Universidade de Aveiro
2010

Departamento de Electrónica, Telecomunicações e Informática

Tiago Simões Batista

Arquitecturas para sistemas de informação baseados em cloud computing

Architectures for cloud computing based information systems

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
2010

Tiago Simões Batista

Arquitecturas para sistemas de informação baseados em cloud computing

Architectures for cloud computing based information systems

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Prof. Doutor Joaquim Manuel Henriques de Sousa Pinto, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Cláudio Jorge Teixeira Vieira, investigador post doc da Universidade de Aveiro.

o júri / the jury

Presidente / President

Doutor José Luís Guimarães Oliveira
Professor associado do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro

Vogais / Examiners committee

Doutor Fernando Joaquim Lopes Moreira
Professor Associado do Departamento de Inovação, Ciência e Tecnologia da Universidade Portucalense

Doutor Joaquim Manuel Henriques de Sousa Pinto
Professor Auxiliar do Departamento de Electrónica e Telecomunicações da Universidade de Aveiro

Doutor Cláudio Jorge Teixeira Vieira
Investigador post-doc da Universidade de Aveiro

agradecimentos

Um abraço para todos os colegas que me acompanharam ao longo do curso, especialmente durante os últimos dois anos.

Um muito obrigado à minha família que me perdoou andar a pastar pela universidade antes de descobrir a minha verdadeira vocação.

E já agora, um grande agradecimento ao Nuno e ao Diogo por me terem aberto os olhos a tempo de eu não desistir.

Abraços para todos lá de casa que me ajudaram nos momentos de humor mais negro, e um beijo especial para a Fátima, que tem a compreensão para não me mandar bugiar apesar do pouco tempo que lhe posso dedicar.

Obrigado aos professores que me deram aulas ao longo do curso, especialmente para aqueles que me estimularam a ser independente, autodidacta, e a não ter medo de confrontar as ideias pré concebidas.

Finalmente, muito obrigado aos meus orientadores pelo estímulo que me deram para sair da minha zona de conforto e explorar vertentes novas desta área maravilhosa!

As pessoas que me ajudaram a chegar até aqui são incontáveis, mas como este espaço é limitado, tenho que me conformar em mandar um abraço a todos, sem poder mencionar nomes!

palavras-chave Nuvem, SOA, IaaS, PaaS, virtualização

resumo

Este trabalho faz um apanhado do panorama actual no que diz respeito a *Cloud computing*. Começa por analisar a definição proposta pelo NIST e categorizar vários serviços comerciais de acordo com as categorias propostas nessa definição.

De seguida, são analisadas as implementações grátis disponíveis em licenças *Open Source* e chega-se à conclusão que para Clouds do tipo IaaS já existem várias implementações, algumas com boa qualidade, mas que na área de PaaS ainda existe muito trabalho a ser feito antes de se chegar a uma implementação com funcionalidade comparável à dos serviços comerciais existentes.

Após uma breve análise sobre a integração de SOA com as facilidades do *Cloud computing*, chegou-se à conclusão que PaaS se apresenta como o modelo de serviço mais adequando para desenvolver aplicações SOA.

Visto que não existe ainda nenhum PaaS livre, e que os existentes apresentam problemas sérios de *vendor lock in*, é especificada uma *framework* completa, portátil e aberta que permitirá implementar um serviço do tipo PaaS em infraestrutura privada ou sobre algum dos IaaS existentes.

O PaaS especificado baseia-se, sempre que possível, em tecnologias existentes, concluindo-se que apenas a tecnologia de armazenamento de dados estruturados está aquém do necessário para a implementação. Deixa-se para o futuro a implementação dos vários módulos que permitirão a integração dos vários componentes da PaaS, no entanto sempre que possível, são sugeridas tecnologias a utilizar de forma a manter a implementação aberta e portátil.

keywords Cloud, SOA, IaaS, PaaS, virtualization.

abstract

This work sums up the current situation of Cloud computing. It starts by performing an analysis of the NIST definition draft, and categorizing some commercial services into the categories proposed by the referred definition.

Next, the free implementations distributed under an Open Source license are analyzed, and the conclusion is that there are some high quality IaaS cloud implementations, but the PaaS area still needs a lot of work before the functionality of a free implementation is comparable to that of the commercial services available.

After a brief analysis of the integration of SOA and Cloud computing, the conclusion is that PaaS presents the most adequate service model for the development of SOA applications.

Given that, up to the moment, there is no free PaaS, and that the existing ones present serious vendor lock in problems, a complete, portable, and open framework that allows the deployment of a PaaS type service on private or on IaaS infrastructure is specified.

The specified PaaS is based on current technology whenever possible, with exception of the storage of structured data that is not up to the requirements yet. The implementation of the modules required to integrate the various PaaS components is left as future work. Yet, whenever possible, suggestions are made about usable technologies that will allow the PaaS to remain portable and open.

Table of Contents

Table of Figures

Table of Tables

Acronym List

ACID	Atomicity, Consistency, Isolation, Durability (set of properties)
BPEL	Business Process Execution Language
CLI	Command Line Interface
CORBA	Common Object Request Broker Architecture
GUI	Graphical User Interface
IaaS	Infrastructure as a Service
IEETA	Instituto de Engenharia Electrónica e Telemática de Aveiro
KVM	Kernel Virtual Machine
NAS	Network Attached Storage
NCSU	North Carolina State University
OCCI	Open Cloud Computing Interface
OGF	Open Grid Forum
OOP	Object Oriented Programming
PaaS	Platform as a Service
RDBMS	Relational Database Management System
REST	Representational State Transfer
RDP	Remote Desktop Protocol
RMI	Remote Method Invocation
SaaS	Software as a Service
SDK	Software Development Kit
SOA	Service Oriented0020Architecture
SSH	Secure Shell
UDDI	Universal Description Discovery and Integration
VM	Virtual Machine
VNC	Virtual Network Computing
WSDL	Web Service Definition Language
WSRF	Web Services Resource Framework
XML	Extensible Markup Language

1 Introduction

With the advent of what has been called web 2.0 [1], many new ideas have risen to become successful business from night to day. This sort of meteoric growth poses a great challenge for system and application administrators not only because of under or over provisioning [2], but also because the architecture of an application sometimes becomes its own greatest problem when scaling.

Over the previous years, two technologies that try to ease both problems - the scalability and the creation of new services - have emerged. They are Cloud Computing and Service Oriented Architecture (SOA), each of them addressing a different problem. SOA is geared towards the creation of new applications using loose coupling design patterns; it also allows the creation of new services by orchestrating available services or modules.

1.1 Motivation

Cloud computing aims to solve the scalability problem by supplying computation power on demand, thus allowing the resources allocated to an application to scale in or out as needed.

The combination of both technologies (Cloud & SOA) will enable application developers to create richer applications, using less development time, and with more confidence that the external modules work as expected.

This requires a good understanding of SOA and cloud computing as well as a good definition of what is to be achieved, because each of the concepts involved is so vast that a full understanding would take longer than the time allowed for the whole project.

1.2 Objectives

The work developed during this dissertation is expected to culminate in two deliverables. The first is a simple Infrastructure as a Service (IaaS) cloud implemented and running at the datacenter. This implementation is useable by itself for teaching and research purposes. By the time this dissertation is finished, this cloud should be working, requiring only small adjustments for a larger scale deployment.

The second deliverable is a specification for a platform running on the cloud that serves two purposes. The first is the creation of a truly vendor agnostic platform that parts with some of the limitations of current commercial implementations. The second is to ease SOA application development and deployment by addressing problems that exist with current public UDDI implementations.

The specified platform should allow the creation of SOA services that seamlessly integrate and take advantage of the cloud scalability features, yet it should be open and fully portable in order to avoid lock in issues.

1.3 Methodology

The first thing that must be explored is Cloud computing. There are some commercial implementations, and several implementations of academic interest. There is much discussion over the definition of cloud computing, yet this is not the place to continue that discussion, therefore the NIST draft [3] will be explored as an introduction to the concept. Some implementations that were able to perform (or are in the process of performing) the transition from datacenter virtualization to cloud computing management will be analyzed based on feature charts, and the best candidate will be used for field-testing on the available resources.

Service Oriented Architecture (SOA) is now a mature software architecture. Its main objectives are code reusability and loose coupling. To achieve that objective, developers must transform each module of their applications into standalone services. When each module is a service, new applications may be built just by recombining a subset of the available modules into a service orchestration. This pattern is therefore the epitome of modular software, where each module becomes truly autonomous from the others.

This architecture has found its way mostly on the Business to Business (B2B) market segment, where integration of new software and legacy systems is a common issue. By wrapping the legacy system on a modern service interface, it becomes possible to include that functionality on a modern service orchestration.

SOA takes advantage of several well known remote invocation patterns such as CORBA, RMI or Web Services to build its interfaces, therefore allowing loose coupling between services.

As the complexity of the required orchestration increases, automatic systems for discovery and orchestration of services are introduced by SOA, commonly UDDI for service discovery and BPEL for service orchestration.

While SOA is a well known concept, there is the need to clarify which aspects of the architecture are more prone to take advantage of a cloud computing environment.

To build a test bed suitable for experimentation with the topics described above, some underused resources from the IEETA datacenter will be used.

The available computing resources range from Pentium III class desktop computers to modern multi CPU servers. The heterogeneity that at first may seem an obstacle is in fact a great opportunity: a) to explore how certain services scale horizontally; b) to simulate off cloud services; or even c) to test the chosen cloud implementation flexibility.

If the early testing stages prove to be successful, it will be necessary to study the possibility of growth through migration / integration of extra computation resources into the cloud to enhance the flexibility, and increase the usage of the available resources. This step is critical especially if the resulting cloud will be used for teaching or research purposes, as planned

1.4 Document organization

This document is split into six chapters and two appendixes. The first chapter introduces the problem, proposes a step towards a possible solution, as well as a methodology to achieve it. The second chapter, complemented by the appendixes, is an in depth analysis of the current state of the art in cloud computing, both in the commercial world as in the open source community.

Chapter three describes the deployment of the selected infrastructure manager, as well as all the decisions made during the process of creating an IaaS testbed. Chapter four proposes a new PaaS architecture, and chapter five compares it to the solutions that are now available.

Finally, chapter six contains a short conclusion and some ideas for the future.

2 Cloud Computing – State of the art

Some people may claim that cloud computing is just a new fancy name for something that already existed.

The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do. I can't think of anything that isn't cloud computing with all of these announcements. (…) I'm not going to fight this thing. But I don't understand what we would do differently in the light of cloud [4].

Others claim that it *represents a true paradigm shift in the consumption and delivery of IT services [5].*

The commercial offerings from the major companies added to the controversy, because they all claim to be Cloud based, yet the services offered vary greatly in scope.

One thing is certain, all the press in the IT area is overpopulated with articles that talk about, try to define, and demystify Cloud Computing. There is much discussion, both in the academic community and in the business world, as to what is the definition of cloud computing.

Because of this, there is a greater and greater need to reach a conclusion on what constitutes a cloud, and what is something else. NIST in cooperation with industry and government is now trying to reach a consensus as to the definition of cloud computing:

Although that definition does not achieve full community consensus, it encompasses many of the initial ideas for the definition on cloud computing, and will be explored here to give a better understanding of what is Cloud Computing and its service models.

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential ***characteristics****, three* ***service models****, and four* ***deployment models*** *[3].*

2.1 Characteristics

2.1.1 On-demand self-service

A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service's provider [3].

This allows for a great deal of automatic flexibility. If the consumer has the privileges to provision for more resources, then the provisioning act can be an automated one responding to external stimuli, such as an increased load on the server, a degraded quality of service, or even a remote site outage.

This is the major difference between Grid computing and Cloud computing. On a Grid, the resources are scheduled, and the allocation is usually made in large chunks and served when available, which is suited to batch computing. On a Cloud, the resources are allocated in small chunks, and are served almost in real time. This makes Cloud Computing a very interesting platform when the amount of required resources for a given task is not known in advance [6].

2.1.2 Broad network access

Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs) [3].

The PC is no longer the only device accessing the internet. We have a plethora of new connected devices, and all need to be catered by the services on the cloud. This means that the cloud must supply its services in a standard way, so that any new device that adheres to the same standards will be able to take advantage of the supplied services with no modification.

2.1.3 Resource pooling

The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, network bandwidth, and virtual machines [3].

This enables the costumer to be focused on the business requirements instead of the infrastructure details. As an example, some datasets are of sensitive nature and cannot cross certain geographic borders in order to avoid a legal nightmare. The multi-tenant model also enables resource sharing, allowing a given resource that is not being used by a costumer to be assigned to another, with the confidence that neither of them will ever interfere with each other's operation.

2.1.4 Rapid elasticity

Capabilities can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be purchased in any quantity at any time [3].

The costumer must be able to quickly scale in or out. If the supplied service has an unexpected success the scaling speed can be daunting, and a traditional datacenter may take days or weeks to provision for a higher load, which may lead to big loss of revenue. Even if the provision is done right on the first time, the application supplier still needs to provision for peak load, wasting resources most of the time. If the application is an unexpected flop, the provider will end up with a server infrastructure that may never be used, wasting even more resources on the initial investment.

Using cloud computing, the scaling of the infrastructure allocated to the application is very fast, allowing the costumer to scale according to the current application load, not according to his expectation of the application acceptance by the public.

The rapid elasticity means that the provider infrastructure must be provisioned for peak load. This may pose a problem for private clouds (see section 2.3.1), where the company that uses the cloud services performs provisioning of the datacenter. But on the case of large public cloud providers (see section 2.3.3) that serve worldwide clients, that problem is alleviated because peak load hours vary for each client and the available resources can therefore be reallocated throughout the day.

2.1.5 Measured Service

Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service [3].

As with any other utility, the supplier usually charges the client on a usage basis, this means that the usage level must be clearly metered, and the metric used to calculate any charging amount must be understood and accepted by both parties. On a different level, the provider must be able to monitor its datacenters usage patterns in order to plan when and where to invest in order to keep the service level.

2.2 Service Models

Most of the services offered by cloud providers can be split in a few classes of services. A description of those classes follows. These classes are not absolute; some services will fall into more than one category especially if they are composite or orchestrated services.

2.2.1 Cloud Software as a Service (SaaS)

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through a thin client interface such as a web browser (e.g., web-based email). The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings [3].

This model completely hides the underlying complexity from the consumer. From the outside, the system works as if a large computer were servicing all the requests, always maintaining an acceptable quality of service. If the consumer is to be charged, the charging will most likely take the form of a subscription, or a per access payment.

Software as a Service did not emerge with cloud computing [7]. It has been around for years, in the form of subscription services on the internet, some of them are free to the end user, such as Google's Gmail, others are paid services, such as some online scientific libraries that request payment for each article downloaded. Software as a Service gained visibility with Cloud Computing because the flexibility of on demand scaling enabled the software providers to pay only for the resources that are actually used by the application.

2.2.2 Cloud Platform as a Service (PaaS)

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations [3].

This service model offers a development stack that enables the client to take advantage of the provider's infrastructure without the need to learn how to manage it, and in most cases without the need to understand how to take advantage of parallelization. On today's offerings, the provider usually supplies a free SDK for the available programming languages, thus allowing a developer to take full advantage of the cloud just by learning a new SDK. When on a commercial business model, the charging for this service can be quite complex, and is usually based on one or more of the following criteria:

- CPU cycles used
- Number of requests
- Amount of data transferred
- Emailed recipients

This service model usually has a great downside. It completely locks the application developer to the service provider as there is no standard SDK for cloud development, and changing the SDK may mean a full rewrite of the application.

2.2.3 Cloud Infrastructure as a Service (IaaS)

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls) [3].

This service allows a user to provision and spawn an infrastructure and run his selected software stack. This is an on demand service, as the infrastructure can grow or shrink with the load fluctuations. It is usually the client's responsibility to manage most of the software stack and make sure that the application running on the cloud can scale horizontally with the addition of new nodes. The startup and shutdown process of extra nodes is requested via an interface (usually a web service). When in a commercial business model, the charges are calculated on CPU/hour usage. Usually associated with an IAAS is also a storage service, where charges are based on GB transferred and/or stored per month.

2.3 Deployment Models

2.3.1 Private cloud

The cloud infrastructure is operated solely for an organization. It may be managed by the organization or a third party and may exist on premise or off premise [3].

This is an in-house cloud, its full resources are committed to the service of a single organization needs. This type of cloud has the disadvantage of requiring the owner to scale the physical resources for peak load, thus muting some of the cloud computing advantages over other computing models. Yet, if a company has strict security or legal impairments, this may be the only way to take advantage of the cloud model.

2.3.2 Community cloud

The cloud infrastructure is shared by several organizations and supports a specific community that has shared concerns (e.g., mission, security requirements, policy, and compliance considerations).

It may be managed by the organizations or a third party and may exist on premise or off premise [3].

The community cloud has a lot in common with a private cloud, yet the ability to share resources among various organizations can lead to a better utilization ratio of those resources, thus achieving a better utilization ratio of the resources committed to build the cloud.

2.3.3 Public cloud

The cloud infrastructure is made available to the general public or a large industry group and is owned by an organization selling cloud services [3].

This is the most flexible model. The provider is able to take advantage of scale economy, and pass the savings on to the clients. Usually the provider has a resource pool so large that from any client point of view, it looks like a nearly infinite amount of resources.

2.3.4 Hybrid cloud

The cloud infrastructure is a composition of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds) [3].

This deployment model has some similarities with grid federation where two or more grids under different administration domains [8-9] are presented to the end user as an uniform resource. This means that the end user does not need to worry as to the particular interface of each cloud that forms the hybrid cloud, nor about different pricing schemes. From the user point of view, only one cloud is visible. The cloud provider has to deal with different pricing schemes from each of the upstream providers, as well as with the task of presenting its service as unified as possible, abstracting its clients from the particularities of each of the upstream clouds.

There is debate on the lease vs. buy infrastructure matter. In some cases the company may go with a hybrid model, acquiring enough infrastructure to take advantage of the locality effect, or to accommodate the base load, and leasing as the load peaks. Whether this leads to monetary savings is not part of this study, yet when this happens, the resulting computing environment is called a hybrid cloud.

