UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

VISOR
Virtual Machine Images Management Service for Cloud Infrastructures

João Daniel Raposo Pereira

Dissertation to obtain the Master degree in the specialty
Computer Science
(2^{nd} cycle of studies)

Supervisor: Prof^{a}. Doutora Paula Prata

Covilhã, Junho 2012

Acknowledgments

This dissertation would not have been possible without the valuable help and support of several important people, to whom I would like to express my gratitude.

I would like to extend my deepest gratitude to my supervisor, Prof^{a}. Doutora Paula Prata, for her valuable support and advices throughout the project development stages. Thanks also for providing me the opportunity to explore new ideas in order to keeping me overcoming myself.

To the people from Lunacloud, most precisely to its CEO, Mr. António Miguel Ferreira, and all the development team, whom have provided me with the opportunity to have privileged access to their Cloud Computing resources prior to the Lunacloud infrastructure public launch date. I was also pleased with the opportunity to visit their data center. Many thanks to them.

I am also very grateful to my closest family, specially to my parents, sister, brother-in-law and last but not least to my girlfriend, for always giving me the needed support to overcome all the barriers during the already elapsed academic route, this project and beyond.

Resumo

A Computação em Nuvem (”Cloud Computing”) é um paradigma relativamente novo que visa cumprir o sonho de fornecer a computação como um serviço. O mesmo surgiu para possibilitar o fornecimento de recursos de computação (servidores, armazenamento e redes) como um serviço de acordo com as necessidades dos utilizadores, tornando-os acessíveis através de protocolos de Internet comuns. Através das ofertas de ”cloud”, os utilizadores apenas pagam pela quantidade de recursos que precisam e pelo tempo que os usam. A virtualização é a tecnologia chave das ”clouds”, atuando sobre imagens de máquinas virtuais de forma a gerar máquinas virtuais totalmente funcionais. Sendo assim, as imagens de máquinas virtuais desempenham um papel fundamental no ”Cloud Computing” e a sua gestão eficiente torna-se um requisito que deve ser cuidadosamente analisado. Para fazer face a tal necessidade, a maioria das ofertas de ”cloud” fornece o seu próprio repositório de imagens, onde as mesmas são armazenadas e de onde são copiadas a fim de criar novas máquinas virtuais. Contudo, com o crescimento do ”Cloud Computing” surgiram novos problemas na gestão de grandes conjuntos de imagens.

Os repositórios existentes não são capazes de gerir, armazenar e catalogar images de máquinas virtuais de forma eficiente a partir de outras ”clouds”, mantendo um único repositório e serviço centralizado. Esta necessidade torna-se especialmente importante quando se considera a gestão de múltiplas ”clouds” heterogéneas. Na verdade, apesar da promoção extrema do ”Cloud Computing”, ainda existem barreiras à sua adoção generalizada. Entre elas, a interoperabilidade entre ”clouds” é um dos constrangimentos mais notáveis. As limitações de interoperabilidade surgem do fato de as ofertas de ”cloud” atuais possuírem interfaces proprietárias, e de os seus serviços estarem vinculados às suas próprias necessidades. Os utilizadores enfrentam assim problemas de compatibilidade e integração difíceis de gerir, ao lidar com ”clouds” de diferentes fornecedores. A gestão e disponibilização de imagens de máquinas virtuais entre diferentes ”clouds” é um exemplo de tais restrições de interoperabilidade.

Esta dissertação apresenta o VISOR, o qual é um repositório e serviço de gestão de imagens de máquinas virtuais genérico. O nosso trabalho em torno do VISOR visa proporcionar um serviço que não foi concebido para lidar com uma ”cloud” específica, mas sim para superar as limitações de interoperabilidade entre ”clouds”. Com o VISOR, a gestão da interoperabilidade entre ”clouds” é abstraída dos detalhes subjacentes. Desta forma pretende-se proporcionar aos utilizadores a capacidade de gerir e expor imagens entre ”clouds” heterogéneas, mantendo um repositório e serviço de gestão centralizados. O VISOR é um software de código livre com um processo de desenvolvimento aberto. O mesmo pode ser livremente personalizado e melhorado por qualquer pessoa. Os testes realizados para avaliar o seu desempenho e a taxa de utilização de recursos mostraram o VISOR como sendo um serviço estável e de alto desempenho, mesmo quando comparado com outros serviços já em utilização. Por fim, colocar as ”clouds” como principal público-alvo não representa uma limitação para outros tipos de utilização. Na verdade, as imagens de máquinas virtuais e a virtualização não estão exclusivamente ligadas a ambientes de ”cloud”. Assim sendo, e tendo em conta as preocupações tidas no desenho de um serviço genérico, também é possível adaptar o nosso serviço a outros cenários de utilização.

Palavras-chave

Computação em Nuvem, Infra-estrutura como um Serviço, Imagens de Máquinas Virtuais,

Serviços Web, Transferência de Estado Representacional, Programação Orientada a Eventos, Sistemas de Armazenamento.

Abstract

Cloud Computing is a relatively novel paradigm that aims to fulfill the computing as utility dream. It has appeared to bring the possibility of providing computing resources (such as servers, storage and networks) as a service and on demand, making them accessible through common Internet protocols. Through cloud offers, users only need to pay for the amount of resources they need and for the time they use them. Virtualization is the clouds key technology, acting upon virtual machine images to deliver fully functional virtual machine instances. Therefore, virtual machine images play an important role in Cloud Computing and their efficient management becomes a key concern that should be carefully addressed. To tackle this requirement, most cloud offers provide their own image repository, where images are stored and retrieved from, in order to instantiate new virtual machines. However, the rise of Cloud Computing has brought new problems in managing large collections of images.

Existing image repositories are not able to efficiently manage, store and catalogue virtual machine images from other clouds through the same centralized service repository. This becomes especially important when considering the management of multiple heterogeneous cloud offers. In fact, despite the hype around Cloud Computing, there are still existing barriers to its widespread adoption. Among them, clouds interoperability is one of the most notable issues. Interoperability limitations arise from the fact that current cloud offers provide proprietary interfaces, and their services are tied to their own requirements. Therefore, when dealing with multiple heterogeneous clouds, users face hard to manage integration and compatibility issues. The management and delivery of virtual machine images across different clouds is an example of such interoperability constraints.

This dissertation presents VISOR, a cloud agnostic virtual machine images management service and repository. Our work towards VISOR aims to provide a service not designed to fit in a specific cloud offer but rather to overreach sharing and interoperability limitations among different clouds. With VISOR, the management of clouds interoperability can be seamlessly abstracted from the underlying procedures details. In this way, it aims to provide users with the ability to manage and expose virtual machine images across heterogeneous clouds, throughout the same generic and centralized repository and management service. VISOR is an open source software with a community-driven development process, thus it can be freely customized and further improved by everyone. The conducted tests to evaluate its performance and resources usage rate have shown VISOR as a stable and high performance service, even when compared with other services already in production. Lastly, placing clouds as the main target audience is not a limitation for other use cases. In fact, virtualization and virtual machine images are not exclusively linked to cloud environments. Therefore and given the service agnostic design concerns, it is possible to adapt it to other usage scenarios as well.

Keywords

Cloud Computing, IaaS, Virtual Machine Images, Web Services, REST, Event-Driven Programming, Storage Systems.

Contents

List of Figures

List of Tables

List of Listings

Acronyms and Abbreviations

AJAX	Asynchronous Javascript and XML
AKI	Amazon Kernel Image
AMI	Amazon Machine Image
API	Application Programming Interface
ARI	Amazon Ramdisk Image
AWS	Amazon Web Services
CLI	Command-Line Interface
CRUD	Create, Read, Update and Delete
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
EBS	Elastic Block Storage
EC2	Elastic Cloud Compute
Ext	Extended file system
FG	FutureGrid
FGIR	FutureGrid Image Repository
GFS	Google File System
HATEOAS	Hypermedia as the Engine of Application State
HDFS	Hadoop Distributed Filesystem
HMAC	Hash-based Message Authentication Code
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IaaS	Infrastructure-as-a-Service
I/O	Input/Output
ID	Identifier
IP	Internet Protocol
iSCSI	Internet Small Computer System Interface
JSON	JavaScript Object Notation
LCS	Lunacloud Storage
LVM	Logical Volume Manager
MAC	Media Access Control
MD5	Message-Digest algorithm 5
MIF	Mirage Image Format
NFS	Network File System
NIST	National Institute of Standards and Technology
NoSQL	Not only SQL
NTFS	New Technology File System
OS	Operating System
PaaS	Platform-as-a-Service
PID	Process Identifier
POP	Post Office Protocol
RAM	Random Access Memory
REST	Representational State Transfer
RDP	Remote Desktop Protocol
ROA	Resource Oriented Architecture

S3	Simple Storage Service
SaaS	Software-as-a-Service
SCP	Secure Copy
SHA1	Secure Hash Algorithm 1
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSH	Secure Shell
TDD	Test-Driven Development
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTC	Universal Time Coordinated
UUID	Universally Unique IDentifier
VAS	VISOR Auth System
VCS	VISOR Common System
VDI	Virtual Disk Image
VHD	Virtual Hard Disk
VIS	VISOR Image System
VISOR	Virtual Images Service Repository
VM	Virtual Machine
VMDK	Virtual Machine Disk Format
VMM	Virtual Machine Monitor
VMS	VISOR Meta System
VWS	VISOR Web System
W3C	World Wide Web Consortium
WS	Web Services
WSDL	Web Service Definition Language
WSRF	Web Services Remote Framework
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language-Remote Procedure Call

Chapter 1

Introduction

Cloud Computing has been defined by the U.S. National Institute of Standards and Technology (NIST) as ”a model for enabling ubiquitous, 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” [1]. In this way, Cloud Computing allows users to access a wide range of computing resources, such as servers presented as virtual machine (VM) instances, whose number varies depending on the amount of required resources. Where sometimes running a simple task may require just a single machine, other times it may require thousands of them to handle workload peaks. To address elastic resources needs, Cloud Computing brings the appearance of limitless computing resources available on demand [2].

An Infrastructure-as-a-Service (IaaS) is the founding layer of a cloud. It is the framework responsible for managing the cloud underlying physical resources, such as networks, storage and servers, offering them on demand and as a service. Through an IaaS provider, customers only need to pay for the amount of provisioned resources (e.g. number of VMs, number of CPUs per VM, network bandwidth and others) and for the time they use them [3]. Thus, instead of selling raw hardware infrastructures, cloud IaaS providers typically offer virtualized infrastructures as a service, which are achieved through virtualization technologies [4].

Virtualization is applied to partitioning physical server’s resources into a set of VMs presented as compute instances. This is achieved by tools commonly known as hypervisors or Virtual Machine Monitors (VMMs). In fact, virtualization is the engine of a cloud platform, since it provides its founding resources (i.e. VMs) [5]. Server’s virtualization has brought the possibility to replace large numbers of underutilized, energy consumers and hard to manage physical servers with VMs running on a smaller number of homogenised and well-utilized physical servers [6].

Server virtualization would not be possible without VM images, since they are used to provide systems portability, instantiation and provisioning in the cloud. A VM image is represented by a file, which contains a complete operating system (OS). A VM image may be deployed on bare metal hardware or on virtualized hardware using a hypervisor, in order to achieve a fully functional VM that users can control and customize [7]. Therefore, a VM image is generated in order to deploy virtual compute instances (i.e. VMs) based on it. On a simplistic overview, we see the process of instantiating a VM in a cloud IaaS as contemplating three main components: the raw material, the manufacturer and the delivery. The raw material are VM images, handled by the manufacturer, which is the hypervisor, that in turn produces a fully functional VM instance to be presented to users through a delivery service, which is the IaaS interface.

Since VM images are a key component of the Cloud Computing paradigm, in which VM instances are built upon, the management of large amounts of VM images being deployed over multiple distributed machines can become an exponential bottleneck. For that purpose, most IaaS offers embed its own VM image repository [8]. An image repository holds images that can be used for VMs instantiation, as well as providing mechanisms to distribute those images to hypervisors. VM images in an image repository are placed on the hypervisor by the provisioning system (i.e. IaaS). These repositories are commonly based on the IaaSs own storage systems.

1.1 Motivation

As said by Ammons *et al.* from IBM Research [9], the rise of IaaS offers have brought new problems in managing large collections of VM images. These problems stem from the fact that current cloud IaaS frameworks do not provide a way to manage images among different IaaSs [8]. Their image repositories are tied to their own constraints and needs, without commonly offering services to store VM images and catalogue their associated metadata (i.e. information about them) across different IaaSs in the same repository.

Thereby, limitations and incompatibilities arise while trying to efficiently manage VM images on environments containing multiple heterogeneous IaaSs and their own storage systems, or when migrating between different IaaSs. Facing the cloud computing paradigm, users should not see themselves limited in shifting or incorporating multiple heterogeneous IaaSs and their storage systems in their own environment. In fact, such interoperability and vendor lock-in constraints are among the major drawbacks pointed to Cloud Computing [10, 11, 12], where cloud providers offer proprietary interfaces to access their services, locking users within a given provider. As said by Ignacio M. Llorente, the director of the OpenNebula cloud platform [13], ”the main barrier to adoption of cloud computing is cloud interoperability and vendor lock-in, and it is the main area that should be addressed” [14].

Besides interoperability and compatibility mechanisms among heterogeneous IaaSs, there is also the need to efficiently catalogue and maintain an organized set of metadata describing images stored in an image repository [15]. As stated by Bernstein *et al.* from CISCO, ”the metadata which specifies an image is a crucial abstraction which is at the center of VM interoperability, a key feature for Intercloud” [16]. It is also said that an open, secure, portable, efficient, and flexible format for the packaging and distribution of VM images is a key concern.

All the concerns around VM images management are becoming increasingly important since Cloud Computing and virtualization technologies are increasing in adoption, thus it becomes a matter of time till IaaS administrators face a collection of thousands of VM images [17]. Furthermore, since VM images can be cloned (in order to clone VMs) versioned and shared, they are expected to continuously increase in number on an IaaS, and their management becomes then a key concern [6]. In fact, as said by Wei *et al.* from IBM, VM images sharing is one of the fundamental underpinnings for Cloud Computing [18]. Moreover, the efficient VM image management is a crucial problem not only for management simplicity purposes but also because it has a remarkable impact on the performance of a cloud system [19].

Besides the already stated problems, it is also required to pay attention to the VM images management as a service rather than as embedded IaaS functionalities. In fact, most IaaSs embed the image management functionalities in some monolithic component, instead of isolating those functionalities in an isolated service (as will be described further in Chapter 2). Some important researchers and companies have already stated such need. As stated by Metsch, from Sun Microsystems (now Oracle) on an Open Grid Forum report [20], there is the need to have methods to register, upload, update and download VM images. Wartel *et al.* from the European Organization for Nuclear Research (CERN) have also bring attention to the need for image management services. They say that ”one of the challenges of maintaining a large number of hypervisors running possible different VM images is to ensure that a coherent set of images is maintained” [21]. They also state that it is needed to provide a central server which would maintain and provide access to a list of available VM images, providing a view of the available images to serve hypervisors, including a set of metadata describing each image, such as its name, OS, architecture and the actual image storage location.

1.2 Objectives

Considering all the exposed problems in the previous section, we propose an agnostic VM image management service for cloud IaaSs, called VISOR (which stands for Virtual Images Service Repository). Our approach differs from IaaS tied VM image repositories and embedded services, as VISOR is a multi-compatible, metadata-flexible and completely open source service. VISOR was designed from bottom to top not to fit in a specific platform but rather to overreach sharing and interoperability limitations among different IaaSs and their storage systems.

It aims to manage VM images and expose them across heterogeneous platforms, maintaining a centralized generic image repository and image metadata catalogue. We have targeted a set of IaaSs and their storage systems, but given the system modularity, it is possible to easily extend it with other systems compatibility. With a unified interface to multiple storage systems, it is simpler to achieve a cross-infrastructure service, as images can be stored in multiple heterogeneous platforms, with seamless abstraction of details behind such process.

Furthermore, placing cloud IaaSs as the main target audience is not a limitation for other use cases. In fact the need to manage wide sets of VM images is not exclusively linked to cloud environments, and given the service agnostic design concerns, it is possible to adapt it to other use cases. Also, we are looking forward to achieving a remarkable service not only for its concepts and features but also for its performance.

1.3 Contributions

This dissertation describes a cloud agnostic service through which VM images can be efficiently managed and transferred between endpoints inside a cloud IaaS. During the described work in this dissertation, these were the achieved contributions:

- By studying the existing IaaS solutions and the isolated services towards the management of VM images in cloud environments, we were able to compile an overview of their architecture, VM image management functionalities and storage systems, where images are saved in.

- Since VISOR aims to be a high performance and reliable service, prior to addressing its development, we have conducted an analysis of both Web services and I/O (Input/Output) concurrency handling architectural approaches, a work which may fit in a future research publication.

- The proposed VISOR service was implemented and it is now fully functional, with the source code repository and documentation being freely exposed through the project home page at `http://www.cvisor.org`.

- We have proposed some innovative VISOR design concepts, including the isolation of data communication formats conversion and authentication mechanisms on pluggable middleware, highly increasing the service modularity and compatibility. There were also included abstraction layers, responsible for providing seamless integration with multiple heterogeneous storage and database systems. Finally, we have also addressed data transfer approaches, in order to assess how could us speed up image downloads and uploads, while also sparing servers' resources, which was achieved through chunked streaming transfers.

- Finally, we have also conducted an intensive VISOR testing approach, evaluating its performance and resources usage rate. During these tests we have also assessed the underpinning storage systems performance (where VM image files are stored in), which is a comparison that we have not found yet among published work.

1.3.1 Scientific Publication

The contributions of this dissertation also include a scientific publication, containing the description of the proposed VISOR service aims, features and architecture [22]:

J. Pereira and P. Prata, *"VISOR: Virtual Machine Images Management Service for Cloud Infrastructures"*, in The 2nd International Conference on Cloud Computing and Services Science, CLOSER, 2012, pp. 401–406.

We have already submitted another research paper, containing the detailed description of the VISOR architecture, implementation details and the conducted performance evaluation methodology and obtained results. We are also expecting to submit another research paper soon, based on the related work research presented in this dissertation.

1.3.2 Related Achievements

- In part due to the work presented in this dissertation, the author has been elected as one of the 100 developers and system administrators all around Europe, to integrate a two--phase beta testing program of the now launched Lunacloud [23] cloud services provider. It has exhaustively tested the Lunacloud compute (i.e. VMs) and storage services. For the storage service tests, the author has used VISOR and the same testing methodology employed to test VISOR and its compatible storage backends described in this dissertation.
- Due to the acquired deep knowledge of the Ruby programming language [24] during the development of the proposed service, the author has been invited to address the development of a future proposed Ruby virtual appliance (a VM image preconfigured for a specific software development focus) for the Lunacloud platform.
- The author has also given a talk about Cloud Computing entitled *"Step Into the Cloud - Introduction to Cloud Computing"*, during the XXI Informatics Journeys of the University of Beira Interior, April 2012.

1.4 Dissertation Outline

This dissertation is organized as follows. In Chapter 2 the state-of-the-art for Cloud Computing IaaS solutions, their storage systems and the existing isolated VM image management services are described. Chapter 3 introduces the necessary background concepts for understanding the Cloud Computing paradigm. It also contains a review of our VISOR image service implementation options regarding Web services and I/O concurrency architectures. In Chapter 4, the design, architecture and development work of the proposed VISOR image service are described in detail. Chapter 5 contains the discussion of the VISOR performance evaluation tests methodology and the obtained results, while we also compare them with other related VM image service published performance. Finally, in Chapter 6 we outline our work conclusions and future work.

Chapter 2

Related Work

In this chapter we will present the state of the art for Cloud Computing IaaSs and independent VM image services found in the literature. For IaaSs we will compare them and describe their features, architecture, storage systems and embedded VM image services (if any). We will also compare the independent VM image services by describing their aims, strengths and limitations. Finally, we will provide a summary of all the outlined problems found among current solutions and how we will tackle them within our approach, the VISOR image service.

2.1 Infrastructure-as-a-Service (IaaS)

IaaSs are the lowest layer of a cloud. They manage physical hardware resources and offer virtual computing resources on demand, such as networks, storage and servers [1] through virtualization technologies [5]. Among all existing IaaSs [25], Amazon Elastic Compute Cloud (EC2) was a pioneer and is the most popular offer nowadays. For open source offers, Eucalyptus, OpenNebula, Nimbus and OpenStack stand out as the most popular IaaSs [26, 27, 28, 29, 30]. A summary comparison between these IaaSs can be observed in Table 2.1 in the end of this section.

It is possible to identify eight main common components in an IaaS: hardware, OS, networks, hypervisors, VM images and their repository, storage systems and user's front-ends, which can be described as follows:

- **Hardware and OS:** An IaaS, like other software frameworks, relies and is installed on physical hardware with previously installed OSs.
- **Networks**: Networks include the Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP), bridging and the physical machines subnet (a logically subdivision of a network) arrangement. DHCP, DNS and bridges must be configured along with the IaaS, as they are needed to provide virtual Media Access Control (MAC) and Internet Protocol (IP) addresses for deployed VMs [28].
- **Hypervisor:** A hypervisor, provides a framework allowing the partitioning of physical resources [5]. Therefore a single physical machine can host multiple VMs. Proprietary hypervisors include VMware ESX and Microsoft Hyper-V. Open source hypervisors include Virtuozzo OpenVZ, Oracle VirtualBox, Citrix Xen [31] and Red Hat KVM [32] [26]. A virtualization management library called *libvirt* [33] is the tool commonly used to orchestrate multiple hypervisors [28].
- **VM Images:** A VM image is a file containing a complete OS, which is deployed on a virtualized hardware using a hypervisor, providing a fully functional environment that users can interact with [7].
- **VM Image Repository:** A VM image repository is where VM images are stored and retrieved from. Commonly, such repositories rely on a storage system to save images. In almost all the addressed IaaSs, the VM images management functionalities are not presented as

an isolated service but rather being integrated somewhere in the IaaS framework. The exception in this set of IaaSs is OpenStack, the most recent of them all.

- **Storage System:** Usually, an IaaS integrates a storage system or relies on an external one, which is intended to store and serve as image repository, while also being used to store raw data (IaaS operational data and user's data).

- **Front-ends:** Front-ends are the interfaces exposing the IaaS functionalities (i.e. VMs, storage and networks) to clients. These include Application Programming Interfaces (APIs), Command-Line Interfaces (CLIs) and Web services and applications.

2.1.1 Elastic Compute Cloud (EC2)

Amazon Web Services (AWS) is a proprietary set of compute, storage, load balancing, monitoring and many other [34] cloud services provided by Amazon. AWS services can be accessed through the AWS Web interface. Optionally they can also be accessed through their exposed Simple Object Access Protocol (SOAP) [35] and Representational State Transfer (REST) [36] Web service interfaces, over the HTTP protocol [37]. AWS was one of the cloud IaaS pioneers and has served as model and inspiration for other IaaSs.

Amazon Elastic Compute Cloud (EC2) [38] is a AWS Web service providing the launching and management of VM instances in the Amazon data centers' facilities, using the AWS EC2 APIs. Like in other cloud IaaSs, in EC2 the access to instantiated VM instances is mainly done using the SSH protocol (mainly for Linux instances) or the Remote Desktop Protocol (RDP) protocol [39] (for Windows instances). It is also possible to communicate and transfer data to and from VM instances using common communication and transfers protocols, such as SCP, POP, SMTP, HTTP and many others [38]. Users have full control of the software stack installed in their VM instances as equally in their configurations, such as network ports and enabled services. EC2 instances are deployed based on Amazon Machine Images (AMIs) which are machine images preconfigured for EC2 and containing applications, libraries, data and configuration settings. Users can create custom AMIs or use preconfigured AMIs provided by AWS.

EC2 instances can run with either volatile or persistent storage. An EC2 has volatile storage if running directly backed by its physical host storage. However, this option is falling in disuse since all the instance data is lost when it is shutdown [38]. To circumvent this, AWS provides a service called Elastic Block Storage (EBS) [40], which provides persistent storage to EC2 instances. EBS storage are network attached volumes that can be mounted as the filesystem of EC2 instances, thus persisting their data. EC2 instances can be placed in different locations to improve availability. These locations are composed by Availability Zones and Regions. Availability Zones are independent locations engineered to be insulated from failures, while providing low latency network connectivity to other Availability Zones in the same Region. Regions consist of one or more Availability Zones geographically dispersed in different areas or countries.

The Amazon Simple Storage Service (Amazon S3) [41] is where EC2 VM images (i.e. both user's custom and Amazon's own AMIs) and user's data are stored. S3 is a distributed storage system where data is stored as "objects" (similar to files) grouped in "buckets" (similar to folders). Buckets must have an unique name across all existing S3 buckets and can be stored in one of several Regions. S3 Regions are useful for improving network latency (i.e. minimizing the distance between clients and data), minimize costs or address regulatory requirements [42]. To instantiate a VM instance on EC2, a VM image is picked from S3 and deployed as a fully functional VM using the Xen hypervisor [43].

2.1.2 Eucalyptus

Eucalyptus (Elastic Utility Computing Architecture for Linking Your Programs to Useful Systems) [44], is a framework for building cloud infrastructures. It was developed at Santa Barbara University (California) and is now maintained by Eucalyptus Systems, Inc. The overall platform architecture and design was detailed by Nurmi *et al.* [45]. Eucalyptus was developed to provide an open source version of Amazon EC2 [28] and has now both an open source and enterprise versions. It implements an Amazon EC2 compatible interface. Eucalyptus is one of the most popular IaaS offers and probably the most widely deployed [44]. It provides its own distributed storage system, called Walrus.

Walrus is the Eucalyptus distributed storage system, which is primarily used to store and serve VM images [46]. Besides VM images it is also used to store raw data (e.g. users and system data). Walrus mimics Amazon S3 in its design (organizing data in buckets and objects) and interfaces. Thus it implements S3 compatible REST and SOAP APIs [45]. Walrus can be accessed by both VM instances and users.

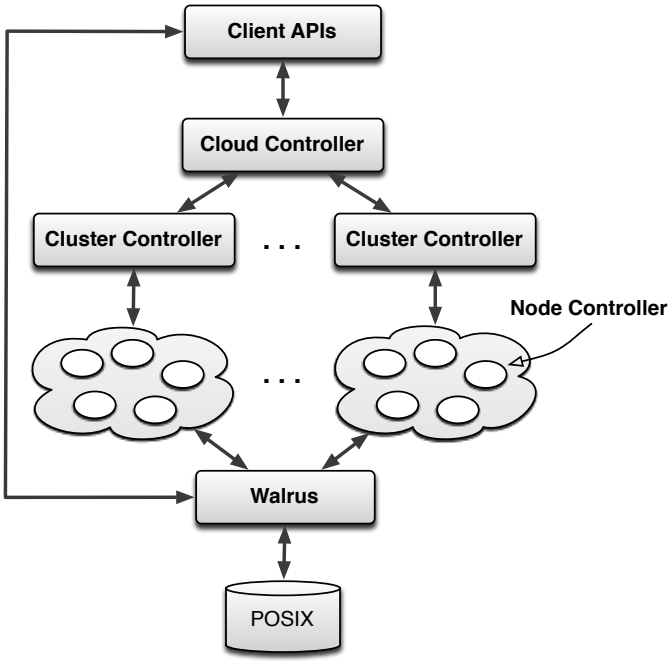

Figure 2.1: Architecture of the Eucalyptus IaaS.

In Figure 2.1 it is pictured the Eucalyptus IaaS architecture. In Eucalyptus *client APIs* are the interfaces connecting clients to the Eucalyptus platform, which include Amazon EC2 and S3 compatible interfaces. Through them users can access compute (i.e. VM instances) and storage (i.e. Walrus) services. The *cloud controller* is the core of an Eucalyptus cloud and is a collection of Web services towards resources, data and interface management. It is also responsible for managing the underlying cluster controllers. *Cluster controllers* execute as a cluster front-end for one or more node controllers, and are responsible for scheduling VMs execution and managing the virtual networks where VMs operate. *Node controllers* execute on every logically connected node that is designated for hosting VM instances. A node controller delivers data to the coordinating cluster controller and controls the execution and management of VM instances hosted on the machine where it runs.

2.1.3 OpenNebula

OpenNebula [13, 47] is another open source framework for building cloud infrastructures. Although it is mainly used to manage private clouds (accessed from inside an organization only) and to connect them with external clouds [30]. It has not been designed to be an intrusive platform but rather to be extremely flexible and extensible, so it can easily fit in network and storage solutions of an existing data center [29]. Such flexibility allows OpenNebula to provide multiple compatible storage systems, hypervisors and interfaces, with the last ones being the Open Cloud Computing Interface [48] and EC2 compatible interfaces [14, 13]. Compared with Eucalyptus, OpenNebula is stronger in support for high numbers of deployed VMs [30].

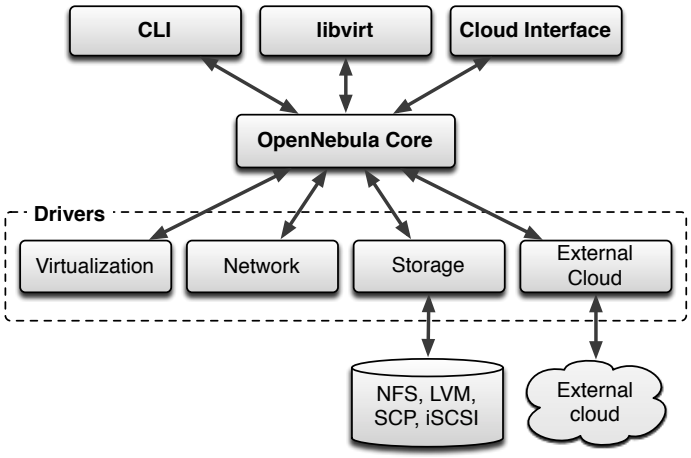

Figure 2.2: Architecture of the OpenNebula IaaS.

The OpenNebula IaaS architecture is pictured in Figure 2.2. OpenNebula exposes its functionalities through three main interfaces. These are a *Command-line interface*, an interface for the open source *libvirt* [33] VMs management library and a *cloud interface* including an Amazon EC2 compatible API.

Since OpenNebula is highly modular, it implements tools compatibility through *drivers* (i.e. plugins). The *OpenNebula core* is the responsible for controlling VMs life cycle by managing the network, storage and virtualization through pluggable drivers.

Regarding the drivers layer, *Virtualization* drivers implement compatibility with hypervisors, including plugins for Xen, KVM, VMware and Hyper-V [13]. The *network* driver is responsible for providing virtual networks to VMs (managing of DHCP, IPs, firewalls and others) [47]. *Storage* drivers manage the storage systems where VM images and users data are stored. These include plugins for the Network File System (NFS), Logical Volume Manager (LVM), SCP and Internet Small Computer System Interface (iSCSI) backends and protocols [14]. Lastly, an OpenNebula cloud can communicate with external clouds through the *external cloud* drivers. This feature makes it possible to supplement an OpenNebula cloud with an external cloud computing capacity, in order to ensure the needed compute capacity to attend demands. These include Amazon EC2 and Eucalyptus plugins [47].

2.1.4 Nimbus

Nimbus [49] is an open source framework (developed at University of Chicago) combining a set of tools to provide clouds for scientific use cases. Like OpenNebula, Nimbus is highly customizable. However, while OpenNebula allows users to switch almost all components (e.g. storage

systems, network drivers), Nimbus focus on low level customizations by administrators and high level customizations by users. Therefore, tools like the storage system, user's authentication mechanism and SSH to access instances are immutable [28]. In this way, Nimbus appears to sit somewhat in the middle of Eucalyptus and OpenNebula regarding customization.

