SUBBIAH, S., GU, X., AND WILKES, J. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. ACM Symposium on Cloud Computing – SOCC2011.

[45] STECCA, M., BAZZUCCO, L., AND MARESCA, M. Sticky session support in auto scaling iaas systems. IEEE World Congress on Services.

[46] VAQUERO, L., RODERO-MERINO, L., AND BUYYA, R. Dynamically scaling applications in the cloud, 2011.