2.4 Comparing Cloud to Grid

Up to now, it is established that Cloud Computing is a highly flexible computing environment that is able to scale close to real time to the computational needs of the task at hand.

On the other hand, Grid computing is a way of harnessing the power of a number of computational resources to perform a given task.

In fact, according to [10], in 2004 the definition for Grid computing did include many of the characteristics that are now publicized as Cloud computing. Yet some differences arise with time and with a closer look.

Recent comparisons [6, 11] show that Grid computing never quite fulfilled the real time scalability promise that some expected it to. Due to the fact that Grid computing often works on federated scenarios, real time scalability becomes even more difficult as different administrative domains may have very different resource allocation policies. This means that Grid computing became popular especially among those that previously used clusters or supercomputers to tackle problems that would otherwise take a long time to solve, which lead to complex scheduling and reservation policies on top of heterogeneous compute resources that cross various administrative domains.

Cloud computing on the other hand removed much of the complexity from the grid solutions, and focused on scaling the resources dedicated to a given task almost in real time. This means that usually a Cloud does not transverse administrative domains, does not allow the allocation of resources ahead of time (although some vendors do allow it) and focuses primarily on managing the usage of a large pool of resources from a single company, by a large number of users with shifting needs.

One difference that is now visible between the Grid and the Cloud paradigms lies on the administrative domains and target audience. Grid computing pools a set of resources from different administrative domains, and exposes it to a given community. On the other hand, Cloud computing exposes the computing resources of a single organization to the general public.

Another great difference between the two paradigms is on usual size and scheduling of the allocations. Grid computing usually serves large allocations, it is not uncommon for an allocation to span all of the resources, for a scheduled amount of time, to a small group of users. This means that Grid allocations are seldom possible at the time they are requested, and therefore complex scheduling is required to take the most advantage of the available resources. As for Cloud computing, allocations are usually small, and are served on a best effort basis. Service Level Agreements (SLA) are possible, but usually will be more expensive.

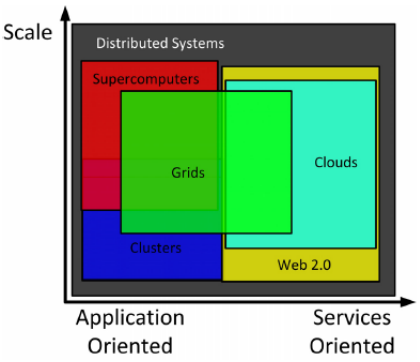

Figure 1 - Cloud and Grid compared[1]

As Figure 1 shows, Grid computing overlaps most if not all of the distributed computing technologies. In fact, many are now harnessing the possibility of using IaaS nodes as Grid nodes on demand, or even building fully virtualized Grids over IaaS.

2.5 Comparison of free IaaS implementations

Each of the implementations reviewed on this document (see Appendix A) has its own set of features. Those features can be a strength or a weakness for a given implementation, depending on the intended usage scenario. A deeper analysis of the differences and similarities of those implementations is necessary if one is to choose a single one for deployment.

2.5.1 Architecture

Not being exactly a feature, the architecture of an application says a lot about its possible future. A clear, modular, and extensible architecture eases the addition of new features and the never-ending process of finding and solving bugs. How the modules communicate among them is also of capital importance, determining how much of the system can be distributed and/or duplicated for reliability and load balancing.

The analyzed implementations' architecture range from a traditional three tier architecture [12] to a fully pluggable architecture [13]. This proves that all of the architectures are viable to start with, yet only time will tell if all of them will keep up with the fast pace of technology.

[1] Taken from [11] I. Foster, Y. Zhao, I. Raicu *et al.*, "Cloud Computing and Grid Computing 360-Degree Compared," *Gce: 2008 Grid Computing Environments Workshop*, pp. 60-69, 2008.

	Architecture	Brokers
VCL	Three Tier	
Globus/Nimbus	Monolithic tools	Java interfaces
OpenNebula	Three tier tools	Text protocols
Eucalyptus	Three tier	
Enomaly		
OpenQRM	Plugin based	
ConVirt	Plugin based	

Table 1 - Infrastructure managers architecture

Table 1 shows the referred projects architecture and brokers. Unfortunately not all of the information required to properly build the entire table is available on the public documentation. When the required information could not be gathered either from the official project documentation or from the associated development communication mediums (usually mailing lists), the field is left empty.

2.5.2 Guest Operating Systems

While running only POSIX guests may be enough for most use case scenarios, a really flexible implementation must be guest-agnostic. Not all implementations achieve this at the moment, yet the tendency seems to add support for as much guest operating systems as possible.

At the moment, the greatest problem seems to be the configuration of user accounts and networking settings during the boot process. Most implementations are now addressing this problem, with some of them already supporting Windows guests to some extent.

The support for Windows and other legacy operating systems is on most cases conditional to the existence of virtualization extensions on the underlying hardware, because most hypervisors require such extensions to run those systems unmodified.

Table 2 shows how well each of the projects supports each class of operating system. It should be noted that none of the existing (commercial or open source) implementations mention support for the OS X operating system, although it is very similar to other BSD systems.

	Linux	Other UNIX [2]	Windows
VCL	Yes	Yes	Yes
Globus/Nimbus	Yes	N/A[3]	No
OpenNebula	Yes	N/A	Yes [14]
Eucalyptus	Yes	N/A	Yes [15]
Enomaly	Yes	N/A	Yes[4]
OpenQRM	Yes	N/A	Yes[5]

Table 2 - Guest operating systems supported

2.5.3 Virtualization Technologies

This type of technology enables the creation of a uniform resource pool where the infrastructure can be deployed. Where available, the virtualization implementation may take advantage of the

[2] This group includes *BSD, (Open)Solaris, and other POSIX compliant systems, excluding OS X
[3] No information available
[4] At least on the commercial version
[5] Only on version 3.x

virtualization extensions present on modern CPUs. However some of the implementations, such as KVM may actually require that these extensions are present [16].

All cloud computing implementations support at least two of the available virtualization technologies; some of them support all of the major virtualization technologies.

	Xen	KVM	VMware	Others
VCL	No	No	Yes	Physical hosts
Globus/Nimbus	Yes	Upcoming [17]	Yes	No
OpenNebula	Yes	Yes	Yes	No
Eucalyptus	Yes	Yes	No	No
Enomaly	Yes	Yes	No	Qemu
OpenQRM	Yes	Yes	Yes	Vserver

Table 3 - Virtualization backends supported

2.5.4 Public Interfaces

The major difference between a private IaaS cloud and legacy datacenter virtualization is on whom has the ability to control what infrastructure is deployed at any given time. On a cloud computing scenario, that control is shifted towards the client. Therefore, the public interfaces become of paramount importance.

A web interface is a great thing to have if a human is to interact directly with the cloud, however most of the current use cases of IaaS require a programmatic self service interface to the available operations, therefore a web interface is not essential.

Currently, the most popular public interfaces seem to be based either on a web service or on a web application. Among the web services, the EC2 interface specified by Amazon seems to be the most popular one, and most of the projects implement it to some extent.

	Web application	Web Service	Other
VCL	Yes	No	XML-RPC
Globus/Nimbus	No / Upcoming? [18]	EC2 / WSRF	No
OpenNebula	No	OGF OCCI / EC2	XML-RPC
Eucalyptus	Yes	EC2	No
Enomaly	Yes	REST	No
OpenQRM	Yes		

Table 4 - Public interfaces

Table 4 shows the public interfaces exposed by the projects. OpenQRM did not document any programmatic interfaces for its functionality, which severely cripples its usability on a cloud scenario.

2.5.5 Licensing

As this is an academic project, and there is no allocated budget, both the licensing and the price become a factor when choosing the infrastructure manager. All of the projects analyzed have a free version, those are the versions that will be compared.

The terms of the license agreement are also important as they may hinder a possible commercial use of the end results. As an example, the GNU Affero license states that any software that supplies a service over a network, should have its source available. This is an extension of the usual GPL license terms to networked software.

	License
VCL	Apache License v2.0
Globus/Nimbus	Apache License v2.0
OpenNebula	Apache License v2.0
Eucalyptus	BSD up to v1.5.1 / GPL3 from v1.5.2
Enomaly	GNU Affero General Public License v3
OpenQRM	GNU GPL v3

Table 5 - Licensing terms

2.6 Choosing one infrastructure manager

Due to budget constraints, all of the software used must be free (as in beer), therefore all the projects that require payment are automatically excluded. A search for projects that implement the required functionality showed that a great number of them started as grid management platforms and are now turning to cloud interfaces, leveraging previous knowledge on distributed computing platforms. Some are quite new, and do not yet gather community consensus as to their quality.

The ones that stood the initial challenge are compared bellow. An initial look at the contenders shows that there are many interpretations of what are the most important features of an IaaS cloud.

The contenders are Apache VCL, Nimbus, OpenNebula, Eucalyptus, Enomaly, OpenQRM and ConVirt. Of these, ConVirt was quickly dismissed as an option as at the time of deployment, the documentation states that it crashes intermittently. However, its feature list is quite large and this project may be worth revisiting at a later time.

Apache VCL started as an infrastructure manager for the North Carolina State University. Among all of the projects, it is the one with the most advanced scheduler, as its main purpose is the allocation of computing resources for classes. The two major downsides of this project are the lack of a publically documented interface, and the lack of support for open source hypervisors.

Nimbus is a great candidate; it supports the EC2 API, and has special functions to deploy pre configured clusters. However, this project only supports the Xen hypervisor (KVM support is being written), which limits the freedom of choice.

OpenNebula features both the EC2 and the OCCI interfaces. It also supports a wide choice of hypervisors. Its driver based architecture allows for easy development of new features as well as the replacement of internal features with ones that provide the same interface.

Eucalyptus also features a good specification list, but has the downside of only providing a single public interface. The Eucalyptus interface is compatible with EC2, the most widely used interface, however this may change as new interfaces are being proposed as standards.

Enomaly is a commercial project with a crippled community edition. From the developer point of view, the only point of contact is the mailing list. This severely reduces the ability to use this project on an academic environment where modifications to the core functionality may be required at any time.

OpenQRM features an architecture similar to the one of OpenNebula, although the language used to describe it is quite different. OpenQRM calls it a plug in based architecture. The downside of this implementation is that no interface other than the web interface is documented, which takes it off the cloud computing scenario and into the datacenter virtualization market. However if this project ever introduces one or more programmatic interfaces (web-services or rpc), it will automatically become a great candidate for an IaaS management solution.

From the platforms described above, OpenNebula stands out as the best all rounder, it supports Xen, KVM and VMware as its virtualization backends and can use them simultaneously if care is taken to mark each virtual machine template with the appropriate hypervisor requirement. On the interface side, it features an RPC2 interface on top of which other interfaces are implemented such as part of the EC2 and OCCI interfaces. As this project is part of the larger RESERVOIR project, it should be around and be supported for quite some time, and an ecosystem is forming around it with satellite projects that add to the base functionality.

2.7 PaaS implementations

While there is a plethora of projects that claim to deliver a good IaaS implementation, and some of them actually deliver a good product, that is not the case when speaking about implementations of the PaaS service model.

In fact, no free and functional PaaS implementation was found for inclusion on this project, therefore this is for sure a field that is lacking on academic exploration.

Some PaaS implementations should however be mentioned here as they supply a framework that is deployable on private infrastructure, meaning that the product itself is a platform, not usage of a private platform as a service.

2.7.1 Manjrasoft's Aneka

Aneka is a .NET based, multi platform PaaS implementation produced by Manjrasoft [19]. It is not a free product, therefore no experiments were performed on this software.

This product features a very complex resource scheduling and reservation algorithm that was expected on a grid middleware instead of on a cloud PaaS; it also features a very strong authentication mechanism that is used throughout its inner services. Due to its programming models, a traditional .NET application requires some porting before it can take advantages of the features of the

platform. The application data is stored on a RDBMS that is connected to the platform via a plugin, meaning that alternative data storage models are a possibility [20-21].

The provisioning model used by this project is heavily market oriented, it allows complex accounting, and the establishment of Service Level Agreements (SLA) with clients [20]. This is a major advantage if Aneka is to be used to host a public PaaS. However, its usage of a RDBMS may hinder such a wide public deployment.

Although Aneka has some market oriented features, no one created a public cloud based on this project up to the moment. This may be due to licensing issues, as it is a proprietary project and therefore it is not easy to modify its core to cater to a specific company needs.

Another reason for the lack of a public PaaS based on Aneka can be that this project does not actually show off a cloud usage scenario on its web page, instead it publicizes its usage on batch oriented parallel computing, a workload that is more akin to grid computing.

2.7.2 JBoss Cooling Tower

The JBoss community is developing the Cooling Tower [22], and advertising it as a turnkey PaaS solution. This project is based on a traditional JBoss Application Server cluster, and the automatic deployment of the required software stacks to scale said cluster.

For the storage of application data, Cooling Tower recommends Infinispan, a persistent extension to the java.util.Map interface, that allows a simple data model, yet it features very interesting core features such as automatic management of data copies and balancing the data among available nodes [23].

This project should allow the usage of most of the J2EE specification, with exceptions for every part of the standard that relies on the existence of a relational database for persistence, such as the Java Persistence API (JPA).

2.7.3 Google's AppEngine

This product's business model is described in some detail on Apendix A.2. On the technical side, it supports both Pyton and Java programming languages, and it features an impressive structured storage solution.

AppEngine's data storage solution, BigTable, when accessed from a java program features an interface similar to JPA, yet it supports a different query language similar to SQL but without some of the complex queries, deemed Google Query Language (GQL) [24-25].

3 OpenNebula Deployment

In order to perform any kind of study on the subject of cloud computing, a cloud is required! As there is no knowledge to be gained by starting with one of the available commercial IaaS offerings (and there is money to be lost!), the first step is to deploy a small test cloud on the available hardware. Its resources need not be large, it just needs to be large enough to prove that the concepts that are to be explored are valid and functional.

On section 2.6, the choice for OpenNebula as the best all rounder of all of the infrastructure managers is made. OpenNebula proved to be truly open, well documented, flexible and simple. Those are the main requirements for the base IaaS cloud that is to be created as a testbed to explore new software architectures.

Over the next sections, all of the design choices made for its deployment are detailed. Some of the choices are in fact not the best choices, but they are the ones that allow a greater level of flexibility while playing with the resources that are allocated for the project.

OpenNebula has a core that manages its several drivers, a scheduler that performs the matching between VM's and hosts on its simplest form, and several drivers that actually take the actions required to fulfill a request [26].

3.1 Testing deployment configurations

In the beginning, two hardware systems were available. One, a Pentum III class desktop that was used as an OpenNebula frontend, the other, a Dell PowerEdge 4400 with two xeon processors (Pentium III family) and five SCSI hard drives.

The desktop system was used to deploy the OpenNebula software, and the PowerEdge was used as a virtualization host. During this phase, the only tests performed were on the integration with different distributions of Xen, and with VMware ESX 3i.

The search for a good Xen distribution that runs on dated hardware was quite lengthy. It involved several full installations plus the time to understand the specifics of each installed distribution added to the time required to understand what distribution specific configuration options are needed to integrate each of them with the OpenNebula manager.

Several distributions were installed, but most of them did not fully support the Xen hypervisor as the kernel recommended by Xen is quite old and not all the distributions have the manpower required to forward port the patches to recent kernels. This problem should become mute soon as the bulk of the Xen patchset is being integrated with the mainline kernel [27]. The only distributions that supplied a recent dom0 kernel with Xen support were openSUSE and Fedora Core. OpenSUSE lead to server hard locks after a few days running a VM. No attempt was made to figure out

what was the problem due to time and resources limitations. Fedora Core was not installed as it is known for being a bleeding edge distribution often used to try out experimental features [28].

Of all the tried distributions, the ones that seemed to perform better were the RHEL based CentOS and Scientific linux. The second was chosen over the first as it is already installed on other servers at the datacenter. The system was installed on a logical volume created on a software RAID 1 spanning two of the server's disks. On the other three, a RAID 5 configuration was created to store the cloud system images.

3.1.1 Choosing the hypervisor

The choice of a hypervisor can make or break such a project, especially if the hardware is considered legacy hardware. Before installing anything, a little research showed that the available hypervisors all have advantages and disadvantages.

Xen is the most mature of the open source hypervisors, many production systems use it, and its usage is well documented. The downside of this hypervisor is the lack of support for it on the vanilla kernel, which means that each linux distribution must supply their set of patches. This leads to a fragmentation of the code, meaning that there will be distribution specific problems on top of version specific problems. Meaning that the choice of a linux distribution must be made with extra care if Xen is to be used.

KVM is regarded as many as the future standard linux hypervisor. Although it is already used on many datacenters in a production environment, the version shipped with the enterprise distributions generally used on server class hardware is quite dated. Also, this hypervisor only recently added support for hardware without virtualization extensions which disqualifies it from the usable hypervisors on this deployment.

VMware ESX 3i does run on older systems and its binary rewrite virtualization technique allows it to run legacy guests without modification, however it does have a major drawback. After the initial trial period, if the free serial number is entered, the public API used to manage the server will become read-only, disallowing management operations via that interface [29].

Given ESX 3i limitations, the choice was reduced to Xen and KVM. Choosing Xen means that quite a few linux distributions must be tested until a good balance between age and ease of management is found. Choosing KVM means that a recent distribution that includes the new paravirtualization code must be chosen. The distributions that include that code are at the moment targeted for the desktop market, with a very fast pace of updates, leading to a greater slice of time allocated for system administration in order to keep up with the constant releases. Preliminary tests using desktop oriented linux distributions also showed some problems while booting on some of the available hardware, so KVM was discarded for the time being.

3.2 Configuring OpenNebula

On an early stage, the front node for OpenNebula was installed on an old Pentium III desktop. During this stage, the documentation on the OpenNebula web site was an invaluable resource that covered almost all of the problems found [30]. The only real problem found during the install process that was not covered by the documentation was the resolution of the dependencies for some of the ruby gems required by the software.