The Nimbus storage system is called Cumulus, and was described by Bresnahan *et al.* [50]. It is used to serve as repository for VM images and to store raw data. Cumulus implements an Amazon's S3 compatible REST interface (like Eucalyptus' Walrus), and extends it as well to include usage quota management [49]. Cumulus is independent of the overall Nimbus architecture, thus it can be installed as a standalone storage system for generic use cases. It has a modular design and lets administrators choose the backend storage system to use with the Cumulus storage service. By default Cumulus stores data on a local POSIX filesystem but it also supports the Apache Hadoop Distributed Filesystem (HDFS) [51].

HDFS is the distributed and replicated storage system of the Apache Hadoop software framework for distributed computing [52]. HDFS was designed to store huge amounts of data with high reliability, high bandwidth data transfers and automated recovery [51]. HDFS is an open source storage inspired by the Google File System (GFS) [53]. GFS is the proprietary file system powering Google's infrastructure.

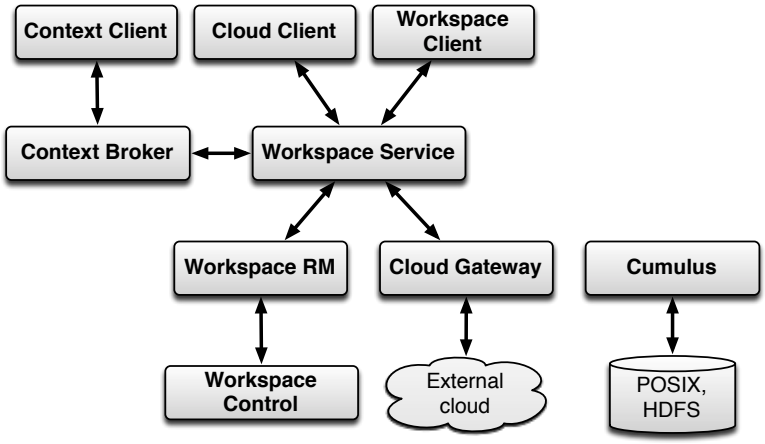

Figure 2.3: Architecture of the Nimbus IaaS.

The architecture of the Nimbus IaaS is detailed in Figure 2.3. Nimbus implements three main clients: context clients, cloud clients and workspace clients. The *context client* is used to interact with the *context broker*, which allows clients to coordinate large virtual clusters deployments automatically. The *cloud client* is the easiest client interface, and it aims to provide fast instance launching to users [49]. The *workspace client* is intended to expose a command-line client to Nimbus.

Both cloud and workspace client tools communicate with the *workspace service*, which implements different protocol front-ends. Front-ends include an Amazon EC2 SOAP-based compatible interface and a Web Services Remote Framework (WSRF) protocol. The *workspace RM* (resource manager) is what manages the platform underlying physical resources. The *workspace control* is the responsible for managing VMs (with hypervisors) and virtual networks. As Eucalyptus, Nimbus can connect to an external cloud through the *cloud gateway* to increment its computing capacity in order to fulfil extra demand needs. To instantiate a VM instance on Nimbus, a VM image is picked from *Cumulus* and deployed as a functional VM using one of the Xen or KVM hypervisors [28].

2.1.5 OpenStack

OpenStack [54] is an open source framework jointly launched by Rackspace and NASA, providing software for building cloud IaaS. It currently integrates three main services: OpenStack Compute, Object Store and Image Service. Besides these three main projects there are also the smaller OpenStack Dashboard and Identity services. Compute (codenamed *Nova*) is intended to provide virtual servers on demand (i.e. VM instances). Object Store (codenamed *Swift*) is a distributed storage system. Image Service (codenamed *Glance*) is a catalogue and repository of VM images which is exposed to users and OpenStack Nova. Dashboard (codenamed *Horizon*) is a Web interface for managing an OpenStack cloud. Finally, Identity (codenamed *Keystone*) provides authentication and authorization mechanisms to all other OpenStack services.

Swift is a distributed storage system with built-in redundancy and failover mechanisms [55]. It is used to store VM images and other data on an OpenStack cloud. Swift is distributed across five main components: proxy server, account servers, container servers, object servers and the ring [55]. The *proxy server* is responsible for exposing the Swift REST API and handling incoming requests. Swift also have an optional pluggable Amazon S3 compatible REST API. *Account servers* manage users' accounts defined in Swift. The *Container servers* manage a series of containers (i.e. folders) where objects (i.e. files) are mapped into. *Object servers* manage the objects on each one of many distributed storage nodes. Finally, the *ring* is a representation (similar to an index) containing the physical location of the objects stored inside Swift.

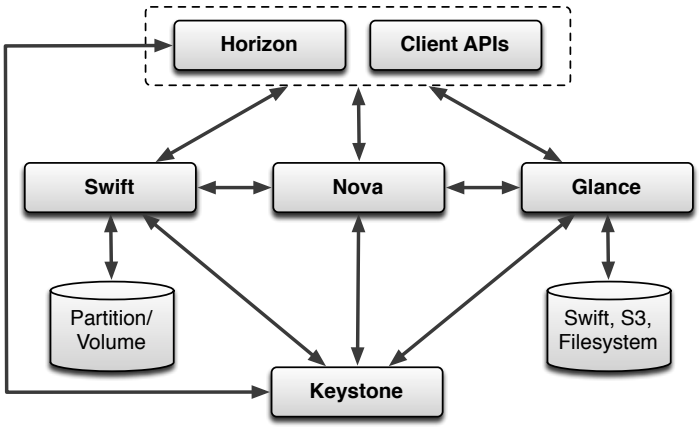

Figure 2.4: Architecture of the OpenStack IaaS.

The OpenStack architecture is pictured in Figure 2.4. All services interact with each other using their public APIs. Users can interact with an OpenStack cloud using Horizon or the exposed *client APIs* for each service. All services authenticate requests through *Keystone*.

Nova encompasses an OpenStack API and an Amazon EC2 compatible API on its internal *nova-api* component. It also comprises the *nova-compute*, *nova-network* and *nova-schedule* components, responsible for managing VM instances via hypervisors, virtual networks and the scheduling of VMs execution, respectively. Besides that, there are also *queues* to provide message communication between processes and SQL *databases* to store data.

Swift is used by the other services to store their operational data. *Glance* can store VM images in Swift, Amazon S3, or in its host's local filesystem (as will be described further in Section 2.2.1). To instantiate a VM instance, Nova queries and picks an image from the Glance image service and deploys it using one of the Xen, KVM, LXC and VMware ESX hypervisors [54].

Table 2.1: Cloud Computing IaaS summary comparison.

	EC2	**Eucalyptus**	**Nimbus**	**OpenNebula**	**OpenStack**
Authors	Amazon Web Services	Eucalyptus Systems, Inc.	University of Chicago	OpenNebula Project Leads	Rackspace and NASA
First Release	2006	2008	2008	2008	2010
Supported OS	Linux, Windows	Linux, Windows	Linux	Linux, Windows	Linux, Windows
Hypervisors	Xen	Xen, KVM, VMware ESX	Xen, KVM	Xen, KVM, VMware ESX, Hyper-V	Xen, KVM, VMware ESX
Image Storage	S3	Walrus	Cumulus	NFS, SCP, iSCSI	Swift, S3, Filesystem
Isolated Image Service	-	-	-	-	Glance
License	Proprietary	Proprietary, GPLv3	Apache 2.0	Apache 2.0	Apache 2.0

2.2 Image Services

2.2.1 OpenStack Glance

OpenStack [54] was the first IaaS framework (in the best of our knowledge) that has isolated the VM images management on a separated service (OpenStack Glance [56]). Therefore, the VM images management was outsourced, instead of maintaining such functionalities integrated in some ”monolithic” platform’s component. Since OpenStack is one of the most recent cloud frameworks, it already materializes the need to efficiently manage and maintain VM images on a cloud IaaS. This has become latent in OpenStack since it is a recent project appearing when the Cloud Computing adoption is reaching the masses, leading to the increasing on the number of VM images to manage.

OpenStack Glance is an image repository and service intended to manage VM images inside an OpenStack cloud, responding to the OpenStack Nova compute service in order to gather images and deploy VM instances based on them. Through Glance, users and the Nova service can search, register, update and retrieve VM images and their associated metadata. Metadata attributes include the image ID, its owner (i.e. the user which has published it), a description, the size of the image file and many others [56]. Authentication and authorization is guaranteed by the OpenStack Keystone authentication service [57]. In Glance users can be managed by roles (i.e. administrators and regular users) and linked by groups membership [56]. This is useful as in an OpenStack cloud there can be many projects from many teams, and administrators want to enforce security and isolation during OpenStack usage.

Glance is implemented as a set of REST Web services, encompassing the *glance-api* and *glance-registry* services. Javascript Object Notation (JSON) [58] is used as the data input/output format and the complete Glance API is described in [57]. *Glance-api* serves as front-end for the Glance clients (end-users and OpenStack Nova) requests. It manages VM images and their transfer between clients and compatible storage backends. Compatible storage backends encompass the host’s local filesystem (default backend), OpenStack Swift and optionally Amazon S3. It can also gather VM images from an HTTP URL. *Glance-registry* is the component responsible for managing and storing VM images’ metadata on an underlying SQL database [56]. When a request for registering a VM image or to update an existing one reaches the glance-api, the image metadata is sent to glance-registry which will handle and record it on an underlying database. Glance also integrates and maintains a cache of VM images, in order to speed up future requests for cached images. The Glance API calls may also be restricted to certain sets of users (i.e. administrators and regular users) using a Policy configuration file. A Policy file is a JSON document describing which kinds of users can access which Glance API functionalities.

Users are given with client APIs and a CLI from which they can manage Glance. Through them users have access to the full Glance REST API functionalities [57]. Thus they can add, retrieve, update and delete VM images in the Glance repository. Administrators can manage Glance through a specific command-line interface, having the ability to perform backups of the glance-registry database, manage the Glance server’s status (i.e. start, stop and restart) and other administration tasks. The Glance service functionalities are also integrated in the OpenStack Horizon dashboard Web interface.

In summary, Glance is a key service on the OpenStack platform. It is the intermediary between the Nova compute service and the available VM images. It has interesting features as the user’s management roles and groups membership, images caching and multiple available

storage backends for VM images. Its REST API is also concise and well formed. Although being open source, Glance is tied to OpenStack requirements, development plans, protocols, tools and architecture. As expected, Glance was designed to seamlessly fit with other OpenStack services, mainly with Nova and Keystone. It is also constrained by a rigid metadata schema and support for SQL databases only. This serves well the purposes of OpenStack but could be a limitation for other platforms. Its API is also limited to the JSON data format, which although being a popular data format nowadays, could be a severe limitation for existing clients which only communicate with older data formats, such as the Extensible Markup Language (XML) [59].

2.2.2 FutureGrid Image Repository

The FutureGrid (FG) platform [7] provides Grid and Cloud test-beds for scientific projects. It is deployed in multiple High-Performance Computing (HPC) resources distributed across several USA sites. FG is intended to be used by researchers in order to execute large scale scientific experiments. To use the platform, users need to register for an account on the project Web portal [60] and create or join a project from there after. FG enables users to conduct intensive computational experiments by submitting an experimentation plan. It handles the execution of experiments, reproducing them using different Grid and Cloud frameworks. FG can deploy experiments on cloud infrastructures and platforms, grids and HPC frameworks, as equally on bare metal hardware. The FG available cloud infrastructures are Eucalyptus [44], Nimbus [49] and OpenStack [54]. Cloud platforms include Hadoop [52], Pegasus [61] and Twister [62]. Therefore, users gain the ability to compare frameworks in order to assess which of them is best suited for their experiments or to test possible migrations between frameworks.

Among many other components, FG integrates the FutureGrid Image Repository (FGIR) in its architecture. FGIR is intended to manage VM images inside FG across all available cloud infrastructures and platforms. Laszewski *et al.* has done a quick overview of FGIR aims [8] and further described it in detail [15]. FGIR has focused on serving four kinds of client groups: single users, user groups, system administrators and other FG subsystems. Single users are those conducting experiments on the FG platform. They can create and manage existing VM images. Groups of users refer to groups of FG users collaborationg in the same FG project, where images are shared between those collaborators. System administrators are those responsible for physically manage the FGIR, which have the ability to manage backups, the FGIR server status (i.e. start, stop and restart) and other administration tasks. FG subsystems are other FG platform components and services that rely on FGIR to operate. One of this components is the FG RAIN, which is the service responsible for picking up an image from the FGIR and deploy it as a virtual instance on a test-bed inside the FG platform.

FGIR is implemented as a Web service and lets clients search, register, update and retrieve VM images from the repository. The core of FGIR includes mechanisms for users usage accounting (i.e. tracking user's service usage) and quota management (e.g. controlling used disk space), image management functionalities and metadata management. FGIR manages both image files and their metadata, where metadata is used to describe each image's properties. Image metadata is stored on a database and includes attributes such as the image ID, its owner (i.e. user which has published the image), a description, the size of the image file and many others [15]. The FGIR provides compatibility with many storage systems, since the FG platform needs to interact with different infrastructures and platforms. The compatible cloud storage systems comprise OpenStack Swift [63] and Nimbus Cumulus [50]. There is also the possibility to store images on FGIR server's local filesystem and in the MongoDB database system [64, 65].

The FGIR functionalities are exposed to clients through several tools and interfaces. Client tools include the FG Web portal [60], a CLI and an interactive shell, from which users are able to access the exposed service functionalities (i.e. search, register, update and retrieve VM images). These tools communicate with the repository through the FGIR REST interface (although not yet implemented at the FGIR description paper publishing time [15]) and its programming API. Further details about the client interfaces functionalities can be found at [15]. Security is guaranteed by the FG platform security services, which include a Lightweight Directory Access Protocol [66] server handling user accounts and authentication requests.

In summary, FGIR is an image service towards the management of VM images in the FG platform. Although it has been implemented to address a variety of heterogeneous cloud infrastructures, it is limited in the number of compatible cloud storage systems to Cumulus and Swift [15]. Even though the MongoDB storage option makes use of GridFS [67] (which is a specification for storing large files in MongoDB), its performance has shown [15] that it is not a viable option for a VM image file storage backend. FGIR also suffers from flexibility limitations, given the fact that the whole repository functionalities are located on a single service, rather than split across several distributed Web services, which would increase the service scalability, modularity and isolation. This would be possible for example, by outsourcing the image metadata management to a separate Web service. In fact, as expected FGIR is tied to the FG platform requirements, since it is not exposed as an open source project and we have not found any indicators that FGIR is used in any other projects or platforms besides FG.

2.2.3 IBM Mirage Image Library

Ammons *et al*. [9] from IBM Research have recently described the Mirage image library. Mirage is described as a more sophisticated VM image library than those typically found in some IaaSs, and pluggable into various clouds. Mirage provides common image service features such as image registering, searching, retrieving and access control mechanisms. However, Mirage's main feature and purpose is the ability for off-line image introspection and manipulation. Off-line means that there is no need to boot up an image to see inside it, which can be done with images on a dormant state. Therefore Mirage is able to search for images which contain certain software or configuration options, generating a report listing the images that have matched the search. Such features are made possible by indexing the images filesystem structure when they are pushed to the Mirage library, instead of treating them like opaque disk images.

During the indexing procedure, images are not saved in their native format but rather in the Mirage Image Format (MIF) [6]. This is what enables the off-line image introspection features. MIF is a storage format also from IBM, which exposes semantic information from image files. Thus, images are not stored in Mirage as a single file but rather storing each image file's contents as separate items. An image as a whole is represented by a manifest, which is intended to serve as recipe for rebuilding an image from its content chunks when it is required for download. Besides image introspection, MIF also provides Mirage with the ability to exploit images similarities. Therefore Mirage is able to save storage space by storing the contents of an image only once, even if the same content appears in other images [6].

In order to provide portability, Mirage converts MIF images to standard machine image formats on-the-fly when they are requested. The Mirage *image indexer* component is the responsible for converting images in both directions, this is from both MIF to some standard image format and vice-versa. The indexer was designed with a plugin architecture in order to provide

compatibility with many standard machine image formats. However, the Mirage compatible image formats are not described in [9].

Mirage has introduced another interesting feature: an image version control system. This makes it possible to track and retain older versions of an image. Mirage maintains a provenance tree that keeps track of how each image has derived from other versions. Whenever an image is updated, a new entry is added to its versions chain and the last version becomes the one from which the image should be generated (i.e. converted to a proper format) and provided from. Image metadata is maintained on the Mirage *catalog manager* and stored on an underlying SQL database. Metadata attributes include an image identifier, the state of an image (i.e. active or deleted), its creation timestamp and its derivation tree from previous versions, among others. When an image is marked as "deleted" it becomes eligible for garbage collection, another interesting feature of Mirage. Regarding users interaction, Mirage provides a set of *library services* containing both user and administrator functions to manage the image library. As expected user functionalities include the registering, searching and retrieving of images. The administrator functionalities provide the ability to manage the Mirage server status (i.e. start, stop and restart), manage the garbage collection, lock images (i.e. prevent changes) and other tasks.

In summary, Mirage is an image library with many interesting features not found in IaaS image services like OpenStack Glance [56]. Although since Mirage uses the MIF format for storing images, it always needs to convert images to make them usable by most hypervisors. As expected, such conversion results in service latency. Even though Mirage maintains a cache of popular images already converted to usable formats [9], whenever an image is required and not present in the cache (or present but with a different format of that required) it always needs to reconstruct that images from the MIF data. Moreover, although it is said [9] that Mirage can be plugged in many clouds, we have only found mentions to the IBM Workload Deployer product [68], where Mirage serves images in clients' private clouds, and to the IBM Research Compute Cloud [69], which is a private cloud for the IBM Research community. It is also not stated the integration of Mirage as a service for IaaSs but rather as "pluggable into the hypervisor platforms that customers already have in their data centers".

We have found Mirage to be used in other publications and experiments around VM images [18, 17, 6, 70]. In [6], Mirage is addressed along the description of MIF. In [18], it is described the security approach and features of the Mirage image library. In [17], it is proposed a novel tool named *Nüwa*, intended to patch and modify dormant VM images in an efficient way. Nüwa was built as a standalone tool integrated on top of Mirage and evaluated on the IBM Research Compute Cloud. In the same way, in [70] it is exploited how to efficiently search dormant VM images at a high semantic level, using the Mirage image library. In such evidences of Mirage usage, we always found IBM Research authors and products to be involved in such publications. This fact and the absence of a project homepage or code repository may well indicate that Mirage is a proprietary closed source project. Furthermore, Mirage stores images in its host's local filesystem only [9], thus being incompatible with all the IaaSs own storage systems (e.g. Nimbus Cumulus). It is also stated in the latest research publication around Mirage [70] that MIF only supports Linux ext2 or ext3 filesystem formats [71] and Windows NTFS. Considering all these facts, Mirage imposes flexibility and broad compatibility constraints, being tied to the IBM requirements. The above outlined research also indicates that it is best suited for VM images introspection and off-line patching use cases than for a broad compatible Cloud Computing IaaS image service.

2.3 Our Approach

Unlike the observed compatibility limitations in Glance, VISOR is not intended to fit in a specific IaaS framework but rather to overreach sharing and interoperability limitations among different IaaSs and their storage systems. Furthermore, it is also not a proprietary service like FGIR and Mirage, but rather an open source project that can be freely customized.

Considering that at least Glance just supports JSON as the data communication format (there is no mention to the FGIR and Mirage data formats), in VISOR we intend to provide compatibility with at least JSON and XML data formats. However, since such compatibilities will not live in the service core but on pluggable middleware which acts upon requests, the modularity will be highly increased, and it can be easily extended with additional formats.

We also aim to tackle the rigid metadata structure by providing a recommended elastic schema, through which users have the ability to provide any number of additional metadata attributes, while also being able to ignore some non useful ones. We also want to go further in compatibility extensions with metadata storage. Therefore, we want VISOR to provide compatibility with heterogeneous database systems, even with database system without the same architectural baseline. Thus, we will support relational SQL and NoSQL databases [72] through an abstraction layer with seamless abstraction of details behind such integration process.

Furthermore, we also want to provide compatibility with more storage systems than those provided by Glance, FGIR and Mirage, including remote online backends (such as Amazon S3) and cloud storage systems (such as Eucalyptus Walrus, Nimbus Cumulus and others). This compatibility will also be provided through an abstraction layer, providing a common API to interact with all currently (and others that show up in the future) supported storage systems.

Another key concern when addressing the VISOR design is to provide a highly distributed service. Thus, instead of incorporating all service functionalities in the same monolithic service component (like in FGIR [15]), we will split them across several independent Web services. In this way, we expect to increment the service isolation, fault tolerance and scalability capabilities.

Chapter 3

Background

In this chapter we will introduce the Cloud Computing paradigm and related concepts that will be assumed along the description of the VISOR image service in Chapter 4.

We will also discuss and justify our options for two key concerns regarding the VISOR design implementation: current Web services and I/O (Input/Output) concurrency architectural options. Since VISOR was developed as a set of Web services, we needed to assess the currently available Web services architectural options. In the same way, since VISOR aims to be a fast and reliable service, we needed to assess how it can handle I/O concurrency in order to maximize throughput and concurrency-proof. Therefore, we have reviewed the literature for both of these two (i.e. Web services and I/O concurrency) architectural styles in order to determine the most suitable options for the VISOR service implementation.

3.1 Cloud Computing

Since the Internet advent in the 1990s, the ubiquitous computing paradigm has faced major shifts towards better computing services, from the early Clusters to Grids and now Clouds. The underlying concept of Cloud Computing was first sighted by John McCarthy, way back in the 1960s, when he said that ”computation may someday be organized as a public utility” [3]. Thus, it is the long-held dream of computing as utility. Although the Cloud Computing term adoption was only in 2006, when Google’s CEO Eric Schmidt used it to describe a business model of services provided through Internet [42]. Since then, the term has seen multiple definitions from different researchers and organizations. The coexistence of such different perspectives seems to be linked to the fact that Cloud Computing is not a new technology, but rather a new paradigm mixing existing mature technologies in an innovative way to fulfil the computing as utility dream.

Armbrust *et al.* from the Berkeley RAD Lab, in the most concise Cloud Computing overview to date (in the best of our knowledge) [73], has defined Cloud Computing as both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services. Another widely accepted definition has come from the U.S. NIST, since the U.S. government is a major consumer of computer services. NIST has defined Cloud Computing as ”a model for enabling ubiquitous, 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” [1].

Cloud Computing is then an abstraction of a pool of resources (e.g. storage, CPU, network, memory and other resources delivered as a service) to address user’s needs, providing hardware and software on demand. It is distinguished by the appearance of virtual and limitless resources, with abstraction of the underlying physical systems’ specifications. Cloud Computing users pay for the services as they go and for what they need, with services being delivered to them through common Internet standards and protocols. Cloud Computing has also appeared to

increase the computing economic efficiency through improvements on the resources utilization rate, while also optimizing their energy consumption. Furthermore, individuals and organizations with innovative ideas for building new services, no longer require to make heavy up-front investments in physical infrastructures and resources on-premises in order to develop and deploy those Internet-delivered services. Cloud Computing has also been mentioned as the future of Internet and the fifth generation of computing after Mainframes, Personal Computers, Client--Server Computing and the World Wide Web [74].

3.1.1 Computing Models

Although Cloud Computing has been emerging as a new shift in computing, it shares some concepts with the early cluster, grid and utility computing approaches [75, 76, 42]. It distinguishes itself from these approaches with features to address the need for flexible computing as utility. Thus, we need to assess both similarities and disparities between these computing approaches in order to better understand the Cloud Computing paradigm.

- **Cluster Computing:** In early times, high-performance computing resources were only accessible for those who could afford highly expensive supercomputers. This has lead to the appearance of Cluster Computing. A cluster is nothing more than a collection of distributed computers linked between themselves through high-performance local networks [76]. They were projected for arranging multiple independent machines working together in intensive computing tasks, which would not be feasible to execute on a single machine. From the user point of view, although they are multiple machines, they act as a single virtual machine.

- **Grid Computing:** Grid Computing has come to provide the ability to combine machines from different domains. Grids can be formed by independent clusters in order to tackle a heavy processing problem and can be quickly dismantled. Buyya *et al*. [77] defines a grid as a specific kind of parallel and distributed system enabling the dynamical sharing and combining of geographically distributed resources, depending on their availability, capacity and users requirements. It aims to build a virtual supercomputer, using spare compute resources.

- **Utility Computing:** A concept embedded on the Cloud Computing paradigm is the Utility Computing. It has surged to define the pay-per-use model applied to computing resources provided on demand [75, 42], which is one of the founding characteristics of Cloud Computing. Examples of common utility service in our daily life are water, electricity, gas and others, where one uses the amount of resources he wants for the time he wants, paying to that service providers based on services usage.

- **Cloud Computing:** Considering Cloud Computing, Buyya *et al*. [10] has described it as a specific type of distributed system of a collection of interconnected virtualized machines. Contrariwise to grids, Cloud Computing resources are dynamically provisioned on demand, forming a customized collection based on a service-level agreement and accessible through Web service technologies. Thus, Cloud Computing arranges to overreach the characteristics of its computing model predecessors in order to provide computing as utility.

3.1.2 Advantages

According to the NIST definition of Cloud Computing [1], this computing model has five vital characteristics: on-demand self-service, broad network access, resource pooling, rapid elasticity and measured service. These cloud features become some of its biggest advantages and represent the shift from previously introduced computing models:

- **On-Demand Self-Service:** Users are able to manage and provision resources (e.g. compute, storage) as needed and on demand, without service providers personnel interference. Thus users become independent and can manage resources by their own.

- **Broad Network Access:** Access to cloud resources is conducted over the network using standard Internet protocols, providing an access mechanism independent of client's platforms, device types (e.g. smartphones, laptops and others) and user's location.

- **Resource Pooling:** Cloud providers instantiate resources that are pooled in order to serve consumers, with resources being dynamically allocated as the required demand. Resource pooling acts as an abstraction of the physical resources location. Processing, memory, storage and network bandwidth are resources examples.

- **Rapid Elasticity:** Resources can be quickly and elastically provisioned, with the system scaling to more powerful computers or scaling across a higher number of computers. From the consumer point of view, resources seem nearly infinite, being available to scale according to demand.

- **Measured Service:** A metering system is used in order to monitor, measure and report the use of cloud resources, achieving usage transparency and control over service costs. A client is charged only on what it uses, based on metrics such as the amount of used storage space, number of transactions, bandwidth consumed and compute nodes uptime.

The above described set of features represent the vital Cloud Computing characteristics, thus, a computing model needs to respect these features in order to be considered a cloud service. Besides such features (which can be seen as cloud advantages), one should also be aware of some other cloud advantages, as reliability, ease of use and lower costs:

- **Reliability:** The scale of Cloud Computing networks and their load balancing and failover mechanisms makes them highly reliable. Using multiple resource locations (i.e. sites) can also improve disaster recovery and data availability [3].

- **Easy Management:** Cloud Computing lets one concentrate in its resources management, having someone else (i.e. cloud service provider's IT staff) managing the underlying physical infrastructure. Furthermore, since cloud services are exposed through Internet, the required user skills to manage them are decreased.

- **Lower Costs:** Cloud Computing can reduce IT costs, which can become a drastic reduction for small to medium enterprises [3]. Using resources on demand will make it possible to eliminate up-front commitment, the need for planning ahead and purchasing resources that will only be required in future at some point. It also guarantees that whenever the demand decreases, costs with underused resources can be avoided by shrinking resources.

3.1.3 Challenges

Cloud Computing has arrived with myriad advantages, but it incurs in specific limitations. Although such limitations or disadvantages are not structural but rather challenges that remain unsolved by now. Several studies in literature have analysed such challenges. Armbrust *et al.* [73] has identified the top ten obstacles for Cloud Computing: availability, data lock-in, confidentiality and auditability, transfer bottlenecks, performance unpredictability, scalable storage, bugs in large-scale distributed systems, scaling quickly, reputation fate sharing and software licensing. Zhang *et al.* [42] also presents a survey which identifies several design and research challenges along the cloud roadmap. We will address some of these challenges here:

- **Security:** Security has been cited as the major roadblock for massive Cloud Computing uptake. Nevertheless, most security vulnerabilities are not about the cloud itself but about its building technologies, being intrinsic to the technologies or prevalent in their state-of-the-art implementations [78]. Examples of such vulnerabilities are obsolete cryptography mechanisms and VMs isolation failures.

- **Availability:** When someone deposits valuable services and data on the cloud, their availability becomes a major concern. However clouds load balancing and failover mechanisms make them highly reliable, since data and services are accessible over the Internet, the multiple components on the connection chain increase the risk of availability interruption.

- **Vendor Lock-In:** Due to proprietary APIs and lack of standardization across multiple cloud providers, the migration from one cloud to another is a hard task. In fact, vendor lock-in is one of the biggest concerns for organizations considering the cloud adoption [73]. The effort and investment made on developing applications for a specific cloud makes it even harder to migrate them if relying on specific cloud development tools [79].

- **Confidentiality:** Confidentiality is related to both data storage and management [4], as there is the need to transfer data to the cloud and the data owner needs to rely on the cloud provider to ensure that will not happen any unauthorized access. Furthermore, the majority of clouds are based on public networks, which expose them to more attacks [73].

3.1.4 Service Models

Cloud Computing can be classified by the provided service type. Basically, in cloud anything is provided as a service, thus the Everything-as-a-Service or X-as-a-Service taxonomy is commonly used to describe cloud services. Although many definitions can surge, as Database-as-a-Service, Network-as-a-Service and others, there are three main types of services universally accepted [12, 4, 74, 1], which constitute the founding layers of a cloud: Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS).

On Figure 3.1 it is pictured the arrangement of the three Cloud Computing service models. Over the physical resources is were it is placed the IaaS. PaaS is placed on top of IaaS in order to rely on its services to provide the founding features for the upper SaaS. Despite this hierarchical separation, components and features of one layer can be considered in another layer, which may not necessarily be the immediately upper or lower one [79]. For example, storage is also present on PaaS, and a SaaS can be built directly on top of a IaaS instead of a PaaS.

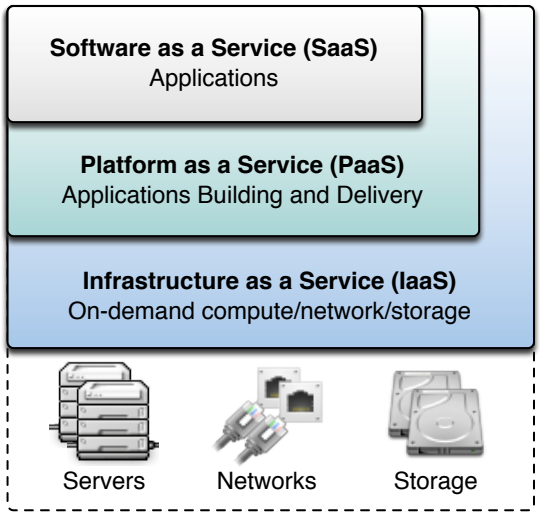

Figure 3.1: Cloud Computing service models and underpinning resources.