The configuration of a Xen host was also very well documented, and it was possible to validate that the process required to deploy a new system into the cloud was fully functional. At this point it was not possible to test all of the cloud's functions as some, such as migration, require at least two virtualization hosts.

In order to compare the functionality of the Xen backend with that of the VMware backend, the virtualization host was backed up, and a copy of ESX 3i was installed.

Adding support for VMware on the OpenNebula frontend requires both the VMware SDK and the OpenNebula VMware driver to be compiled. Although the process is documented, the documentation on VMware's site was not simple to follow as each modification to the environment is described in detail, but no step by step instructions or installation script was found.

Once the driver was installed and the compute node was added to OpenNebula, the problems began! The first of those problems was a licensing issue; the VMware API becomes read-only when the free serial number is entered. In order to proceed, the serial number was removed and the host went back into trial mode. The only possibility of using ESXi with OpenNebula is by acquiring one of their enterprise licenses, which leaves all of the functionality of the API intact.

The second problem faced was with the OpenNebula driver. The driver supplied with the current release (1.4) does not have all of the functionality that is present on the Xen or KVM drivers. It misses quite a few options and has a serious security fault. One of the missing functions was the attachment of a CDROM drive, a process used on the contextualization of virtual machines. This rendered the driver useless as no contextualization could be performed during the boot process (setting up the IP address and administrative login). The security fault was that the administrative username and password for the ESXi hosts were included on the command line when invoking the driver, this way any user on the machine that runs the front end would be able to easily access that information.

In order to compare the hypervisors, the VMware driver was improved with better CPU management, support for context and regular CDROM, and a security flaw was removed in the process. The patch was submitted upstream and is expected to become part of the next major release of OepnNebula.

3.3 Driver comparison

The drivers were compared based mostly on subjective terms as the deployment of each one has a few differences mostly on the distributed storage area that would invalidate any benchmark results. It must also be noted that whatever the results of such comparison, the Xen hypervisor would be chosen due to economical constraints, given that the cost of a VMware license is so high that it would be cheaper to buy new servers!

The time spent by a systems administrator managing the virtualization servers is quite precious, as that time could be spent managing and adding more functionality to the cloud itself.

ESXi requires very little of the system administrator's time, given that it is distributed as a firmware image. The only time spent on the configuration of this server was the time required to enable the (unsupported) SSH console, the time that takes to create a user, and the time required to configure the storage backend.

Xen on the other hand requires a few hours of system administration before a virtualization host can be added to OpenNebula. A user must be created, that user must be given some administrative abilities required to manage Xen, the storage must be properly configured, some utility software must be installed, and finally the system must be regularly updated as any linux system.

Another detail that must be mentioned is the driver functionality. Even with the patches created especially for this project, the functionality of the VMware driver is below that of the Xen driver because it does not support the migrations of virtual machines. This function is not essential for regular operation, however it is essential if one of the virtualization hosts must be taken down for scheduled maintenance.

KVM was not tested as the ability to perform paravirtualization on this hypervisor is newer than the kernel supplied with the linux distribution used, and the hardware does not have virtualization extensions. This hypervisor uses different strategies than Xen, yet its functionality is about the same, and it also requires a full linux install to function, so most of what was said about Xen administration also applies.

All of the above means that the choice of the hypervisor will seriously impact both how the time of the system administrator is spent and the reliability of the deployed virtual machines. If the manpower required to manage full linux hosts is available, Xen is at the moment the best choice, as it allows for paravirtualization on dated hardware, and will take advantage of virtualization extensions as they become available with the migration to new hardware.

However if the system administration time is a problem, there is a VMware enterprise license available, and if the Virtual machines deployed can be taken down at any time, the choice is VMware, as it is an "install and forget" environment, as most of the time required by this solution is on the compilation of the OpenNebula driver.

3.4 The next step: more hardware!

After the initial evaluation of the software, comes the time to actually create a computation cloud, and there is no way a cloud with a single virtualization node can be called a cloud!

This means that the cloud is starving for more virtualization nodes. At this point a “new” server became available, a PowerEdge 2850, with two Xeon CPUs 2GB of memory and two 30GB SCSI drives on a RAID controller.

The Linux install process was effortless on this server. But a decision had to be taken about where to place the OpenNebula frontend as the old desktop was not nearly strong enough to manage a large number of virtual machines or virtualization hosts and still perform its original duties.

With a clean linux install on both the servers, both the systems were configured to run the frontend and storage if required. At this point a closer look at the performance of each of the servers was taken. The older of the two has more disk space available; however a benchmark revealed that its performance was worse than that of the new server with its disks on a hardware raid controller. On the downside, the tests also revealed that when deploying a large number of VM’s at the same time, the newer server shows a load spike, rendering it nearly unusable for the time it takes to copy all the files.

On the networking side, the older server has a 100Mbit Ethernet card, while the newer one has two gigabit interfaces. Even with only one of those interfaces connected, the newer server has an advantage. If it is to manage a large number of virtual machines, the distributed storage must have enough network throughput to allow the remote virtual machines to operate on their disks.

The newer server was chosen due to the fact that it has a higher network throughput and faster disks. This means that a large amount of disk space is wasted on the older server with a higher disk capacity.

As of now two more storage solutions remain to be tested. One of those is the decoupling of the storage from the frontend node, placing it on the datacenter’s NAS. The other is the usage of a cluster file system. Those solutions should be tested if this project is to be enlarged.

The NAS solution was not tested due to the high amount of IO operations that would be performed. There is the fear that if this solution is to be used, the remainder of the department grinds to a near halt while the files required to deploy a large number of new virtual machines are copied.

The cluster filesystem solution was not tested because that solution does not make sense with only two nodes as reliability would be severely reduced. However if dedicated storage nodes become available, other solutions will be promptly tested.

3.5 Contextualization

The contextualization of virtual machines is an essential step of the deployment process on a cloud. This process is composed of two steps. The first is the configuration of the network interface with the required IP address, and the second is the configuration of the administrative login for the virtual machine. Further steps may be required such as installing a software package, or deploying an application on an application server, those steps are not yet implemented, however once the initial contextualization scripts are in place, it becomes easier to perform other administrative tasks.

OpenNebula achieves this by attaching an ISO image with the required information during the boot process. The ISO image always contains a file used to set a number of environment variables. Added to that, a number of user defined files can be added to the image. The scripts used at the moment setup networking, and put an SSH key in place for the root user in order to allow the remote operation of Linux hosts. The virtual machine image must be previously configured to look for and use these scripts during the boot process as part of the preparation required to run it on a cloud environment.

Another way to perform a similar task is by using a customized init script that sets up part of the system during the initial boot process, leaving the remaining tasks to the contextualization script.

A mix of both of the above contextualization solutions is used when booting Linux systems, using the first script to set up some basic networking information to be used during the boot process, and the second to perform more complex tasks such as setting a host name, or even installing a software package, as those tasks require external information.

3.6 Networking

The OpenNebula documentation always uses a bridged scenario that requires the gateway to route traffic to all of the IP networks used on the cloud. However at the IEETA datacenter, routable IP addresses are a valuable resource. This restriction lead to the usage of a private network for all of the deployed VM's, which means that they could communicate with each other, but not with the outside world. Although communication with the outside world is not strictly needed for this project, it would ease the administration of the virtual machines as it allows one to customize a machine at will, shut it down, and reuse its image as a new deployable image.

In order to connect all of the virtual machines to the internet, some sort of NAT must take place. The first idea that comes to mind is usually performing that on the network gateway, but on this case the gateway is off limits. The second option would be applying NAT to all outgoing traffic on each of the virtualization nodes, but that would break the connectivity between virtual machines deployed on different virtualization nodes.

The solution to this problem technically simple, however the final network architecture ends up quite complex as the physical network ends up with two logical networks and two routers. Given that there is no administrative access to the datacenter router, one of the hosts that can be modified must be chosen as a router for the virtual machines. The choice was to use the newer server for this task as it has a gigabit interface that should be more than enough to route internet traffic for several virtual machines. However not all of the outgoing traffic should be translated, so the IPtables rule set does not modify outgoing traffic for other virtualized hosts, but translates all traffic that is directed to other hosts that are not on the same IP network.

The final step required to allow full connectivity between all the hosts and the internet is adding the IP address selected as the gateway to the router's ethernet interface. Now that the IP connectivity is possible, the addition of the host's resolv.conf file to the contextualization ISO image finishes the process, and allows the virtual machines to connect to each other as well as to the internet.

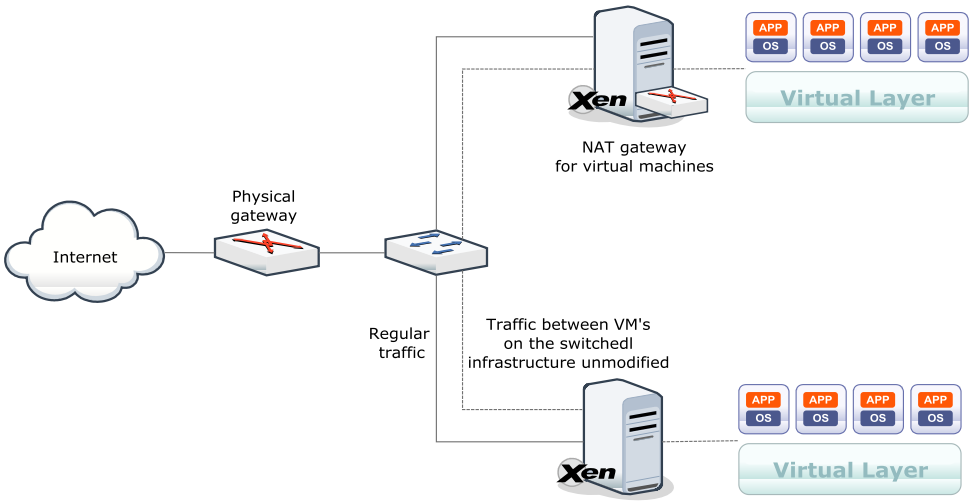

Figure 2 - Network setup

This networking setup allows full internet connectivity and requires no modifications to the existing network infrastructure, therefore adding new private networks becomes a matter of modifying the IPtables script and adding an IP alias to a host.

Figure 2 illustrates the logical connections between physical hosts and the internet, and between the virtual hosts and the internet. Virtual hosts must route their internet traffic via the NAT gateway, otherwise access will fail as the physical gateway (department router) has no knowledge of the existence of the private network used by the VM's.

3.6.1 Remote access

OpenNebula supports various ways of accessing the console of the virtual machines. At the moment only SSH is used as the VM's are on a non public network, and therefore in order to access

them, one must first access one of the physical hosts. If a set of routable IP addresses are allocated to this project, accessing the VM's via a remote access protocol such as VNC or RDP becomes possible.

At that time, the core of the OpenNebula will be modified in order to add further information to the user table on the database to allow the addition of a hook that sends an email to the owner of a VM with the required remote access information.

At the moment, no steps were performed to add this functionality as the OpenNebula team is working on a whole new user management system that is to replace the current one on the next major release.

3.7 Extending the functionality

All the steps described previously lead up to the correct setup of a functional virtual infrastructure manager. It is not yet possible to supply any service to the public as no public interface exists as of yet.

As said on section 2.5.4, the greatest difference between datacenter virtualization and a IaaS cloud is whom has the ability to control what infrastructure is deployed on any given moment. On a cloud computing scenario, that control is shifted towards the clients, which are given the ability to deploy an arbitrary software stack on a virtual machine on demand.

In order to be able to supply such service to the public, an interface has to be exposed. OpenNebula enables by default its own RPC2 interface, but that is a proprietary interface that is specific to this implementation. For that reason, two other interfaces can be enabled. The first and most known of them is the EC2 interface, widely known due to Amazon's compute cloud. Another interface supplied with OpenNebula is the OCCI interface. OCCI is an interface developed by the Open Grid Forum, and it is expected to become a standard interface in the future [31].

Each of these interfaces has limitations. The implementation of the EC2 interface is at the moment incomplete and does not work with most of the tools designed to work with Amazon's implementation. The OCCI interface is not widely used and therefore there are few client tools available.

However limited, support for both protocols was enabled. Both of the protocols work well with the core infrastructure manager, but it is not recommended to use both of them at the same time (in the same management application) as that would severely hinder application portability given that no other infrastructure manager implements both of these interfaces.

There are quite a few public interfaces for clouds, with more being proposed... As the time passes, only a few of these interfaces are expected to become widely used. At that time, It is expected that OpenNebula will fully implement some of those interfaces with their full functionality [32]. Until then,

OpenNebula supplies a set of client tools that are usable with the current implementation of each interface.

4 Proposal of a new PaaS architecture

There are a few commercial IaaS offerings. Those have the advantage of not locking in an application to a given vendor; however they also end up being more expensive when compared to a PaaS offering if one wants the deployed application to be scalable.

The above is an empiric statement that may not hold true for all the applications deployed on a cloud computing scenario, yet the deployment of a high availability typical application on an IaaS cloud requires a few instances to be available at all times. At the minimum, a load balancer, an application server, and one database server need to be deployed if the application is to be available and have the possibility of scaling with load increase or decrease.

When compared to a PaaS this is quite expensive, especially if the application has a low traffic volume. Commercial PaaS offerings also have a large downside, vendor lock in. As stated earlier, if one wants to change from one vendor to another, all of the features that rely on the vendor platform must be reimplemented. This may end up being almost a full application rewrite, but at the minimum it would mean a rewrite of the persistence and session layers.

4.1 Common application deployment architectures

A simplistic scalability definition states that scalability is the ability of an application to maintain a given service level when the load increases. There are several ways to achieve scalability, most of them falling in two scenarios, scale up or scale out [33-34].

Scale up is the solution that most small companies will use for their in house applications, where the maximum load is a factor of the size of the company. This technique consists on *throwing money at the problem* by building servers with faster storage, more CPU's, and more RAM to support the same application. This will work well if the application has a known peak usage, but may fail if the load increases above the server capacity.

Scaling out, on the other hand, means building an architecture that will see its performance increased simply by the addition of new machines to any of the application layers. This solution is becoming more and more popular for IaaS cloud deployments, where the available virtual machines have limited resources, yet are available in quantity [35].

The typical application scaling pattern for applications that started small and used leased hardware collocated on a remote datacenter used to follow the steps described below.

For small to medium applications the usual deployment scenario requires one application server and one SQL database. For very small deployments, the services can even be hosted on the same machine. As the application load grows, the deployment becomes more and more critical.

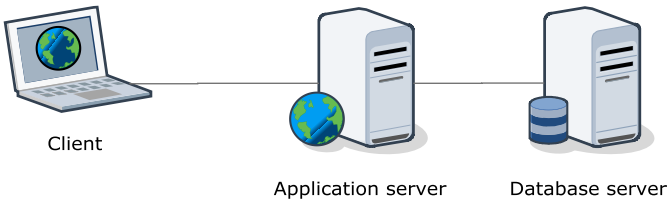

Figure 3 - Small application deployment

As the application grows, the database and application services must be split into two different machines. This allows for a little more load, yet it does not yet perform well if an application has a load peak as it is not a scalable architecture.

Scalable architectures require that all of the application state is saved on the database, as well as a modified application server that knows where to look for that information. This allows the distribution of the load between all of the available servers, allowing a larger number of requests to be served on the same amount of time. Nowadays, most application servers already allow such deployment scenarios [36].

As the load increases, the database must also become distributed over several machines in order to allow an increased performance. Most of the SQL solutions available do allow some sort of clustering but do not support any sort of real time scalability, meaning that in order to take advantage of any new server the database must be rebuilt [37-38].

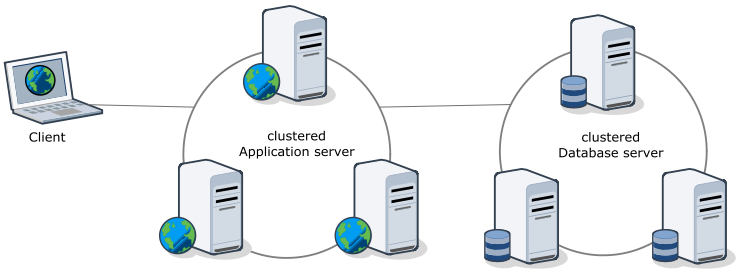

Figure 4 - Scalable application deployment

If an application follows the above patterns, it will in fact scale out to a limit. That limit is the limit of the SQL server's ability to scale with the addition of new node, as performing a complex query on data spread over several nodes severely hinders the performance of the current SQL servers.

4.2 A multi tenant elastic platform

In order to take advantage of the possibilities that cloud computing brought to the computing world, a new application deployment architecture must be devised. Previously, simple reasoning showed that PaaS was the ideal service model to develop new SOA applications. However, PaaS comes with the disadvantage of vendor lock in.

Over the next pages, the old and the new come together to form a new PaaS based on well known technologies. Such a platform has several advantages both for existing applications as for new applications.

Existing applications will transition easily to the platform proposed here, without fear of vendor lock in as the platform is well known, and easy to deploy on a private infrastructure if needed.

New applications can take advantage of the economy of scale, as hosting on a multi tenant platform is usually cheaper than renting the required infrastructure, even on a pay as you go model.

At any point, an application can switch vendors without modification. Cloud computing is a business model, and as such, prices will always follow the supply and demand laws. Using this platform, service providers are able to stimulate direct competition, leading to platform improvements and cheaper prices.

A major problem that should be avoided by a public PaaS specification is the strategy commonly known as “Embrace, Extend and Extinguish”, where a powerful company takes over a given platform, and extends it with proprietary functionality making sure that interoperability breaks in order to extinguish the concurrency [39].

A novel improvement over existing PaaS is also presented, as the proposed platform supports SOA application deployment via the integration of a modified UDDI service with the original design. Such integration allows an increased reliability of the information stored on the UDDI server as that information is gathered and validated upon application deployment.

4.2.1 Storing the data

The persistence layer of an application can rely on a series of technologies. From flat files to full Relational Database Management Systems (RDBMS), every solution has its strengths and drawbacks.

The most common solution used for web applications is a relational database. However, with the advent of cloud computing, alternative storage solutions with better horizontal scalability are being proposed.

Those alternative storage solutions, relinquish some of the ACID properties of a traditional RDBMS system for speed and scalability. Two commercial examples with some success are Google’s BigTable, and Amazon’s SimpleDB.

The open source community is now starting to build similar products, such as Cassandra and MongoDB. Although some of those are on production on applications such as Facebook or Twitter, they are in fact still under heavy development and lack some of the features that are required on a real

multi tenant platform, such as data separation or a reliable query parser. The noSQL[6] site keeps an up to date list of non relational data store implementations.

4.2.1.1 Key Value data stores

These are the simplest of data stores. These solutions are very similar to the already widely known memcached data store, but they offer persistence, something that memcached does not offer.

These solutions store a scalar value under a given key, and can only retrieve the value based on the key. This means that it is not possible at all to perform complex queries, in fact it is not even possible to query the datastore based on a property of the stored data.

This type of datastore is very scalable and fast due to its simple architecture. Its major downside is also what allows it to be scalable and fast, its simplicity.

In order to use such a datastore to build an application persistence layer, complex coding must be done to overcome some of the datastore limitations, and it is predictable that certain applications, even if they do not need data integrity, need a way to store complex data and query it based on various data properties.

4.2.1.2 Document stores

Document stores add to the key value stores because they allow the storage of complex documents. The documents are a series of attribute-value pairs, where the values can be complex data types.

Document stores are quite scalable and can deliver good performance levels, as the stored data can be indexed. Yet at the moment, none of the available document stores is mature enough to be useable on a multi tenant platform where strict data isolation and a reliable interface is a must.

Still, the data model offered by this class of systems is very close to the serialization of an object on an OOP language, turning it into a very interesting data storage model.

4.2.1.3 Column oriented data stores

This type of data store can hold a series of records with a well defined schema, yet more properties can be defined on a per record base during runtime. These data stores, contrary to most RDBMS systems that store data on a per row basis, store all of the values for each column together.

This storage model has advantages over row based storage if the operations performed on the data tend to be column oriented instead of row oriented. It is commonly used for data wharehousing solutions where analytical processing is more important than transaction processing.

[6] http://nosql-database.org/

However these systems allow complex data models, which means that they start to suffer from the potential scalability problems described next for traditional RDBMS systems, but lack the amount of development that the traditional systems received while trying to attenuate them.

4.2.1.4 RDBMS systems

These are the most commonly used storage systems for application data. They are well known, feature full ACID compliance, and most of the commonly used OOP languages feature some type of widely used abstraction layer on top of the relational model.

The weakest point of these systems lay on the horizontal scalability, as they feature complicated locking mechanisms and allow complicated queries that may become quite slow when several nodes must be contacted to form the result.

The major vendors are now working on better horizontal scalability, and some even on elasticity, each with a different approach. This work is quite promising, but up to now some scalability problems still arise, especially on large database clusters.

4.2.1.5 Object oriented databases

This type of database offers the same ACID properties that are available on a RDBMS. Using such a database for data storage is quite transparent, as the programmer is already dealing with objects on the code, the difference being that the database objects allow transactions and locking, which also solves concurrency and persistence problems.

The downside of these databases lies on the available implementations, as none of the known open source implementations allows the distribution of data among several nodes. This severely impairs the scalability of the persistence layer for a multi tenant scenario.

4.2.1.6 Other datastores

Data storage is a very complex field, and no single solution solves all of the requirements for every application. The solutions presented above are the ones with wider community adoption, or which seem to have better potential, but in no way is the above to be used as a comprehensive list of data storage models.

Other solutions are available, such as graph databases or memory only databases. Those are not presented here because they either lack some required features by design, because none of the known implementations features a community of users large enough to allow a steady development plan, or even because the storage model itself does not present any advantages on a distributed scenario.

4.2.1.7 The choice

On a IaaS cloud, all of the machines have low CPU and memory, as well as small disks, so it is not possible to scale up efficiently. The data storage model is also of great importance, as using an obscure data storage model does not capitalize previous knowledge and may in fact deter the adoption of a platform.

The chosen solution must have a few characteristics. Data isolation is a must, given that the final product is a multi tenant platform. A shared nothing data storage model is required because of the characteristics of the cloud computing model. Elasticity is a must, as the datastore must not be put offline just to add a new node. Data redundancy is a plus as one should always plan on having some nodes failing. Robustness is required because no sanitation is to be performed on deployed applications, and a malformed query (intentional or not) must not affect the datastore normal operation.

The requirement list presented above is quite large and in fact, none of the available free solutions seems to fulfill all of them. Some commercial solutions such as Objectivity/DB claim to fulfill all of them, but such claims cannot be verified as they would require extensive testing.

Of the above, document datastores seem to be the most interesting solutions as they are very close to the OOP model, and the community is developing solutions with some very interesting features. Yet at the time there are still a few problems with the available solutions.

CouchDB is very interesting performance wise, and it does scale well if there is enough space on each host's disks, but its replication model does not allow for data partitioning, which is one of our must have requirements [40].

The solution that is to be chosen must support an authentication system that allows each application to access its own data, and no other. At the moment, the MongoDB datastore does not support such a feature when using the sharding configuration [41].

Key value datastores are indeed fast, but due to their simplistic data model, they are not suited to perform as a generalist datastore. This means that they are an option for a future feature, but not required on the initial stages of development.

More unusual solutions such as graph databases are just too unknown or have licenses too restrictive to make them a viable solution. As an example, Neo4j was not discussed due to its relative obscurity, but it solves some quite interesting problems [42].

The solutions referred above are all part of a recent trend to move away from relational databases to alternative storage models such as document stores, key value stores, or even a graph model for large scale storage solutions. Most of those solutions provide horizontal scalability features be it via replication or sharding. All of them are extremely fast when used to power a solution that takes

advantage of their intrinsic capabilities, yet none of them is mature enough to power a multi tenant platform without some serious development work.

At the moment the only storage solutions that feature a shared nothing architecture, and have the features required by a multi tenant application are RDBMS. These solutions do have some scalability issues, but a lot of work is being put into its mitigation.

A closer analysis of the resources that could be potentially made available to this project also shows that it would never hit the scalability limitations of the freely available and time proven RDBMS's. This leaves us with a choice between a PostgreSQL or a MySQL based datastore.

Although this conclusion does not take into account all the use cases for the actual platform that is to be deployed, it ends up being a step towards large scale adoption of the platform, as it capitalizes on common knowledge. However, choosing an RDBMS as the storage model does hinder the platform's ability to scale to massive numbers of applications.

So, the ideal solution will be an RBMS that features sharding over several nodes, and is able to withstand the failure of any node.

At the moment, both of the major open source RDBMS are working on ways to simplify sharding for horizontal scalability, as well as to make it transparent to the end user.

On the MySQL side, the spider storage engine automatically partitions data according to some rules on the database schema. This storage engine requires very little modifications to the actual application, and avoids the usage of sharding code on the application logic, such as hibernate sharding. MySQL cluster also claims to achieve horizontal scalability and elasticity, and features a shared nothing architecture, which is a plus for the projected scenario.

On the PostgreSQL project, several plugins try to support some sort of horizontal sharding. One of the most promising projects is gridSQL. Another interesting project is pl/proxy by skype, but using this project requires heavy modifications to the application logic.

Both GridSQL and the spider storage engine require slight modifications of the persistence layer in order to be used efficiently, and the current state of integration with ORM like hibernate is unknown, but GridSQL seems to be at a disadvantage as it does not expose all of the postgreSQL dialect and is optimized for data warehousing.

MySQL cluster is apparently the most transparent solution, and is the recommended solution for the current deployment scenario. However extra work must be put into the development of an architecture that does the most with the available resources, such as using a transparent cache in front of the database to speed up access to the most accessed resources [43].

With the maturing of different technologies, other solutions are expected to become available. The technology that seems to be the most promising is the document datastore, as it is close to the OOP paradigm, and a lot of community effort is going into the development of some implementa-

tions of this data storage model. This means that a stored document may contain all of the properties of an object at any given time, therefore being enough to restore an object state from the datastore.

Future work may either add a document store to the architecture, or prove that the SQL backend is not scalable enough and completely replace it. However, if it is to be replaced, it must be done as soon as possible, preferably before public availability of this PaaS to avoid future application migration problems.

4.2.2 The application server

4.2.2.1 Programming languages

Choosing an application server for a cloud computing platform is not an easy task. Several implementations are available, but none of the open source ones are geared towards providing a PaaS.

The first choice has to be a programming language. Many options are available, but the choice must leverage in house and community knowledge. As most of the programming disciplines currently taught at the University of Aveiro use java as a programming language, it seems to be a good choice. Other languages can be added later if this architecture proves itself useful. Both .NET and Ruby are good candidates as those languages are becoming fierce competitors for the established java *status quo*.

The Ruby programming language is compilable to java bytecode, this means that a platform that fully supports the java standard is easily customizable to support this language. Yet no research was performed on this subject as the initial user base is geared towards java and .NET.

In due time, .Net poses a great addition to the supported technologies, however, as it is expected that some modifications must be performed to the core of the language or to the application server, the only viable implementation is based on Mono, an open source implementation of the .Net technology [44]. But there are still concerns among the open source community as to what is the legality of the project outside of Novell's agreement with Microsoft [45-46]. As well as to the legality of the ASP.NET and ADO implementations that are built on top of the base .NET CLI and are not covered by the agreement at all [47-48].

So, for the initial implementation effort, the choice falls on a java application server, as the available implementations are free, modifiable, and have the advantage of running on multiple platforms unmodified.

4.2.2.2 Application server architecture

The first solution that comes to mind is a cluster of application servers, commonly known as a *web farm.* However, this setup requires that each of the application servers must have access to the full pool of deployed applications. Given that most of the current IaaS implementations either do not support the attachment of persistent storage, or provide it at a cost, the only solution would be building a distributed storage solution using a large number of persistent storage instances. As this would severely increase the runtime cost due to the allocation of extra compute nodes just for storage, an alternative solution must be found.

Given that the clustering facilities cannot be used to scale the application server layer, decisions have to be taken as to how much of the J2EE standard can be supported, and which application server should be used to support it. In order to do so, we start by looking at how the current application servers allow stateful communication over a stateless protocol.

HTTP is a stateless protocol, so in order to implement stateful functionality over HTTP, some information has to be stored by the server. Each of the application servers available implement a session manager. This component usually stores session data on the memory of the java virtual machine (JVM) that is running the application server.

In order to share the HTTP session between several application servers, that information must either be replicated in memory, or stored on a backend database. In fact, all of the application servers implement and use those features when working in clustered mode, however as said before, their clustering facilities require a large amount of disk space which is not available on regular cloud VM's. This means that the standard implementation of the session manager and other components that allow stateful operation must be replaced with versions that store its information on a persistent backend, or the application server must be modified in such a way that "partial clustering" becomes a possibility.

The modification of the individual classes is preferred over the modification of the application server clustering code because this solution allows deployed clouds to take advantage of new application server releases without modifying the server, an advantage because modifying a complex application server is likely introduce bugs.

Higher up the J2EE specification are the stateless EJB's. Due to their stateless nature, there is no need to actually persist any information, so they can be used without modification.

Entity beans are another problem area, as there are different policies to deal with them. Some containers populate theirs information once, and modify them in memory until a time comes where all modifications are persisted. If state is to be shared among all of the application servers, extra care has to be taken by the programmer to persist any and all modifications if they are to be seen by a subsequent request that may be served by an arbitrary server. How to achieve this is dependent on the chosen application server.

Stateful beans are the most problematic area, clustered servers usually route all of the requests for a given EJB instance to the same server, but that is not an option on this scenario. This means that the state for every instance of an EJB also has to be persisted after every single request.Due to the expected complexity of dealing with the stateful EJB state, the support for this feature is left off the initial implementation effort, as this feature is not required to prove the validity of the platform.

No application server available today implements the required features. This means that modifications must be made to any of the available servers.

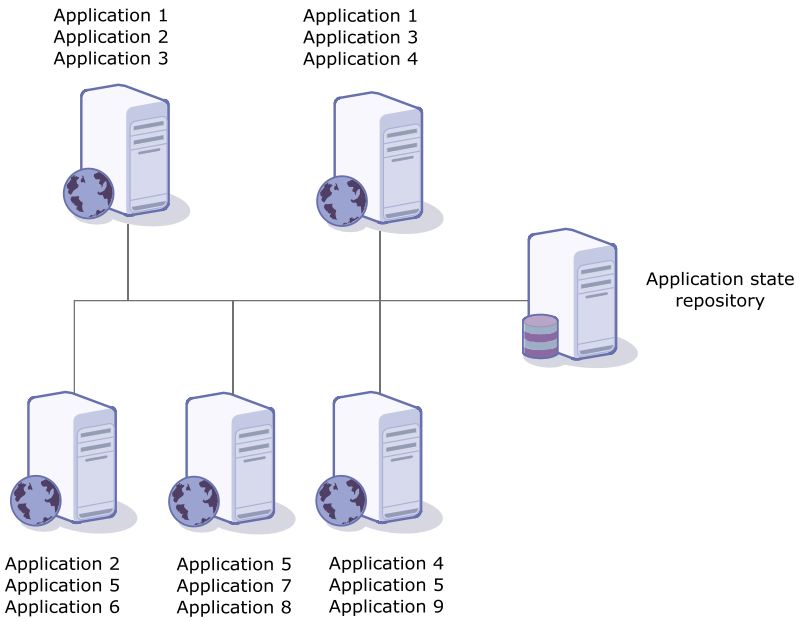

Figure 5 - Application deployment

Figure 5 depicts the expected result of the proposed modifications. A *semi clustered* application server, where a given server has a set of applications that may differ from those deployed on another server, yet the state information required by the application servers is kept on a shared repository containing the state of all the deployed applications. This means that each application server can operate without the need to exchange information with other servers, therefore removing the need for a traditional cluster and all the additional state synchronization chat that is required to maintain it.

4.2.2.3 Choosing the server

A survey of the available java application servers shows that they are split into two categories, the simpler category is the servelet and JSP server, where Apache's tomcat is the only serious contender, and full blown J2EE servers (either version 5 or 6 of the standard), where there is fierce competition between Glassfish, WebSphere and Jboss.

The requirements detailed above lead to the conclusion that any of the available J2EE application servers available must be modified in order to be useable in this scenario as none of them separate the possibility of persisting session information from the clustering logic.

This means that the chosen implementation must be as simple and well documented as possible, in order to allow a speedy development.

The most simple of the well known java application servers is Apache's Tomcat, however it does not implement the full J2EE specification. It is a great starting point precisely because it is the simplest of the available application servers.

Other free application servers that are widely used include GlassFish, Jboss and Geronimo. These application servers support the full J2EE standard and are certified either for version 5 or version 6 of the standard.

The choice of the application server must take into account the requirements of the platform. If it is to become a plain JSP/servlet container, then Tomcat is the appropriate choice. However if it is required to run a full J2EE application, then one of the certified application servers is required. If that is the case, a closer look at the specification and session management techniques used by each of them is required.

At the early stages of deployment, Tomcat is the best option as it is simpler than the other application servers. The modifications that are required on this server will allow the evaluation of the modifications required to integrate a certified J2EE server, while allowing initial development of simpler servelet based applications that can be used to test the platform on the early development stages.

After the early development stages, it will be time to move to a full J2EE server. When that time arrives, the choice should fall on JBoss as it allows integration with tomcat. However, a deeper analysis should be performed at the time, as the contenders keep evolving and new features may turn this into a void statement.

4.2.3 Becoming a multi tenant platform

All of the above does not specify the full architecture of the platform; it only deals with problems that must be solved in order to achieve full horizontal scalability using small servers.

In order to offer a scalable system, other considerations must be made. Data storage and application servers are of the utmost importance because they supply the programming interfaces that the platform users will use. However the true strength of PaaS is on its automatic scalability.

The previously described architecture for the application server requires that a single instance of the application server receives, processes, and replies to a client request. This requirement is in place because the clustering capabilities are not to be used. This means that the remaining functionality of the cluster must also be implemented by the platform.

4.2.3.1 Information repository

The information repository must be available to the entire infrastructure. Due to its nature, it is the single most important piece of the PaaS architecture. It contains all the information required to identify what infrastructure is deployed at any time, the information about the deployed applications, plus the information about what nodes are scaling out or scaling in.

Due to the nature of the information stored on this repository, either a RDBMS or a document store can be used.

At the moment, a document store seems the best option, with a preference for MongoDB for its data storage model. This document store does not support authentication on a sharded environment yet, but that feature is planned for an upcoming release, in the meanwhile it can be used on a single instance. This will allow a simpler data model even if at the cost of denormalization, and it will enable the writing of some documentation on using a non relational data store, to be used later by application developers.

When the authentication issue of mongoDB is resolved, it will be easy to migrate all of the data to a distributed datastore, taking immediate advantage of the sharding of data between several servers, and the robustness that comes with that feature is welcome.

The data model is quite simple. The document data model fits almost perfectly with the object oriented programming model, so the properties of each document class are the same as the properties of a given object class. The document classes that have been identified so far are:

- **User** – Contains all of the user information: username, email, password, application list…
- **Application** – Contains all of the application information: size, tarball hash, document root, webservice information, WSDL path…
- **Application server** – Contains the information about a given application server: List of deployed applications, CPU load, IOwait, network load, IP address, hostname…
- **Load balancer** – Contains information similar to the application server class.
- **Datastore server** – Not used yet as the datastore is not elastic at the moment but would store metrics that are similar to the application server.
- **Information repository server** – Contains the same information as the datastore server class, but will be used to scale the information repository.

If stronger authentication mechanisms are required, this service can either be wrapped by a secure web service, or replaced by a RDBMS. Due to this uncertainty added to the fact that many of the details that are to be persisted will only become apparent during the implementation, it is not yet possible to specify an information schema.