3.1.4.1 Infrastructure-as-a-Service (IaaS)

The IaaS offers computing resources on demand, such as networks, storage and servers through virtualization technologies, achieving a full virtualized infrastructure in which users can install and deploy software (i.e. operating systems and applications). Thus, it is the lowest cloud layer, which directly manages the underlying physical resources. Commercial offers of public IaaS include Amazon EC2 [38], Rackspace Cloud Servers [80], Terremark [81], GoGird [82] and the newcomers HP Cloud [83] and Lunacloud [23]. Open source implementations for building IaaS clouds include Eucalyptus [44], OpenStack [54], Nimbus [49], OpenNebula [13, 14] and the newcomer CloudStack [84].

3.1.4.2 Platform-as-a-Service (PaaS)

Sitting on top of IaaS is the PaaS, providing application building and delivery services. More precisely, a PaaS provides a platform with a set of services to assist application development, testing, depoyment, monitoring, hosting and scaling. Each PaaS supports applications built with different programming languages, libraries and tools (e.g. databases, web servers and others). Through a PaaS an user has no control on the underlying IaaS, since he can only control the PaaS and deployed applications. Commercial offers of PaaS include SalesForce Force.com [85] and Heroku [86], Google App Engine [87] and Microsoft Windows Azure [88]. Open-source offers include VMware Cloud Foundry [89] and the newcomer Red Hat OpenShift [90].

3.1.4.3 Software-as-a-Service (SaaS)

Being powered by the PaaS layer, SaaS is the upper layer of Cloud Computing. It intends to provide software over the Internet, eliminating the need to install and run applications on user's own systems. On a SaaS the consumer cannot manage the underlying infrastructure (i.e. the IaaS) neither the platform (i.e. the PaaS). It can only use the exposed applications and manage some user's specific settings. Today many users are already using multiple services built around the SaaS model without even knowing it. Examples of SaaS are all over the Web, including applications like Dropbox [91] (which relies on Amazon to store data), Google Apps [92] (e.g. Gmail, Google Docs, etc.) and many more.

3.1.5 Deployment Models

Following the NIST definition of Cloud Computing [1], a deployment model defines the purpose, nature, accessibility and the location where a cloud resides. A cloud can comprise single or multiple clouds, thus forming a single-cloud or a multi-cloud environment.

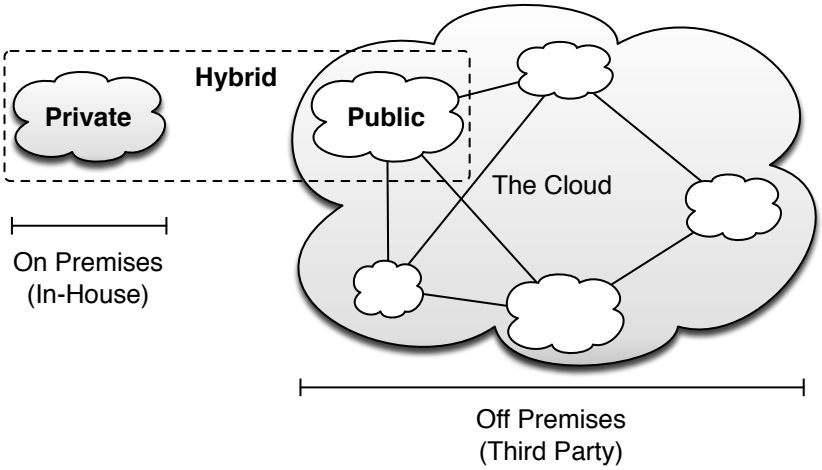

Figure 3.2: Cloud Computing deployment models.

The cloud deployment models are pictured in Figure 3.2. These models define whether a cloud is public, private, hybrid or communitarian (although some authors [4, 74, 30] ignore the community model). We will address and describe each one of the deployment models.

3.1.5.1 Public Cloud

In a public cloud, the infrastructure is owned and managed by the service provider. Such clouds are exposed abroad for open use by generic consumers via Internet, and exist on premises of cloud providers. Users only need to pay for using the cloud. This model raises concerns from client's side, as they lack of fine-grained control over data, network and security [3, 42].

3.1.5.2 Private Cloud

Private clouds refer to internal data centers of a company or organization. Thus it is not exposed abroad to public, and resides on the premises of the cloud owner. However it can be managed by the owner organization, a third party or a combination of both [1]. Security is improved since only the company or organization users have access to the cloud. Sometimes private clouds are criticized and compared to standard server farms as they do not avoid the up-front capital investment [42].

3.1.5.3 Community Cloud

A cloud is considered to follow the community model when it was designed and deployed to address the needs or requirements of one or many jointly organizations. As in private clouds, a community cloud can be managed by one or more involved organizations, a third party or a combination of both. Although a community cloud may exist on or off premises.

3.1.5.4 Hybrid Cloud

A hybrid cloud is achieved from the combination of two or more public, private or community clouds. A hybrid cloud can behave as a single entity if involved standards are common to all constituent clouds. It lets an organization to serve its needs in the private cloud and if needed it can use the public cloud too (e.g. for load balancing between clouds) but requires careful planning about the split of public and private components [42].

3.1.6 Enabling Technologies

As already stated, Cloud Computing does not represent a technological shift but rather a revolutionary computing model. Thus, Cloud Computing is enabled by many already existing and long employed technologies. Its main enabling technologies include virtualization, Web applications, Web services and storage systems [93, 94, 78]. We will conduct an overview of each one of these technologies in the next sections.

3.1.6.1 Virtualization

Facing the Cloud Computing paradigm, resources are pooled and presented as virtual resources, which are abstracted from physical resources such as processors, memory, disk and network. The art of such abstraction is delivered by virtualization technologies. Through virtualization, physical resources are mapped to logical names which point to those resources when needed. Thus, virtualization enables a more efficient and flexible manipulation of resources with multitudinous benefits such as flexibility, isolation and high resources utilization rate [26].

In Cloud Computing, virtualization is applied to partitioning server's resources into a set of VMs presented as compute nodes, providing transparent access to resources (e.g. providing a public IP address for an instantiated VM), and offering abstraction for data storage across several distributed devices. Virtualization can be seen as the engine of Cloud Computing, providing its founding resources (i.e. VMs). In fact, according to Bittman [5], virtualization is the enabling technology of the service-based, scalable and elastic, shared services, metered usage and the Internet delivery characteristics of Cloud Computing.

In virtualization, the hypervisor (also known as VMM) is the tool which partitions resources, allowing a single physical machine to host multiple VMs. The hypervisor is the one controlling the guest VMs accesses to the host physical resources. Commercial hypervisors include VMware ESX and Microsoft Hyper-V. Open source hypervisors include Virtuozzo OpenVZ, Oracle VirtualBox, Citrix Xen [95] and Red Hat KVM [32] [26].

3.1.6.2 Virtual Machine Images

The virtualization mechanism would not be possible without VM images. They are the component used to provide systems portability, instantiation and provisioning in the cloud. An image is represented by a single container, such as a file, which contains the state of an operating system. The process of image creation is commonly known as machine imaging. In Cloud Computing an image is generated in order to deploy compute instances (i.e. VMs) based on it. It is also common to restore an instance from a previously taken snapshot of it (which is stored on a new image file) or to clone a deployed instance. It is also common to see previously configured images with specific installed tools and configurations for specific purposes (e.g. Web development or others). These special images are called virtual appliances.

Images can have different formats, depending on the provider or target hypervisor. Common image formats are the ISO 9660 [96] (".iso"), Virtual Disk Image (VDI) from Oracle VirtualBox, Virtual Hard Disk (VHD) from Microsoft and Virtual Machine Disk Format (VMDK) from VMware [97]. In some cases it is common to find three main kinds of images: machine, kernel and ramdisk images. Each one of these image types fulfils a specific system state storing need:

- **Kernel Image:** A kernel image contains the compiled kernel executable loaded on boot by the boot loader. A kernel constitutes the central core of an operating system. It is the first thing loaded into volatile memory (i.e. RAM) when a virtual instance is booted up.
- **Ramdisk Image:** A ramdisk image contains a scheme for loading a temporary filesystem into memory during the system kernel loading. Thus it is used before the real root filesystem can be mounted. A kernel finds the root filesystem through initial driver modules contained on a ramdisk image.
- **Machine Image:** A machine image is a system image file that contains an operating system, all necessary drivers and optionally applications too. Thus, it contains all the post-boot system data. Kernel and ramdisk images can be specified when deploying an instance based on a machine image.

An example of system state split across these types of images can be found in the Amazon EC2 cloud [38]. In EC2 instances are created based on an AMI (Amazon Machine Image). There are AMIs for multiple operating systems, such as Red Hat Linux, Ubuntu and Windows. Each AMI can has an associated AKI (Amazon Kernel Image) and ARI (Amazon Ramdisk Image) images.

3.1.6.3 Web Applications and Services

SaaS, PaaS and IaaS would not be possible without Web applications and Web services technologies, since Cloud Computing services are normally exposed as Web services and typically have a related Web application too [93]. In fact SaaS applications are implemented as Web applications (e.g. Dropbox, Gmail), so all the service functionalities are exposed through them. For PaaS, instead of being built as Web applications, they provide the development environments to build them. Furthermore, typically, a PaaS may expose a Web application as the platform control panel, and Web service APIs to provide the ability to interact with the platform remotely. IaaS would also not be possible without Web applications and services, since IaaS are exposed to customers through Web applications and typically also expose Web service APIs.

3.1.6.4 Storage Systems

In Cloud Computing, one of the key components of all service models (i.e. SaaS, PaaS and IaaS) are the storage systems. In fact, as data continues to grow, storage is a key concern for any computing task. In 2011, IDC Consulting in partnership with the EMC Company have announced a study where they have estimated a total of 1.8 Zettabytes of data on earth in that year, with the same study predicting this amount to grow to 35 Zettabytes till 2020 [98]. It is also predicted that a very significant fraction of this data is or will be in the cloud.

Cloud storage is like regular storage but with on demand space provisioning and accessed through Internet using a Web browser or a Web service. It provides Internet accessible data storage services on top of the underlying storage systems. When interacting with cloud storage, users have the sense that data is stored somewhere in a specific place. However, in cloud data

has no specific place and can be stored anywhere in one of the cloud enabling servers. It may even move from machines frequently as the cloud manages its storage space [99]. Cloud storage presents several advantages when compared to local storage, such as the ability to access data from everywhere with just an Internet connection and the ability to share data with others.

Although many cloud storage systems do not share a common well defined and standardized set of characteristics [99], they may fall in one of five main categories: unstructured, structured, persistent blocks, and database storages [100, 101]. Some authors [100] include message queues in the cloud storage types too. Although we will not mention them since they are not really storage systems but rather integration services supporting communication and messages exchange between processes and applications in the cloud [101].

Unstructured Storage: Unstructured storage is based on the concept of simply putting data in the cloud and retrieving it latter, supporting only basic operations (i.e. read/write). Typically, in these storages, files can be up to 1TB in size [100]. These systems have some limitations when compared to regular file systems, such as the absence of support for nested directories.

An example of such storage system is the Amazon S3 [41]. S3 stores objects (i.e. files) in buckets (like folders) residing in a flat namespace and exposes its functionalities through a Web service. Examples of other unstructured storages are Microsoft Windows Azure Blob [88] and Google BlobStore [87]. Many cloud IaaSs have their own unstructured storage system, with some of them being inspired in the S3 API [102], such as Eucalyptus Walrus [45], Nimbus Cumulus [50], OpenStack Swift [63] and Lunacloud Storage (LCS) [23].

Structured Storage: Structured storage has been introduced in cloud providers to circumvent the fact that most relational databases do not scale much [100]. These systems are commonly known as NoSQL Key-Value stores [72] as objects are described by ket/value pairs. They implement a non-relational data type providing schema less aggregation of objects. These storages are optimized to scale and provide fast data look up and access. Contrariwise to that observed on common relational database systems, structured storage systems do not fully comply to the SQL standard, as they do not support joins and queries for performance reasons [43]. They are optimized to scale and be much more flexible than any relational database systems. Examples of such storage systems are Amazon SimpleDB [103], Microsoft Windows Azure Table [88] and Google AppEngine DataStore [87].

Persistent Block Storage: Persistent block storages act similar to traditional file systems and provide a storage service at block or file level to users [100]. VMs started in the cloud typically rely on volatile local storage (i.e. physical server/rack storage drives), in which data is only kept during instances uptime [43]. Therefore, block storage is commonly used as network attached storage devices for deployed compute instances in order to persist their data. Examples of such storage systems that can be attached to compute instances are Amazon EBS [40] and Microsoft Windows Azure Drive [88].

Although not commonly provided as a service to users, many other file systems have been developed to operate on cloud computing environments [101]. Examples of such systems are the Google GFS [53] and the Hadoop HDFS [51, 104]. They are fault tolerant and replicated file systems with automated recovery, meant to process huge amounts of data. GFS is the proprietary file system powering the Google infrastructure and HDFS is an open source storage inspired by GFS and is heavily used by companies such as Yahoo!.

Database Storage: These storages include relational and NoSQL document-oriented databases. Many popular relational databases are supported by cloud providers, such as Microsoft SQL Server and MySQL [105]. They are deployed and provided as distributed databases. Examples of such services are the Amazon Relational Database Service (RDS) [106], which can be powered by MySQL, Oracle or Microsoft SQL Server databases and Microsoft Windows Azure SQL [88], which is based on the SQL Server database system. There are also the NoSQL document databases, such as MongoDB [64, 65], CouchDB [107] and Cassandra [108], in which objects are described with key/value pairs attributes stored in documents.

3.2 Web Services

Among several other definitions, a Web service has been defined by W3C as a software system designed to support interoperable machine-to-machine interaction over the network [109]. Thus, Web services are the key of integration and interoperability for applications built around heterogeneous platforms, languages and systems. They provide a service independent of the underlying system particularities. Following the Service Oriented Architecture (SOA) [110], service providers publish the interfaces of the services they offer in order to expose them abroad. SOA is then a paradigm for managing distributed computational resources, the so-called services.

There are several technologies to build Web services. In [111], Adamczyk *et al*. advocates that these technologies can be grouped across five categories: REST, WS-*, XML-RPC, AJAX and others. The others category embraces mainly RSS (Really Simple Syndication) feeds, Atom and XMPP (Extensible Messaging and Presence Protocol). These are the simplest services, used for reading and writing data on the Internet. AJAX (Asynchronous Javascript and XML) is used for interactive services (e.g. satellite maps). Another category is represented by XML-RPC (Extensible Markup Language-Remote Procedure Call), which was the first attempt to encode RPC calls in XML. However, the two main architectures for developing web services are the WS-* standards approach, mainly known as SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). With the Web 2.0 advent, many Web applications are opening their data, acting as services. Therefore, it is needed to provide APIs in order to make Web applications accessible over the Web. While REST services are gaining momentum as a lightweight approach for building services on the Web, SOAP already has a large enterprise background [112]. According to the Programmableweb.com portal, nowadays 68% of APIs available on the Web follow the REST architecture, while 19% are SOAP services, with the remaining 13% falling in the other web services category.

Considering that VISOR is implemented as a collection of Web services (as will be described further in Chapter 4) and based on our research work, it is necessary to argue the correct choice between SOAP and REST. A similar overview of both REST and SOAP approaches regarding their advantages and limitations will be outlined. However, since our research has indicated REST as the most suitable option for VISOR, its description will be expanded to provide the needed background to better understand the VISOR service implementation.

3.2.1 SOAP

SOAP is a standard protocol proposed by W3C [35] to create Web services and has appeared to extend the older XML-RPC protocol. A Web service created with SOAP sends and receives messages using XML and is based on a messaging protocol, a XML-based language and a platform-

independent framework. The message protocol is used for data exchange between interacting applications/processes and is called SOAP. A XML-based language is used to describe the service interface functionalities and is called Web Services Description Language (WSDL). A WSDL document contains a XML description of the interface exposed by a service, detailing the available methods, their parameters, types of data and the type of responses that the Web service may return. Lastly, the platform-independent framework is used for registering, locating and describing Web service applications. This framework is called Universal Description Discovery and Integration (UDDI).

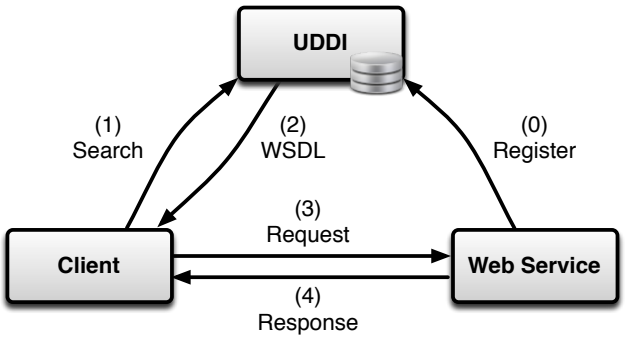

Figure 3.3: Dynamics of a SOAP Web service.

A SOAP service provides an endpoint exposing a set of operations on entities. Operations are described in a WSDL document, while semantics is detailed in additional documents. Thus clients can understand and know how to design a client accordingly to the service assumptions. In Figure 3.3 it is pictured the dynamics of a simple SOAP Web service. First a Web service must register itself on UDDI (0) in order to be exposed abroad. For a new request, the client searches for the wanted service in UDDI (1), receiving a WSDL document containing the service description (2). After that, the client generates a SOAP/XML request and sends it to the Uniform Resource Locator (URL) specified in the WSDL document (3). Finally, the service processes the request and returns the response back to client (4). During requests, client-server interaction state is kept by the server, thus SOAP services are *stateful*, a characteristic which impacts negatively the scalability and the complexity of the service [112].

3.2.1.1 Advantages

SOAP services are known to be a good fit for high quality of service, reliability and security requirements [113]. They have long been employed in enterprise grade applications and present a set of advantages when facing other Web service architectures:

- **Formal Contracts:** In case that both provider and consumer have to agree on the exchange format, SOAP provides rigid type checking specifications for this type of interaction, adhering to contracts accepted by the two sides (i.e. provider/consumer).
- **Built-in Error Handling:** When facing an exception, a SOAP service can rely on built-in error handling features. Whenever an error is faced, a SOAP fault message is generated and sent to clients, containing the fault code and optionally the fault actor and details.
- **Platform and Transport Independent:** SOAP web services can be used and deployed in any platform. Another of its biggest advantages is the capability to rely on any type of transport protocols (e.g. HTTP, SMTP and others).

- **Security:** Security and reliable messaging are key SOAP advantages. SOAP services may rely on the WS-Security framework [114], an application layer (end-to-end) security mechanism. WS-Security offers measures for authentication, integrity, confidentiality and non-repudiation, from message creation till its consumption.

3.2.1.2 Disadvantages

Although being a widely accepted Web services standard, SOAP incurs in specific constraints representing its disadvantages when compared to other technologies, which we outline here:

- **Interface:** Data and useful information are encapsulated inside the service, thus some of the information can not be directly accessed, affecting the system connectedness. Also all operations rely on the HTTP POST method only [115].
- **Complexity:** The serialization and de-serialization of the native program language and SOAP messages are complex tasks, with time consuming concerns. Also, the SOAP tightness to XML is a disadvantage because of the XML verbosity issues and the time needed to parse messages formatted with it [113].
- **Interoperability:** In SOAP an interface is unique to a specific service. Thus, clients need to adapt to different interfaces when dealing with different services, facing hard to manage WSDL documents changes.
- **Performance:** The network communication volume and server-side payload is greatly increased by redundant SOAP and WSDL information encapsulated inside the service [115], which also requires more time-consuming operations.

3.2.2 REST

REST (Representational State Transfer) was first introduced by Roy Fielding, one of the main authors of the HTTP specification versions 1.0 and 1.1 [37], in his doctoral dissertation [36]. REST is an architectural style for distributed hypermedia systems [116], such as the World Wide Web. Web services implemented following the REST architecture are commonly named RESTful Web services. REST aims to provide the capability to model resources as entities, providing a way to create, read, update and delete them. These operations are also known as CRUD (Create, Read, Update and Delete).

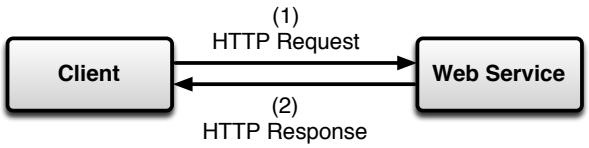

Figure 3.4: Dynamics of a REST Web service.

The REST architecture, as pictured in Figure 3.4, consists of a simple client-server communication, with clients making requests to servers (1) and servers responding by acting upon each request and returning appropriate responses to clients (2). In REST there are the principles of *resources*, manipulation of resources through their *representations*, self-descriptive *messages* and the hypermedia as the engine of application state (abbreviated as HATEOAS). A *resource* can be any coherent and meaningful concept that may be addressed [117] (e.g. users of a social network). In response to some request for a given resource, a RESTful service returns a

representation of the resource, which is some data about the current state of a resource (e.g. current user account data), encoded in a specific data format, as XML, JSON or others. These resources are then manipulated through *messages*, which are the HTTP methods. Restful services are implemented using the HTTP protocol and based on the REST architecture principles, exposing objects (i.e. resources) that can respond to one or more of the HTTP GET, HEAD, POST, PUT, DELETE and OPTIONS methods. Some of these HTTP methods correspond to the CRUD operations, as matched in Table 3.1.

Table 3.1: Matching between CRUD operations and RESTful Web services HTTP methods.

CRUD Operation	HTTP Method
Create	POST
Read	GET
Update	PUT
Delete	DELETE

Finally, the HATEOAS principle implies that the state of any client-server communication is kept in the hypermedia that they exchange (i.e. links to related resources), without passing that information in each message, keeping both clients and server stateless [111].

3.2.2.1 Resource-Oriented Architecture

In order to provide guidelines to implement the REST architecture, Richardson and Ruby in [118] have documented the Resource-Oriented Architecture (ROA). ROA is a set of guidelines based on the concepts of resources, representations, Uniform Resource Identifiers (URIs), which are the name and the address of a resource, and links between resources. Considering ROA, there are four vital properties and guidelines for a RESTful Web service:

1. **Addressability:** Services should expose a URI for every piece of information that they want to serve. Resources are addressed with standard URIs, and can be accessed by a standard HTTP interface. Therefore they do not require any specific discovery framework like the UDDI used by SOAP services.

2. **Statelessness:** Every request is completely isolated and independent from others, so there is no need to keep states between them in the server. This is possible through the inclusion of all necessary information in each request.

3. **Connectedness:** Clients only need to know the root URI or a few well formed URIs in order to be able to discover other resources. Resources are related with each other by links, so its easy to follow them. Also, as the Web does not support pointers [119], URIs are the way to connect and associate resources on the Web.

4. **Uniform Interface:** The HTTP protocol is the service uniform interface. Given the URI of a resource, the HTTP methods can be used to manipulate it. The GET method is used to retrieve a representation of a resource, PUT or POST methods to a new URI to create a new resource, PUT to an existing URI to modify a resource and DELETE to remove an existing resource. Furthermore, they do not require any serializing and de-serializing mechanism.

3.2.2.2 Resources and Representations

Recalling the already introduced REST principles, resources are identified by an URI. Through HTTP methods, the representation of resources are returned to clients, or clients modify existing resources or add new resources, depending on the used method. Thus, different operations can be issued to the same resource URI, only switching between the HTTP methods.

Table 3.2: Examples of valid URIs for a RESTful Web service managing user accounts.

URI	Description
servicedomain.com/users	The users resources URI.
servicedomain.com/users/john	The user John URI (structural form).
servicedomain.com/users?name=john	The user John URI (query string form).

A resource URI should be formed structurally or with query strings. We demonstrate an example of both approaches for an example Web service which manages user accounts in Table 3.2. Therefore, a valid URI should be of the form `<service address>/<resource>/<identifier>` or `<service address>/<resource>?<query>`. In possession of a resource URI, one can issue HTTP methods to it, in order to manage that resource data (i.e. representation).

If a resource supports multiple representations, it is possible to generate responses in different data formats. Therefore, in HTTP a client can specify the preferred format in the request *Accept* header, so the service is able to identify the wanted response data format. Since RESTful Web services comply with the HTTP protocol, they support multiple response formats as the Web does (e.g. JSON, XML, octet-stream and others).

Listing 3.1: Sample GET request in JSON. The *Accept* header was set to *application/json*.

```
GET /users/john
HTTP/1.1 200 OK
Content-Type: application/json
{
  "user": {
      "name":"John",
      "email":"foo@bar.com",
      "location":"Portugal"
    }
}
```

Listing 3.2: Sample GET request in XML. The *Accept* header was set to *application/xml*.

```
GET /users/john
HTTP/1.1 200 OK
Content-Type: application/xml

<?xml version="1.0"?>
  <user>
    <name>John</name>
    <email>foo@bar.com</from>
    <location>Portugal</location>
  </user>
```

Considering the example of a RESTful Web service to manage user accounts, following the URI samples in Table 3.2, lets consider that this service is internally designed to support and serve the resources representations in both JSON and XML data formats. In Listings 3.1 and 3.2, it is shown a sample response in both JSON and XML data formats respectively. In these examples, a GET request was issued to the URI */users/john*. For simplicity purposes we have hide the service address part. If a client adds the *Accept* header set to *application/json* to the HTTP request headers, the service would return a representation of the John user account encoded as a JSON document (Listing 3.1). If the request header is set to *application/xml*, the service returns it encoded in a XML document instead (Listing 3.2).

3.2.2.3 Error Handling

Initially, many RESTful Web services have copied the error handling strategy from SOAP [111]. Thus, instead of using HTTP status codes [37] (e.g. ”404 Not Found”) in order to describe the status of a response, they always returned ”200 OK”, only describing the real status somewhere in the response body. Furthermore, there were also services using their own custom status codes, which has forced users to develop specific clients to them, thus strangulating clients interoperability. Although, nowadays most of RESTful Web services do use HTTP status codes [111], adding only additional status codes when the error situations are not comprised in the HTTP status code set. Thus they behave like regular Web applications.

Listing 3.3: Sample RESTful Web service error response for a not found resource.

```
GET /users/fake
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "code": 404,
  "message": "User fake was not found."
}
```

When a request is successful processed, a 200 status code is sent with the response (as in Listings 3.1 and 3.2). Whenever an issue is faced, a RESTful service should generate and return the error code and a message describing it through the response, as in the Listing 3.3 example.

3.2.2.4 Advantages

Besides the addressability, connectedness and uniform interface REST advantages already introduced in the ROA guidelines, there are other advantages around RESTful Web services:

- **Simplicity and Abstraction:** Clients are not concerned with the server internals, thus clients portability is improved. Servers are also not concerned with the user interface, thus servers can be simpler. Moreover, both servers and clients can be replaced and developed independently, as long as the service interface remains unchanged.
- **Standards:** REST leverage in well-funded standards like HTTP and URI, so there is no need to use other specific standards. This also provides the ability to rely on existing libraries and make services useful within the World Wide Web context. As stated by Vinoski in [120], ”the fact that the Web works as well as it does is proof of these constraints effectiveness”.
- **Data Format:** REST provides the flexibility to use more than one data format. This is possible because there exists a one-to-many relationship between resources and their representation [119], so they are independent. In this way, we can leverage in the HTTP content negotiation to define data format. The same is not true for traditional Web services, which rely on a standard format.
- **Performance:** As REST does not requires any time consuming mechanism such as serialization, the server payload and communication size are smaller. Also, as it is stateless, it can easily support conditional GET operations and provide seamless support to data compression and caching, greatly reducing the resources access time.

3.2.2.5 Disadvantages

Besides all the advantages and hype around RESTful Web services, they also suffer from some disadvantages, as any other existing approach. We will now describe them in detail:

- **Not an Universal Solution:** REST does not fits all service models by default. Although it fits most of them, it is not tailored for services which are required to be stateful, support rigid data models, asynchronous operations or transactions [115].
- **Tied to HTTP:** Instead of being transport agnostic like SOAP, RESTful Web services need to rely on the HTTP protocol. Although in most cases this should be an advantage, in others it could be a limitation.
- **Lack of Embedded Security and Reliability:** In REST there is no built-in mechanisms for security and reliability. With the lack of a standard method to handle messaging reliability, it becomes difficult to manage operations that need, for example, to be tracked in the amount of times they are performed.

3.2.3 Applications and Conclusions

In the Web services early adoption, the term ”Web service” was quickly and exclusively perceived as a SOAP-based service [121]. Later, REST has come to provide a way to eliminate the complexity of WS-* technologies, showing how to rely on the World Wide Web to build solid services. REST is considered to be less complex, require fewer skills and having lower entry costs [111]. Although SOAP is the traditional standards-based (i.e. WS-*) approach, nowadays REST services dominate among publicized APIs, like the ones from Twitter, Facebook, Yahoo! and others, with some of them offering both REST and SOAP interfaces, like Amazon [113].

In fact, SOAP and REST philosophies are very different. While SOAP is a protocol for distributed computing, REST adheres to the Web-based design [113]. Some authors, like Pautasso *et al.* in [122], recommend using REST for ad-hoc integration and SOAP/WS-* for enterprise-level applications, where reliability and transactions are mandatory. However, in a similar evaluation, Richardson and Ruby in [118] show how to implement transactions and messages reliability and security mechanisms in REST, relying on HTTP. Thus, it shows that REST disadvantages can be tackled efficiently, even when considering enterprise-level applications. For security requirements, while SOAP has the WS-Security framework [114], RESTful services can rely on common long and successfully employed HTTP security mechanisms. For example, it is possible to rely on HTTPS [123] to address confidentiality and in HTTP basic and digest-based authentication [124] or service key-pairs to address authentication. Furthermore, it is also possible to address message-level security through cryptography and timestamps to improve confidentiality and prevent replay attacks, as Amazon does in its S3 API [102]. Also, some performance tests [115] have shown REST as a good solution for high availability and scalability, handling many requests with minimal degradation of response times when compared to SOAP.

By relying on the Web technology stack [125], REST provides a way to build services which can easily scale and adapt to demands. Furthermore, by relying on hypermedia, it is possible to achieve significant benefits in terms of loose coupling, self-description, scalability and maintainability [119]. In fact, as stated by Vinoski [126], it is possible to say that only the RESTful web services are really ”made of the Web”, when others, based on classic architectures are just ”made on the Web”. Together, all these facts make REST the most suitable Web service architectural model for our VISOR service implementation.

3.3 I/O Concurrency

Building highly concurrent services is inherently difficult, due to concerns on how to structure applications in order to achieve high throughput. It is also required to guarantee a peak throughput when demand exceeds a certain level, with high number of I/O requests placing large demands on the underlying resources. Applications that need to scale and handle multiple concurrent requests can rely on I/O concurrency provided by two common models: threads and events [127]. Threads allow developers to write their applications code relying on the OS to manage I/O and schedule those threads. Contrariwise, the events approach, commonly known as event-driven, lets developers manage the I/O concurrency by structuring the code as a single threaded program. The program will then react to events, as non-blocking I/O operation results, messages or timers.

Before starting to develop VISOR, we investigated what the most suitable programming architecture for it is, considering its aims and purposes, providing high performance and reliability. In this chapter we present an overview of the multithreading and event-driven programming model approaches, in order to identify their advantages and disadvantages, always considering the VISOR image service requirements.

3.3.1 Threads

Developers needing to deal with I/O concurrency have long employed the multithreading programming architecture. Threads have become popular among developers as they make it possible to spawn multiple concurrent processing activities from a single application. Therefore, they appear to make programs less painful to develop, maintain and scale, without a significant increase of the programming logic. Moreover, threads promised to let applications handling large amounts of I/O, improve the available processing resources usage. Indeed, such resources usage has become even more important when dealing with modern multicore systems, achieving true parallelism between tasks running in different processor cores.