4.2.3.2 Load Balancing

Regular load balancers have a pool of homogeneous resources, and direct a given request to a given resource based on some pre established metric.

This platform however does not require that all application servers have access to all of deployed applications, as that would require an enormous amount of storage per server. This means that on any given time, each server will have a different set of applications deployed.

In order to make sure that there is no clash on the namespace of the platform, each application must have a unique root path on the context of the application server. This approach avoids other options such as creating multiple virtual hosts, one per account. Simple application name sanitation must therefore be performed before a new application is deployed on the platform.

The load balancer must be made aware of which servers may respond to a request for a given application. So, a reverse proxy that is able to redirect a request based not only on the usual metrics but also on the path being requested is suited to balance the load among the real application servers.

This raises two more problems. The first is that a single load balancer will not be enough to route requests for an unspecified number of application servers, so an experimental ratio must be set. The number of public IP addresses that those load balancers expose to the public must also be higher than the number of load balancers running. This will allow real time scaling without the latency that comes with the addition of a new DNS record. In such a scenario, scaling a load balancer that has two IP addresses assigned to its interface becomes as simple as removing one of the addresses from the interface and assigning it to the new load balancer. When removing a load balancer, the heartbeat software automatically takes care of assigning the no longer used address to one of the running balancers. A traditional clustering architecture can be used to manage the assignment of IP addresses to the various hosts.

The second problem is that the status of each application server as well as the list of deployed applications must be stored on a location that is readable by the load balancer, and must be kept up to date at all times under penalty of losing some requests.

A first proposal for a load balancer is a modified reverse proxy based on Lighttpd, a small web server that already features most of the features required, and is fairly easy to modify. Its load balancing schedules are varied, and it is very easy to add a new one.

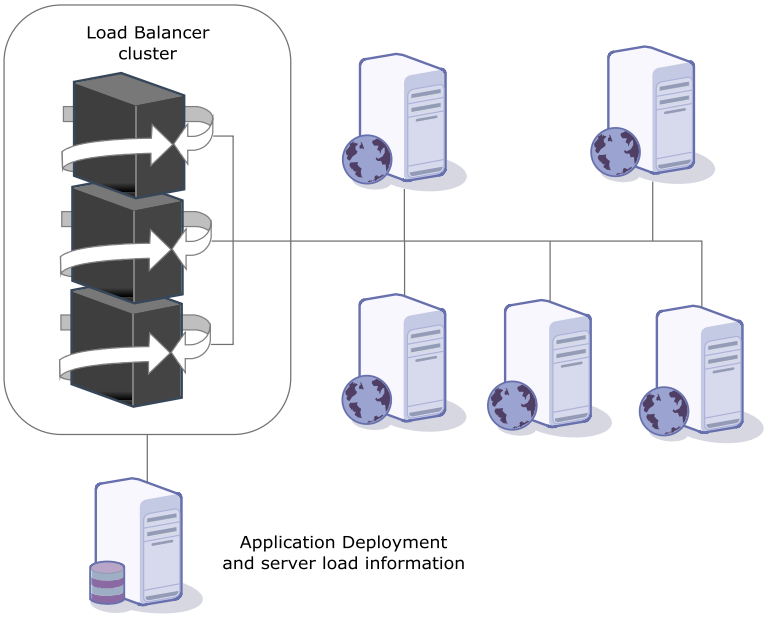

Figure 6 - Load balancer architecture

Figure 6 shows the architecture of the load balancer deployment. This deployment scenario raises one more problem, who has the responsibility of monitoring the several existing nodes of the infrastructure.

4.2.3.3 Monitoring the infrastructure

The information about the load of all the infrastructure can easily be gathered via a protocol such as SNMP. Extra information such as the list of deployed applications on each application server can either be monitored via a custom SNMP MIB, via a remote SSH session, or via a web service, with preference to the custom SNMP MB because it allows a looser coupling between the information gatherer and the information source and it is a very simple protocol.

The gathered information is to be placed on an information repository described in detail on section 4.2.3.1.

4.2.3.4 Deploying an application

All of the application servers described earlier feature the possibility of performing automatic application deployment. That said, the possibility of two application names clashing is a real problem that must be avoided at all times.

So, a unified interface must be exposed that allows a client to upload a standard application package, such as a WAR. Upon receiving such a package, the root path of the application must be replaced by a unique identifier that is to be used later by the load balancer. An example of a simple

identifier is one that contains the username and the application name separated by an underscore, if care is taken to disallow underscores both on the username and on the application name.

Such an identifier could also be used to name a database to be created on the datastore. That database must be created on the datastore when receiving the application for the first time, and must be erased when the application is removed from the platform.

After receiving a new application for deployment, performing the required sanity checks and creating the database for the application, the application can be copied to a common storage area that holds a copy of every single application deployed or potentially deployable on the platform.

The common storage area does not need to be very fast, or shared among all the application servers. In fact it does not even need to be a contiguous storage area; it just needs to be a space where all application packages can be found for deployment on an application server.

4.2.3.5 Managing the load

Previously, when discussing the available datastores, the chosen solution was a RDBMS, which is not a solution that can be scaled in real time (human intervention is required). This means that the amount of servers to be deployed on the data storage layer must be calculated for peak load, which parts with the real time scalability of the cloud computing paradigm, however that datastore should be complemented or replaced by a more flexible implementation as soon as the available ones reach a level considered mature enough to be used on a multi tenant environment.

When the datastore is replaced or complemented, it will be possible to scale it using similar methodologies to those described here, but at the time, the discussion that follows only applies to the application servers and the load balancers.

There are four metrics that must be watched to get a service with a reasonable quality from a server. They are IOwait , CPU load, network usage, and finally available disk space.

The available disk space is only a metric of how many more applications can be deployed on a given server. It is not indicative of the quality of the service performed by that server.

The CPU load increases with the complexity of the business logic of all the applications deployed on a given server. When this value goes over a certain threshold, the performance deterioration will be noticed, especially if there is a human being waiting for the result of a calculation.

Given that the servers have a very limited amount of memory it is unlikely that filesystem cache is useful. This means that almost all the requests require the server to read the necessary files from disk, which takes time. That time is called IOwait, and meters how long the CPU is idle waiting for data from the disk. This value is likely to increase especially when the server receives a large number of requests with short replies, but that require disk access.

The final metric is how much of the available bandwidth is being used by a given server. If that value goes over a given threshold, the perceived performance will be seriously degraded, even if the server has enough of the other resources to deal with the load.

All of these metrics must be monitored at small intervals, and the most up to date values stored at the information repository. If SNMP is used for monitorization, traps can be set up to start the deployment of an extra application server.

When deploying a new application server, the choice of what applications should be deployed on that server is not an easy one. An easy way to cover all the bases is by setting a threshold number of applications. If the number of applications installed on a given server is above a certain threshold, then those applications should be split between the two servers. However, if the server has a low number of applications, all of the applications should be deployed on the new server as well.

A cleaner approach is possible if an appropriate SNMP MIB is available for each of the components, such as the J2EE-MIB [49] for the monitoring of the J2EE application servers. Such a MIB allows fine grained monitoring of the resources used for each application, therefore allowing better scaling heuristics, such as deploying a copy of the most accessed application on the least overloaded server, as well as the application of a business model, by setting thresholds on the available resources for each application. Fine grained information is also at the core of the implementation of a Service Level Agreement (SLA) if the platform is viable.

Again, if all metrics fall below a given threshold, it is time to consolidate. At that time, the two servers with the freest resources should be consolidated into one, which means deploying the applications that are only on one of the servers on the other, and shutting down the first. This consolidation behavior should be independent of the monitoring method used.

Similarly for load balancers, SNMP traps can be used to trigger the scaling. Scaling in is simple if common high availability methodologies are employed on the load balancers, as one of the available ones should take over the tasks of the disappearing ones quickly and automatically. The problem is the deployment of a new load balancer, as the common high availability tools usually require human intervention when adding a host to the cluster, but it is possible to configure them as to allow nodes to automatically join the cluster.

There are two major entities at play here. The first is a monitoring agent. That agent is responsible for collecting the required information from all of the reachable application servers and storing it on the information repository. That information must contain the metrics described above, and the list of applications deployed on a given server. This can be used by the platform administrators to evaluate the platform's behavior as a hole, and to tweak the scaling metrics in an effort to optimize performance. The number of monitoring agents deployed should also be a function of the size of the monitored infrastructure.

The second agent is a load manager. It is responsible for listening for SNMP traps and responding appropriately by deploying or consolidating servers.

On an early stage, the agents manager can be a single program instance, however they should be planned to work on a high availability scenario, using an external datastore to keep track of their actions and state. This architecture will later enable a higher degree of robustness to the cloud as the failure or shutdown of an agent instance must not interfere with the operation of the cloud as a whole. As such, the usage of a Distributed Hash Table (DHT) is suggested as a way to keep the available infrastructure evenly distributed among the available agents.

4.2.3.6 Extra value – Web services

Brief introduction to SOA

As the complexity of the applications increased with the decrease in the price of computing, patterns that allowed code reuse started emerging. CORBA and RMI are standards that allowed building modular distributed applications whose modules communicated with each other using binary interfaces. Both of these technologies have two major weaknesses, they use binary protocols that are not friendly to network administrators, and they do not offer an internet wide means of discovery. This makes them less than optimal to build the interface for a public service [50].

Many companies brought IT to the core of its business process, therefore many of the B2B problems that required standard communications between them could be replaced by automated computer to computer communications, without any human intervention [51].

However, traditional component based design used binary protocols at its core, and was not suitable for B2B applications as binary protocols are frowned upon by network administrators as all of the security must be implemented on the application, and because there was no standard way of discovering the interface of a service without human intervention.

Web Services were the answer to both of the above problems. They use HTTP and XML at the communications level, which allows network administrators to inspect and filter traffic, and WSDL to specify its interfaces so that no human intervention is required to find the interface of a service.

Wrapping legacy software with a web service also provided a quick way to expose the functionality of a legacy module or application as a business service, a quick way to use legacy services on modern services without the need to rewrite all of the business logic to implement a different interface.

The downside of web services is also one of the technologies that add more value to it. XML is a text based protocol, which means that it requires a lot more traffic to send the same information that would be sent over a binary protocol. Therefore using web services as a service interface will cause It to be somewhat slower than using one of the binary counterparts [52].

Service Oriented architecture (SOA) is an architecture pattern that uses the above technologies (among others) to allow loose coupling between application components. By using standard communication protocols between modules, many of the problems of component based design go away. The need to use a single programming language to avoid integration issues becomes mute [53-54], and the need to run all of the components on the same machine is no longer an issue [55].

But SOA is not only about building loosely coupled component based applications. It takes a step further, and sees each component as a service that can be discovered and used by other applications.

The inherently distributed application model recommended by SOA introduced many new possibilities, as the orchestration of several services to form a new service, together with the possibility of replacing any of the individual services that build an application with another that implements the same interface. Those problems are solved by the architecture itself, by specifying layers for service discovery and for service orchestration [7, 56].

The discovery of services is performed via the query of an UDDI server, this server contains the necessary business and technical information required to bind and use the listed services. However, most of the public UDDI servers contain outdated information, and the protocol has some scalability issues. These problems are being addressed by alternative UDDI implementations [57-59].

The orchestration of web services is performed by BPEL, a XML based language whose purpose is to integrate several, sometimes disparate, functions into an integrated service that performs as a whole [60].

How to bring SOA to the cloud computing age

SOA and Cloud computing aim at solving different problems. SOA aims at building better quality software by identifying real and artificial dependencies, and getting rid of as many of the artificial dependencies as possible. Cloud computing aims at solving scalability problems, whether on the infrastructure or on the application level.

So, when building an SOA application for the cloud, what is the service model that fits best? The IaaS model is not abstract enough, as it still requires that some active party, be it the application manager or the application itself, to monitor the load and scale appropriately.

The SaaS model is closer to the end result expected form an SOA application or service running on the cloud.

The PaaS model seems to fit perfectly with the requirements to build such an application. It features a well known programming interface (and usually a data store), and it is easy to implement a scalable application using this service model, as the cloud scales the application automatically according to the load on the service.

The problem with the PaaS model is vendor lock in. None of the current commercial implementations is interoperable with another. This means that in order to switch between vendors, the application provider must rewrite a (possibly significant) part of the application.

So, the ideal cloud deployment model for a SOA application would be a PaaS cloud, preferably integrated with an UDDI system, that is completely open and vendor agnostic. This would take the advantages of no vendor lock in that come with an IaaS, as well as the ease of development that comes with a PaaS.

Integrating UDDI with the PaaS

Currently there are a myriad of web services available throughout the web, and several attempts have been made to bring some order to the chaos. Current UDDI public registries contain mostly unreliable information, meaning that the technical information that describes a service is incorrect and therefore unusable for the composition of new services [61-62].

There are two solutions to this problem. The first is a matter of changing the development practices of the application developers, adding the extra step of updating every UDDI with the updated application information and removing such information when the service is no longer online. This does not seem to be working as the information on most of the registries is still outdated!

Another way to solve the problem of stale information is by using an UDDI registry that periodically monitors the applications that are registered on its database. This approach works better, but it has the disadvantage of requiring extra work by the registry. The distribution of the service seems to be a good way of lessening the burden on the central server, as each local registry may check the accuracy of a subset of the registered services [57-58].

What is proposed is an hybrid between a traditional passive UDDI, and an active distributed UDDI. Given that the application upload and deployment takes place on a controlled environment, the required information about the service supplied by the application (white and yellow pages) can be updated on application upload. On the other hand, the technical information about the application can easily be retrieved in runtime.

To retrieve the technical information (green pages), a local agent running on each application server can monitor new applications upon deployment, and if a (correct) web service is found, it can then update the information repository with the correct WSDL information. A mandatory file placed at the root of the application can point to the WSDL location to facilitate the discovery process.

The frontend UDDI server can then serve external requests. Given that the UDDI is running in a controlled environment, no application registration is required as that action is performed on deployment. Also, the available information will always be correct as the local agents are in charge of keeping the technical information up to date, while the white pages are associated with the user profile, and the yellow pages with the application profile.

Traditional UDDI directories organize its information in categories and sub categories, on a tree structure. This may in fact be detrimental, as there is a possibility that the required service for a given application may not be found due to a misinterpretation during the navigation of the tree [63]. Therefore, it is proposed that the UDDI interface should organize its contents based on a semantic ontology instead of a tree. This approach will in fact require extra work, as making sure that the semantics of the information associated with each service is accurate takes administration time, but the advantages are enormous.

The advantage of organizing the information on a semantic ontology is that the required services can be easier to locate. The more information is contained in the query, the more likely it is that the returned results will match the exact requirements. This is better than keyword search as it is difficult to add all possible keywords to a given service. Another advantage of the semantic ontology is that the same service may appear on several points of the ontology, without the need to re-register the service [63].

The organization of the information on a semantic ontology lead to another question. What about the services that cannot be performed by a computer? Lets say that one company is looking for a human resources management contractor. Usually, that company would look at the yellow pages (the ones on paper!) and start placing calls for each company. However the UDDI service already has the ability of performing the same function as the yellow pages and more. This lead to the conclusion that the semantic ontology colud easily be able to separate services that are intended to be consumed by a device from services that are meant to be consumed by a human, meaning that there is no problem in listing both kinds of services on the same directory. The only modification required is supplying a web interface where an identified user may publish his services to the world. Such a service should only allow the input of information about the company and the service, with no possibility of adding technical information meant for computer consumption.

The problem with services that are meant to be consumed by humans is that there is no way to make sure that the supplied information is accurate and up to date. As to the accuracy of the application, very little can be done. It is up to the one that publishes the information to make sure that it is in fact accurate.

The problem of maintaining the information up to date on the other hand is quite easy to solve. Human services, unlike computer services tend to be quite stable over time. A company does not change its points of contact often, and the life time of a supplied service is commonly counted in years. This means that a simple expiricy policy may solve the staleness of the data. As an example, if a user registered a service, that registration should be valid for six months. However, after three months the system should automatically start sending monthly mails reminding the user that his service would be removed from the listing after that period. Such warnings would grant plenty of time for a service provider to log in and confirm or deny that he still provides the service, therefore reducing the duration of the stale data to a maximum of six months.

4.2.4 The platform infrastructure manager

The previous sections identified and defined the various entities required to build a scalable PaaS using an existing IaaS cloud, as well as the flow for some of the actions that are commonly performed on a PaaS system. The major differences between existing PaaS services and the one specified are the inclusion of the UDDI service as part of the core design, and the portability of the PaaS itself as it can be deployed on any infrastructure be it cloud based or not.

Most of the described entities are based on common freely available software and standards, and therefore are achievable simply by introducing slight modifications to the existing software in order to fulfill the new requirements. Others are new software, but are in fact quite simple to develop such as the user interface for deploying a new application.

Operating system portability is achieved because all of the components are built to be cross platform, or run on a JVM that is itself cross platform.

Yet, at the heart of this system lies the development of a completely new application, a Platform Infrastructure Manager (PIM).

The role of this manager is to scale the platform, removing the need for human intervention. This software must be designed on a distributed way, with the possibility of replicating each module on a different machine, therefore taking advantage of the cloud's scalability properties.

This new application will therefore take advantage of the patterns of SOA when possible, not by using web services, as those are better suited for B2B applications but by using loose coupling strategies between each of the available modules. This makes it easy to extend and add features to the manager in the future. Some ideas for such features include an external (non managed) hop-In/hop-out nodes as an integrant part of the platform infrastructure.

4.2.4.1 The core of the Manager

At the core of the manager lies the information repository described on 4.2.3.1. This information repository's distributed nature allows for the distribution of the state of the PIM. Meaning that the application itself must be developed with care to store all of the relevant state information on the repository as a way of enabling horizontal scaling and elasticity.

The core application of the PIM is to be a small application that may run on any of the managed nodes in parallel with every other application. It must be kept as simple as possible, to ease maintenance, and must be fully fault tolerant.

Each of the core application instances must register itself on the central datastore and update its own status on a regular base. The status information contains a timestamp, and a list of managed modules, as well as the status of each of the managed modules.

Each instance must also run a cleanup and reassignment of active applications at random intervals with a configurable probability. Such probability must be automatically decreased as the number of deployed copies increases, and the information about when the last cleanup process occurred should be stored to avoid running such a process at very small intervals.

Upon the failure of an application core, all of its modules must be distributed among all of the remaining cores. This task is to be performed by whatever core performs the cleanup process. Extra care should be taken to make sure that no information is lost if an application fails when performing a cleanup, as an example, to migrate one module from one manager to the other, the new information should be written before the old one is removed.

Other than that, each application core should be responsible for managing and monitoring a series of autonomous modules. Each module type must be properly designed to store all of the required information on the datastore as well, in order to allow for automatic horizontal scalability.

Each of the application modules should have a type and a subtype, as well as a target and host operating system, used to indentify its function. As an example, a module that monitors the CPU load of a server should have the *monitor* type and the *CPU* subtype. If the module depends on the operating system of the target server, it should specify it on the target operating system property, and if the module runs only on a given operating system, it should be noted on the appropriate section as well.

Another property that each module should have is a class. At the moment, the two classes that are envisioned are the *independent* and the *agent*. Where the independent module may run on any server and the agent is to be deployed together with a given application or server type instance. This means that modules of the class *agent* should not be scaled according to any of the metrics previously described; instead they should be scaled with the application they are to be deployed with.

Several module types and subtypes should be available for deployment. Some of those will be described next, but others will most likely be required later if the system is to be fully implemented.

4.2.4.2 Monitor load

The load monitor belongs to the class *independent*. Its purpose is to monitor the load of a set of servers, and scaling those servers appropriately. Each module is to be responsible for a single type of server, be it an application server, a datastore server, or a PIM host. When deploying the load monitor, the deployer (another load monitor or a cloud operator) must configure that load monitor with enough information to allow the new process to access the information repository, and start operation. Such information comprises a cloud template for the deployment of a new server of the class that is to be monitored, how much time to wait before deploying a new server to avoid performing many scaling operations on a row because of the same server.

At the time it was not defined whether this module actively monitors the other servers or if it waits for the servers to send it trigger information. Yet the most likely solution includes a mix of both, with periodic monitoring used to gather statistics and make sure that the monitored services are running, while the critical conditions should be triggered by the monitored host itself, informing the monitor that the load passed a threshold that is supposed to cause the monitor to act upon it.

The required information should be added as part of the contextualization of the virtual machine to be deployed if running under an infrastructure manager, or sent to the server (possibly via SNMP).

When scaling out, the module should first boot a new instance of a given server, then call a scaling function that is server specific and actually performs any required work to distribute the load over the two servers.

When scaling in, the module should call a scaling function before destroying the server. That function should remove the server from any cooperative tasks it may be performing before allowing the scale in process to continue. When possible, the scale in process should take place on the two servers with the lowest load possible, consolidating the tasks performed by both of them on a single server.

Other server types such as the load balancers do not require any complicated actions to be performed on them; this means that an empty function should be called for those modules.

If unmanaged nodes are to become a possibility, the need to uniquely identify a given node becomes paramount as the IP address or even domain name of the server may change in runtime. This requires the usage of a robust name resolution system, far more sophisticated than DNS. The XRI/XDI architecture has the required abilities to handle such naming scheme modifications.

This naming resolution system supports, among other things, the reuse of the same identifier without the fear of mixing the identity of two hosts, therefore allowing per server accounting even if a given server changes its IP address and domain name to one that was previously assigned to another host. This technology already proved its applicability for this scenario as it is at the core of the OpenID v2.0 specification precisely to allow a safe and robust resolutions of the user's identity [64].

Deploy application on new server

After deploying a new application server, a set of applications must be deployed on the new server if the load is to be distributed between the two. This is the task of this module.

If the number of applications deployed on a given server is over a configurable threshold, the applications should be split among the two servers. However, if the application count is bellow that threshold, the module replicates the full application list on the new server.

Consolidate application servers

Before scaling in two application servers, one must be chosen to keep on running. The applications that are running on the server that is to be shut down must be deployed on the other server if not deployed already.

The load managers must remove the server that is to shut down from their rotation, and some time must be allowed to avoid shutting down the server while in the middle of answering a request. Only then is it possible to shut down one of the application servers.

Deploy a new Datastore server

Deploying a new datastore process is not supported on the initial stages of the development as the application datastore is based on a RDBMS that requires human intervention and the information repository is based on a monolithic datastore. So at the moment this module should just send an email to an operator informing that the datastore is overloaded.

Later, when a new (and improved) datastore is deployed, information about the existing datastore deployment should be added to the new instance, and possibly, the rebalancing process should be triggered.

Consolidate the datastore server

On the initial stages this is not recommended as the chosen datastore does not allow on line consolidation, yet when the new datastore is in place, it will be the task of this function to remove the node from the datastore, therefore distributing its information over the remaining datastore.

4.2.4.3 Deploy new application

This module exposes an internal cloud service. It performs the actual sanitation of a new application submitted by a user, setting the correct application root, and performing any other checks that are required.

If the application is marked as a web service by the user, it is also this module's responsibility to check the sanity of the user provided application information.

This module then deploys the application on a configurable number of application servers, and if required adds the required information to the UDDI servers. However, the presence and validity of the WSDL file must be checked before adding any information to the UDDI server.

4.2.4.4 Delete application

This module takes an application name and username and deletes that application from the whole platform. This encompasses deleting all of the application specific information from the repository and the UDDI if the application is a web service, undeploying every application instance, and finally deleting the application WAR from the central application repository.

4.2.4.5 Other modules

The modules and functions described above are expected to be operating system agnostic, however on the real world, the need for operating system specific modules may arise, especially if a non POSIX compliant system is ever used as an application server host.

Other modules may be required, as the platform development begins and the requirements and system limitations become evident. However, the modules described above should provide an initial base functionality that should be enough to evaluate the viability of the platform, as well as to evaluate any scalability problems that may arise due to architecture mistakes.

4.2.5 Portability isssues

All of the current PaaS implementations come with the downside of vendor lock in. This comes from the fact that none of the services actually license the code that comprises the platform, which would allow a client to deploy that very same platform on another vendor's infrastructure.

The proposed PaaS parts radically with this trend. By using only open source components, and licensing the code that is to be developed under an appropriate license, such as the GNU Affero license, the freedom to deploy individual components of the whole PaaS on any infrastructure is granted to all that make use of it.

The proposed PaaS was designed with the possibility of running on a large number of small virtual machines in mind, precisely to make sure that it will run on any available infrastructure without requiring top end servers.

If needed, a partial deployment is possible on physical hardware, just by removing the agents that perform the scaling of the infrastructure. On the other hand, deployment on different IaaS suppliers is possible by the implementation of different client API's on the scaling agents, therefore turning this into a very flexible PaaS.

As to host operating system, all of the required software is multi platform. This is accomplished either by recompilation for the required platform, or by running on a java/python/ruby virtual machine. In either case, supporting different host operating systems should be as easy as recompiling a few modules, and creating a new distribution package for the given operating system. Neverthe-

less, during the specification phase, Linux was always seen as the most viable host for such a system, as (taking EC2 as an example) running a linux instance on a IaaS is cheaper than running windows on the same hardware, and because it is a very well documented and widely used system for server purposes.

5 Comparison

The comparison of the proposed platform with existing products is not a simple task. The first problem is that the maturity of the existing projects is quite varied, and so are the target audiences for each one.

A famous title in software development states that "*there is no silver bullet" [65]*. Brooks became famous for this not only due to the content of the article itself, but also due to the fact that this principle is applicable not only to software engineering methods, but to many other aspects of the software lifecycle, from the choice of a programming language, to the final product.

The platform proposed by this dissertation work is an example of the concept. Many decisions were taken when specifying the platform, and many more need to be taken when the development cycle of each component begins. Those decisions are unlikely to please all possible audiences, yet care was taken to try to cater to the needs of the enterprise J2EE application as much as possible.

If this approach is successful, the addition of the .Net programming language should cater to the needs of another large community, and therefore the needs of most of the enterprise market should be covered.

There are some key points that define the scalability of the cloud architecture. The first of those is the ability to move data in and out of the cloud, the available throughput that must be shared by all the cloud applications and users. The second is how the application servers itself work and share information, this may eventually lead to a point where adding more servers may decrease performance instead of increasing it. The third factor is the structured storage solution used for application data persistence, this is most likely the field where the greatest number of implementations are available. There are many solutions to particular problems that plague data storage, yet no single public domain implementation solves all the issues up to the moment.

5.1 Gateways

The gateway system to the AppEngine PaaS is not clear and the other implementations currently use a clustered application server to respond to all the requests.

The proposed platform parts with this concept, as the fact that the cloud bandwidth is saturated does not imply that a new application server should be deployed, nor does the fact that the network has little load mean that the application servers are idling.

On the most common scenario, the proposed platform should be deployed on an infrastructure that features better internal links than internet links for each host.

By separating the gateway from the application server, this platform allows the usage of different templates in the case of deployment on a IaaS cloud, which could reduce the total running costs.

The gateway load balancers should run as a cluster not to share state information, but to scale IP addresses appropriately as adding and removing IP addresses from the DNS system may take a long time due to the architecture of the DNS system itself.

This is an improvement over current implementations where any application server is in charge of responding to its own requests and routing those that must be replied elsewhere, as it separates the task of replying to a request from the task of routing the requests, therefore allowing the scaling of each layer in separately.

5.2 Application server

Scaling the application servers is a crucial feature of a PaaS. The available and known implementations scale the application servers using the well established cluster technology to allow session state to be maintained, with the exception of Google's AppEngine that forces each request from the user to be completely stateless, or the developer to explicitly store session information on the BigTable data store.

The proposed implementation parts with the cluster architecture, as it requires constant state and routing information sharing, and taking a page form the shared nothing architecture, it states that no information should be shared by the application servers, meaning that each server is autonomous from the others while serving a request. This removes the need for a high performance shared storage where the full pool of deployed applications are stored after deployment, but it adds the need for an intelligent load balancer to route a request to one of the available applications servers that have the required application deployed. This may add a bit of latency as it requires a round trip to a persistent data store, yet the communication protocol used to perform this task should be as fast as possible in order to minimize perceived latency.

The proposed PaaS does allow the maintenance of session information, yet it does that in a completely different way. It hides the complexity of storing and fetching session information on a persistent data store from the user. This does lead to an increase on the latency for any request that uses this feature, but that increase is justified due to the fact that it should not degrade with load.

5.3 Structured data storage

Application data can be stored using several models, and within each data model several implementations are available. This is an area where a lot of effort is being placed by the developers in order to add the required functionality to supply a really scalable service, with the possibility of storing and querying complex data.

Current PaaS implementations developed different solutions for this problem. JBoss's Paas uses a key value data store, therefore trading data complexity for scalability. Unfortunately, complex data storage is required on most of the real world applications that may take advantage of such a PaaS.

Aneka's solution is a full RDBMS for application data storage. This solution allows a large data complexity and allows the application to take advantage of the ACID properties. The downside of this choice is that the current scalability and elasticity of the available RDBMS systems is quite limited, and therefore this may become a bottleneck for a busy application.

Google's BigTable is currently the best available solution of the available PaaS's for application data. The data model of this solution takes advantage of the fact that most developers were already using some sort of object-to-relational solution such as hibernate or LINQ, and exposes a programmatic interface very similar to such products.

Although many criticize Google due to the strict limits placed upon the queries performed, there is no doubt that if an application is programmed taking into account the artificial limitations placed upon the queries, this data store can scale to extremely large volumes of queries without showing any impact due to the increased load.

The proposed PaaS specification tried to part with the RDBMS system with no success, as currently none of the implementations of complex data stores allow the storage of complex data on a multi tenant environment. Yet, it is expected that the MongoDB will reach that point in a little time, therefore the RDBMS is just a temporary solution.

5.4 UDDI

Up to this moment none of the available PaaS implementations really integrated an UDDI service into its core. This may be a major flaw, as many of those implementations are trying to enter the business space where SOA is actually used.

As such, this proposal encompasses the integration of an UDDI service in the core of the cloud itself. This service can be used for internal service publication in the case of services that are related to the management of the cloud itself. But the main reason to add a directory to the core of the implementation was the proposal of a new concept of UDDI management, where the published information is always up to date without the need of constant monitoring as is the case of active UDDI.

This is probably the most innovative concept introduced by this work, as it effectively allows the integration of applications deployed on the PaaS into existing or new service orchestrations designed to build a new service, or even an enterprise business layer.

Another novelty introduced by this work is the way in which the services are searched within the UDDI server itself, that parts with the traditional tree like organization and takes a semantic approach, therefore increasing the chances of finding an adequate service.

6 Conclusions & Future Work

6.1 IaaS

The first objective of this dissertation was to assert the state of the current IaaS implementations. The state of the commercial implementations is widely documented on the supplier's web sites, but that is not always the case with the open source implementations, where the documentation often lags behind the implementation features.

The chosen implementation was OpenNebula. It is still quite a simple infrastructure manager, yet it is developing quite fast as the users realize that it features a clean and easily extensible architecture. In fact, in order to compare the VMware driver with the Xen driver on even grounds, some modifications were necessary.

The resulting deployment of OpenNebula is nearly useable as a production system, with very few modifications required for such operation. Those modifications consist mainly in the addition of new virtualization hosts and moving the shared storage to a dedicated appliance, or using an image management system that copies the source images to all virtualization nodes and deploys a copy of the local image on the node.

An interesting addition to the existing deployment is a project from the ecosystem that consists on a web console that is usable to manage the cloud. However, tests performed on the very early stages of the implementation of the console did not show any advantage for the current work, so no effort was placed on helping the development team.

When compared with existing IaaS deployments, the one performed at IEETA can be at most classified as a proof of concept due to its small size, and already a plethora of new ideas are bubbling around it. With the addition of new resources, the possibility of sharing the deployment among several investigation groups, or using it for educational purposes becomes real and appealing.

An interesting test performed by CERN also shows that OpenNebula is able of deploying about one VM per second when using LVM deployment and pre transferred images, which is quite an impressive number for such a simple manager, yet not enough for CERN's requirements.

The core of OpenNebula also has some flaws that reduce its usability for a large public deployment. At the moment, high availability is not yet part of the immediate design goals of the OpenNebula team. This means that in the case of a failure of the node that runs the scheduler, deployment of new images will be impossible. Another flaw lies on user and permissions management that is too simplistic. The next release should include the ability to specify access lists for each user or VM image, meaning that it will be a major step towards the deployment of a public cloud.

The IaaS cloud that resulted from this dissertation work is suitable for wide deployment, with very small modifications that lie especially on the mass storage used to store system images. It can be

used either for teaching or research purposes, where a large number of small machines are a common requirement.

It is also proof that the IaaS open source implementations are now at a stage that allows them to be used for initial production applications, but also that there is a lot of work that still needs to be performed by the development community.

6.2 PaaS

The lack of an open PaaS cloud is a major drawback to the adoption of this service model as a hosting platform for the enterprise business logic due to the fear of vendor lock in, among other factors.

The PaaS specification envisioned by this document explores existing technologies to build a modern platform that features well known technologies that are widely used on the enterprise market segment. It also brings the UDDI server into the cloud, taking advantage of existing application information to avoid the publication of outdated or incorrect information on that server.

This PaaS steps forward the existing ones as it solves some of the most important problems that pose as a barrier to PaaS adoption, the problem of vendor lock in and the problem of portability. Those problems seem to be the two most important barriers to wide enterprise PaaS adoption.

6.3 Future work

The OpenNebula community is now tackling what are perceived as its current weaknesses. Work is focusing on the public API that is incomplete and does not expose the full functionality of the software. The core is now a single point of failure, architecture modifications that will allow a high availability core are under consideration. The user management is currently too simplistic, ACL support is being added to the core of the system, as well as alternative authentication mechanisms.

The work on OpenNebula is a community effort that is doing quite well. From personal experience, the support mailing lists are quite friendly, bugs are not a common place and are solved quite quickly. Configuration issues seem to be the most common problem, but those have an easy solution most of the times.

The refinement of the specification and the implementation of the PaaS proposed here would be a very interesting project, however due to its length and amount of technologies to be approached, that task is better suited for a medium sized project than for a single person’s work.

References

[1] T. OReilly, “What is Web 2.0: Design patterns and business models for the next generation of software,” 2007.

[2] M. Armbrust, A. Fox, R. Griffith *et al.*, “Above the clouds: A Berkeley view of cloud computing,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[3] P. Mell, and T. Grance. "Draft NIST Working Definition of Cloud Computing v15," September 10, 2009; http://csrc.nist.gov/groups/SNS/cloud-computing/.

[4] L. Ellison. "Oracle's Ellison nails cloud computing," September 9, 2009; http://news.cnet.com/8301-13953_3-10052188-80.html.

[5] IBM. "IBM - Cloud Computing," September 9, 2009; http://www.ibm.com/cloud/.

[6] T. Biske. "Cloud versus Grid," April 2010; http://www.biske.com/blog/?p=460.

[7] M. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions."

[8] A. Rajsekar, M. Wan, R. Moore *et al.*, “Data grid federation,” *PDPTA, Las Vegas NV*, 2004.

[9] R. Ranjan, A. Harwood, R. Buyya *et al.*, “Grid Federation: An Economy Based, Scalable Distributed Resource Management System for Large-Scale Resource Coupling,” *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia*, 2004.

[10] M. Bote-Lorenzo, Y. Dimitriadis, and E. Gómez-Sánchez, "Grid characteristics and uses: a grid definition." pp. 291-298.

[11] I. Foster, Y. Zhao, I. Raicu *et al.*, “Cloud Computing and Grid Computing 360-Degree Compared,” *Gce: 2008 Grid Computing Environments Workshop*, pp. 60-69, 2008.

[12] A. Aarsten, D. Brugali, and G. Menga, "Patterns for three-tier client/server applications."

[13] J. Mayer, I. Melzer, and F. Schweiggert, “Lightweight plug-in-based application development,” *Lecture Notes in Computer Science*, pp. 87-102, 2003.