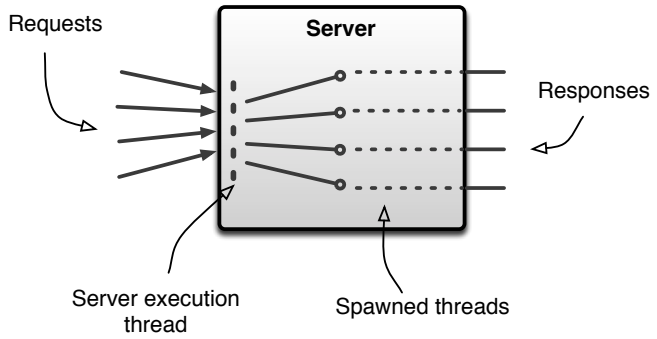

Figure 3.5: Dynamics of a typical multithreading server.

In Figure 3.5 it is pictured the dynamics of a typical multithreading server. The server application runs on a single thread of execution and usually, whenever a request is received, the server will spawn and hand off each request to individual threads. Then, the request handling thread will execute the task associated to that request and will return its result. In this case, one thread per request (or task) is created. The operating system is the one deciding how to schedule the server spawned threads and how long one can run.

3.3.1.1 Advantages

Being a mainstream approach to I/O concurrency, threads have some explicit benefits that can be gathered from its inherent maturity and standardization across multiple platforms:

- **Sequential Programming:** Thread-based programs preserve the common appearance of serial/sequential programming. In this way, they manage to be a comfortable option for programmers, since they do not require any specific program code architecture.
- **Maturity:** Threads have become a dominant approach for concurrency processing, being standardized across the majority of OSs. Indeed, it is well sustained, with high quality tools and supported by the majority of programming languages.
- **Parallelism:** One of the biggest threads strengths is the ability to scale along the number of processors. Running multiple concurrent threads on a modern multicore system becomes a straightforward task, with each core simultaneously executing a different task, achieving true parallelism [128].

3.3.1.2 Disadvantages

Even though heavily used in production applications, multithreading has been far from being recognized by developers as an easy and pain free programming choice [128]. In fact, it is tightly coupled with hard to deal problems, regarding resources sharing for I/O concurrency.

- **Locking Mechanisms:** A thread-based program uses multiple threads in the same single address space. Therefore, it manages to provide I/O concurrency by suspending some thread which is blocking the I/O, and then resuming the execution in another different thread [129]. This procedure requires locking mechanisms to protect the data that is being shared, which is a task of programmer's responsibility.
- **Data Races and Deadlocks:** Threads concurrency implies some synchronization concerns between threads, leading to less robust programs, as they are almost always prone to data race conditions and deadlocks [130]. A data race conflict is when two different threads concurrently access the same memory location in a read/write operation and at least one of them is a write. On the other hand, a deadlock problem represents a blocking state where at least one process cannot continue to execute since it is waiting for the release of some resource being locked by another process, which in turn is also waiting to access another blocked resource.
- **Debugging:** Multithreaded programs can be very hard to debug, specially when facing problems like data races, since a multithread program can exhibit many different behaviours, even when facing the same I/O operations during its execution [131]. As Savage *et al.* has stated in [132], in a multithread program it is easy to make synchronization mistakes but it is very hard to debug them after.
- **Memory Consumption:** In regard to memory consumption, the main concern is the stack size, as most systems use by default one or two memory pages for each thread stack [129].
- **Performance:** For a threaded server, the simultaneous requests number is dictated by the number of available server threads. Thus, with enough long running operations, a server would eventually get out of available threads. Furthermore, the system overhead increasing due to scheduling and memory footprint for a high number of threads would decrease the system overall performance [133].

3.3.2 Events

Event-driven programs must be structured as a state machine that is driven by events, typically related to I/O operations progress. Event-driven programming design focuses on events to which the application should react when they occur. When a server cannot complete an operation immediately because it is waiting for an event, it registers a *callback*, a function which will be called when the event occurs (e.g. I/O operations completion, messages or timers). An event-driven program is typically driven by a loop, which polls for events and executes the appropriate callback when the events occur [129]. This is done by relying on event notification systems and non-blocking/asynchronous primitives, which allow a single execution context to use the full processor capacity in an uninterrupted way. Thus, an event-driven application is responsible for tracking I/O operations status and managing resources in a non-blocking manner.

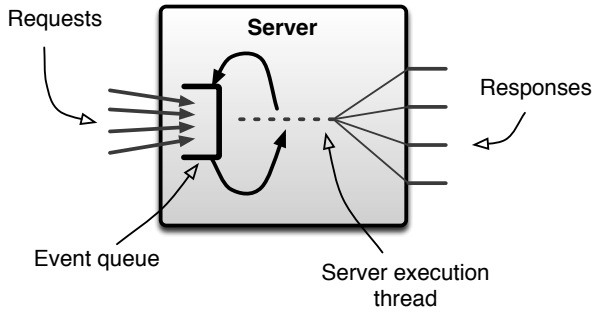

Figure 3.6: Dynamics of a typical event-driven server.

In figure 3.6 it is pictured the dynamics of an event-driven server. The server uses a single thread of execution and incoming requests are queued. The server is structured as a loop, continuously processing events from the queue, giving to each task a small time to execute on the CPU, switching between them till their completion. With events, a callback function is assigned to each I/O operation, thus the server can proceed with the execution of other tasks and when the I/O operation has finished, the corresponding callback is picked up. For example, if a server makes a set of HTTP requests to an external service, a callback is assigned to each request. Then the server can proceed with the processing in a non-blocking manner, without blocking its thread of execution waiting for the I/O operations to complete. Only when a response is received for a specific request, the corresponding callback is picked up in order to process that response.

3.3.2.1 Advantages

With events, disadvantages identified in the multithreading approach are avoided and become its biggest advantages. In fact, the resource usage and scalability limits of threads implementations has led many programmers to opt for an event-driven approach [133].

- **Locking Mechanisms:** In event-driven programs there is no need for locking mechanism. Resources access requests are queued when those resources are locked by another request.
- **Data Races and Deadlocks:** These problem does not apply to event-based programs since this approach uses a single thread of execution. Problems like resources deadlocking are avoided since the event-based approach implies the queuing of incoming events that cannot be served immediately due to resources usage.

- **Memory Consumption:** This issue is reduced by allocating only the required memory for the callback function responsible for I/O processing. Thus, the memory footprint is considerably lower than when allocating a full thread stack [129].
- **Performance:** Network servers spend a lot of time waiting for I/O operations to complete and event-driven programming takes advantage of this fact. With this in mind, event-driven frameworks use a single event loop that switches between tasks, running I/O operations for each one of them at time. When some I/O operation is finished, the corresponding callback is picked up.

3.3.2.2 Disadvantages

Even though events fill almost all the disadvantages introduced by threads, it is not an immaculate approach, as expected. Thus, there are a few drawbacks commonly pointed to the event-driven architecture, which we outline here:

- **Programming Complexity:** The most audible drawback pointed by the community seems to be its inherent programming complexity, due to specific constraints such as callback functions, non-sequential code arrangement and variables scope [128].
- **Blocking Operations:** As an event-driven server uses a single line of execution, we need to take care to do not hang it on some blocking I/O operation. Therefore, many programming languages libraries are now asynchronous, so they do not block the event loop. However, there is still some absence of specific communication protocols libraries in some languages (e.g. handling filesystem operations).
- **Debugging:** As a single thread is responsible for processing all tasks in disjoint stages, debugging can become more difficult than with threads, as the stack traces do not represent the complete processing flow of an isolated task [133].

3.3.3 Applications and Conclusions

By avoiding common thread problems, event-driven programs provide I/O concurrency with high performance non-blocking/asynchronous operations. Studies that compare multithreading servers against event-driven ones [134, 127, 133] agree that multithreading implies higher overhead when dealing with large number of concurrent requests, due to threads scheduling and context-switching. Although this can be attenuated by limiting the number of concurrent threads, this would mean that a server would be restricting the accepted number of connections. Furthermore, under heavy workloads, multithreading servers require large number of threads to achieve good performance, reducing the time that is available to serve incoming requests. Moreover, as already addressed, common threads resource sharing and memory footprint constraints are greatly attenuated in the events approach. It is also stated that an event-driven server is able to exceed the throughput of a threaded one, but even more important is the fact that the performance does not degrade with increased concurrency [133].

We have already stated that the inherent event-driven programming complexity has been the major drawback pointed to this approach. Trying to tackle such disadvantage, some event-driven frameworks based on the Reactor pattern [135, 136] have been successful in maximizing the events strengths and minimizing its weaknesses. They have become important tools for scale-aware applications, as they provide the ability to develop high-performance, concurrent applications relying on asynchronous I/O. Among all them, Node.js [137] for JavaScript,

Twisted [138] for Python and EventMachine [139] for Ruby seem to be the most popular options. Node.js [128], the most recent of the three frameworks, has been gaining much attention lately, as it is powered by the Google V8 JavaScript engine. However, it is a recent project without much solid and mature documentation. On the other hand, Twisted has been long accepted as a standard for event-driven programming in Python, being used (and sponsored) by Google, among others, to create high performance services. Last but not least, the EventMachine framework has also been widely accepted, being now used in many large projects, powering many enterprise API servers, like PostRank (now acquired by Google). It is also used in cloud applications as the VMWare CloudFoundry PaaS [89]. Together, all these facts make events the most suitable I/O concurrency architectural model for our VISOR service implementation.

Chapter 4

Developed Work

In this chapter we address our development work towards VISOR, a virtual machine images management service for cloud infrastructures. We start by providing an overview of VISOR and some of its underlying concepts which will be assumed along this chapter.

Next to this overview, we describe in detail each one of the VISOR subsystems, being the VISOR Image, Meta, Auth, Common and Web. Each one these subsystems plays a specific role towards the overall service functionalities. Finally we will described the used development tools and libraries. Throughout this chapter we will sometimes mention VISOR as ”the service” or ”the system” as it becomes more suitable.

Besides the detailed theoretical description contained in this chapter, all the service source code is thoroughly documented and available on the project homepage at `http://cvisor.org`. Whenever it becomes needed, we will provide direct links to some of that source documentation. During the VISOR development, we have followed a Test-Driven Development (TDD) [140] approach in order to build a reliable and heavily tested service. Currently, VISOR counts on many unitary and integration tests already written to ensure the service functionalities preservation in further development stages.

4.1 VISOR Overview

The VISOR main purpose is to seamlessly manage VM images across multiple heterogeneous IaaS and their storage systems, maintaining a centralized agnostic repository and delivery service. In this section we will conduct an overview over the service features, concepts and architecture.

4.1.1 Features

The VISOR service was designed considering a set of features, defined based on its aims and purposes. These features were already described by the authors on a previous research publication [22] and we will enumerate them as follows:

- **Open Source:** If anyone wants to contribute, fork and modify the source code, customize or just learn how it works, it can be done freely. The development process is community-driven, everyone is welcome to contribute and make suggestions to improve the system.
- **Multi-Interface:** The service provides more than one interface, exposing its functionalities abroad to a wide range of users, being either end-users, developers, other services or system administrators.
- **Modular:** The service was designed and implemented in a modular way, so all subsystems are isolated and can be easily customized and extended by users and researchers aiming at improving it.

- **Extensible:** The service provides to the community the ability to extend it, or to be used in order to build new tools relying on it. This is done by exposing the multiple service interfaces.

- **Flexible:** It is possible to install the system by requiring minimal setup procedures, relying on its high modularity, while providing the possibility to do it strategically close to the needed resources.

- **Secure:** The service is fairly secure and reliable, with robust authentication and access control mechanisms, while preserving the ability to be integrated with external services.

- **Scalable:** The service was designed from bottom to top with scalability in mind, thus, modularity, flexibility and involved technologies offer the ability to adapt to high load requirements.

- **Multi-Format:** The service provides compatibility with multiple VM image formats, being able to handle and transfer image files regardless their format. This is possible by treating VM images as opaque files.

- **Cross-Infrastructure:** The system provides a multi-infrastructure and unified image management service, capable of sitting in the middle of multiple heterogeneous IaaS. This is possible given the system openness and multi-format and multi-storage features.

- **Multi-Storage:** It provides compatibility with multiple cloud storage systems, by relying on an abstraction layer with a seamless common API over multiple storage systems plugins.

4.1.2 Introductory Concepts

Along this chapter, while conducting the description of VISOR, there will be some omnipresent concepts common to all the service subsystems. These concepts characterize the VM images managed with VISOR and can be grouped on image entities, permissions and status concepts.

4.1.2.1 Image Entities

A VM image registered in VISOR is represented by two entities, the image file and its metadata (information about that image). These are the image entities managed by the service:

- **Image File:** The most important object managed by the service is the VM image, which is a compressed file in some format, as the ISO 9660 format (i.e. iso) for example, used to bootstrap VMs in cloud IaaS. VISOR is responsible for the seamless management and transfer of VM image files between heterogeneous storage systems, client machines and vice versa.

- **Image Metadata:** In order to maintain a centralized image repository and catalogue, it is needed to describe the stored VM images. This is done by registering image metadata, which is a set of attributes describing a certain image file (i.e. its name, version and others). VISOR is responsible for managing and record the metadata in a secure database, maintaining an organized image catalogue.

4.1.2.2 Image Permissions

In VISOR an image can be of two different types, public or private. When registering a new image in the service, the user should choose between one of these types, as they rule the image access permissions, determining who can see and retrieve that image.

- **Public:** Images registered as public images can be seen and managed by everyone, so any service user is able to see, manipulate and retrieve it.
- **Private:** Images registered as private images can only be seen and managed by their owners, which are the users who registered and uploaded them.

4.1.2.3 Image Status

At registering time, an user can provide both image metadata and its file, or provide only metadata if he wants to provide the file later. Considering this, an image can have one of the following status, which define the image availability.

- **Locked**: The locked status means that metadata was already registered in VISOR, and now the service is waiting for the image file upload.
- **Uploading**: An image with the uploading status means that its image file is currently being uploaded to VISOR.
- **Available:** An available image means that both metadata and file are already in the VISOR repository, so it is available for retrieving.
- **Error:** The error status informs that an error has occurred during the image file upload, so it is not available for download.

4.1.3 Architecture

VISOR is a distributed multi-subsystem service, each one being an independent web service playing a specific role towards the whole service functionality. The service main subsystem is the VISOR Image System (VIS) and it is the responsible for handling all the exposed service functionalities. Thus, VIS is the front-end for VISOR users. It handles users authentication and image uploads and downloads between clients and supported storage systems. Furthermore, it is also responsible for coordinating the image metadata management, submitting registering requests to another subsystem, the VISOR Meta System (VMS). Thus, the VMS is the subsystem responsible for image metadata, which is recorded on a database managed by it. The VMS, being another web service, exposes an API, which is used by the VIS when it wants to accomplish some metadata operation. The other service subsystem is the VISOR Auth System (VAS), which maintains and exposes a database containing user accounts information. Whenever the VIS wants to authenticate some user, it communicates with the VAS in order to search and retrieve user account credentials.

Besides the VIS, VMS and VAS, the service incorporates other two subsystems, the VISOR Common System (VCS) and the VISOR Web System (VWS). The VCS integrates a set of utility modules and methods used across all other subsystems, so its source code gets required by them in order to access such utility functions. Lastly, the VWS is a prototype web application, which aims to be a graphical user interface to display statistical data about the service usage and provide dynamic reports about the images registered in VISOR.

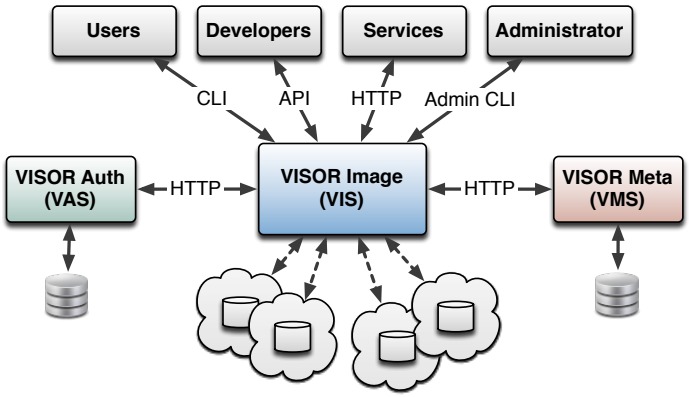

Figure 4.1: Architecture of the VISOR service.

The VISOR architecture and its main subsystems arrangement is pictured in Figure 4.1. The VIS, VMS and VAS are all independent RESTful Web services [36, 116], exposing their functionalities through the HTTP protocol. It is possible to observe that both VAS and VMS communicate with an database, where they register user accounts in the *users* table and image metadata in the *images* table, respectively. It is administrator's choice to deploy the VAS and VMS relying on the same database (containing both users and images tables) or with a database for each one.

Besides the service internals, VISOR also maintains a set of files towards logging and configuration. All subsystems' servers log and debug requests and responses to log files. While VISOR is being installed, it creates a template configuration file, which is detailed in Appendix C. This configuration file should be customized by users in order to provide the needed configuration parameters. These configurations are loaded by all the VISOR subsystems servers when they are started, and include the host addresses and ports to deploy each subsystem server, user's database and storage systems credentials and logging options.

4.1.3.1 Client Interaction

The users interaction with VISOR is handled by the VIS and accomplished through its user interfaces. The VIS stack includes a set of client tools, exposing the service to end-users, developers, external services and administrators. These clients can use the VIS CLI, the programming API, the HTTP interface and an administration CLI, respectively. All the main subsystems servers are powered by application servers, thus the administration CLI is the tool used to start, stop and restart their server applications. All the VISOR subsystems expose their own administration CLI, so the service administrator can use each them to manage the VIS, VAS and VMS application servers status independently. Since all subsystems administration CLIs have the same functionalities of the VIS one, we will only mention these kind of interfaces in the VIS description, as the same applies to the other subsystems.

4.1.4 Metadata

In order to describe images in the VISOR repository, the VMS supports a large range of fields within which we can describe images. For achieving a cross-IaaS image service, one key requirement is the flexibility of the metadata schema. Therefore we purpose a reference schema listed in Table 4.1, but we maintain it as much flexible as possible, where many of the user-managed attributes are made optional. Furthermore, users can provide any number of additional attributes (besides those listed in Table 4.1) which are encapsulated in the *others* attribute.

Table 4.1: Image metadata fields, data types, predefined values and access permissions.

Field	Type	Predefined Values	Permission
id	String		Read-Only
uri	String		Read-Only
name	String		Read-Write
architecture	String	i386, x86_64	Read-Write
access	String	public, private	Read-Write
status	String	locked, uploading, error, available	Read-Only
size	String		Read-Only
type	String	kernel, ramdisk, machine	Read-Write
format	String	iso, vhd, vdi, vmdk, ami, aki, ari	Read-Write
store	String	s3, cumulus, walrus, hdfs, lcs, http, file	Read-Write
location	String		Read-Only
kernel	String		Read-Write
ramdisk	String		Read-Write
owner	String		Read-Only
checksum	String		Read-Only
created_at	Date		Read-Only
uploaded_at	Date		Read-Only
updated_at	Date		Read-Only
accessed_at	Date		Read-Only
access_count	Long		Read-Only
others	Key/Value		Read-Write

4.1.4.1 User-Managed Attributes

For registering a new image, users must provide the *name*, which should preferably contain the OS name and its version, and the platform *architecture*, being it either 32-bit (i386) or 64-bit (x86_64). To define whether an image should be public or private, the *access* permission should be provided. If not, images will be set as public by default. For uploading an image file, the *store* attribute must be provided, indicating in which storage the image should be stored. Further details on the storage backends will be presented in Section 4.2.8. After the image has been uploaded, the *location* attribute will be set to the full image storage path.

Optionally, users can provide the *format* of the image and its *type*, with the last one defining if it is a kernel, ramdisk or a machine image. If an image has some associated *kernel* or *ramdisk* image already registered in the repository, users can also associate that image to the one being registered or updated, providing their *id* for these fields. Finally, users can also provide additional key/value pair attributes, which will be encapsulated in the *others* image attribute.

4.1.4.2 Service-Managed Attributes

For new images, the service defines an Universally Unique IDentifier (UUID) [141] for them (the *id*) and an URI (the *uri*), which defines the path to the location of an image in VISOR. The *status* of the image is also defined by the system and the *owner* attribute is set to the username of the user that has registered the image. Furthermore, the service defines the *size* of the image and its *checksum* (a MD5 hash [142] calculated at upload time). VISOR also maintains some tracking fields useful to increase security, track user actions and to provide the ability to mine statistical data about the service usage and images life cycle. These fields are the *created_at*, *uploaded_at*, *updated_at* and *accessed_at* (last image access) timestamps, and the number of image accesses counted through the *access_count* attribute.

4.2 VISOR Image System

The VIS is the VISOR main subsystem, as it is the entry point for all image management operations, being them metadata requests, image files related requests, or both of them. Thus, whenever a user wants to manage the VISOR repository, it should communicate with VIS, using one of its multiple interfaces. In this section we describe the VIS architecture and each one of its internal components in detail.

4.2.1 Architecture

The VIS architecture, as represented in Figure 4.2, is composed by three main layers. These are the client interfaces to interact with the system, the server application (which is the core of the system) and the storage abstraction layer that implements a storage system agnostic API.

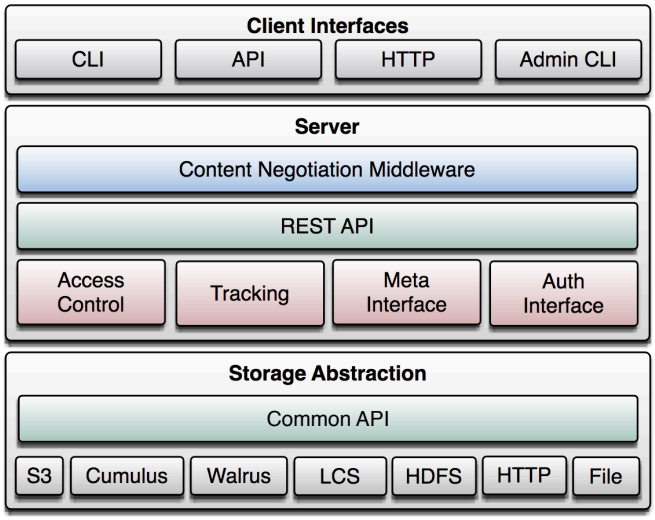

Figure 4.2: VISOR Image System layered architecture.

Sitting on top of the VIS are the provided set of interfaces, exposing the system to a wide range of clients. These are the main CLI, the programming API, the HTTP interface and an administration CLI. The server itself comprises the content negotiation middleware, the REST API, the access control and tracking modules, and the interfaces to communicate with the VMS (Meta Interface) and VAS (Auth Interface). Lastly, the storage backend abstraction layer comprises all the supported storage systems individual plugins, with a common seamless API implemented on top of them. We will describe each one of the VIS architecture layers and theirs internal pieces, starting by the internal server components and followed by the storage abstraction layer and client's interfaces.

4.2.2 REST API

The server exposes the VISOR functionalities abroad through its REST interface[1], defined in Table 4.2. Through this set of HTTP methods and paths, it is possible to manage both image metadata and files. The VIS server supports both JSON and XML output data formats, with JSON being the default one. The supported data formats will be addressed in detail while describing the content negotiation middleware in Section 4.2.4.

[1] http://cvisor.org/Visor/Image/Server

Table 4.2: The VISOR Image System REST API methods, paths and matching operations.

Method	Path	Operation
HEAD	/images/<id>	Return detailed metadata of a given image.
GET	/images	Return brief metadata of all public and user's private images.
GET	/images/detail	Return detailed metadata of all public and user's private images.
GET	/images/<id>	Return the metadata and file of a given image.
POST	/images	Add new metadata and optionally upload a file.
PUT	/images/<id>	Update the metadata and/or file of a given image.
DELETE	/images/<id>	Remove the metadata and file of a given image.

Whenever a request is successfully processed or an exception occurs during it, the server will handle and return a response containing either a success code, with the requested response, or one detailed error message, from a well defined set of possibilities. These response codes, prone methods and a description of the returned response are listed in Table 4.3.

Table 4.3: The VISOR Image System REST API response codes, prone methods and description. Asterisks mean that all API methods are prone to the listed response code.

Code	Prone Methods	Description
200	*	Successful request.
400	POST, PUT	Failed metadata validation.
400	POST, PUT	Both location header and file provided.
400	POST, PUT	Trying to upload a file to a read-only store.
400	POST, PUT	No image file found at the given location.
400	POST, PUT	Unsupported store backend .
400	POST, PUT	Neither file or location header were provided.
403	*	User authentication has failed.
403	DELETE	No permission to manipulate image.
404	HEAD, GET, PUT, DELETE	Image metadata was not found.
404	GET, DELETE	Image file was not found.
404	GET	No images were found.
409	GET	Image file already downloaded to the current path.
409	POST, PUT	Image file already exists on store.
409	PUT	Cannot assign file to an available image.

If no error occurs during requests processing, a response with a 200 status code, along with the request result is returned to clients. If an error occurs, error messages are properly encoded and returned through the response body.

Regarding error handling, besides those contemplated by the VIS REST API (Table 4.3) it is important to understand how does VISOR handles errors during VMs file uploads. Whenever the server receives a request implying the upload of a VM file, both server file caching and the following upload to the storage are strictly monitored, based on the image metatada *status* attribute. At registering time of a new image, the status attribute is set to *locked*. Further, when an image file is provided during registering, the server promptly updates the status to *uploading*. Moreover, when the server faces an exception, it ensures that the image status is set to *error* prior to execution abort. The status is set to *available* only after a successful upload.

In the next sections we describe all the image management operations exposed by the VIS REST API (Table 4.2). The presented request result samples were collected with the Curl [143] Unix tool, with the VIS server running on the 0.0.0.0 host address and listening on port 4568. These are the raw API outputs, when interacting directly with the server using the HTTP protocol.

4.2.2.1 Retrieve an Image Metadata

When issuing HEAD requests, the server will ask VMS for the metatada of the image with the given ID and then will encode that metadata in valid HTTP headers, embedding them into the response. This encoded headers are of the form `X-Image-Meta-<Attribute>:<Value>`.

Listing 4.1: Sample HEAD request.

```
HEAD http://0.0.0.0:4568/images/d186965e-b7e3-4264-8462-7d84c2cac859
HTTP/1.1 200 OK
x-image-meta-_id: d186965e-b7e3-4264-8462-7d84c2cac859
x-image-meta-uri:http://0.0.0.0:4568/images/d186965e-b7e3-4264-8462-7d84c2cac859
x-image-meta-name: Ubuntu Server 12.04 LTS
x-image-meta-architecture: x86_64
x-image-meta-access: public
x-image-meta-status: available
x-image-meta-size: 751484928
x-image-meta-format: iso
x-image-meta-store: file
x-image-meta-location: file:///VMs/d186965e-b7e3-4264-8462-7d84c2cac859.iso
x-image-meta-created_at: 2012-05-01 20:32:16 UTC
x-image-meta-updated_at: 2012-05-01 20:32:16 UTC
x-image-meta-checksum: 2ea3ada0ad9342269453e804ba400c9e
x-image-meta-owner: joaodrp
Content-Length: 0
```

As we can see in the above Listing 4.1 sample output, when we retrieve the metadata of the image with the ID `d186965e-b7e3-4264-8462-7d84c2cac859`, we receive a HTTP 200 status code, as the request was successful. We then see the response headers, where it is encoded the image metadata, followed by an empty body, as shown by the *Content-Length* header.

4.2.2.2 Retrieve all Images Metadata

When receiving a GET request on */images* or */images/detail* paths, the server will ask the VMS, seeking the metatada of all public and user's private images, returning a brief or detailed description (respectively to the request path) of each matched image. In this case, the result is passed as a set of encoded documents through the response body.

Listing 4.2: Sample GET request for brief metadata.

```
GET http://0.0.0.0:4568/images
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 214

{
  "images": [
    {
      "_id": "d186965e-b7e3-4264-8462-7d84c2cac859",
      "name": "Ubuntu Server 12.04 LTS",
      "architecture": "x86_64",
      "access": "public",
      "format": "iso",
```

```
14        "size": 751484928,
15        "store": "file"
16      },
17      {
18        "_id": "4648b084-8311-40f8-9c4f-e15453acccf8",
19        "name": "CentOS 6.2",
20        "architecture": "x86_64",
21        "access": "private",
22        "format": "iso",
23        "size": 851583221,
24        "store": "file"
25      }
26    ]
27  }
```

In the above Listing 4.2, we request the brief metadata of all public and user's private images. Thus, we receive a JSON document through the response body, containing all matched images, where in this case there were only two images in VISOR, one public and one private (owned by the request user). For detailed metadata, the path to be used is */images/detail*, receiving a similar output but with more displayed attributes. The difference between brief and detailed metadata will be discussed further in Section 4.3.2.1.

These two operations can handle HTTP query strings for filtering results, sort attributes and sort direction options. Thus, one can ask for a subset of public images that match one or more attributes value, or choose the attribute with which results should be sorted by, as equally providing the sorting direction (ascending or descending). Thus, if we want to return brief metadata of 64-bit images only, with results sorted by the name of the images in descending order, we issue a GET request to the path `/images?architecture=x86_64&sort=name&dir=desc`.

4.2.2.3 Register an Image

For POST requests, the server will look for metadata encoded in the request headers, decode them and then prompting the VMS to register that new metadata. The sent metadata headers from client to server are of the form of `X-Image-Meta-<Attribute>:<Value>`, as shown in Listing 4.1. Besides metadata, users may reference the corresponding image file through two distinct methods, providing its location URI through the *Location* attribute header, or by streaming the image file data through the request body. As expected, users can only use one of these methods, either providing the image file location or the image file itself.

Providing the Image Location: One should pass the location header if the associated image file is already stored somewhere in one of the compatible storage backends (listed further in Section 4.2.8), providing the image file path as the value of the `X-Image-Meta-Location` header.

Listing 4.3: Sample POST request with image location providing.

```
1 POST http://0.0.0.0:4568/images
2 HTTP/1.1 200 OK
3 x-image-meta-name: Fedora 16
4 x-image-meta-architecture: x86_64
5 x-image-meta-access: public
6 x-image-meta-format: iso
```

```
x-image-meta-store: http
x-image-meta-location: http://http://download.fedoraproject.org/pub\
/fedora/linux/releases/16/Live/x86_64/Fedora-16-x86_64-Live-Desktop.iso
```

So, if we want to register in VISOR an image mapped to the last release of the Fedora 16 operating system, we can do it by issuing the request present in the Listing 4.3. After providing the *name*, *architecture*, *access* and *format* values, we then inform the VIS that the image is stored on a HTTP *store*, at the URL provided in the *location* header.