[14] "[one-users] Does OpenNebula support Windows guest VM?," September 18, 2009; http://lists.opennebula.org/pipermail/users-opennebula.org/2009-May/000428.html.

[15] "Running Windows on Eucalyptus « Ajmf's Weblog," September 18, 2009; http://ajmf.wordpress.com/2009/08/05/running-windows-on-eucalyptus/.

[16] "Main Page - KVM," October 1, 2009; http://www.linux-kvm.org/page/Main_Page.

[17] "[workspace-user] KVM Status in Globus Workspace," September 18, 2009; https://lists.globus.org/mailman/htdig/workspace-user/2009-June/000850.html.

[18] "google-summer-of-code-2009-globus," September 18, 2009; http://code.google.com/p/google-summer-of-code-2009-globus/downloads/list.

[19] "Manjrasoft," May 2010; http://www.manjrasoft.com/.

[20] C. Vecchiola, X. Chu, and R. Buyya, “Aneka: a software platform for .NET-based Cloud computing,” *High Performance & Large Scale Computing, Advances in Parallel Computing. IOS Press, Amsterdam*, 2009.

[21] X. Chu, K. Nadiminti, C. Jin *et al.*, "Aneka: Next-generation enterprise grid platform for e-science and e-business applications."

[22] M. Neale. "Cooling Tower - JBoss community," May 2010; http://community.jboss.org/wiki/CoolingTower.

[23] "INFINISPAN - Open Source Data Grids - JBoss community," May 2010; http://www.jboss.org/infinispan.

[24] F. Chang, J. Dean, S. Ghemawat *et al.*, "Bigtable: A distributed storage system for structured data."

[25] Google. "GQL Reference - Google App Engine - Google Code," http://code.google.com/appengine/docs/python/datastore/gqlreference.html.

[26] B. Sotomayor, R. Montero, I. Llorente *et al.*, “Capacity leasing in cloud systems using the opennebula engine,” *Cloud Computing and Applications,* vol. 2008, 2008.

[27] J. Fitzhardinge. "Re: [Xen-devel] What is the current state of Dom0 kernel support?," March, 2010; http://lists.xensource.com/archives/html/xen-devel/2009-06/msg01193.html.

[28] "Overview - Fedora project," March, 2010; http://fedoraproject.org/wiki/Overview.

[29] M. DiPetrillo. "UPDATE: VMware RCLI now writes to ESXi Free Hosts," March, 2010; http://www.mikedipetrillo.com/mikedvirtualization/2008/12/update-vmware-rcli-now-writes-to-esxi-free-hosts.html.

[30] "OpenNebula: The Open Source Toolkit for Cloud Computing - Documentation," March 2010; http://www.opennebula.org/documentation:documentation.

[31] O. O. C. C. I. W. Group. "OGF Open Cloud Computing Interface Working Group :: start," October 21, 2009; http://www.occi-wg.org/doku.php.

[32] "Open Cloud Manifesto," March 2010; http://www.opencloudmanifesto.org/.

[33] A. B. Bondi, “Characteristics of scalability and their impact on performance,” in Proceedings of the 2nd international workshop on Software and performance, Ottawa, Ontario, Canada, 2000, pp. 195-203.

[34] M. Michael, J. E. Moreira, D. Shiloach *et al.*, "Scale-up x Scale-out: A Case Study using Nutch/Lucene." pp. 1-8.
[35] D. Oppenheimer, and D. Patterson, “Architecture and dependability of large-scale Internet services,” *IEEE Internet Computing*, pp. 41-49, 2002.
[36] V. Cardellini, M. Colajanni, and P. Yu, “Dynamic load balancing on web-server systems,” *IEEE Internet Computing,* vol. 3, no. 3, pp. 28-39, 1999.
[37] A. Delis, and N. Roussopoulos, "Performance and scalability of client-server database architectures." pp. 610-610.
[38] M. Stonebraker, “The case for shared nothing,” *Database Engineering Bulletin,* vol. 9, no. 1, pp. 4-9, 1986.
[39] J. West, and J. Woodard, “Strategic Responses to Standardization: Embrace, Extend or Extinguish?,” 2009.
[40] "Configuring_distributed_systems," April 2010; http://wiki.apache.org/couchdb/Configuring_distributed_systems.
[41] D. Merriman. "Security and Authentication," April 2010; http://www.mongodb.org/display/DOCS/Security+and+Authentication.
[42] B. Scofield, ""Comics" is hard: On domains and databases," 2009.
[43] Q. Yao, and A. An, "Using user access patterns for semantic query caching." pp. 737-746.
[44] "Mono project," May, 2010; http://www.mono-project.com/Main_Page.
[45] "Novell and Microsoft Collaborate," May 2010; http://www.novell.com/linux/microsoft/faq.html.
[46] B. Smith. "Microsoft's Empty Promise - Why Worry About C#?," May 2010; http://www.fsf.org/news/2009-07-mscp-mono.
[47] Microsoft. "ECMA C# and Common Language Infrastructure Standards," May 2010; http://msdn.microsoft.com/en-us/netframework/aa569283.aspx.
[48] R. Schestowitz. "Novell Wants to Bring Microsoft, Moonlight, and Mono to Linux Phones (Android)," May 2010; http://techrights.org/2010/03/17/android-mono-silverlight-danger/.
[49] Sun, "J2EE-MIB," 2002.
[50] H. Petritsch, “Service-Oriented architecture (SOA) vs. component based architecture,” *White Paper, Vienna University of Technology*, 2008.
[51] T. Mukhopadhyay, and S. Kekre, “Strategic and operational benefits of electronic integration in B2B procurement processes,” *Management Science,* vol. 48, no. 10, pp. 1301-1313, 2002.
[52] N. Gray, "Comparison of Web Services, Java-RMI, and CORBA service implementations."
[53] A. Cleary, S. Kohn, S. Smith *et al.*, “Language interoperability mechanisms for high-performance scientific applications,” 1998.
[54] M. Weiser, A. Demers, and C. Hauser, "The portable common runtime approach to interoperability." p. 122.
[55] F. Leymann, “Web services: Distributed applications without limits,” *Business, Technology and Web, Leipzig*, 2003.
[56] L. Chen, B. Wassermann, W. Emmerich *et al.*, "Web service orchestration with BPEL." p. 1072.
[57] B. Sujata, B. Sujoy, G. Shishir *et al.*, “Scalable Grid Service Discovery based on UDDI,” in Proceedings of the 3rd international workshop on Middleware for grid computing, Grenoble, France, 2005.
[58] Z. Du, J. Huai, and Y. Liu, “Ad-UDDI: An active and distributed service registry,” *Technologies for E-Services*, pp. 58-71.
[59] W. Tsai, R. Paul, Z. Cao *et al.*, “Verification of web services using an enhanced UDDI server,” 2003.
[60] "Web Services Business Process Execution Language Version 2.0," May 2010; http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm.
[61] M. Clark. "UDDI weather report," April 2010; http://www.webservicesarchitect.com/content/articles/clark04.asp.
[62] S. Kim, and M. Rosu, “A survey of public web services,” *E-Commerce and Web Technologies*, pp. 96-105.
[63] R. Akkiraju, R. Goodwin, P. Doshi *et al.*, "A method for semantically enhancing the service discovery capabilities of UDDI." pp. 9-10.
[64] D. Reed, L. Chasen, and W. Tan, "OpenID identity discovery with XRI and XRDS." pp. 19-25.
[65] F. Brooks, “No silver bullet: Essence and accidents of software engineering,” *IEEE computer,* vol. 20, no. 4, pp. 10-19, 1987.
[66] J. Barr. "Amazon EC2 Beta," October 9, 2009; http://aws.typepad.com/aws/2006/08/amazon_ec2_beta.html.
[67] J. Barr. "New EC2 Features: Static IP Addresses, Availability Zones, and User Selectable Kernels," October 9, 2009; http://aws.typepad.com/aws/2008/03/new-ec2-feature.html.
[68] J. Barr. "Amazon EBS (Elastic Block Store) - Bring Us Your Data," October 9, 2009; http://aws.typepad.com/aws/2008/08/amazon-elastic.html.
[69] Amazon. "Amazon Simple Storage Service (Amazon S3)," October 21, 2009; http://aws.amazon.com/s3/.

[70] M. Palankar, A. Iamnitchi, M. Ripeanu *et al.*, "Amazon S3 for science grids: a viable solution?." pp. 55-64.
[71] "Welcome to Google Apps," October 9, 2009; http://www.google.com/apps/.
[72] I. Blau, and A. Caspi, "What type of collaboration helps? Psychological ownership, perceived learning and outcome quality of collaboration using Google Docs." pp. 48-55.
[73] "Google App Engine - Google Code," October 8, 2009; http://code.google.com/intl/pt/appengine/.
[74] "Quotas - Google App Engine - Google Code," October 8, 2009; http://code.google.com/intl/pt/appengine/docs/quotas.html.
[75] Microsoft. "Windows Azure Platform," October 21, 2009; http://www.microsoft.com/windowsazure/.
[76] H. E. Schaffer, S. F. Averitt, M. I. Hoit *et al.*, "NCSU's Virtual Computing Lab: A Cloud Computing Solution," *Computer,* vol. 42, no. 7, pp. 94-97, 2009.
[77] "Academic Partners | Virtual Computing Lab (VCL)," September 11, 2009; http://vcl.ncsu.edu/academic-partners.
[78] "Apache VCL," April 2010; http://incubator.apache.org/vcl/.
[79] "Home | Virtual Computing Lab (VCL)," September 18, 2009; http://vcl.ncsu.edu/.
[80] "How it Works | Virtual Computing Lab (VCL)," September 12, 2009; http://vcl.ncsu.edu/help/general-information/how-it-works.
[81] "VCL XML RPC," September 18, 2009; http://people.apache.org/~jfthomps/xmlrpcdocs/xmlrpcWrappers_8php.html.
[82] "Creating base windows XP image," April 2010; http://vcl.ncsu.edu/help/applications-images/creating-windows-xp-base-images.
[83] "FAQ - Nimbus Open Source IaaS Cloud Computing Software," April 2010; http://www.nimbusproject.org/docs/?doc=2.2/faq.html#nimbus.
[84] "Publications - Nimbus Open Source IaaS Cloud Computing Software," April 2010; http://www.nimbusproject.org/papers/.
[85] "[workspace-user] Does Nimbus support Windows VM?," September 14, 2009; http://lists.globus.org/mailman/htdig/workspace-user/2009-June/000835.html.
[86] "TP2.2 Extensibility Guide - Nimbus Open Source IaaS Cloud Computing Software," October 1, 2009; http://workspace.globus.org/vm/TP2.2/plugins/index.html.
[87] "OpenNebula: The Open Source Toolkit for Cloud Computing - about," April 2010; http://www.opennebula.org/about:about.
[88] "Reservoir: Home," September 14, 2009; http://www.reservoir-fp7.eu/.
[89] "OpenNebula: The Open Source Toolkit for Cloud Computing - Architecture," April 2010; http://opennebula.org/documentation:rel1.4:architecture.
[90] "Haizea - An Open Source VM-based Lease Manager," September 16, 2009; http://haizea.cs.uchicago.edu/whatis.html.
[91] "Eli Lilly, NASA Build Eucalyptus Clouds - Plug Into The Cloud - InformationWeek," September 16, 2009; http://www.informationweek.com/cloud-computing/blog/archives/2009/06/eli_lilly_nasa.html.
[92] "EucalyptusOverview - Eucalyptus," April 2010; http://www.eucalyptus.com/products/overview.
[93] "The Eucalyptus Story | Eucalyptus Systems Inc," October 21, 2009; http://www.eucalyptus.com/about/.
[94] "FAQs | Eucalyptus community (Interface)," April 2010; http://open.eucalyptus.com/wiki/FAQ#interface.
[95] "FAQs | Eucalyptus community (kvm)," April 2010; http://open.eucalyptus.com/wiki/FAQ#kvm.
[96] "Enomaly Developers Wiki," September 17, 2009; http://src.enomaly.com/.
[97] Intel, "Intel® Cloud Builder Deployment Guide for ECP," Intel® Cloud Builder Deployment Guide for ECP.
[98] "OpenQRM," September 17, 2009; http://www.openqrm.com/.
[99] "Features | openQRM," September 12, 2009; http://openqrm.com/?q=node/2.
[100] "Architecture of openQRM | openQRM," September 17, 2009; http://www.openqrm.com/?q=node/42.
[101] "FAQ - ConVirt," September 23, 2009; http://www.convirture.com/wiki/index.php?title=FAQ#Q._What_is_ConVirt.3F.
[102] "Documentation - ConVirt," September 23, 2009; http://www.convirture.com/wiki/index.php?title=Documentation#Features.
[103] "Bugs and caveats - ConVirt," September 23, 2009; http://www.convirture.com/wiki/index.php?title=Bugs_and_caveats.
[104] "Get Involved - ConVirt," September 23, 2009; http://www.convirture.com/wiki/index.php?title=Get_Involved#Docs.

Appendix

Appendix A - Commercial Cloud implementations

A.1 Amazon

Nowadays it is impossible to circumvent this company when speaking about the subject of cloud computing. Amazon is offering two major cloud services, EC2 and S3. The first is a computation (IaaS) cloud, its purpose is to supply IT infrastructure on demand. The S3 service supplies storage both to the public, and to the EC2 cloud. These two clouds complement each other, and allow for a great flexibility on the final service presented to the user.

The interface for this commercial offering is based on web services and there are multiple client implementations, ranging from command line applications to browser plugins. This interface is in fact becoming a *de facto* standard, as most free and open source implementations of this deployment model are now offering a compatible interface.

The EC2 service is based on virtual machine provisioning. Amazon took the concepts of datacenter virtualization a step further into the web 2.0 era by providing a user centric interface to their virtual machine provisioning service [66]. For the first time, users were able to provision full software stacks in a matter of minutes, being charged on a *pay as you go* model instead of requiring that the application owner predict how many servers would be required for any given task during peak load. This model was an evolution over the rental of virtual private servers because it allows a greater granularity on the pricing model, a faster allocation of the requested resources, and due to the ability to create pre configured system images, it even offloaded some of the system administration time used to roll out updates to running systems.

Initially, the virtual machines did not have any persistent storage and interruption of a task cloud lead to data loss. This was later resolved with the addition of new features, among them the Elastic Book Store, which can be used as an attachable volume on the virtual machines instances [67-68].

S3 is a data storage service that is accessible from any internet connected device [69]. This service is commonly used as a data gateway for Amazon EC2, as most of the workloads to be processed require the availability of datasets and the transfer of data between S3 and EC2 is free.

The S3 service interface is kept as simple as possible; each file is called a data object, and each directory a bucket. The primary interface is HTTP based (REST and SOAP), a bittorrent interface is also provided to lower the costs for high scale distribution [70].

This data storage service has a SLA guaranteeing reliability, yet it should not be mistaken for a high availability distributed storage solution, as the bandwidth allowed per object is not very high, and because the security system does not allow one to set granular access permissions on a bucket or object [70].

A.2 Google

Google offers both PaaS and SaaS. Most of the products in the Google portfolio are free to use until a threshold is passed, and others are completely free. This is a major factor when experimenting with new products or services as no upfront investment is needed to test the product.

On the SaaS front, Google offers among others [71], the Google Docs suite, a set of online productivity tools. Although this suite does not try to compete with similar desktop products on the level of complexity and amount of advanced features, many users are choosing it due to the great collaboration features that are only possible on an online product [72]. Google Docs, as well as other services from this company were built using a distributed cloud platform. This platform was later made available to the public, and branded as App Engine.

The PaaS called App Engine [73], only supported python in its early days but has been upgraded to support the java programming language. In order to use this platform, the developer must use a SDK provided by Google and follow some simple sandbox rules that prevent abuse of the resources. The download and usage of the SDK is free, and the development platform can run on the most common desktop operating systems. The developed application is allowed to use a limited amount of system resources to prevent it from hogging resources needed by others. The limitation is based on per minute and per day quota of each of the resources available. If an application exceeds its daily quota usage of a resource, usage of that resource is denied, which may leave the application on a crippled or even unavailable state. In order to prevent the denial of usage due to going over quota, the application owner can set up billing. In this situation, if the resource is used beyond a given quota the application owner gets charged, the value of the maximum daily budget is configurable to avoid exceeding the owner expectations [74]. There are also hard limits on the amount of computation needed to serve any given request, to avoid starvation of other applications.

If all aforementioned constraints are met, the application will scale seamlessly with the increase or decrease of load in such a way that no performance degradation is perceived by the end user. This capability is one of the major features of thee PaaS as a service model, as no special care must be taken to allow the application to scale with the load.

A.3 Microsoft

Microsoft Azure has some resemblances with the service provided by Amazon on the architecture point of view [75]. Yet on the business side, the way the service is exposed to users is completely different. On this service, a user only has a few system images he may choose from, and those images are based on roles, such as an application server or an SQL server. Therefore, it becomes simpler to build standard application architectures at the cost of some flexibility. This service model is somewhere between the IaaS and the PaaS as a user can select instance types, yet the software stack for each instance type is fixed.

This service capitalizes on the .NET knowledge and tools that are already deployed at the users premises. With the addition of a SDK, the application can be deployed on the cloud platform.

The charging model is quite complex, based both on deployment time and on resources consumption. For the application server, that Microsoft calls .”compute instance”, the pricing is based on uptime. For the storage services, charges are based on the amount of Gb stored per month with an extra fee for each 100000 transactions. SQL servers are charged monthly according to the number of databases used. AppFabric, a set of communication buses that allow integration with external applications is charged based on the number of connections and authentication requests. Finally, data transfer is charged per Gb.

Appendix B - Academic or open source IaaS implementations

Most universities are now on their first steps towards understanding the advantages of cloud computing. The open source community is also active; several implementations allow cloud computing, some of those even claim to be of production quality. This means that now is the time to start looking at those implementations with a critical eye, trying to understand the advantages and shortcomings of each one. Most of the available implementations focus on the IaaS model, the model that enables a smooth scalability of PaaS and SaaS by creating a uniform resource pool that can be scaled on demand.

In order to compare features, one must rely heavily on each implementation web page, mailing lists and other communication forms, as the feature list is usually a moving target and scientific publication does not always keep up with the latest improvements.

B.1 Apache VCL

This implementation started some time before of the cloud computing hype. The North Carolina State University (NCSU), when faced with escalating requirements and costs to support its IT infrastructure, realized that some of the problems they were facing were solvable by loading a system image on a blade on demand [76]. With the advent of virtualization, deployment on virtual machines was added, and the service grew organically to its current form which fits on the definition of cloud computing. Virtual Computing Lab (VCL) is now hosted by the Apache foundation, and is used in production by some universities [77].

The VCL is an open source system designed to provision and broker access to a compute environment requested by an end user. The provisioned computer can range from a virtual machine to a physical computer, the computer can be housed on a datacenter, in which case remote access tools must be used, or it can be located on a computing lab under VCL management [78].

The primary goal of this project is to deliver a dedicated computing environment to a user for a limited amount of time. The supplied computing environment can range from a simple virtual machine running productivity software to a blade server running high end software, or even a cluster of compute nodes. At the NCSU there are several environments available to faculty and students, that include linux, solaris and windows operating systems [78-79].

B.1.1 VCL Architecture

VCL has a three tier architecture, consisting on a web server for the frontend, a database server, and one or more management nodes. Each management node controls one or more compute nodes that actually run the software images requested by the end users [80].

B.1.2 Web server

The web server is running Linux and uses Apache to serve PHP files to clients. It serves as a frontend to the database and provides tools to request, manage, and govern all VCL resources [80].

Browsing the source also shows that a XML RPC interface exists, but how much of the full VCL functionality is covered by this interface is not clear [81].

B.1.3 Database server

The database consists on a MySQL server running on Linux. It holds all the system reservations, access controls, machine and environment reservations, log history, etc [80].

B.1.4 Management nodes

These are the nodes that actually perform all the work, they run a VCLD daemon written in perl, that controls a series of compute nodes, these can be physical machines or VMware machines [82]. The management node processes reservations/jobs assigned on the web portal, and ensures that the requested image is available for the user when he connects [80].

B.1.5 Conceptual Overview

Figure 7 illustrates VCL's conceptual overview. The users connect to the web portal to schedule a desired application environment. The environment consists of a full operating system loaded with a selected suite of applications. The user can choose to host his environment on blade servers, vmware virtual machines or standalone machines, which include the traditional computing labs [80].

When the scheduled time arrives, the user connects to the requested resource using standard remote desktop technology [80].

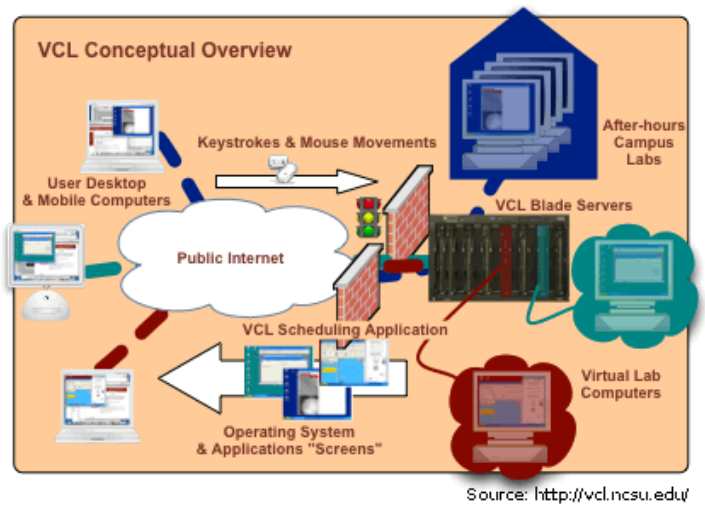

Figure 7 - VCL conceptual overview

B.2 Nimbus

This implementation is a set of tools wrapped in a package that eases installation and maintenance of an IaaS cloud computing solution. Its main purpose is to cater the needs for scientific clouds on universities [83]. A look at the early publications listed denotes that the project was initially geared towards the grid community, later shifting its focus to the cloud paradigm [84]. This implementation,

has however, a major drawback for production deployment; a search on the mailing list shows that at the moment, there is no support for virtual machines running the Windows operating system [85].

B.2.1 Nimbus Architecture

This set of tools allows for very loose coupling, allowing each component to be replaced by a similar one that exposes the same interface [86]. On the public side, it exposes two interfaces, one is a partial implementation of EC2, and the other is a web service (WSRF) interface that exposes the full functionality of the service. This way, a user can select from a number of clients that use either the EC2 semantics, or the WSRF protocol. A context broker allows clients to coordinate large virtual cluster launches automatically and repeatedly. The RM API translates the EC2 or WSRF commands to specific site manager commands. The site manager then communicates with workspace control or workspace pilot. The workspace pilot is a small program that runs on the Virtual Machine Manager and allows the system to be part of a batch processing system, integrating with scheduling systems such as PBS. Workspace control is responsible for the execution of commands on the host that runs the virtual machine, currently only Xen can be controlled like this, but KVM control is implemented and expected to be released soon [83].

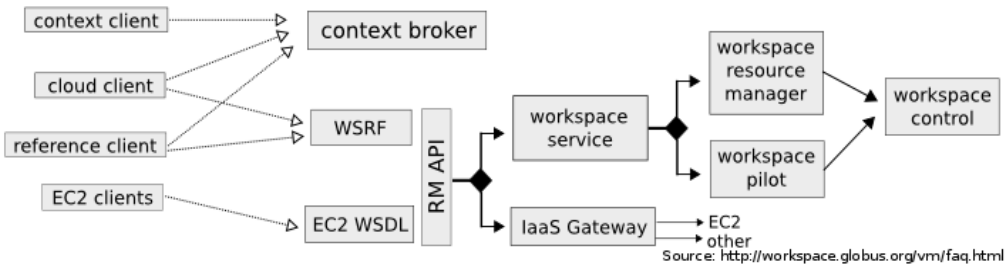

Figure 8 - Nimbus components and interaction diagram

Figure 8 shows how all the components interact. Loose coupling is an integrant part of the design, thus allowing each component's replacement by another as long as the same interfaces are respected.

B.3 OpenNebula

This project is an IaaS implementation that orchestrates network, storage and virtualization technologies to supply a single service to its end users. OpenNebula is able to orchestrate datacenter resources, as well as remote resources according to allocation policies. It is part of the RESERVOIR project, an European Union project that aims to enable massive scale deployment and management of complex IT services [87-88].

OpenNebula enables the creation of private and hybrid clouds, allowing any existing deployment to grow beyond existing physical resources. This is a major advantage in the event of an unexpected load peak.

B.3.1 Architecture

OpenNebula has a three tier architecture, composed of tools, core, and drivers. The tools layer consists on a series of tools that can be used by a client or system administrator to query the interfaces provided by the core. The core consists on a management layer and a SQL backend used to store management information. The driver allows the plugging of different virtualization, storage and monitoring technologies and Cloud services to the core [89].

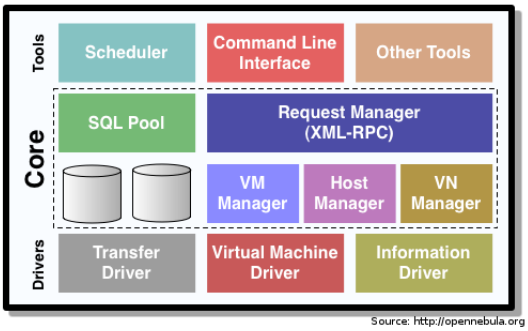

Figure 9 - OpenNebula Architecture

B.3.2 Tools

This layer consists of tools that are either distributed with OpenNebula such as the command line interface (CLI) or the scheduler, and third party tools that are created using either the XML-RPC interface or the REST OpenNebula Cloud API (OGF OCCI) [31, 89].

The Command Line Interface (CLI) is the client that allows clients and system managers to manipulate the infrastructure.

The scheduler is a replaceable module that allows the definition of several policies related to the load or resources availability. This module can be replaced with Haizea [90] to allow a more sophisticated scheduling policy.

B.3.3 Core

Several modules compose the core, their objective is to control and monitor the virtual machines, storage network and hosts. The core performs its actions by invoking a suitable driver [89]. A Request Manager exposes an XML-RPC interface, decoupling the requests from the underlying architecture. A Virtual Machine Manager manages and monitors virtual machines. A Transfer Manager transfers files needed for the correct deployment of virtual machines. A Virtual Network Manager handles IP, MAC addresses, and all virtual Network operations. A Host manager manages and monitors all the physical hosts. A Database keeps the current state on a persistence layer to allow recovery in case of failure, and permits the deployment of any custom accounting modules that may be needed.

This core architecture means that there is a high flexibility for module replacement and instrumentation, therefore easing the process of extending or debugging the system.

B.3.4 Drivers

To allow for a pluggable architecture, when the core needs to interact with a middleware, that interaction is performed through a specific adaptor, that OpenNebula designates as a driver. The driver then performs the requested operation on the specific middleware. OpenNebula has VM drivers that allow it to control Xen KVM and VMware Virtual machines, it also features a driver for libvirt.

B.4 Eucalyptus

This implementation aims at production quality code, some big companies are using it right now, and paid technical support is available for the enterprise version [91].

Eucalyptus is interface compatible with Amazon EC2, but it is designed in such a way that other interfaces may be added at a later time. The platform is implemented using common Linux tools and web services, in an effort to make it easy to install and maintain [92].

This project started at the University of California as part of another research project. As the parent project ended, the original developers formed a company to sell commercial support for the platform, and to continue the development of the product [93].

B.4.1 Architecture

This implementation exposes an EC2 interface, because it *seemed to be the best documented of the available choices at the time we began development and also the most commercially successful [94],* yet its architecture is modular, and other interfaces may be added later. On the virtualization side, both Xen and KVM are supported [95].

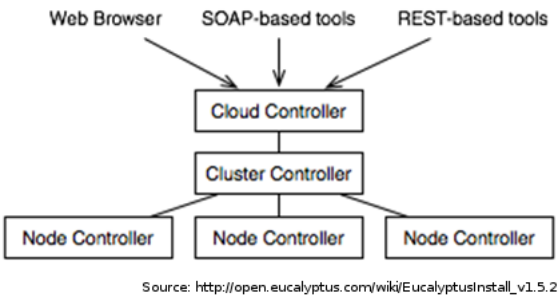

Figure 10 - Eucalyptus architecture

The internal architecture is layered, comprising of a cloud controller that serves as a front end for the whole cloud, a Cluster controller that manages a cluster, and a Node controller that instantiates and stops virtual machines. A Eucalyptus cluster is the equivalent to an Amazon EC2 availability zone, which means that one single cloud controller can control several cluster controllers, each of them controlling several node clusters.

B.5 Enomaly

Enomaly claims to be a cloud infrastructure designed to work alongside the existing virtual datacenter on enterprises of all sizes. It aids in the design, deployment and management of applica-

tions in the cloud, while reducing administrative workload. It features a web based interface that makes it a simple utility for administrators [96].

This is a commercial implementation with a community edition. The community edition is the one that will be analyzed here, as the commercial versions are not free and therefore not usable for most academic purposes.

The documentation from the developer point of view is almost inexistent, which leads to a situation where the mailing list is the only point of contact for developers. The lack of a central knowledge base usually leads to the repetition of questions and scattering of knowledge where the parting of a developer may cause a big impact.

The documentation for the user and system administrator is outstanding; this is most likely due to the commercial version of the project.

B.5.1 Architecture

The community website that was previously used by the enomaly community, that contained some scarce development information has been removed since the initial writing of this document. This limits this project usability even further.

The architecture is not clear, yet the frontend is web based, with a REST API exposed to the user. On the cloud deployment guide, the need for a database becomes apparent; this means that the persistence layer is most likely on that database. Still on the installation guide, there is mention of support to Xen, KVM (and Qemu), as the virtualization backends [97].

Multiple backends and frontends are typical on three tiered architectures; however, from the available documentation nothing can be said about the number of modules, or how those modules communicate among them.

B.6 OpenQRM

OpenQRM is an open source implementation that features a pluggable architecture. It supports multiple virtualization backends, and aims to be a single management console for the complete IT infrastructure with a well defined API that can be used to integrate third party tools and plugins [98].

This implementation features an impressive feature set [99]. It supports migration of images between physical and virtual hosts, which may be an advantage on some scenarios. Support for windows VM is only available for version 3.x which may become a limitation.

B.6.1 Architecture

OpenQRM is separated on a ‘base’ an ‘plugins’, the base just manages the plugins and provides the framework for the interaction with the plugins, but the functionality is all provided by its plugins [100].

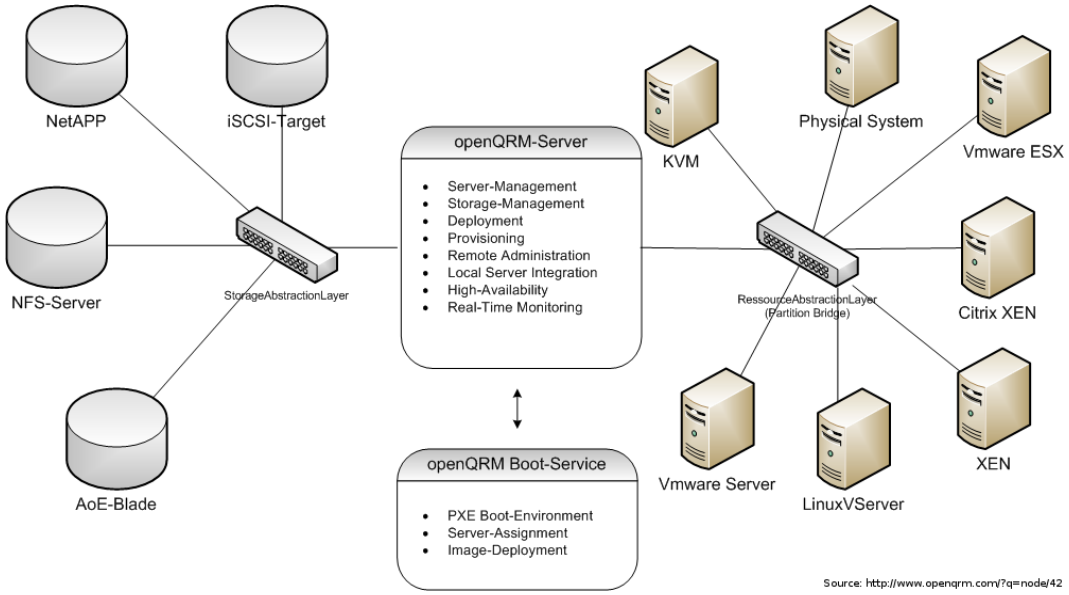

Figure 11 - OpenQRM architecture

Figure 11 shows the architecture of OpenQRM, each of the features depicted is implemented by a plugin, therefore achieving a high degree of modularity.

This sort of architecture eases the implementation of new features, and allows a modular approach to bug tracking, as features are fully contained in modules.

The downside is that an upgrade on the module management system that breaks backward compatibility will lead to a big module rewrite as most (if not all) of the interfaces must be rewritten. This sometimes leads to missing functionality on a newer version as not all modules are ported to the new version (as is the case of Windows VM support).

B.7 ConVirt

ConVirt is an easy to use, yet sophisticated tool that aims to administer and monitor virtualized environments that range from a few Virtual Machines on a single workstation to thousands of virtual machines spread across hundreds of servers [101].

This implementation is not well known among the academic community, a search on Google scholar returns very little articles that mention ConVirt. It has a list of features [102] that rival with the other implementations mentioned on this document, however its stability is not as good as might be expected from a package intended to control a large number of managed servers as it *crashes or hangs intermittently* [103].

B.7.1 Architecture

ConVirt features a pluggable architecture, allowing easy extension and the addition of plugins that implement new features. The communication mechanism between the plugins and the core is not clear, and the developer documentation available seems to be oriented towards a power user instead of a developer as there is no API documentation [104].

The core architecture is well documented, with a clear UML diagram showing the various packages and the relation between them.

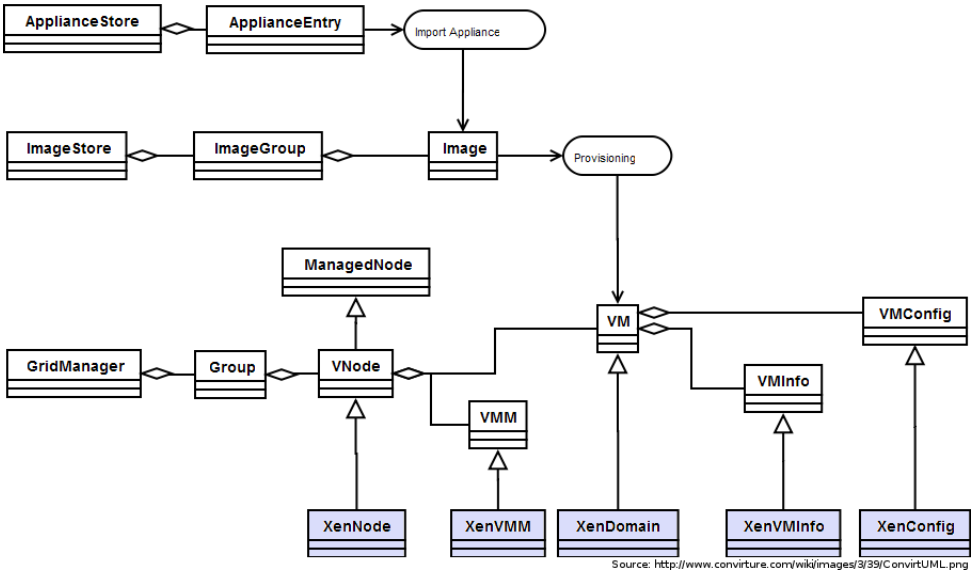

Figure 12 - ConVirt core architecture