Listing 4.4: Sample POST request response with image location providing.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 570

{
  "image": {
    "_id": "36082d55-59ee-43ed-9434-f77a503bc5d0",
    "uri": "http://0.0.0.0:4568/images/36082d55-59ee-43ed-9434-f77a503bc5d0"
    "name": "Fedora 16",
    "architecture": "x86_64",
    "access": "public",
    "status": "available",
    "size": 633339904,
    "format": "iso",
    "store": "http",
    "location": "http://download.fedoraproject.org/pub/fedora/linux/
                releases/16/Live/x86_64/Fedora-16-x86_64-Live-Desktop.iso",
    "created_at": "2012-05-11 17:51:51 UTC",
    "updated_at": "2012-05-11 17:51:51 UTC",
    "checksum": "2f5802-25c00000-9544b400",
    "owner": "joaodrp",
  }
}
```

When receiving such request, the server will perform an analysis on the provided location URL, trying to find if the resource is a valid file, its checksum (if any) and the resource's real location. This is done by seeking the URL, following redirects up to a certain deepness level and finally, parsing and analysing the HTTP headers of the true resource location.

After the request is processed, the VIS server would return the response described in the Listing 4.4. From there we can see that VISOR has defined an UUID for the image (*id*) and has generated the image URI. We can also see that it has detected the image size and its checksum, by analysing the provided location URL. Further, if we issue a GET request to this image, we will always being downloading the last available release of Fedora 16 from its HTTP URL.

Uploading the Image: Otherwise, if wanting to upload the image file an user must provided the same metadata through the request headers (as in Listing 4.3) but this time without providing the *location* header and modifying the *store* attribute value to indicate the storage system in which the image should be stored. Besides the metadata included in the request headers, the image file data to upload should be included in the request body.

When receiving such request, and after detecting that the request body is not empty (i.e. the client has also sent an image file for upload) the server will cache data chunks, as they arrive, in a secure local temporary file. It will also create and update a MD5 hash [142] (the *checksum*) during this process, guaranteeing the uploading file integrity. After the upload has been completed, it will look in the provided metadata for the description of the wanted storage service, promptly streaming the image file to that storage system. When the upload ends, the server returns the already inserted and updated image metadata through the response body, as shown in Listing 4.4, then closing the connection.

4.2.2.4 Retrieve an Image

For GET requests on the */images/<id>* path, the server asks the VMS for the metadata of the image with the given ID and tries to find if it has some already uploaded image file. If an image has no associated file, then the server returns a response with the image metadata encoded in HTTP headers and an empty body, as there is no associated file. If the image has an associated file, the server will look for it, parsing the metadata *store* and *location* attributes. Following, the server sends the file from its host storage system to client through the request body.

Listing 4.5: Sample GET request for metadata and file response.

```
1  GET http://0.0.0.0:4568/images/d186965e-b7e3-4264-8462-7d84c2cac859
2  HTTP/1.1 200 OK
3  Content-Type: application/octet-stream
4  x-image-meta-_id: d186965e-b7e3-4264-8462-7d84c2cac859
5  x-image-meta-uri:http://0.0.0.0:4568/images/d186965e-b7e3-4264-8462-7d84c2cac859
6  x-image-meta-name: Ubuntu Server 12.04 LTS
7  x-image-meta-architecture: x86_64
8  x-image-meta-access: public
9  x-image-meta-status: available
10 x-image-meta-size: 751484928
11 x-image-meta-format: iso
12 x-image-meta-store: file
13 x-image-meta-location: file:///VMs/d186965e-b7e3-4264-8462-7d84c2cac859.iso
14 x-image-meta-created_at: 2012-05-01 20:32:16 UTC
15 x-image-meta-updated_at: 2012-05-01 20:32:16 UTC
16 x-image-meta-uploaded_at: 2012-05-01 21:32:16 UTC
17 x-image-meta-checksum: 2ea3ada0ad9342269453e804ba400c9e
18 x-image-meta-owner: joaodrp
19 Transfer-Encoding: chunked
20
21 ????????????????????????????????????????????????????????????????????????...
```

As shown in the Listing 4.5 above, after finding the image file, the server opens a response passing the image metadata in HTTP headers, and the image file through the response body. In the above output, the body is shown as a set of "?" characters, as images are binary files.

4.2.2.5 Update an Image

When handling PUT requests, the process will be similar to that used for handling POST requests. The differences are the fact that the server needs to ask the VMS for the metadata of the image

with the given ID and there is also an image upload process constraint. The VIS server, after confirming the existence of the referenced image metadata, will prompt the VMS to do the required metadata updates. Besides metadata operations, image file upload goes in the same way as for POST requests. The image upload process constraint is that users can only provide an image file to a registered image with status set to *locked* or *error*. This is, it is only possible to assign an image file to images that were registered without an image file (locked) or to that images where an error has occurred during the last image file upload try (error). After the update process finishes, the server returns the already updated image metadata through the response body. The response output format for a PUT request will also be similar to that observed for POST request on Listing 4.4.

4.2.2.6 Remove an Image

When receiving a DELETE request, the server prompts the VMS for the metadata of the image with the given ID. After that, it parses the image file location and proceeds with its deletion (if any). Finally, it asks the VMS to delete the metatada and returns it through the response body.

Listing 4.6: Sample DELETE request response.

```
DELETE http://0.0.0.0:4568/images/d186965e-b7e3-4264-8462-7d84c2cac859
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 592

{
  "image": {
    "_id": "d186965e-b7e3-4264-8462-7d84c2cac859",
    "uri": "http://0.0.0.0:4568/images/d186965e-b7e3-4264-8462-7d84c2cac859"
    "name": "Ubuntu Server 12.04 LTS",
    "architecture": "x86_64",
    "access": "public",
    "status": "available",
    "size": 751484928,
    "format": "iso",
    "store": "file",
    "location": "file:///VMs/d186965e-b7e3-4264-8462-7d84c2cac859.iso",
    "created_at": "2012-05-01 20:32:16 UTC",
    "updated_at": "2012-05-01 20:32:16 UTC",
    "uploaded_at": "2012-05-01 21:32:16 UTC",
    "accessed_at": "2012-05-01 21:03:42 UTC",
    "access_count": 2,
    "checksum": "2ea3ada0ad9342269453e804ba400c9e",
    "owner": "joaodrp",
  }
}
```

As observed in the Listing 4.6, the server returns the deleted image full metadata through the response body. The benefit of returning an already deleted image metatada is to provide the ability to revert the deletion by resubmitting the image metadata through a POST request.

4.2.3 Image Transfer Approach

The way to efficiently handle large image file transfers, from clients to VISOR and from VISOR to storage systems and vice versa becomes a critical requirement. The two biggest concerns that arise are the ability to handle multiple long living connections in a high concurrency environment and data caching problems in all endpoint machines. For this purpose we have investigated how could us improve VISOR in order to avoid such problems without compromising performance.

4.2.3.1 Standard Responses

Relying on standard download and upload responses, a service handling large data transfers, as VISOR, would incur in significant latency and major overhead problems. We have outlined a standard download request and data transfer between client and server in Figure 4.3. When a client launches a download request of a large image file to the server, this one would need to cache the whole image in its address space, prior to sending it to client. In the same way, the client would need to cache the image file in memory, when receiving it from server, then closing the connection (represented with the "X" on the client timeline). Thus, the client is only able to write the file to disk after caching it in memory.

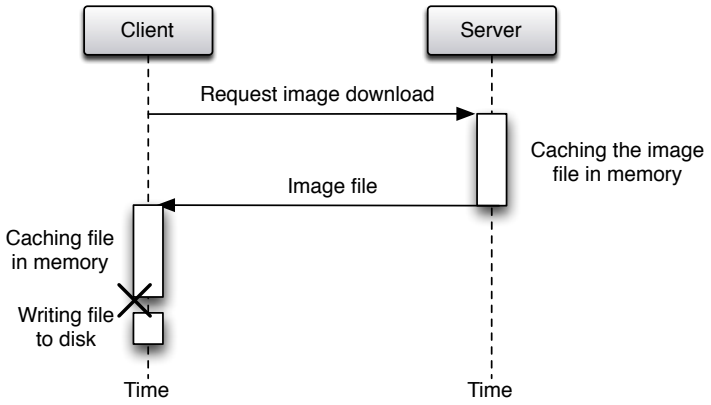

Figure 4.3: A standard image download request between client and server.

Such caching and latency issues would not be so critical for some users, as they may not want to download or upload a wide set of images at the same time. Therefore, most machines would probably have enough free memory to handle such requests. Indeed, memory is cheap. Although, when considering VISOR, we have designed it to be a reliable, high performance and concurrency proof service. Furthermore, besides the inherent high latency charges, with enough concurrent clients requesting to download different images, the host server in Figure 4.3 would definitely get out of memory caching all files.

4.2.3.2 Chunked Responses

Unlike standard responses, HTTP chunked responses, a feature of the HTTP version 1.1 [37], makes it possible to efficiently handle large data transfers without incurring in latency or host's memory overflow problems. Instead of returning a *Content-Lenght* header in the response, for a chunked transfer, a server sends a *Transfer-Encoding* header set to *chunked*, followed by length-prefixed data chunks in the response body. Furthermore, all chunks are streamed in the same persistent connection, avoiding the costly creation of a new connection per chunk. We outline an image file transfer request from clients to server in Figure 4.4.

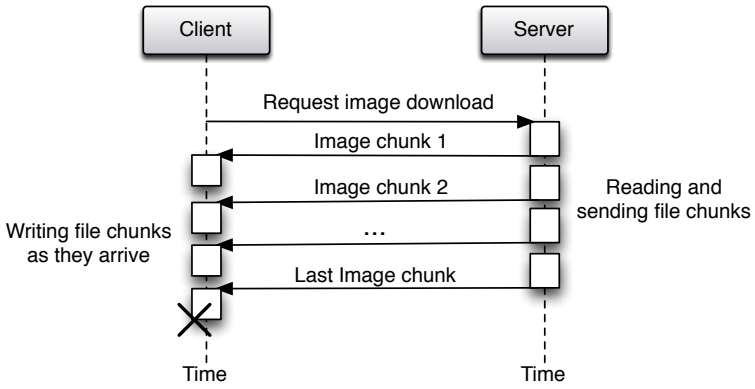

Figure 4.4: An image download request between a client and server with chunked responses.

On the opposite to the observed behaviour in Figure 4.3, for chunked responses it is possible to have both client and server processing data chunk by chunk. Thus, both server and clients are able to process the data without incurring in latency and high memory footprints.

4.2.3.3 VISOR Approach

Considering the outlined research above, we have chosen to integrate two-way chunked transfer capabilities in VIS client tools and server application. Thus, it is possible to handle image uploads and downloads from clients to VIS and storage systems, avoiding caching issues and achieving high performance responses. We outline the VIS behaviour for an image request in Figure 4.5.

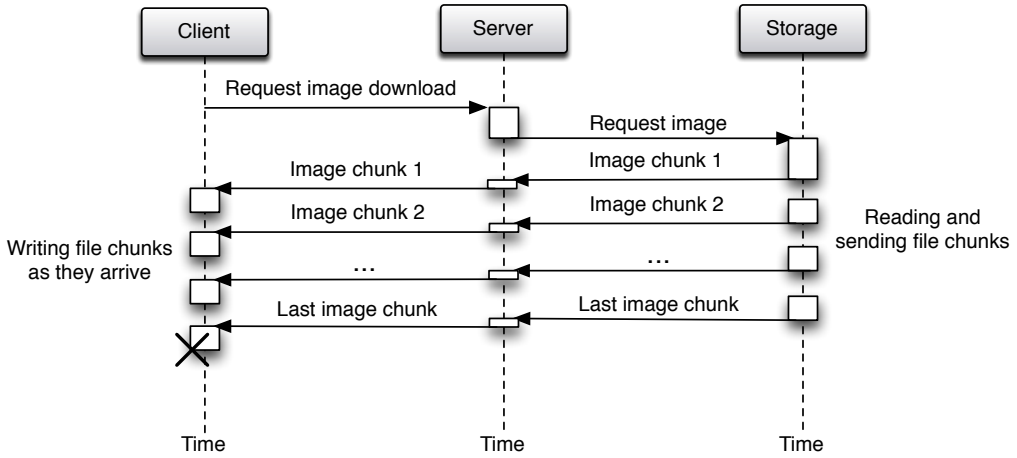

Figure 4.5: A VISOR chunked image download request encompassing the VIS client tools, server application and the underlying storage backends.

When a client requests an image download, the server indicates that it will stream the image file in chunks through the response body and that the connection should not be closed till the transfer ends. Then the server performs a chunked transfer from storage systems, passing the image chunks to clients, which write chunks to a local file as they arrive.

For uploads the process is similar, but the client is the one streaming the image in chunks to the server. The server caches data chunks in a secure local temporary file as they arrive, while calculating the image checksum to ensure data integrity. Having a local copy of the image will ensure that the server can retry images uploads to storages when facing issues, without prompting clients to re-upload the image (although not yet implemented). As soon as that process completes, the server will stream the image in chunks to the proper storage system.

4.2.4 Content Negotiation Middleware

The VIS server API is multi-format, most properly it supports automatic content negotiation with seamless responses encoding in JSON [58] and XML [59] formats, both for metadata and error messages. This process is done through the VIS server's content negotiation middleware, which internal encoding processes are detailed in Figure 4.6.

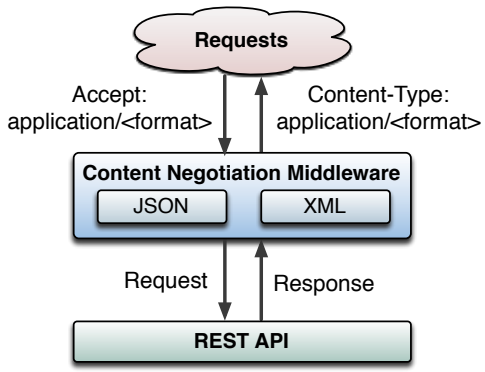

Figure 4.6: Content negotiation middleware encoding process.

This layer will auto negotiate and encode the response metadata and error messages in the proper format, based on the HTTP request's *Accept* header. When a request is received by the VIS server, this middleware will parse it, locate the *Accept* header (if any) and retains its value, being it *application/json* or *application/xml*. As the request reaches the REST API methods, it is processed in the below components and the returned results from it are automatically encoded by the negotiation middleware, either in JSON or XML, depending on the wanted format. It will also set the proper HTTP response's *Content-Type* header. If no *Accept* header is provided in the request, the API will encode and render responses as JSON by default.

By providing a multi-format API it becomes easier to dilute the heterogeneity of clients, supporting their own data format requirements. Additional format encoders can be plugged into the system, but by default, we ship VIS with built-in support for both JSON and XML formats only, which should be enough for the large majority of usage environments.

4.2.5 Access Control

The access control module is responsible for ensuring that users requesting some operation are properly authorized to do so. When a request reaches the VIS server, it is passed through the access control, which will look for the user authenticity and authorization. This module also manages the sharing of VM images. Furthermore, at registering time, VM images may be set as *public* or *private* (concepts previously described in Section 4.1.2.2).

4.2.5.1 Authentication Approach

The VIS authentication approach has been implemented following the Amazon S3 authentication mechanism [144], with only slightly customizations to better fit VISOR's purposes. The S3 authentication mechanism is being used by many other cloud applications like Walrus [45] and Cumulus [50], in order to provide compatibility with such cloud provider standards. In VISOR, requests are allowed or denied firstly by the identity validation of the requester. For that purpose, VISOR stores user accounts information in the VISOR Auth System (VAS), and the VIS is the

one responsible to interact with it in order to collect user account information. That information is then used to validate the identity of the requester.

Following the Amazon API, we use a scheme based on access keys and secret keys, with SHA1 [145] based HMAC [146] signatures, which has been an heavily tested combination for message hashing [147, 148]. The first step to authenticate a request is to concatenate some request elements, using the user's secret key to sign that information with the HMAC-SHA1 algorithm. Finally, the signature is added to the request header, preceded by the user's access key.

When the server receives an authenticated request, it will look for the user with the given access key in the users database through the VAS, fetching its secret key, which is expected to only be known by both VAS and user. It then tries to generate a signature too, like done in client side, using the information and the secret key that it has retrieved for the user with the given access key. Following, the server compares the signature that it has generated with the one embedded in the request. If the two signatures match, the server can be sure that the requester is who he claims to be and it proceeds with the request.

Listing 4.7: Sample authentication failure response.

```
GET /images
HTTP/1.1 403 Forbidden
Content-Type: application/json; charset=utf-8
Content-Length: 52

{
  "code": 403,
  "message": "Authorization not provided."
}
```

Otherwise, if the mentioned user does not exists, the signatures do not match or the authorization header is not provided, the server raises an authorization error, as the authentication was not successful. A sample authentication failure response is detailed in Listing 4.7, where in this case the user has not included an authorization signature in the request. We will describe in detail the authentication and the request signing process in both client and server sides, providing a more in-depth analysis of the VISOR authentication features.

4.2.5.2 Client-Side Request Signing

When using VISOR (through the VIS), an user must provide a set of items, properly treated and encapsulated in the request HTTP headers. The VIS client CLI will do this process automatically. We will enumerate and describe each of the following items:

- **Access Key:** The user's access key, identifying who some requester is claiming to be.
- **Signature:** A request signature is calculated based on the user's secret key and a request information string, which contains the request HTTP method, the path and a timpestamp, indicating the UTC date and time of the request creation.
- **Date:** A request should also contain the *Date* header, indicating the UTC timestamp of the request creation, which should be the same of that used to generate the signature.

With these items, the server will have all the needed information to verify the user's identity. We describe the full requests authentication mechanism from clients side in Figure 4.7.

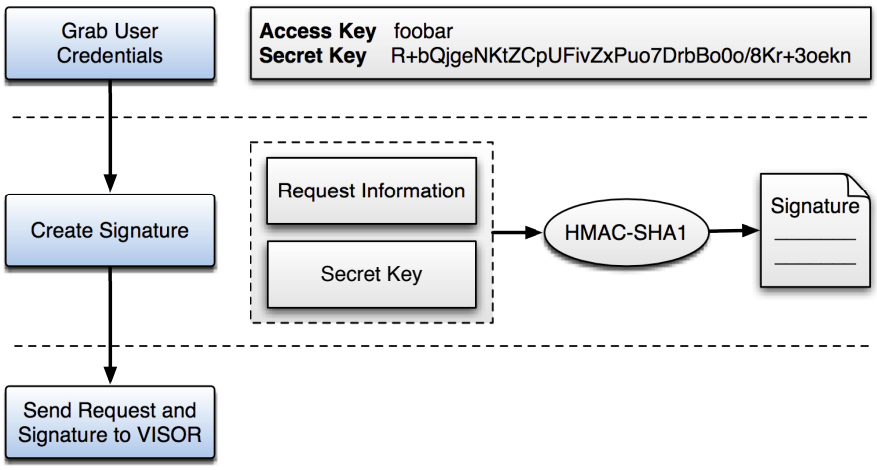

Figure 4.7: VISOR client tools requests signing.

The first step is to grab the user credentials, which VISOR client tools can find in the VISOR configuration file. In possession of user credentials, the next step is to create a valid signature. A signature is generated based on the user secret key and request information. In Table 4.4 it is shown an example of a request and its corresponding information to be signed.

Table 4.4: Sample request and its corresponding information to be signed.

Request	Request Information to Sign

`GET /images HTTP/1.1`
`Date: Tue, 20 May 2012 15:11:05 +0000`
	`GET\n`
	`\n`
	`\n`
	`Tue, 20 May 2012 15:11:05 +0000\n`
	`/images`

The pattern of the request information is to type the request method in uppercase, followed by three line breaks, the Universal Time Coordinated (UTC) timestamp, followed by a new line break and finally the request path. After achieving this request information string, the VIS client tools sign it using the user's secret key, generating in this way a valid *Authentication* header string of the form `"VISOR <user access key>:<request signature>"`. Considering the request example listed in Table 4.4, we present a VISOR authenticated HTTP request for it in Listing 4.8.

Listing 4.8: Sample authenticated request and its Authorization string.

```
1 GET /images HTTP/1.1
2 Date: Tue, 20 May 2012 15:11:05 +0000
3 Authorization: VISOR foobar:caJIUsDC8DD7pKDtpQwasDFXsSDApw
```

Considering the request on the Listing 4.8, in line 1 we have the request method, in this case it is a GET request, followed by the request path, which is '*/images*', so we are requesting to receive brief metadata of all images. Then in line 2 we have the *Date* header, with an UTC timestamp of the request creation. Finally, in line 3 we have the *Authorization* header, which contains a string of the form `"VISOR <user access key>:<request signature>"`. This signature string, embedded in the *Authorization* header, is the process of signing the request information (Table 4.4) following the request signing process pictured in Figure 4.7. After computing this request, clients can send it to the VIS, which will try to validate the request and the requester identity by analysing the request signature.

4.2.5.3 Server-Side Request Authentication

On the server side, the requests authentication procedure is similar to the signing process in client side, but here, it is needed to fetch the requester secret key from the VAS, in order to reproduce a valid signature. We illustrate the server-side authentication process in Figure 4.8.

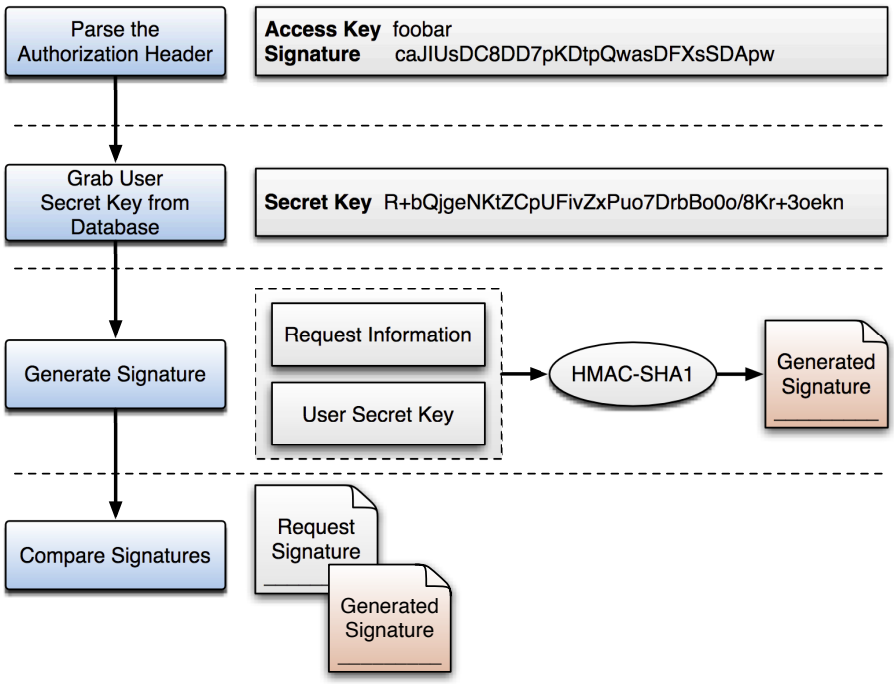

Figure 4.8: VISOR Image System server requests authentication.

When a request arrives, the server's access control module will look for an *Authorization* header, parsing it, or promptly denying the request if it was not provided. Further, knowing the requester access key, it will fetch the corresponding private key from the VAS. As it already knows the access key, it parses the *Date* header and collects the request information to sign, as done in the client side. Then, having these items and the private key retrieved from the VAS, it is able to generate a signature. After that, the server will compare the signature that it has generated, which is guaranteed to be valid, and the signature sent along with the request. If both signatures match, then the requester is the one it claims to be and the request proceeds, otherwise the request is denied with an authentication error message being raised (Listing 4.7).

4.2.6 Tracking

This component is responsible for tracking and recording VIS API operations, creating a full history of each image life cycle since it was published in VISOR. Also, this component will provide statistical data, which will become very useful for tracking the service and repository usage. This data can help administrators in management tasks, such as detecting images that are never used and can be deleted to save storage space. This is very useful, since maintaining a large repository of VM images can become a hard task, where sometimes there are images never used or outdated. Thus, having statistical data about the repository usage and the images life cycle can greatly improve administrator's capabilities. This module will be the engine of the VISOR Web System (VWS), a Web application providing statistical graphs and the ability to generate custom reports based on some conditions (e.g. repeated or unused images detection).

4.2.7 Meta and Auth Interfaces

The Meta (VMS) and Auth (VAS) interfaces are two components for internal use, towards the VIS server communication with the VMS and the VAS. Whenever the VIS needs to process some metadata operation it uses the VMS interface in order to issue a request to the VMS and receive the corresponding response with the processing result. If the VIS needs to authenticate some user, it uses the VAS interface in order to retrieve the user's credentials from the VAS.

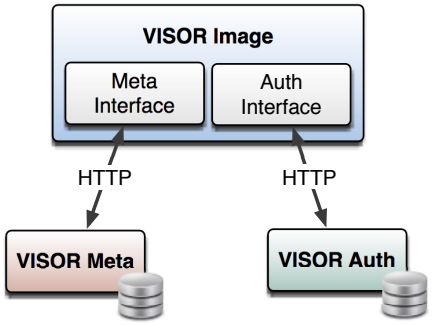

Figure 4.9: The communication between the VISOR Image, Meta and Auth systems.

The VIS Meta Interface[2] comprises a set of methods, listed in Table 4.5, used in order to communicate with the VMS. These set of methods and their description should be self explanatory and can be easily matched with the VMS REST API described further in Section 4.3.2, as this module is a programming API which conforms to its tenets.

Table 4.5: The VISOR Meta interface. Asterisks mean that those arguments are optional.

Method	Arguments	Return
`get_images()`	Query filter*	All public and user's private images brief metadata.
`get_images_detail()`	Query filter*	All public and user's private images detailed metadata.
`get_image()`	Image ID	The requested image metadata.
`post_image()`	Metadata	The already inserted image metadata.
`put_image()`	Metadata	The already updated image metadata.
`delete_image()`	Image ID	The already deleted image metadata.

In the same way as the Meta Interface, the Auth Interface[3] is a programming API, conforming to the tenets of the VAS REST API, which will be described further in Section 4.4.3. Through this interface it is possible to query the VAS web service in order to obtain and manipulate the users database. It is used by the VIS when it needs to query the users database in order to obtain user credentials at authentication time, as described in Section 4.2.5.3.

Table 4.6: The VISOR Auth interface. Asterisks mean that those arguments are optional.

Method	Arguments	Return
`get_users()`	Query filter*	All user accounts information.
`get_user()`	Access key	The requested user account information.
`post_user()`	Information	The already created user account information.
`put_user()`	Access key, Information	The already updated user account information.
`delete_user()`	Access key	The already deleted user account information.

[2] `http://cvisor.org/Visor/Image/Meta`
[3] `http://cvisor.org/Visor/Image/Auth`

4.2.8 Storage Abstraction

This layer is a multi-storage abstraction, providing seamless integration with multiple storage systems. This layer abstracts the heterogeneity of multiple distinct cloud storage systems, in order to provide the ability to build an unified and homogenised API, capable of interacting with multiple platforms, no matter their API complexity or geographical location. All that users need to do is to say in which compatible storage system they want to store an image, and VISOR will seamlessly handle the management of that VM image, acting as a bridge between storage systems and endpoint machines.

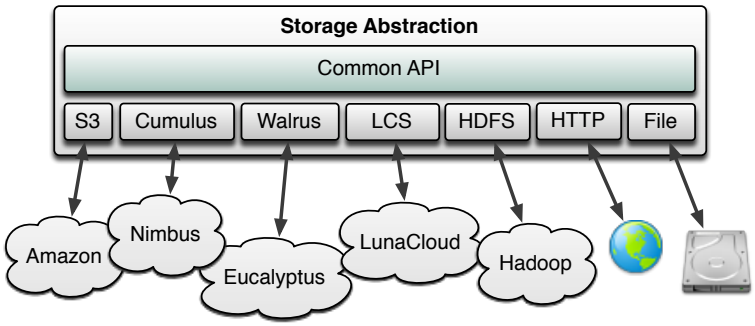

Figure 4.10: The VISOR Image System storage abstraction, compatible clouds and their storage systems.

As pictured in Figure 4.10, the storage layer targets integration with multiple cloud IaaS, namely the Amazon AWS [34], Nimbus [49], Eucalyptus [44, 45], LunaCloud [23] and the Apache Hadoop platform [52]. Thus, the storage layer integrates plugins for the storage systems of these IaaS, which are S3 [41], Cumulus [50], Walrus [45], LCS [23] and HDFS [104, 51] (addressed within Nimbus on Section 2.1.4), respectively. Besides the cloud-based storage systems, we also provide the ability to store images in the server local filesystem and a read-only HTTP backend.

4.2.8.1 Common API

With a unified interface to multiple storage systems it is simpler to achieve a cross-infrastructure service, as images can be stored in multiple distinct storage systems with seamless abstraction of details behind this process. Therefore, VIS provides this seamless API to deal with all the complexity of such heterogeneous environments.

Table 4.7: VISOR Image System storage abstraction layer common API.

Method	Arguments	Return
`save()`	Image ID, file	Save image to the storage system.
`get()`	Image ID	Get image from the storage system.
`delete()`	Image ID	Delete image from the storage system.
`file_exists?()`	Image ID	Find if an image file is in the storage system.

In order to seamlessly manage VM images stored in multiple heterogeneous storage systems, the VIS integrates a common API[4], sitting on top of multiple storage backend plugins. Through this simple API, listed in Table 4.7, the server is capable of managing VM images across all the compatible storage systems. Thus, it is able to save, get and delete images in each supported storage system.

[4] `http://cvisor.org/Visor/Image/Store`

4.2.8.2 Storage Plugins

The storage plugins are classes that implement the common API methods, in order to expose each own storage system functionalities to the top VIS server application. We detail the VIS API supported operations for each one of these storage systems in Table 4.8.

Table 4.8: Compatible storage backend plugins and their supported VISOR Image REST operations.

Store	Supported Operations
S3	GET, POST, PUT, DELETE
Cumulus	GET, POST, PUT, DELETE
Walrus	GET, POST, PUT, DELETE
LCS	GET, POST, PUT, DELETE
HDFS	GET, POST, PUT, DELETE
HTTP	GET
File	GET, POST, PUT, DELETE

As we can see, all storage systems plugins, but the HTTP one, support all the VIS REST API methods. Thus, VIS is able to upload, download and delete VM images from all those storage systems. The HTTP plugin, as it is intended to communicate with an external third-party HTTP server, is a read-only backend. Thus, we can only use it to reference images through an URL when we are registering or updating an image in VISOR, in order to download them later, directly from its web location.

Furthermore, targeting these storage systems is not a limitation, since given the system modularity it is possible to easily extend the system with other storage systems plugins. For extending the service with other storage plugins, it is only needed to know a storage system and its functionalities, in order to create a new plugin class, implementing all the common API methods (Table 4.7). We are also looking forward to add OpenStack Swift [54] to the list of compatible cloud storage systems, which has not been possible due to constraints on available tools to interact with it.

4.2.9 Client Interfaces

The VIS comprises a set of client interfaces, in order to expose the system abroad to a wide range of clients, including end-users, developers, external services and system administrators. Users interact with the system through the main CLI. Developers are those interacting with the system through its programming API, seeking to extend it or rely on it to build new tools. External services directly interact with the system through the HTTP REST interface. Finally, administrators can use the administration CLI to manage the systems status. We following describe each one of the client interfaces in detail.

4.2.9.1 Programming API

This component is a programming interface that conforms to the tenets of the VIS REST API (Table 4.2) and issues HTTP requests to it, properly handling the response back to clients. It provides a complete set of functions, detailed in Table 4.9, to manipulate VM image in VISOR. Every operation that can be done through the REST API can be achieved through this interface, as it is intended to be used by programmers which want to interact with the VIS, extend it, or create external applications relying on it.

Table 4.9: VISOR Image System programming API. Asterisks mean that those arguments are optional.

Method	Arguments	Return
`head_image()`	Image ID	The requested image detailed metadata.
`get_images()`	Query filter*	All public and user's private images brief metadata.
`get_images_detail()`	Query filter*	All public and user's private images detailed metadata.
`get_image()`	Image ID	The requested image detailed metadata and its file.
`post_image()`	Metadata, file*	The inserted image detailed metadata.
`put_image()`	ID, metadata, file*	The updated image detailed metadata.
`delete_image()`	Image ID	The deleted image detailed metadata.

Currently only Ruby bindings are provided, but it is extremely easy to extend the system compatibility with any other programming language client, as such API operates over standard HTTP requests conforming to the VIS REST API. An in-depth documentation of the programming API along with many examples on how to use it can be found at the API documentation page[5].

4.2.9.2 CLI

The main interface for those directly interacting with VISOR is the VIS CLI, named `visor`. Through this interface, it is possible to access all image management operations exposed by the VIS from an Unix command-line. The CLI exposes a set of commands, which are all detailed in Table 4.10. In conjunction with this set of commands, it is also possible to provide a set of options, which are listed in Table 4.11, in order to provide additional parameters.

Table 4.10: VISOR Image System CLI commands, arguments and their description. Asterisks mean that those arguments are optional.

Command	Arguments	Description
`brief`	Query filter	Return all public and user's private images brief metadata.
`detail`	Query filter	Return all public and user's private images brief metadata.
`head`	Image ID	Return the requested image detailed metadata.
`get`	Image ID	Return the requested image detailed metadata and its file.
`add`	Metadata, file*	Add a new image and optionally upload its file too.
`update`	ID, metadata, file*	Update an image metadata and/or upload its file.
`delete`	Image ID	Remove an image metadata and its file.
`help`	Command name	Show a specific command help.

Table 4.11: VISOR Image System CLI command options, their arguments and description.

Option	Short	Argument	Description
`--address`	`-a`	Host address	Address of the VISOR Image server host.
`--port`	`-p`	Port number	Port were the VISOR Image server listens.
`--query`	`-q`	String	HTTP query string to filter results.
`--sort`	`-s`	Attribute	Attribute to sort results.
`--dir`	`-d`	Direction	Direction to sort results ('asc'/'desc').
`--file`	`-f`	File path	Path to the image file to upload.
`--save`	`-s`	Path	Directory to save downloaded image.
`--verbose`	`-v`	-	Enable verbose logging.
`--help`	`-h`	-	Show help message.
`--version`	`-V`	-	Show VISOR version.

[5] `http://cvisor.org/Visor/Image/Client`

Next we will present all the commands and their syntaxes, in which we consider that the server is running on the host and port listed in the VISOR configuration file, so there is no need to explicitly provide these parameters. Elements surrounded by '< >' are those which should be filled by users. If separated by a vertical bar '|', it means that only one of those options should be used as parameter. Finally, those arguments surrounded by '[ ]' can be provided in any number, needing to be separated by a single space between them.

Retrieve Image Metadata: For retrieving an image metadata only, without the need to also download its file, users can use the *head* command, providing the image ID as first argument:

```
prompt> visor head <ID>
```

For requesting the brief or detailed metadata of all public and user's private images, one can use the *brief* and *detail* commands, respectively:

```
prompt> visor <brief | detail>
```

It is also possible to filter results based in some query string. Such query string should conform to the HTTP query string format. Thus, for example, if we want to get the metatada of 64-bit ISO 9660 images only, we would use the query `'architecture=x86_64&format=iso'` in the following command:

```
prompt> visor <brief | detail> --query '<query>'
```

Retrieve an Image: The ability to download an image file along with its metadata is exposed through the *get* command, providing to it the image ID string as first argument. If we do not want to save the image in the current directory, it is possible to provide the *--save* option, followed by the path were we want this image be to stored:

```
prompt> visor get <ID> --save '<path>'
```

Register an Image: For registering and uploading an image file, users can issue the command *add*, providing to it the image metadata, as a set of key/value pairs arguments in any number, separated between them with a single space. For also uploading an image file, users can pass the *--file* option, followed by the virtual image file path:

```
prompt> visor add [<attribute>=<value>] --file '<path>'
```

Otherwise, if users want to reference an already somewhere stored image file, it can be done by including the *location* attribute, followed by the virtual image file URI:

```
prompt> visor add [<attribute>=<value>] location='<URI>'
```

Update an Image: For updating an image, users can issue the command *update*, providing the image ID string as first argument, followed by any number of key/value pairs to update metadata:

```
prompt> visor update <ID> [<attribute>=<value>]
```

Further, if users want to upload an image file to a registered image metadata, it can be done by providing the *--file* option, or the *location* attribute, as done for the *add* command syntaxes:

```
prompt> visor update <ID> --file '<path>'
prompt> visor update <ID> location='<URL>'
```

Delete an Image: To remove an image metadata along with its file (if any), we can use the *delete* command, followed by the image ID, provided as its first argument. Also, it is possible to remove more than one image at the same time, providing a set of IDs, separated by a single space, or by providing the *--query* option, removing in this case all the images that match the provided query (if any):

```
prompt> visor delete [<ID>]
prompt> visor delete --query '<query>'
```

Request Help: Lastly, for displaying a detailed help message for a specific command, we can use the *help* command, followed by a specific command name for which we want to see a help message:

```
prompt> visor help <head | brief | detail | get | add | update | delete>
```

This set of commands and options form the VIS CLI, which is the main interface to interact with the system, exposing its whole capabilities to end-users, who rely on VISOR to manage VM images across their cloud IaaSs.

4.2.9.3 Administration CLI

The last VIS client interface is the administration CLI named `visor-image`, used to administrate the system's server status. Through this administration CLI, it is possible to issue a set of commands: *start*, *stop*, *restart* and require to be informed about the server *status*. Optionally it is possible to provide a set of options to these commands, all of them listed in Table 4.12.

Table 4.12: VISOR Image System server administration CLI options.

Option	Short	Argument	Description
`--config`	`-c`	File path	Load custom configuration file.
`--address`	`-a`	Host address	Bind to host address.
`--port`	`-p`	Port number	Bind to port number.
`--env`	`-e`	Environment	Set the execution environment.
`--foreground`	`-f`	-	Do not daemonize.
`--debug`	`-d`	-	Enable debugging.
`--help`	`-h`	-	Show help message.
`--version`	`-V`	-	Show version.

For starting, stopping or restarting the VIS server with no custom configurations, so all VISOR configuration file options will be used as defaults, one can issue the following command:

```
prompt> visor-image <start | stop | restart>
```

For example, for starting the VIS server on a custom host address and port number, different of those listed in the configuration file, one can use the following command:

```
prompt> visor-image start -a <IP address> -p <port>
```

When issuing the *status* command, the user will see information about the VIS server running state (running or not), the process identifier (PID) and the listening host and port address:

```
prompt> visor-image status
```

If the server was running on the 0.0.0.0 host address and the 4568 port, with a process ID of 1234, the CLI will output a string as ”visor-image is running PID: 1234 URL: 0.0.0.0:4568”. If the server was not running, the output would be ”visor-image is not running”. VMS, VAS and VWS also have a similar administration CLI. The only differences are in its names, were they are called `visor-meta`, `visor-auth` and `visor-web` respectively.

4.3 VISOR Meta System

The VMS is the VISOR subsystem responsible for managing image metadata. It receives and processes metadata requests from the VIS, supporting all CRUD (Create, Read, Update, Delete) operations through a REST interface. In this section we describe the VMS architecture and each one of its components and functionalities.

4.3.1 Architecture

The VMS architecture, as represented in Figure 4.11, is composed by two main layers, the server application and a database abstraction layer. The server implements the REST API and manages a database connection pool in its address space. The database abstraction layer contains metadata validation mechanisms and implements a common API over multiple database plugins.

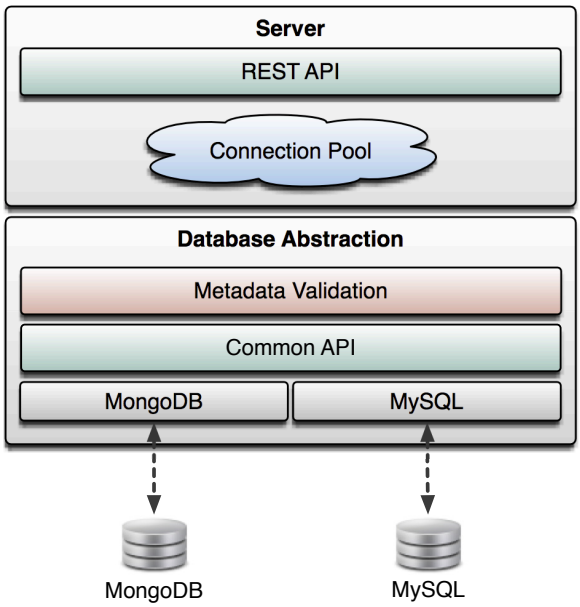

Figure 4.11: Architecture of the VISOR Meta System.

The server application is the responsible for handling incoming requests from the VIS. When a request arrives, its embedded metadata is passed through the metadata validations, which ensure its conformity with the metadata schema already detailed in the Section 4.1.4.

In order to interact with the database where the metadata is stored, the server uses one of the connections already instantiated in the connection pool. This connection pool is a cache of

multiple database connections, created at server's start. The database abstraction layer is the responsible to accomplish metadata operations in the underlying chosen database. This layer provides compatibility with several database systems, even if they have different architectures, like NoSQL databases (as MongoDB [64]) and regular SQL databases (as MySQL [105]). Thus, an user can choose (through the VISOR configuration file) to deploy the VMS backed by one of the currently compatible databases without any further tweaks, as the layer common API homogenizes their heterogeneity. We will describe each one of these components in detail.

4.3.2 REST API

The VMS server exposes the metadata management operations through the REST interface defined in Table 4.13. Through this set of methods, it is possible to retrieve, create, update and delete image metadata records on the backend database.

Table 4.13: The VISOR Meta System REST API methods, paths and matching operations.

Method	Path	Operation
GET	/images	Return brief metadata of all public and user's private images.
GET	/images/detail	Return detailed metadata of all public and user's private images.
GET	/images/<id>	Return metadata of a given image.
PUT	/images/<id>	Update metadata of a given image.
POST	/images	Add a new image metadata.
DELETE	/images/<id>	Remove metadata of a given image.

Regarding error handling, when the server faces an exception during requests processing, or the database raises one itself during queries processing, the server handles these exceptions. After recognizing them, it raises a set of error responses, listed in Table 4.14, containing the error code and an error message.

Table 4.14: The VISOR Meta System REST API response codes, prone methods and their description.

Code	Prone methods	Description
200	GET, POST, PUT, DELETE	Successful image metadata operation.
400	POST, PUT	Image metadata validation errors.
404	GET	No images were found.
404	GET, PUT, DELETE	Referenced image was not found.

These error messages are properly encoded in a JSON document and sent to clients through the response body. An example can be seen in the Listing 4.9.

Listing 4.9: Sample GET request failure response.

```
GET /images
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8

{
  "code": 404,
  "message": "No images were found."
}
```

We will describe all the images management operations exposed by the VMS REST API, being the addition, retrieval, update and deletion of images metadata. For all requests, the accepted input and exposed output metadata is formatted as JSON documents. We will not present here the VMS API request and response examples, as it is not expected to anyone directly interact with the VMS but rather with the already described VIS API (Section 4.2.2).

4.3.2.1 Retrieve all Images Metadata

When receiving a GET request on */images* or */images/detail* paths, the server will query the database for the metatada of all public images and the requesting user's private images, returning a brief or detailed description (respectively to the request's path) of each matched image. For *brief* metadata only the *id*, *name*, *architecture*, *access*, *type*, *format*, *store* and *size* attributes are returned. For *detailed* metadata, all attributes are listed, besides the *accessed_at* and *access_count*, which are intended for internal tracking purposes only. The found images metadata is returned through the response body.

4.3.2.2 Retrieve an Image Metadata

When issuing GET requests to the */images/<id>* path, the server will fetch the image ID from the request path and query the database for the detailed metatada of the image with that ID. Then, it will return its metadata through the response's body.

4.3.2.3 Register an Image Metadata

For POST requests, the server will look for metadata, which should be provided as a JSON document encoded in the request body. Further, the server decodes the metadata, passes it through the metadata validation filters and then, if there were no validation errors, asks the database to register it. After being registered, the image detailed metadata is returned through the response's body.

4.3.2.4 Update an Image Metadata

For PUT requests, the server will fetch the image ID from the request path and look for update metadata, which should be provided as a JSON document sent through the request body. Further, the server decodes the metadata and passes it through the metadata validation filters. If there were no validation errors, it asks the database to update the image record with the given ID with the newly provided metadata. After being updated, the image detailed metadata is returned through the response's body.

4.3.2.5 Remove an Image Metadata

When handling DELETE requests, the server will fetch the image ID from the request path and look for the wanted image metadata in the database. If it founds the image metadata, then the servers asks the database to delete that image metadata and returns it through the response body. It is useful to return the already deleted image metadata, because then users become able to revert that deletion by resubmitting the metadata through a POST request.

4.3.3 Connection Pool

When the VMS server is started, it identifies the database system that the users have chosen to back the server (specified in the VISOR configuration file) and creates a cache with database connections (on a predefined number) to it. We outline an example environment of 3 concurrent clients connected to the database through a connection pool in Figure 4.12, in which can be observed the interaction between clients, connections on the pool and the underlying database.

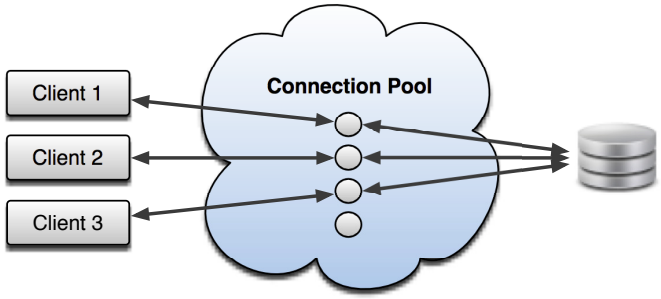

Figure 4.12: A connection pool with 3 connected clients.

This pool is intended to improve the system concurrency-proof and to avoid the costly creation of new database connections at each request, as connections in the pool are kept open and are reused in further requests. The pool maintains the ownership of the physical connections and is responsible for looking for available opened connections at each request. If a pooled connection is available, it is returned to the caller. Whenever the caller ends the communication, the pool sends that connection to the cache instead of closing it, so it can be further reused.

4.3.4 Database Abstraction

This layer implements a common API over individual plugins for multiple heterogeneous database systems, so it is possible to deploy the VMS server backed with one of the currently compatible databases systems without further tweaks. It also maintains a set of validators used to ensure the concordance of user's submitted metadata with the VISOR metadata schema.

4.3.4.1 Metadata Validation

The metadata validations are used whenever a request is received, so we can ensure that the incoming request's metadata is properly validated and in concordance with the VISOR metadata schema (Table 4.1). The launch of database queries is always preceded by these validations, as they are responsible for analysing the user submitted metadata in order to ensure that all provided attributes and their values are valid. Further documentation on the internals of such validation methods can be found at the documentation page[6].

4.3.4.2 Handling Extra Attributes

Considering the VISOR features and aims, we ensured that the VMS can provide a flexible metadata schema. NoSQL databases like MongoDB provide great schema free capabilities [72], with heterogeneous documents (which corresponds to SQL table rows) inside the same collection (which corresponds to a SQL table). Thus, it is possible to add custom metadata fields, not

[6] http://cvisor.org/Visor/Meta/Backends/Base

present on the recommended schema. Although, this is not the case when we consider classic SQL databases, which have a strict data schema. Therefore, we have introduced the capability to provide extra custom metadata attributes also when using a SQL database. This was achieved through an automatic encode/decode and encapsulate/decapsulate procedure, which stores the provided extra attributes into the *others* attribute present on the metadata schema.

Listing 4.10: Sample POST request image metadata JSON document.

```
{
    "name": "Ubuntu Server 12.04 LTS",
    "architecture": "x86_64",
    "access": "public",
    "extra1": "a value",
    "extra2": "another value"
}
```

For example, if an user provides the metadata of the Listing 4.10 for a POST request sent to the VIS, when that metadata reaches the VMS, the abstraction layer will detect the sample *extra1* and *extra2* extra attributes. Further, it will encode those extra attributes in a JSON string as `'{"extra1":"a value", "extra2":"another value"}'`, storing it in the *others* attribute. When looking for an image metadata, the server only needs to read the *others* attribute and parse it as a JSON document. Therefore, VISOR can handle extra metadata attributes not present on the recommended schema, even if relying on strict-schema SQL databases, as it is possible to read, delete and add any extra field to the *others* attribute.

4.3.4.3 Common API

For the VMS we wanted to provide freedom regarding the backend database system choice, in which VISOR metadata should be registered. Thus, it was needed to provide the ability to store metadata on more than one database system, regardless of their architecture (SQL or NoSQL databases) and their own interface constraints. Therefore, we have needed to implement an unified interface to multiple databases systems.

With such an unified interface, the VMS flexibility is considerably increased, as administrators may choose to use one of the currently supported databases to store image metadata, as the common API will dilute their heterogeneity. Thus, we have implemented a common API[7], sitting on top of multiple storage backend plugins. Through this simple API, the server is capable of managing images metadata across all the compatible storage systems.

4.3.4.4 Database Plugins

Currently we provide compatibility with MongoDB and MySQL, although it is extremely easy to extend the system with support to other database systems beyond these two. If someone wants to extend the system with such tools, the only concern is to implement a basic database CRUD API class, respecting the tenets of the centralized metadata schema and everything should work properly without further tweaks.

[7] `http://cvisor.org/Visor/Image/Store`

4.4 VISOR Auth System

The VAS is the VISOR subsystem responsible for managing user accounts. It receives and processes user accounts requests from the VIS, supporting all CRUD operations through a REST interface. In this section we describe the VAS architecture and each one of its components.

4.4.1 Architecture

The VAS architecture, as represented in Figure 4.13, is almost identical to that described in Section 4.3.1 for the VMS. It is composed by two main layers, the server and the database abstraction. The server implements the REST API and manages a database connection pool in its address space. The database abstraction layer implements a common API over multiple database plugins. The system also provides a CLI towards the administration of user accounts.

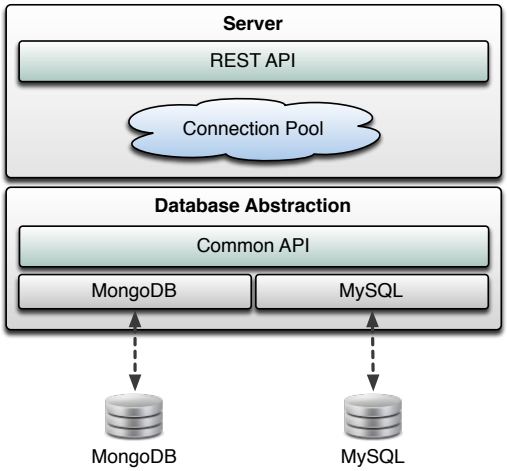

Figure 4.13: Architecture of the VISOR Auth System.

The VAS server is intended to receive requests from the VIS in order to manage user accounts. It also handles requests from the VAS CLI in order to let administrators add, update, list and remove users. This is an important feature since before interacting with VISOR, every user needs to create an user account. As the database abstraction layer and the connection pool share the same concepts of those applied in the VMS, we will not describe them again here.

4.4.2 User Accounts

The VAS server describes VISOR users and their accounts following a schema defined in detail in Table 4.15. As we can see, in VISOR, users are identified mainly by their access and secret keys. There are also the email address and the timestamps of the account creation and last update.

Table 4.15: User account fields, data types and access permissions.

Field	Type	Permission
access_key	String	Read-Write
secret_key	String	Read-Only
email	String	Read-Write
created_at	Date	Read-Only
updated_at	Date	Read-Only

When a new user is being registered, it should provide the username that he wants to be identified by, which is known as the *access_key*. Users should also provide their *email* address, which can be useful for the communication between users and system administrators. Then, the server generates a secure random string, which will be the user's *secret_key*. Besides these attributes, the server generates and maintains other two tracking fields, the *created_at* and *updated_at* timestamps.

4.4.3 REST API

The VAS server exposes the user accounts management functionalities through the REST interface[8] defined in Table 4.16. Through this set of methods, it is possible to retrieve, create, update and delete user accounts on the backend database.

Table 4.16: The VISOR Auth System REST API methods, paths and matching operations.

Method	Path	Operation
GET	/users	Returns all the registered users accounts information.
GET	/users/<access_key>	Returns an user account information.
PUT	/users/<access_key>	Updates the account information of a given user.
POST	/users	Adds a new user account.
DELETE	/users/<access_key>	Removes the account of a given user.

Regarding errors handling, when the server application faces an exception during requests processing, or the database raises one itself during queries processing, the server handles that exceptions. After recognizing them, it raises a set of error responses, listed in Table 4.17, containing the error code and an error message. These error messages, are properly encoded in a JSON document and sent to clients through the response body.

Table 4.17: The VISOR Auth System REST API response codes, prone methods and description.

Code	Prone methods	Description
200	GET, POST, PUT, DELETE	Successful user account operation.
400	POST, PUT	Email address is not valid.
404	GET, PUT, DELETE	Referenced user was not found.
409	POST, PUT	The access key was already taken.

Afterwords, we will describe all the user accounts management operations exposed by the VAS REST API, being the addition, retrieval, update and deletion. For all requests, the accepted input and exposed output data should be formatted as JSON documents.

4.4.4 User Accounts Administration CLI

Through the VAS user accounts administration CLI, named `visor-admin`, it is possible to create, retrieve, update and delete user accounts. Whenever a new user wants to use VISOR, it should first ask for an user account, in order to obtain the required credential to authenticate itself in VISOR against the VIS. The VAS user accounts administration CLI provides the following set of commands listed in Table 4.18. Within these commands it is possible to provide a set of options, all of them listed in Table 4.19.

[8] http://cvisor.org/Visor/Auth/Server

Table 4.18: VISOR Auth System user accounts administration CLI commands, their arguments and description. Asterisk marked arguments mean that they are optional.

Command	Arguments	Description
`list`	Query*	Show all registered users accounts.
`get`	User Access Key	Show a specific user account.
`add`	User Information	Register a new user account.
`update`	User Access Key	Update an user account.
`delete`	User Access Key	Delete an user account.
`clean`	-	Delete all user accounts.
`help`	Command name	Show help message for one of the commands.

Table 4.19: VISOR Auth System user accounts administration CLI options, their arguments and description.

Option	Short	Argument	Description
`--access`	`-a`	Key	The user access key.
`--email`	`-e`	Email Address	The user email address.
`--query`	`-q`	Query	HTTP query like string to filter results.
`--verbose`	`-v`	-	Enable verbose logging.
`--help`	`-h`	-	Show help message.
`--version`	`-V`	-	Show version.

We will now describe all the commands and their syntaxes. As already said in the previously presented VIS CLI examples, elements surrounded by '< >' are those which should be filled by users. If separated by a vertical bar '|', it means that only one of those options should be used as parameter. Finally, those arguments surrounded by '[ ]' can be provided in any number, needing to be separated by a single space between them.

Listing Users: For retrieving all registered user accounts, users can use the *list* command without arguments. For retrieve all user accounts that match a specific query, one can provide that query string too.

```
prompt> visor-admin list
prompt> visor-admin list --query '<query>'
```

Retrieve an User: When trying to retrieve a specific user account, the *get* command should be used, providing to it the user's access key as first argument:

```
prompt> visor-admin get <access key>
```

Register an User: To register a new user account, one can use the *add* command, providing to it the wanted access key and an email address:

```
prompt> visor-admin add access_key=<access key> email=<email address>
```

Update an User: For updating a specific user account, the *update* command should be used, providing to it the user's access key as first argument, followed by the key/value pairs to update:

```
prompt> visor-admin update <access key> [<attribute>=<value>]
```

Delete an User: In order to delete an user account, one can use the *delete* command, providing to it the user's access key. If wanting to delete all users, the *clean* command should be used:

```
prompt> visor-admin delete <access key>
prompt> visor-admin clean
```

Request Help: Finally, if aiming to display a detailed help message for how to use a given command, it can be done through the *help* command, providing the name of the command to display its help message:

```
prompt> visor-admin help <list | get | add | update | delete | clean>
```

4.5 VISOR Web System

VWS is a prototype Web application intended to ease the VISOR repository administration tasks. By now it only integrates dynamic graphs displaying useful statistical information about the images registered in VISOR. It is planned to extend the available statistical graphs and also implement dynamic generation of reports. Such reports would let administrators query VWS by images respecting some condition (e.g. what images where never used and can be deleted, what images some user has published) obtaining a clear report with the matched results. A screenshot of the VWS home page is displayed in Figure 4.14.

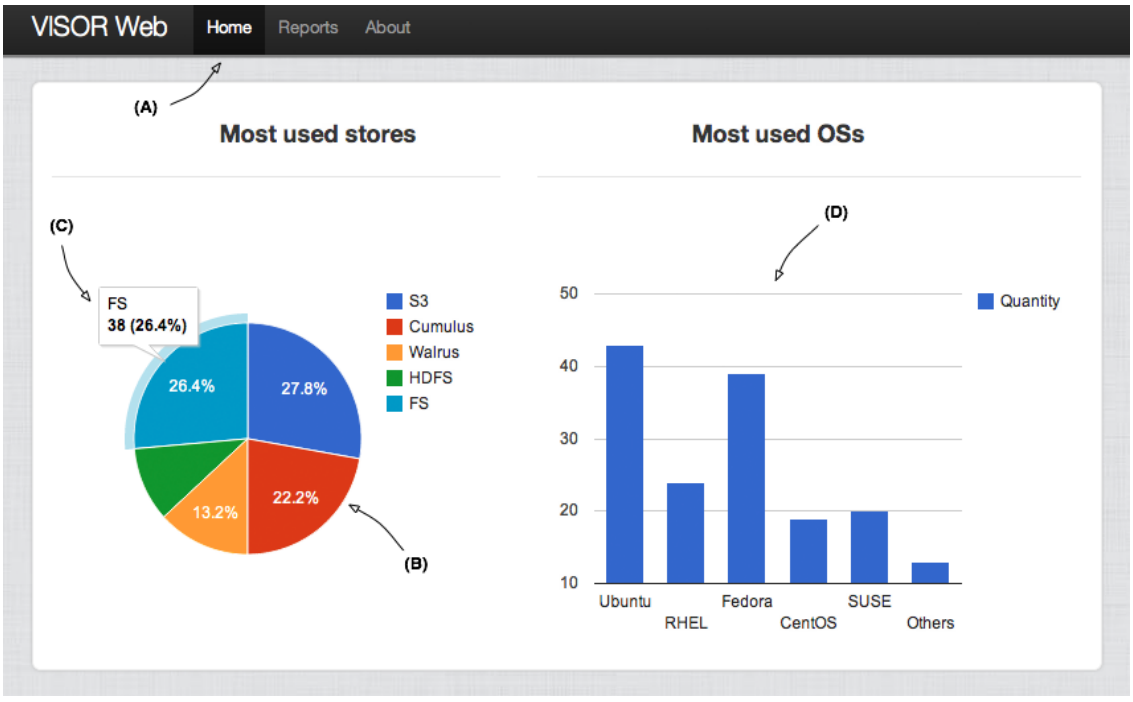

Figure 4.14: VISOR Web System Web portal.

When the report generating functionality becomes available it can be accessed through the VWS portal navigation bar (A), where by now only the home page is enabled. In the above screenshot it is possible to observe a graph displaying how much images are stored in each storage backend (B) and another displaying the most used operating systems among all registered images (D). All graphs are interactive and can easily display precise information about each item on the graphs when the mouse is rolled over them (C).

4.6 VISOR Common System

The last VISOR stack piece to be described is the VCS. This system contains a set of modules and methods used across all the VISOR subsystems, namely VIS, VMS, VAS and VWS. Across all the subsystems, we need a specific set of small utility methods, own exception classes, configuration file parsers and more. Therefore, instead of cloning those utility modules and methods across all subsystems, we have built them into a separate system, the VCS. We will present a brief list of those modules and utility methods. An in-depth description of all VCS modules, classes and methods can be found at the documentation page[9].

- **Configuration:** The configuration module provides a set of utility methods to manipulate configuration and logging files. Thus, all VISOR subsystems use this module in order to search for the VISOR configuration file and parse it. They also use this module when they need to configure a logging file for their server applications.
- **Exception:** The exception modules contains a set of custom exception classes declared and used through all the VISOR subsystems. Thus, whenever an exception is raised in any subsystem, the class of that exception is being grabbed from this module.
- **Extensions:** Considering the programming language own libraries, sometimes it is needed to extend them in order to achieve some custom operations. This module contains methods which are loaded at each subsystem start in order to extend the language libraries methods with additional ones. Examples of such extensions are the additional methods to manipulate hashes data structures[10].
- **Utility:** The utility module provides a set of utility methods to accomplish many simple operations. For example, it contains the function which signs the request credentials in both client and server sides. Besides the request signing methods, it also contains a set of simple methods to do simple operations as comparing objects for example.

Therefore, all the VISOR subsystems have as first dependency the VCS, so when they are being installed they will fetch and install VCS too, in order to be able to use the above described set of modules and their methods.

4.7 Development Tools

VISOR was developed using the Ruby programming language [24, 149]. Ruby has been used to build Cloud Computing projects such as VMware Cloud Foundry [89], OpenNebula [13, 14] and Red Hat Deltacloud [150] and OpenShift [90]. Documentation was generated with the help of YARD [151] and tests were written with the RSpec [152] behaviour-driven development tool. Source code was rolled out using the Git distributed version control system [153].

The VIS was developed using an EventMachine-based [139] non-blocking/asynchronous Ruby Web framework and server called Goliath [154]. The VMS, VAS and VWS were not developed with an event-driven framework, since they are small and have very fast response times. Therefore, since they would not gain from an event-driven architecture, they were developed using a Ruby domain-specific language Web framework called Sinatra [155], with all of them being powered by the Thin [156] event-driven application server.

[9] `http://cvisor.org/Visor/Common`

[10] `http://cvisor.org/Visor/Common/Extensions/Hash`

Chapter 5

Evaluation

In this chapter we present the conducted series of benchmarks in order to assess the performance of VISOR and its subsystems underlying components. We will discuss our evaluation approach and obtained results in order to assess the performance of the two biggest VISOR subsystems, the VIS and VMS. As the VAS is just a tiny web service, with almost the same technologies and architecture of the VMS, we have opted not to test it, as results would be identical to those observed for the VMS. In the same way, neither the VCS nor VWS can be tested, as one is just a collection of programming classes and methods and the other is a prototype web application.

Regarding the VIS, we have conducted a series of benchmarks assessing the system performance while transferring images between clients and supported storage systems. Therefore, we have addressed image registering and retrieving requests. We have performed such tests on two different test beds, one with a single VIS server instance, and another with two load balanced VIS server instances (behind a HTTP proxy). In the single server test bed we address single and concurrent requests, while on the two server instances test bed we reproduce the concurrent tests in order to assess the service scalability. In these tests we assess the VIS server response times, hosts resources usage and storage systems performance. Thus, we assess the VISOR overall performance, testing the image management functionalities.

In order to assess the VMS capabilities (even though the VMS performance is implicit in the VIS tests, as it issues metadata processing requests to the VMS), we need to address the performance not only of the VMS, but also of the compatible database backends. Therefore, we will present the VMS performance benchmarks considering both MongoDB and MySQL backends. In these tests we assess the VMS server throughput and each database system performance.

5.1 VISOR Image System

5.1.1 Methodology

For the VIS tests we have considered Walrus, Cumulus, HDFS and the local filesystem storage backends. Thus, we have not included in these tests the remote S3, LCS and HTTP backends, due to network constraints, as the outbound connection to their networks is limited and suffers from considerable bandwidth fluctuations, which we can not control. We have conducted a series of benchmarks, assessing the VISOR performance under images registering and retrieving requests. These tests were split in single and concurrent requests.

Aiming to provide fair and comparable results, we have used a single-node deployment for all storage systems, due to the high number of required machines to ensure a full multi-node deployment. We have repeated each test 3 times, using the average elapsed time as the reference value. After each round, we have verified the images checksum in order to ensure the files integrity. As the VIS needs to communicate with the VMS in order to manipulate image metadata, the VMS was also deployed, backed by MongoDB. We have chosen MongoDB because the conducted tests (described further in Section 5.3) to access both MongoDB and MySQL backends have showed MongoDB as a winner in terms of performance and service aims.

Typically, data transfer performance tends to be better when handling large files, due to the low payload to overhead ratio of the small ones [50]. Although the optimal file size directly depends and varies on the implementation of each service and used protocols. In order to achieve realistic results, we have chosen to use a set of six image sizes being of 250, 500, 750, 1000, 1500 and 2000MB. Prior to concurrent tests, we have registered in the image repository four images of each one of the six sizes, with image files stored in each one of the four tested storage systems, resulting in a total of 96 images. We have ensured a random selection of one of the four available images of each size in each storage system for each request. Thus, we can guarantee a realistic simulation, where some clients request to retrieve different images and others request to retrieve the same.

5.1.2 Specifications

In these tests we have used four independent host machines, as represented in Figure 5.1. One is used for deploying the VIS, VMS, MongoDB and the local filesystem. On the other three machines we have deployed Cumulus, Walrus and HDFS respectively. For single tests we have used another machine as the host for the VIS CLI client. For concurrent tests we have used the same machines with three additional client machines, leading to a total of four client machines. Moreover, for such concurrent tests, we have used a cluster SSH tool, in order to ensure the requests launch synchrony across all the four clients' CLI.

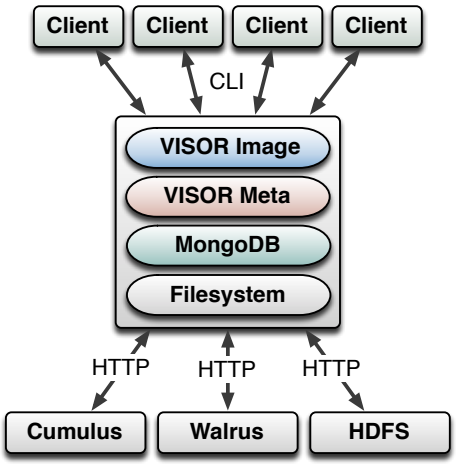

Figure 5.1: Architecture of the VISOR Image System single server test-bed. Each rectangular box represents a machine and each rounded box represents a single process or component.

All the four host servers are powered by identical hardware resources, with Intel i7 eight-core (hyper-threading) processors, 7500RPM disks, 8GB of RAM and ext4 filesystems. Regarding clients hosts, these are all powered by Intel i5 four-core (hyper-threading) processors, 5400RPM disks, 4GB of RAM and ext4 filesystems. All server hosts run the Ubuntu Server 10.04 LTS 64-bit operating system. Tests were carried on a 1Gbit local network, achieving an average transfer rate of $\approx$ 550 Mbit/s between any two machines, according to the *iperf* Unix tool. We have also refer to the *htop* Unix tool in order to collect some performance indicators regarding hosts used resources and average load. Regarding software versions, we have used Nimbus 2.9, Eucalyptus 2.0.3, Hadoop 1.0.1 and MongoDB 2.0.4. We use Ruby 1.9.3 in all the VISOR stack.

5.1.3 Single Requests

The first tests taken were the single sequential requests, testing the registering and retrieving of VM images in VISOR. We will now present and discuss the obtained results from these tests.

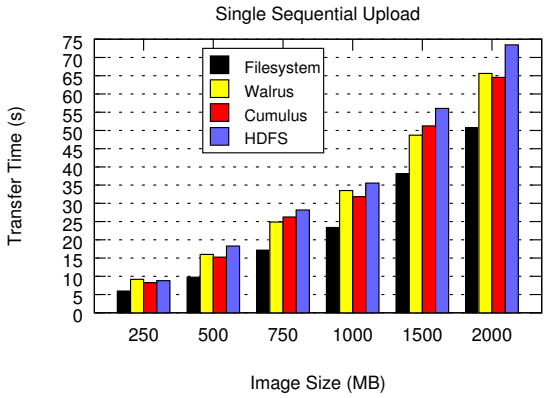

Figure 5.2: Sequentially registering a single image in each VISOR storage backend.

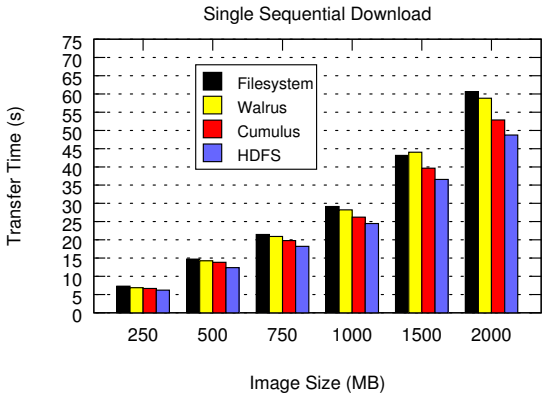

Figure 5.3: Sequentially retrieving a single image from each VISOR storage backend.

5.1.3.1 Images Registering

From the results pictured in Figure 5.2, it is possible to observe that the best performer for all image sizes uploads was the local filesystem. This was expected as the server does not spent any time transferring the image to a remote host. Regarding Walrus and Cumulus, the difference is not quite significant, but we can see Cumulus taking a small advantage on the run for most image sizes. Unfortunately, there is no description of Walrus in the literature (besides a brief mention within Eucalyptus [45]), that would let us predict such results. Regarding Cumulus, we were expecting to see good performance, as it even compares favorably with transfer standards like GridFTP [157], as assessed in [50]. Also, like VIS, Cumulus relies on an event-driven framework. On a different baseline is HDFS, being the worst performer, with the gap between them becoming proportional to the images size. Since HDFS was deployed on a single host, its *NameNode* (which generates an index of all replicated file blocks) and *DataNodes* (containing those file blocks) processes are all in the same machine. In HDFS, a file consists of blocks, so whenever there is a new block, the *NameNode* is responsible for allocating a new block ID and determine the *DataNodes* which will replicate it. The block size is defined to 64 MB, which is a reasonable value, as shown in [158]. Also, each block replica on a *DataNode* is represented by two files, one for data and another for metadata. Thus, the complex HDFS writes and its *single-writer* (only one process writing at a time) model [51] can give a clue about the observed results.

5.1.3.2 Images Retrieving

This test assesses the images download, with results pictured in Figure 5.3. As we can see the worst performer was the local filesystem. Already knowing its results on upload requests, such results become intriguing. Due to a constraint in the evented library, the server was not able to directly stream the image file from disk. Therefore, needing to load the whole image in memory prior to stream it, it incurs in a significant latency. Although, after the transfer begins, it performs faster than any other backend. For all of the other backends, VIS conducts the full streaming process without any issues with residual memory footprint in the VISOR host. When looking at remote backends, it is clear that HDFS was the best performer, followed by Cumulus

and Walrus, with the latter having a slightly poorer performance, specially when handling the largest images. HDFS really stands out for its performance, provided by its architecture and *multiple-reader* (multiple processes can read at a time) model [51]. HDFS performs quiet fast with results increasing linearly while iterating over the images size set. Without further details of the Walrus architecture it is not possible to perform a deeper analysis of its results. Cumulus stands out with the second best performance for all image sizes. In pair with VIS, we see here a possible major influence of the Cumulus event-driven architecture [50].

5.1.4 Concurrent Requests

After testing VISOR against single sequential requests, we have proceeded with the service performance assessments for concurrent requests. Therefore we have used 4 concurrent clients, placed in 4 independent machines, requesting to register and retrieve images.

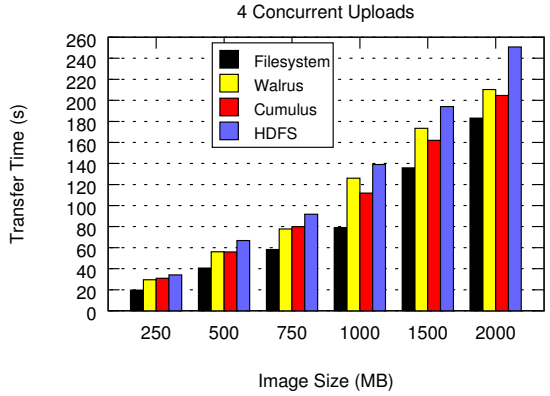

Figure 5.4: Four clients concurrently registering images in each VISOR storage backend.

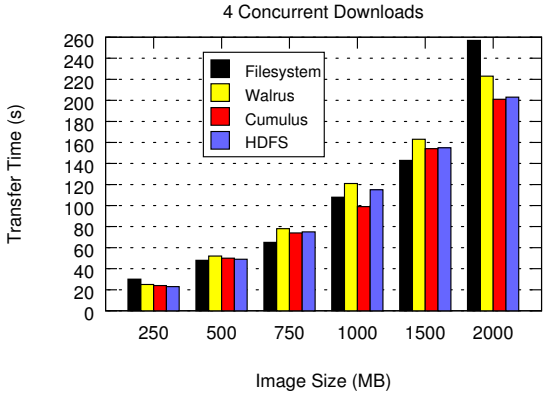

Figure 5.5: Four clients concurrently retrieving images from each VISOR storage backend.

5.1.4.1 Images Registering

For the concurrent image uploads, as we can see in Figure 5.4, the local filesystem remains the best performer. We also can see that Cumulus is the best performer among remote backends, while Walrus performs slightly worst and HDFS remains the worst performer as for single requests. If we take into account the results observed in single upload requests, as pictured in Figure 5.2, we can see that these concurrency results are in some cases (as for 750MB images), even some seconds smaller than 4 times the corresponding single request elapsed time. This gives us an encouraging overview of the system scalability potential.

5.1.4.2 Images Retrieving

For concurrent image downloads, with results pictured in Figure 5.5, we can see that Cumulus is the fastest remote backend for all image sizes but 250 and 500MB. If we refer to the single request tests (Figure 5.3), we can see that Cumulus has handled concurrency better than any other backend, even outperforming HDFS. Walrus was the worst performer when handling concurrent downloads and HDFS stands out with the second best performance. Regarding the filesystem, when handling 2000MB images, we can see a major transfer time peak. As already discussed, a constraint on the images streaming from the local filesystem is currently forcing VISOR to cache the image in memory before streaming it. Although, until VISOR exhausts the host memory (8GB) with 4*2000MB images, it remains one of the fastest backends.

5.1.4.3 Concurrency Overview

Although not being able to test the system against a high number of independent concurrent clients, due to the required number of machines, we still want to provide a wider overview of the VISOR concurrency-proof. Therefore, we have used the same 4 client machines, but we have spawn 2 and 4 threads per client, testing 750MB image requests. Thus, although being limited by the bandwidth and hard disk usage on each client, we were able to simulate 8 and 16 concurrent clients. In Figures 5.6 and 5.7 we present the overview of retrieving and registering 750MB images with 1, 2, 4, 8 and 16 concurrent clients.

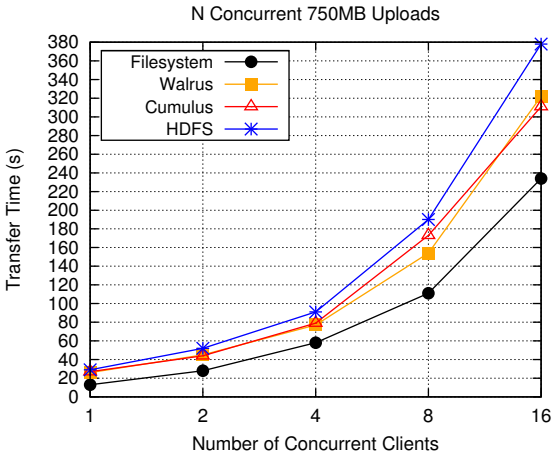

Figure 5.6: N concurrent clients registering 750MB images in each VISOR storage backend.

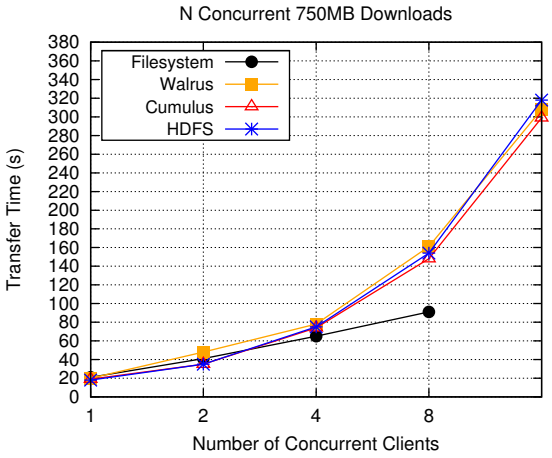

Figure 5.7: N concurrent clients retrieving 750MB images from each VISOR storage backend.

Based on these results, it is possible to say that the VISOR response time has grown in a factor of 2 as the number of clients doubles, for both image registering and retrieving requests. For registering requests (Figure 5.6), it becomes clear the difference between each backend behaviour, where HDFS is the one whose response time grows faster, specially with the highest concurrency level of 16 clients. For retrieving requests all backends become closer regarding response times, although Cumulus stills the fastest backend, followed by Walrus, with HDFS being the worst performer. Therefore, all response times follow the same order as for four concurrent requests, with response times growth becoming proportional to the number of concurrent clients. We were not able to perform 2000MB image retrieving from the filesystem backend due to the memory caching problem already mentioned.

5.1.5 Resources Usage

Regarding used resources during these tests, we have observed that the VISOR stack has only incurred in a small memory footprint in all used host machines. It is worth mentioning that the used memory during tests had only incurred in a residual increment, compared to that observed with the VIS and VMS servers in a dormant state. Knowing the asynchronous VIS server nature, it was expected to see it running on a single thread of execution, saturating 1 processor core. We have seen an average processor load of $\approx 75\%$. We were also able to observe that all of the storage systems hosts were memory stable. In the Cumulus host, as seen in the VIS server host, the processor was only saturated in 1 core, due to its asynchronous processing nature. Also as expected, we have observed that both Walrus and HDFS saturate all the hosts processor cores, due to their multi-threaded architecture [51].

When looking at the client side, the VIS CLI has also only incurred in a residual memory footprint, as it relies on full chunked streaming capabilities. It also saturates only 1 processor core with an average load of $\approx$ 20%. It worth mentioning that during these tests, VISOR and storage backends have not faced critical failures neither become unresponsive at any time. The resources usage for concurrent requests were close to those observed in single requests, with only an expected slight increase of processor usage in the VIS, VMS and storages host machines.

5.2 VISOR Image System with Load Balancing

Considering the encouraging results obtained and described in the previous section, we were aiming to assess the VIS scaling capabilities in order to improve even more the response times under high concurrency loads. Therefore, we have reproduced the concurrency overview tests described in Section 5.1.4.3, using two server instances behind a load balancer HTTP proxy.

Most of the complex web systems use load balancing techniques in order to improve response times and availability [159, 160, 161]. Therefore, we have replicated the VIS server across two independent host machines, placing them behind a HTTP proxy sitting in another host machine. The proxy does all the redirecting and load balancing routing to both server instances. Thus, clients are given with the IP address of the proxy, instead of the VIS server address. Whenever a request reaches the proxy, it will look for the two server instances and load-balance requests between them. For this purpose we have chosen to use HAProxy [162], a fast, high performance event-driven HTTP load balancer. From our research, we have concluded that many of the mainstream HTTP servers and proxies like Apache [163] and Nginx [164] would block the fast VIS streaming chunked transfers, thus incurring in major latency and memory overhead bottlenecks on the proxy machine. Indeed, they are optimized for serving common web applications, which almost always only require small to medium data size transfers. HAProxy is a free tool and has been used by giant companies, as RightScale for load balancing and server fail over in the cloud [165, 166]. It efficiently handles streaming responses without incurring in any latency or caching issues. We have configured it in order to load-balance requests between the two VIS server instances in a factor of 50/50, relying on the well-known Round Robin algorithm [160].

5.2.1 Methodology

These tests were conducted in the same way as the VIS single server tests (methodology described in Section 5.1.1). Although we have only tested registering and retrieving requests for 750MB images, from 1 to 16 concurrent clients. Thus, we can compare these results with those from the concurrency overview tests presented in Section 5.1.4.3. In this way, we expect to assess the VISOR scalability potential and the limits of the single-node storage systems deployments. Results obtained in these tests are pictured in Figures 5.9 and 5.10.

5.2.2 Specifications

In these tests we have used five independent host machines, as represented in Figure 5.8. One is used for deploying the VMS and the MongoDB database. The VIS and filesystem backends being deployed in other two independent host machines. Although we have deployed the VMS and MongoDB in the VIS host in the single server tests, here we have isolated them on a independent machine in order to do not increase the host utilization of one of the two VIS hosts. Thus, both

VIS hosts become comparable and can enjoy the same amount of resources. On another machine we have deployed Cumulus, Walrus and HDFS. For the single server test bed we have deployed each storage on independent hosts, but due to the higher number of required machines to do the same for these tests, we have to deploy all storage systems in the same host. Although, since we test each storage backend at a time, the remaining storage systems processes are idle and have almost no impact on the host available resources. Thus, such deployment is also comparable with the single server test bed. The last of the five host machines was used to deploy HAProxy. For clients we have used another four independent machines with the VIS CLI client.

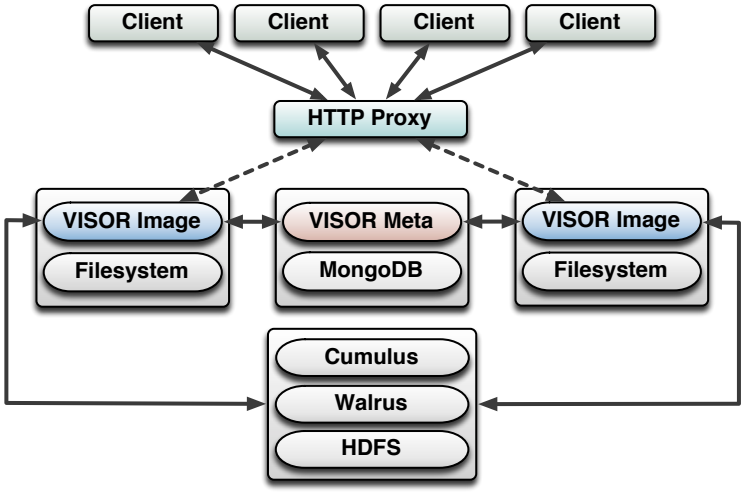

Figure 5.8: Architecture of the VISOR Image System dual server test-bed. Each rectangular box represents a machine and each rounded box represents a single process or component.

We were given with new machines in order to deploy the test-bed pictured in Figure 5.8. The storage systems were deployed in one of the machines used for the single server tests, with an Intel i7 eight-core (hyper-threading) 2.80GHz processor, a 7500RPM disk and 8GB of RAM. The four hosts used for the VIS server instances, VMS, MongoDB and HAProxy are powered by dual Intel Xeon eight-core 2.66GHz processors, 7500RPM disks and 6GB of RAM. Although these machines are not equivalent to those used for the single VIS server tests, they become comparable. Since VIS server instances run on a single core, they do not take advantage of the higher number of cores offered by the dual Xeon processors. Furthermore, being disk processing intensive, VIS servers and filesystem backends use disks with the same 7500RPM speed. All host machines run the Ubuntu Server 10.04 LTS 64-bit operating system and rely on ext4 filesystems. Regarding client hosts, these were the same four machines used for the single VIS server concurrent tests (described in Section 5.1.2). The same applies to network and software version specifications.

5.2.3 Results

5.2.3.1 Images Registering

For image registering requests at Figure 5.9, it is possible to see that compared to the single VIS deployment (Figure 5.6), the elapsed response times for the highest concurrency level (sixteen clients) have decreased by 16% for Cumulus and HDFS, 19% for Walrus and 50% for the local filesystem. Although, when looking at the smaller concurrency levels, it can be seen that the elapsed times are comparable, as it makes almost no difference to have a replicated deployment of two VIS server instances for attending just one or two clients. The much lower response times

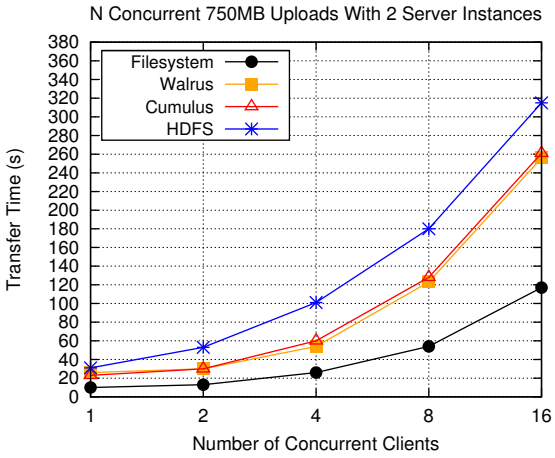

Figure 5.9: N concurrent clients registering 750MB images in each VISOR storage backend, with two server instances.

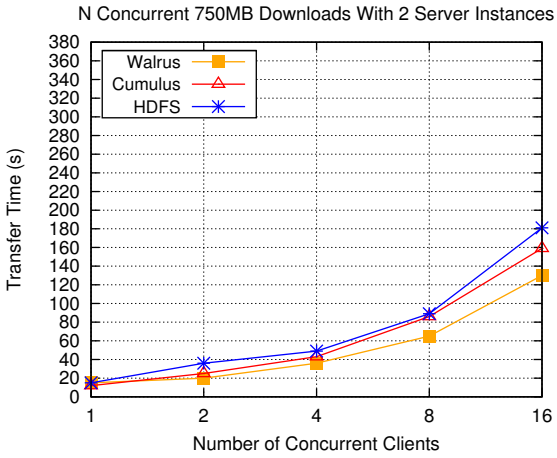

Figure 5.10: N concurrent clients retrieving 750MB images from each VISOR storage backend, with two server instances.

achieved by the local filesystem backend are justifiable by the fact that here we have two VIS server instances, thus each instance writes the images being registered in VISOR in its own local filesystem. Therefore, it was expected to see elapsed times reduced by 50%, as we have two filesystem backends, instead of just one (as for results observed previously in Figure 5.6).

In overall, while comparing these graphs with the ones from the previously concurrency overview tests, we can see steadier curves, with smaller steps between concurrency levels. Although, the elapsed times were not so smaller as expected when dealing with two VIS server instances. We have not seen any performance degradation in the server side. We can then say that we have achieved the maximum throughput from VISOR for images registering, with the bottleneck now being the single node deployment of all storage systems, slowing the service due to concurrent writes. Despite the fact that the number of clients was maintained since the previous non-replicated tests (one to sixteen), here the storage systems attend requests from two clients instead of just one, as we now have two VIS servers requesting to transfer images to backends. Therefore such constraints can also help to understand such results.

5.2.3.2 Images Retrieving

In the results pictured in Figure 5.10, the absence of the filesystem backend stands out. As already said, this backend corresponds to the server local hard drive. Thus, when considering two server instances, we cannot retrieve images from the local filesystem backend, as the proxy is not aware in which of the two instances the image being requested is actually stored. Therefore, we would get an exception when a request reached a server instance which has not the requested image stored in its filesystem. This can be solved by storing images in a remote filesystem, although we have not yet implemented in VISOR such backend plugin.

When looking at the remain storage backends, comparing the obtained results with those already discussed for a single VIS deployment (Figure 5.7), it becomes clear the huge response times decreasing. In these tests Walrus was the best performer, something that has not happened before, with response times decreasing 58%, followed by Cumulus with a decrease of 47% and HDFS with a decrease of 38% in response times, all for the highest concurrency level. As already described, the storage systems reading process is much lighter than the writing one, thus for retrieving request we achieve the expected results with around 50% faster response times.

5.3 VISOR Meta System

5.3.1 Methodology

As already stated, we have chosen to implement a NoSQL (MongoDB) and a SQL (MySQL) database backends for the VMS. Given this, it is needed to assess the performance not only of the VMS server, but also of such backend options. Therefore, we have conducted a set of tests to assess the throughput of the VMS under a simulated concurrent environment, relying on each one of the mentioned database backends. Prior to tests, both databases were filled with 200 virtual image metadata records with randomly generated values, according to the VISOR metadata schema (Section 4.1.4). Tests were performed under a simulated environment of 2000 requests with a concurrency level of 100 clients. We have chosen the simplest, not replicated deployment architecture for these tests, as metadata requests are very small in size and extremely fast in processing. Thus, the simplest single node deployment should be enough for most use cases. If aiming to improve data availability and durability, one can easily deploy the VMS on a replicated environment with multiple server instances (behind a HTTP proxy) and database replicas.

5.3.2 Specifications

The architecture of the tests deployment is detailed in Figure 5.11. As represented, we have used four independent machines, one for simulating the concurrent clients, another two for the VMS, and the last one containing the MongoDB and MySQL databases. One of the VMS instances was configured to run backed by MongoDB and the other by MySQL. Configuration options are described in each one of the VISOR configuration files (one in each machine). Thus, besides the database choice parameter in each configuration file, there was no need to further tweaks.

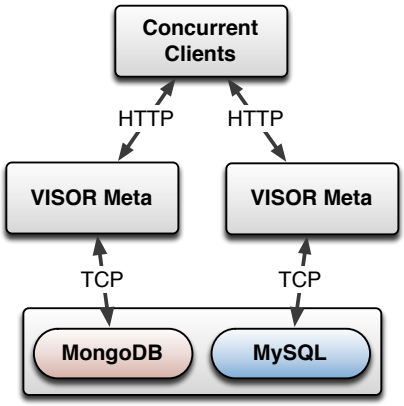

Figure 5.11: Architecture of the VISOR Meta System test-bed. Each rectangular box represents a machine and each rounded box represents a single process or component.

Both databases were configured with only one unique index on the images primary key (*id*). In order to achieve fair comparisons, MySQL was configured to run with the InnoDB engine and the provided ”huge” configuration file. This file is distributed with MySQL and contains improved configurations for hosts with good computing capabilities. Furthermore, we have given enough space to MySQL cache the database in memory (assigned 3GB of memory for InnoDB), thus it can be compared against MongoDB, which by default caches databases in memory.

The test bed pictured in Figure 5.11 was deployed on the same hardware used for the VIS tests (Section 5.1.2). We are using the currently latest releases of both databases, with MongoDB

version 2.0.4 64-bit and MySQL version 5.5.22 64-bit. We have simulated the concurrent clients issuing requests to the VMS using the ApacheBench HTTP benchmarking tool [167], version 2.3.

5.3.3 Results

We will now discuss the test results for all the VMS REST API operations, less the deletion of an image metadata, since the ApacheBench tool is not able to perform DELETE requests. Although we expect them to perform similar to a GET an image metadata request, since before sending the delete query to the database, the VMS server issues a GET to the image metadata and returns that metadata to clients. Thus the elapsed time for deleting an image metadata would only face a small increase due to the database delete query, when compared with the elapsed time for getting an image metadata. Furthermore, the retrieving of all images metadata was restricted to the brief metadata only (*/images* path) instead of detailed (*/images/detail*), as the difference between them two would only be related to the responses size.

5.3.3.1 Retrieve all Images Metadata

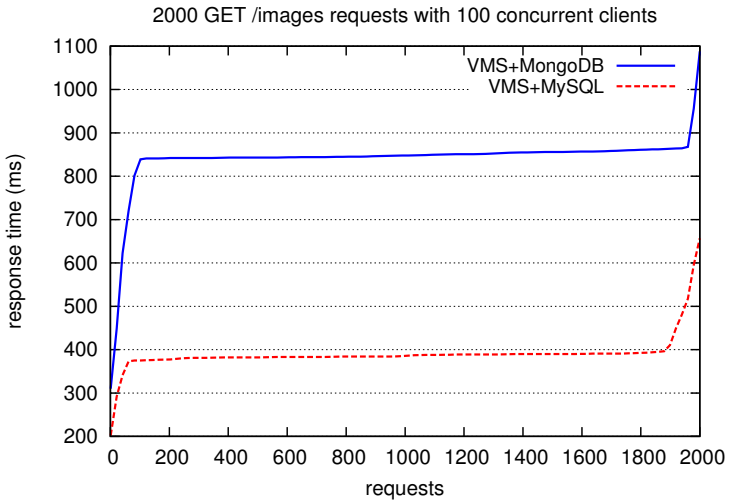

Figure 5.12: 2000 requests retrieving all images brief metadata, issued from 100 concurrent clients.

In Figure 5.12 we present the results for GET requests on the */images* path. For each one of the 2000 requests, all images brief metadata records are retrieved from the database and then returned to clients, all in the same response body. Thus, it is expected to see higher response times, considering that we are returning 200 image metadata records in the same response. As we can see, the VMS instance backed by MySQL (VMS+MySQL) has outperformed the MongoDB one (VMS+MongoDB) by far. The VMS+MongoDB has served 117.19 requests per second (req/s), with the VMS+MySQL instance outperforming it with 252.73 req/s. The response sizes were of 27305 bytes for VMS+MongoDB and 29692 bytes for VMS+MySQL. Such disparity in the response sizes is due to the fact that MongoDB, being a free schema database system, do not register metadata fields not provided, as done by SQL databases. Thus, MongoDB will only register the provided metadata attributes, where MySQL will register the provided metadata attributes and will set to *null* those not provided but present in the metadata schema. Therefore, the VMS+MySQL instance returned bigger responses than that returned by VMS+MongoDB. Even with such size discrepancy, MySQL clearly outperforms MongoDB when retrieving all records in batch.

5.3.3.2 Retrieve an Image Metadata

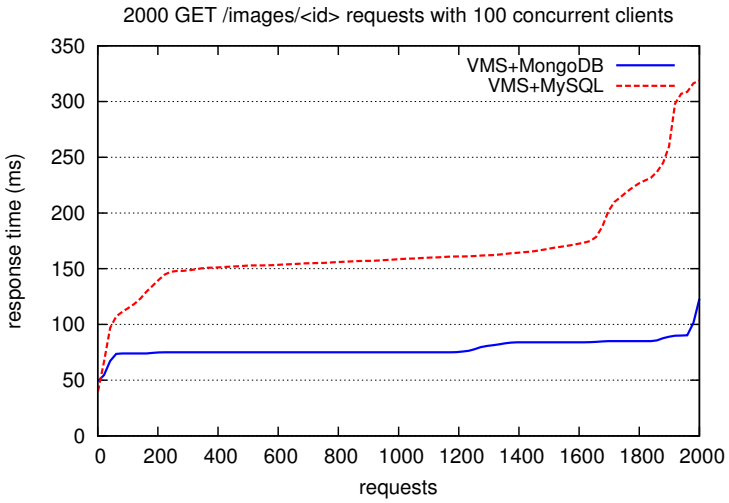

Figure 5.13: 2000 requests retrieving an image metadata, issued from 100 concurrent clients.

In Figure 5.13 we present the results for GET requests in the */images/<id>* path. Here the VMS server returns to clients the detailed metadata of the image with a given ID. During this process, the VMS server will perform an update to the *accessed_at* and *access_count* image metadata timestamps, thus it is expected to see an influence of database writes in these results. Here it becomes clear that VMS+MongoDB outperform the VMS+MySQL combination with a steadier curve. VMS+MongoDB was able to serve 1253.58 req/s, while MySQL only reached 584.39 req/s. In such results, MongoDB takes advantage of its atomic in-place update capability, avoiding the latency involved in querying and returning the whole record from the database in order to modify it and then submit the update to the database. This is applied to do in-place updates of the metadata timestamps. As reference, the response size was of 219 bytes for VMS+MongoDB and 337 bytes for VMS+MySQL, for the same reason stated in the previous test.

5.3.3.3 Register an Image Metadata

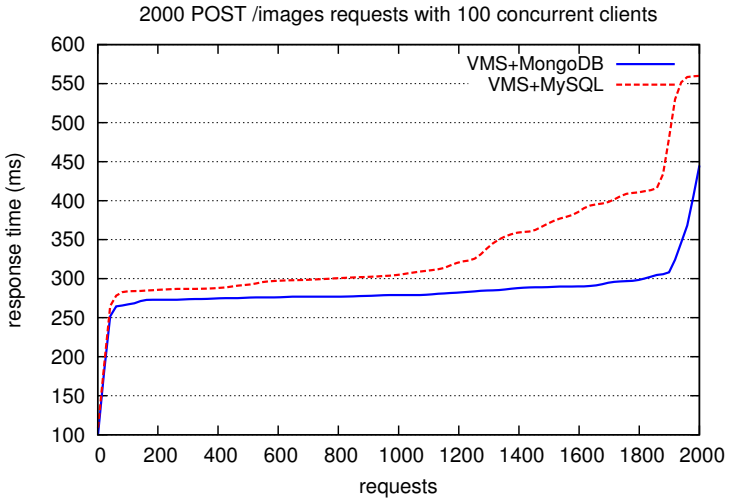

Figure 5.14: 2000 requests registering an image metadata, issued from 100 concurrent clients.

The next VMS operations to be tested are the POST requests on the */images* path, with results in Figure 5.14. Here the VMS receives the metadata to be registered in JSON sent from clients, properly validating and registering it on the underlying database. After that, the server returns to clients the detailed metadata of the record already inserted in the database. Here VMS+MongoDB has served 345.67 req/s, while VMS+MySQL has only served 293.47 req/s. For POST requests, MongoDB seems to take advantage over MySQL through its in-memory writes, only then flushing records to the persistent memory (hard drive) in background.

5.3.3.4 Update an Image Metadata

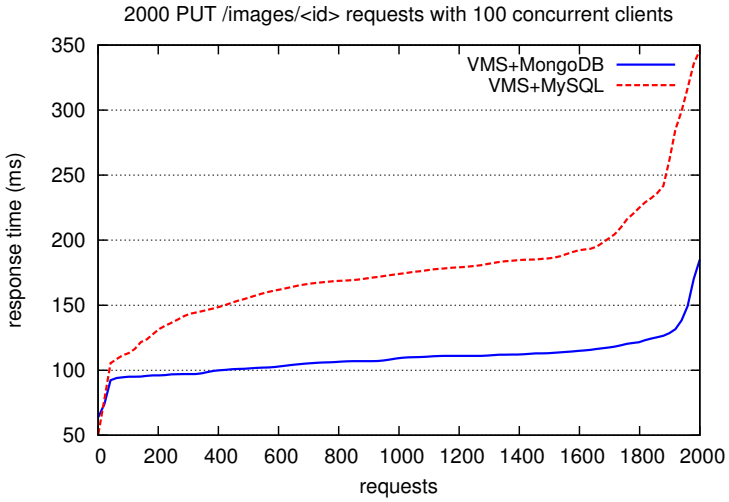

Figure 5.15: 2000 requests updating an image metadata, issued from 100 concurrent clients.

The last tested operation is the PUT request on the */images/<id>* path, with results pictured in Figure 5.15. Here the VMS server receives a JSON string from clients and properly validates and updates the image metadata record with the given ID. After that, it returns to clients the detailed metadata of the record already updated. Here VMS+MongoDB has served 899.67 req/s, outperforming VMS+MySQL, which has only been able to serve 553.89 req/s. Thus, one more time, MongoDB seems to take advantage of its atomic in-place updates.

5.4 Summary

In this chapter we have explained our testing approach towards assessing the performance of the two biggest VISOR subsystems, VIS and VMS. Two of the service main aims are to provide high availability and performance. We have relied in highly scalable technologies like event-driven frameworks and NoSQL databases in order to achieve such requirements. VIS tests have shown VISOR as a stable and high performance image service. VMS tests have shown good throughput from the VMS server, relying in both currently available database backends, with MongoDB being the overall best performer backend.

In [15], Laszewski *et al.* presents the FutureGrid platform own image repository (FGIR) performance tests. They follow a similar testing approach to that used by us in order to assess VISOR performance. They test the service with images of 50, 300, 600, 1000 and 2000MB in size, and FGIR only supports Cumulus, Swift, GridFS and the filesystem storage backends. Although GridFS is not a cloud storage system but rather a specification for storing large files in MongoDB.

FGIR is tested for images registering (uploads) and retrieving (downloads) for single sequential requests, measuring the transfers elapsed times. Although also presented the results for sixteen concurrent image downloads, there is no mention to the concurrent image upload tests.

We can fairly compare FGIR and VISOR performances only for 1GB and 2GB image single downloads and uploads requests, using the Cumulus storage system. We are limited to compare 1GB and 2GB images only since there are no other image sizes common to both tests approaches (FGIR and VISOR). Moreover, we may only compare single requests, since the presented VIS benchmarks for sixteen concurrent clients were obtained spawning four clients processes per client machine (in a total of four). Instead, in FGIR tests, they were given with sixteen independent machines, each one being an independent FGIR client. Finally, both FGIR and VISOR support a filesystem backend, although the FGIR one is not the server local filesystem but a remote filesystem, accessed through SSH. Therefore we can only compare results considering the Cumulus backend, as there are no other compatible storage systems common to both services.

We will refer to [15] for FGIR test results, and to Figures 5.2 and 5.3 for VISOR single upload and download tests, respectively. When looking at 1GB images single uploads to Cumulus, the FGIR takes $\approx$ 40s (seconds) to complete the process, with VISOR outperforming it with only 32s (20% less). For 2GB images, FGIR takes $\approx$ 70s, with VISOR taking only 65s ($\approx$ 7% less). When looking at single download requests the VISOR performance stands out, as for 1GB images, the FGIR spends $\approx$ 45s while VISOR spends only 26s ($\approx$ 42% less). Lastly, for 2GB images, FGIR takes $\approx$ 75s to download them while VISOR takes only 53s ($\approx$ 30% less). As already stated, the sixteen concurrent uploads tests are not comparable. Although, such VIS results (Figure 5.6) are not that far from the ones achieved by FGIR [15]. Even that in such case, our testing client hosts were four times busier than the ones at FGIR tests (as we have four client threads per machine).

These results are very encouraging, even more if we take into account the huge disparity between both test beds resources, with FGIR tests being conducted on the FG Sierra supercomputer (at University of California) machines [15]. Furthermore, in overall, we have seen VISOR tests showing much more stable and predictable response times. During tests, VISOR has handled Terabytes of images being transferred between endpoints, without any notorious signs of slowness or service outages.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This dissertation describes the VISOR image management service, a solution to exploit the need to efficiently manage and maintain a large number of VM images across heterogeneous cloud IaaSs and their storage system solutions. VISOR also addresses the Cloud Computing interoperability limitations, which remain an open research issue [10, 11, 12], by providing compatibility with multiple cloud IaaS storage solutions, database systems and communication formats.

For the service design we have relied on the presented analysis of the most suitable architectural models in order to achieve high performance without compromising reliability. Considering our research and the service specific use case, we have chosen to address its development relying on the event-driven architecture and RESTful Web services. The service design was also conducted in order to address the need for a highly modular, isolated and customizable software stack. Aiming to provide a service as much flexible as possible, we have heavily relied on the principle of abstraction layers. Through such abstraction layers, it was possible to isolate the service compatibility with multiple cloud storage systems (currently Cumulus, Walrus, HDFS, S3, LCS, HTTP read-only and the local filesystem) and database systems (currently MySQL and MongoDB) from the service core features. Therefore, such compatibilities were implemented as individual plugins. We have also shifted the data communication formats support (currently JSON and XML) and authentication mechanisms from the service core to front-end middleware components. In this way, we have achieved a multi compatible service, while isolating such compatibilities from the service API. This makes it possible to improve the service compatibility with other storage systems and other data communication formats. It is also possible to integrate VISOR with custom authentication services besides the VAS (VISOR Auth System). All these customizations only require code level implementations outside the service core.

We have also benchmarked the proposed service in order to assess its performance, stability and resources usage rate. Therefore we have conducted a wide testing procedure addressing both single and concurrent image registering and retrieving requests. We have contemplated Cumulus, Walrus and HDFS storage systems, as equally the server local filesystem as image backends. From the obtained results, VISOR has shown encouraging performance results, even when compared to a production service like the FutureGrid image service [15]. Furthermore, regarding resources usage rate, we have observed only residual memory footprints in both clients and hosts, something that would not be possible without the service two-side streaming approach. Results have therefore justified the design options taken in VISOR's implementation. While benchmarking VISOR, we were also able to assess Cumulus, Walrus and HDFS storage systems performance indicators, something that we have not found in literature till date.

Finally, VISOR is an open source software with a community-driven development process. Therefore, anyone can learn how it works and publicly contribute with code and suggestions for further improvements, or privately customize it to address its particular needs. All the service code and documentation is exposed through its source code repository and the project home page at `http://www.cvisor.org`.

6.2 Future Work

As future work, our service has many ways to be further enhanced. We can group such improvements in three categories: identified issues pursuit, service features and other improvements.

The first topic requiring attention is the constraint observed with the local filesystem backend. When retrieving images stored in the filesystem backend, the streaming process is currently being blocked, which incurs in latency and memory caching. However, this is not a VISOR related issue but rather an EventMachine [139] library limitation. Since EventMachine does not integrate non-blocking/asynchronous filesystem primitives, a solution may be to defer such operations to a background process. However, such approach requires engineering concerns regarding operations synchronization, as a deferred operation gets out of the program control flow. Another solution would be to manually pause and resume the reading of an image file from the local filesystem chunk by chunk in a non-blocking fashion. However, this would transfer the operations pause and resume control from the event-driven framework duties to programmer's concerns. Therefore this is an open research challenge to be tackled as future work.

Besides I/O architectural concerns, there are also some service level improvements that we would like to address in future work. The improvement of users management through roles and groups membership like done by OpenStack Glance [56] would be an interesting feature. The caching of the most requested images like done by Glance and IBM Mirage [9] would also be an interesting feature to reduce the VIS server overhead. Furthermore, the scheduling of VM images deletion through a garbage collection mechanism like done by Glance and Mirage would also become useful to provide the ability to cancel accidental image deletion requests. Security improvements for VISOR are also an important topic to address in future work.

Besides these specific topics, the development of the prototype VWS (VISOR Web System) and the further expansion of the compatible cloud storage systems are also in our development plans. There is also the intention to assess how can be VISOR incorporated and used in another VM image management environments besides cloud IaaSs, thus expanding the service use cases.

Bibliography

[1] P. Mell and T. Grance, "The nist definition of cloud computing. recommendations of the national institute of standards and technology," *NIST Special Publication*, vol. 145, no. 6, pp. 1–2, 2011.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[3] Y. Jadeja and K. Modi, "Cloud computing - concepts, architecture and challenges," in *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, march 2012, pp. 877–880.

[4] S. Patidar, D. Rane, and P. Jain, "A survey paper on cloud computing," in *2012 Second International Conference on Advanced Computing & Communication Technologies*. IEEE, 2012, pp. 394–398.

[5] T. J. Bittman, "Server virtualization: One path that leads to cloud computing," *Gartner RAS Core Research Note G00171730*, 2009.

[6] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, "Opening black boxes: Using semantic information to combat virtual machine image sprawl," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2008, pp. 111–120.

[7] G. von Laszewski, G. Fox, F. Wang, A. Younge, A. Kulshrestha, G. Pike, W. Smith, J. Vockler, R. Figueiredo, J. Fortes *et al.*, "Design of the futuregrid experiment management framework," in *Gateway Computing Environments Workshop (GCE), 2010*. IEEE, 2010, pp. 1–10.

[8] J. Diaz, G. von Laszewski, F. Wang, A. Younge, and G. Fox, "Futuregrid image repository: A generic catalog and storage system for heterogeneous virtual machine images," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. IEEE, 2011, pp. 560–564.

[9] G. Ammons, V. Bala, T. Mummert, D. Reimer, and Z. Xiaolan, "Virtual machine images as structured data: The mirage image library," *HotCloud'11*, 2011.

[10] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009.

[11] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[12] W.-T. Tsai, X. Sun, and J. Balasooriya, "Service-oriented cloud computing architecture," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, april 2010, pp. 684–689.

[13] O. P. Leads. Opennebula: The open source toolkit for cloud computing. [Online]. Available: http://opennebula.org

[14] D. Milojičić, I. Llorente, and R. Montero, "Opennebula: A cloud management tool," *Internet Computing, IEEE*, vol. 15, no. 2, pp. 11–14, 2011.

[15] G. von Laszewski, J. Diaz, F. Wang, A. J. Younge, A. Kulshrestha, and G. Fox, "Towards generic futuregrid image management," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, ser. TG '11. New York, NY, USA: ACM, 2011, pp. 15:1–15:2.

[16] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow, "Blueprint for the intercloud-protocols and formats for cloud computing interoperability," in *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*. Ieee, 2009, pp. 328–336.

[17] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala, "Always up-to-date: scalable offline patching of vm images in a compute cloud," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 377–386.

[18] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, "Managing security of virtual machine images in a cloud environment," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, ser. CCSW '09. New York, NY, USA: ACM, 2009, pp. 91–96.

[19] Y. Chen, T. Wo, and J. Li, "An efficient resource management system for on-line virtual cluster provision," in *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*. IEEE, 2009, pp. 72–79.

[20] T. Metsch, "Open cloud computing interface-use cases and requirements for a cloud api," in *Open Grid Forum, GDF-I*, vol. 162, 2010.

[21] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath, "Image distribution mechanisms in large scale cloud providers," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 112–117.

[22] J. Pereira and P. Prata, "Visor: Virtual images management service for cloud infrastructures," in *The 2nd International Conference on Cloud Computing and Services Science, CLOSER*, 2012, pp. 401–406.

[23] Lunacloud. Lunacloud: cloud hosting, cloud servers and cloud storage. [Online]. Available: http://lunacloud.com

[24] Y. Matsumoto. Ruby programming language. [Online]. Available: http://www.ruby-lang.org/en/

[25] B. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*. IEEE, 2009, pp. 44–51.

[26] M. Mahjoub, A. Mdhaffar, R. Halima, and M. Jmaiel, "A comparative study of the current cloud computing technologies and offers," in *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, nov. 2011, pp. 131–134.

[27] S. Wind, “Open source cloud computing management platforms: Introduction, comparison, and recommendations for implementation,” in *Open Systems (ICOS), 2011 IEEE Conference on*, sept. 2011, pp. 175–179.

[28] P. Sempolinski and D. Thain, “A comparison and critique of eucalyptus, opennebula and nimbus,” in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. Ieee, 2010, pp. 417–426.

[29] D. Ogrizovic, B. Svilicic, and E. Tijan, “Open source science clouds,” in *MIPRO, 2010 Proceedings of the 33rd International Convention*. IEEE, 2010, pp. 1189–1192.

[30] J. Peng, X. Zhang, Z. Lei, B. Zhang, W. Zhang, and Q. Li, “Comparison of several cloud computing platforms,” in *Information Science and Engineering (ISISE), 2009 Second International Symposium on*, dec. 2009, pp. 23–27.

[31] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[32] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the linux virtual machine monitor,” in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[33] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehörster, and A. Brinkmann, “Non-intrusive virtualization management using libvirt,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 574–579.

[34] Amazon. Amazon web services. [Online]. Available: http://aws.amazon.com/

[35] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. (2000) Simple object access protocol (soap) 1.1, w3c note 08 may 2000. [Online]. Available: http://www.w3.org/TR/soap/

[36] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[37] Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. (1999) Rfc 2616: Hypertext transfer protocol, http 1.1. [Online]. Available: http://www.ietf.org/rfc/rfc2616.txt

[38] Amazon. Amazon elastic compute cloud (amazon ec2). [Online]. Available: http://aws.amazon.com/ec2/

[39] Microsoft. Remote desktop protocol: Basic connectivity and graphics remoting specification. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc240445

[40] Amazon. Amazon elastic block store (amazon ebs). [Online]. Available: http://aws.amazon.com/ebs/

[41] ——. Amazon simple storage service (amazon s3). [Online]. Available: http://aws.amazon.com/s3

[42] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[43] E. Mocanu, M. Andreica, and N. Tapus, “Current cloud technologies overview,” in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2011 International Conference on*, oct. 2011, pp. 289–294.

[44] Eucalyptus. Eucalyptus infrastructure as a service. [Online]. Available: http://eucalyptus.com

[45] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, “The eucalyptus open-source cloud-computing system,” in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

[46] C. Kiddle and T. Tan, “An assessment of eucalyptus version 1.4,” 2009.

[47] B. Sotomayor, R. Montero, I. Llorente, and I. Foster, “Virtual infrastructure management in private and hybrid clouds,” *Internet Computing, IEEE*, vol. 13, no. 5, pp. 14–22, 2009.

[48] O. G. Forum. Occi: Open cloud computing interface. [Online]. Available: http://occi-wg.org

[49] U. of Chicago. Nimbus: Cloud computing for science. [Online]. Available: http://nimbusproject.org

[50] J. Bresnahan, D. LaBissoniere, T. Freeman, and K. Keahey, “Cumulus: an open source storage cloud for science,” in *Proceedings of the 2nd international workshop on Scientific cloud computing*. ACM, 2011, pp. 25–32.

[51] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[52] Apache. Hadoop: Software framework for distributed computing. [Online]. Available: http://hadoop.apache.org/

[53] S. Ghemawat, H. Gobioff, and S. Leung, “The google file system,” in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[54] Rackspace and NASA. Openstack: Open source software for building private and public clouds. [Online]. Available: http://openstack.org

[55] OpenStack. Openstack object storage developer documentation. [Online]. Available: http://swift.openstack.org/

[56] ——. Openstack image service developer documentation. [Online]. Available: http://glance.openstack.org/

[57] ——. Glance developer api guide. [Online]. Available: http://docs.openstack.org/api/openstack-image-service/1.0/content/

[58] D. Crockford. (2006) Rfc 4627: The application/json media type for javascript object notation (json). [Online]. Available: http://tools.ietf.org/html/rfc4627.txt

[59] W3C. (2008) The extensible markup language (xml) 1.0 (fifth edition), w3c recommendation. [Online]. Available: http://www.w3.org/TR/xml/

[60] FutureGrid. Futuregrid web portal. [Online]. Available: http://portal.futuregrid.org/

[61] U. of Southern California. Pegasus wms: Workflow management system. [Online]. Available: http://pegasus.isi.edu/

[62] I. University. Twister: Iterative mapreduce. [Online]. Available: http://www.iterativemapreduce.org/

[63] OpenStack. Swift: Openstack object storage. [Online]. Available: http://openstack.org/projects/storage/

[64] 10gen. Mongodb: scalable, high-performance, open source, document-oriented database. [Online]. Available: http://www.mongodb.org/

[65] K. Chodorow and M. Dirolf, *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[66] E. K. Zeilenga. (2006) Rfc 4510: Lightweight directory access protocol (ldap). [Online]. Available: http://tools.ietf.org/html/rfc4510

[67] 10gen. Gridfs: a specification for storing large files in mongodb. [Online]. Available: http://www.mongodb.org/display/DOCS/GridFS

[68] IBM. Ibm workload deployer (iwd). [Online]. Available: http://www-01.ibm.com/software/webservers/workload-deployer/

[69] K. Ryu, X. Zhang, G. Ammons, V. Bala, S. Berger, D. Da Silva, J. Doran, F. Franco, A. Karve, H. Lee *et al.*, "Rc2–a living lab for cloud computing," *Lisa'10: Proceedings of the 24th Large Installation System Administration*, 2010.

[70] M. Satyanarayanan, W. Richter, G. Ammons, J. Harkes, and A. Goode, "The case for content search of vm clouds," in *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*. IEEE, 2010, pp. 382–387.

[71] T. Ts'o and S. Tweedie, "Planned extensions to the linux ext2/ext3 filesystem," in *Proceedings of the Freenix Track: 2002 USENIX Annual Technical Conference*, 2002, pp. 235–244.

[72] R. Cattell, "Scalable sql and nosql data stores," *SIGMOD Rec.*, vol. 39, no. 4, pp. 12–27, 2011.

[73] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and et al., "Above the clouds: A berkeley view of cloud computing," *EECS Department University of California Berkeley Tech Rep UCBEECS200928*, p. 25, 2009.

[74] S. Rajan and A. Jairath, "Cloud computing: The fifth generation of computing," in *Communication Systems and Network Technologies (CSNT), 2011 International Conference on*, june 2011, pp. 665–667.

[75] M. Ahmed, A. Chowdhury, M. Ahmed, and M. Rafee, "An advanced survey on cloud computing and state-of-the-art research issues," 2012.

[76] N. Sadashiv and S. Kumar, "Cluster, grid and cloud computing: A detailed comparison," in *Computer Science & Education (ICCSE), 2011 6th International Conference on*. IEEE, 2011, pp. 477–482.

[77] K. Krauter, R. Buyya, and M. Maheswaran, "A taxonomy and survey of grid resource management systems for distributed computing," *Software: Practice and Experience*, vol. 32, no. 2, pp. 135–164, 2002.

[78] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *Security & Privacy, IEEE*, vol. 9, no. 2, pp. 50–57, 2011.

[79] W. Tsai, X. Sun, and J. Balasooriya, "Service-oriented cloud computing architecture," in *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*. IEEE, 2010, pp. 684–689.

[80] Rackspace. Rackcloud servers: Linux or windows server in minutes. [Online]. Available: http://www.rackspace.com/cloud/cloud_hosting_products/servers/

[81] T. Worldwide. Terremark: Information technology provider. [Online]. Available: http://www.terremark.com/

[82] GoGrid. Gogrid: Complex infrastructure made easy. [Online]. Available: http://www.gogrid.com/

[83] Hewlett-Packard. Hp cloud. [Online]. Available: http://www.hpcloud.com/

[84] Citrix. Cloudstack: Open source cloud computing. [Online]. Available: http://cloudstack.org/

[85] Salesforce.com. Force.com platform as a service. [Online]. Available: http://www.force.com/

[86] J. Lindenbaum, A. Wiggins, and O. Henry. Heroku platform as a service. [Online]. Available: http://www.heroku.com/

[87] Google. Google app engine cloud application platform. [Online]. Available: https://developers.google.com/appengine/

[88] Microsoft. Microsoft windows azure platform. [Online]. Available: http://www.windowsazure.com/

[89] VMware. Cloud foundry: deploy and scale your applications in seconds. [Online]. Available: http://cloudfoundry.com/

[90] RedHat. Openshift: free, auto-scaling platform as a service. [Online]. Available: https://openshift.redhat.com/app/

[91] Dropbox. Dropbox: file hosting service. [Online]. Available: http://dropbox.com/

[92] Google. Google apps. [Online]. Available: http://www.google.com/apps

[93] L. Wang, G. Von Laszewski, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu, "Cloud computing: a perspective study," *New Generation Computing*, vol. 28, no. 2, pp. 137–146, 2010.

[94] L. Wang, J. Tao, M. Kunze, A. Castellanos, D. Kramer, and W. Karl, "Scientific cloud computing: Early definition and experience," in *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on*. IEEE, 2008, pp. 825–830.

[95] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, October 2003.

[96] G. Dhaese and A. Bell. (1995) The iso 9660 file system. [Online]. Available: http://users.telenet.be/it3.consultants.bvba/handouts/ISO9960.html

[97] C. Tang, "Fvd: a high-performance virtual machine image format for cloud," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*. USENIX Association, 2011, pp. 18–18.

[98] IDC and EMC. (2011) The 2011 idc digital universe study. [Online]. Available: http://www.emc.com/collateral/about/news/idc-emc-digital-universe-2011-infographic.pdf

[99] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, "Cloud storage as the infrastructure of cloud computing," in *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*. IEEE, 2010, pp. 380–383.

[100] H. Dewan and R. Hansdah, "A survey of cloud storage facilities," in *Services (SERVICES), 2011 IEEE World Congress on*. IEEE, 2011, pp. 224–231.

[101] E. Mocanu, M. Andreica, and N. Tapus, "Current cloud technologies overview," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2011 International Conference on*. IEEE, 2011, pp. 289–294.

[102] Amazon. Amazon simple storage service api reference. [Online]. Available: http://awsdocs.s3.amazonaws.com/S3/20060301/s3-api-20060301.pdf

[103] ——. Amazon simpledb. [Online]. Available: http://aws.amazon.com/simpledb/

[104] Apache. Hadoop distributed file system. [Online]. Available: http://hadoop.apache.org/hdfs/

[105] O. Corporation. Mysql: The world's most popular open source database. [Online]. Available: http://www.mysql.com/

[106] Amazon. Relational database service (amazon rds). [Online]. Available: http://aws.amazon.com/rds/

[107] A. S. Foundation. Apache couchdb. [Online]. Available: http://couchdb.apache.org/

[108] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[109] W3C. (2004) Web services architecture, w3c working group note 11 february 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/

[110] M. Papazoglou and W. Van Den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB journal*, vol. 16, no. 3, pp. 389–415, 2007.

[111] P. Adamczyk, P. Smith, R. Johnson, and M. Hafiz, "Rest and web services: In theory and in practice," in *The 1st International Workshop on RESTful DesignInternational Workshop on RESTful Design*. Springer New York, 2011, pp. 35–57.

[112] R. Alarcon, E. Wilde, and J. Bellido, "Hypermedia-driven restful service composition," *Service-Oriented Computing*, pp. 111–120, 2011.

[113] P. Castillo, J. Bernier, M. Arenas, J. Merelo, and P. Garcia-Sanchez, "Soap vs rest: Comparing a master-slave ga implementation," *Arxiv preprint arXiv:1105.4978*, 2011.

[114] K. Lawrence, C. Kaler, A. Nadalin, R. Monzillo, and P. Hallam-Baker, "Web services security: Soap message security 1.1 (ws-security 2004)," *OASIS, OASIS Standard, Feb*, 2006.

[115] J. Meng, S. Mei, and Z. Yan, "Restful web services: A solution for distributed data integration," in *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*. IEEE, 2009, pp. 1–4.

[116] D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10.

[117] S. Pérez, F. Durao, S. Meliá, P. Dolog, and O. Díaz, "Restful, resource-oriented architectures: a model-driven approach," in *Web Information Systems Engineering-WISE 2010 Workshops*. Springer, 2011, pp. 282–294.

[118] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, 2007.

[119] S. Parastatidis, J. Webber, G. Silveira, and I. S. Robinson, "The role of hypermedia in distributed system development," in *Proceedings of the First International Workshop on RESTful Design*, ser. WS-REST '10. New York, NY, USA: ACM, 2010, pp. 16–22.

[120] S. Vinoski, "Rest eye for the soa guy," *IEEE Internet Computing*, vol. 11, pp. 82–84, January 2007.

[121] C. Pautasso and E. Wilde, "Restful web services: principles, patterns, emerging technologies," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 1359–1360.

[122] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. big'web services: making the right architectural decision," in *Proceeding of the 17th international conference on World Wide Web*. ACM, 2008, pp. 805–814.

[123] E. Rescorla. (2000) Rfc 2818: Http over tls. [Online]. Available: http://www.ietf.org/rfc/rfc2818.txt

[124] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. (1999) Rfc 2617: Http authentication: Basic and digest access authentication. [Online]. Available: http://www.ietf.org/rfc/rfc2617.txt

[125] WC3. Architecture of the world wide web, volume one. [Online]. Available: http://www.w3.org/TR/webarch/

[126] S. Vinoski, "Rpc and rest: Dilemma, disruption, and displacement," *IEEE Internet Computing*, vol. 12, pp. 92–95, September 2008.

[127] L. Soares and M. Stumm, "Exception-less system calls for event-driven servers," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 10–10.

[128] S. Tilkov and S. Vinoski, "Node. js: Using javascript to build high-performance network programs," *Internet Computing, IEEE*, vol. 14, no. 6, pp. 80–83, 2010.

[129] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris, “Event-driven programming for robust software,” in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ser. EW 10. New York, NY, USA: ACM, 2002, pp. 186–189.

[130] C. Flanagan and S. N. Freund, “Fasttrack: efficient and precise dynamic race detection,” *SIGPLAN Not.*, vol. 44, pp. 121–133, June 2009.

[131] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and precise datarace detection for multithreaded object-oriented programs,” *SIGPLAN Not.*, vol. 37, pp. 258–269, May 2002.

[132] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: a dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, pp. 391–411, November 1997.

[133] M. Welsh, S. Gribble, E. Brewer, and D. Culler, “A design framework for highly concurrent systems,” *University of California at Berkeley, Berkeley, CA*, 2000.

[134] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton, “Comparing the performance of web server architectures,” in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys ’07. New York, NY, USA: ACM, 2007, pp. 231–243.

[135] D. C. Schmidt, *Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 529–545.

[136] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley, 2000.

[137] Joyent. Node.js: Evented i/o for v8 javascript. [Online]. Available: http://nodejs.org

[138] G. Lefkowitz. Twisted: event-driven networking engine for python. [Online]. Available: http://twistedmatrix.com/trac/

[139] F. Cianfrocca. Eventmachine: fast, simple event-processing library for ruby programs. [Online]. Available: FrancisCianfrocca

[140] S. Hammond and D. Umphress, “Test driven development: the state of the practice,” in *Proceedings of the 50th Annual Southeast Regional Conference*. ACM, 2012, pp. 158–163.

[141] Leach, M. Mealling, and R. Salz. (2005) Rfc 4122: The universally unique identifier (uuid). [Online]. Available: http://tools.ietf.org/html/rfc4122.txt

[142] R. Rivest. (1992) Rfc 4627: The md5 message-digest algorithm. [Online]. Available: http://tools.ietf.org/html/rfc1321

[143] D. Stenberg. curl: Command line tool for transferring data with url syntax. [Online]. Available: http://curl.haxx.se/

[144] Amazon. (2006) Amazon simple storage service, developer guide, api version 2006-03-01. [Online]. Available: http://awsdocs.s3.amazonaws.com/S3/20060301/s3-dg-20060301.pdf

[145] D. Eastlake and P. Jones. (1992) Rfc 3174: Us secure hash algorithm 1 (sha1). [Online]. Available: http://tools.ietf.org/html/rfc3174

[146] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions: The hmac construction," *RSA Laboratories' CryptoBytes*, vol. 2, no. 1, pp. 12–15, 1996.

[147] S. Contini and Y. Yin, "Forgery and partial key-recovery attacks on hmac and nmac using hash collisions," *Advances in Cryptology-ASIACRYPT 2006*, pp. 37–53, 2006.

[148] J. Kim, A. Biryukov, B. Preneel, and S. Hong, "On the security of hmac and nmac based on haval, md4, md5, sha-0 and sha-1," *Security and Cryptography for Networks*, pp. 242–256, 2006.

[149] D. Thomas, C. Fowler, and A. Hunt, *Programming Ruby 1.9 (3rd edition): The Pragmatic Programmers' Guide*. Pragmatic Bookshelf, 2009.

[150] RedHat. Deltacloud: Many clouds, one api. [Online]. Available: http://deltacloud.apache.org/

[151] L. Segal. Yard: A ruby documentation tool. [Online]. Available: http://yardoc.org/

[152] D. Chelimsky. Rspec: Behaviour-driven development tool for ruby programmers. [Online]. Available: https://www.relishapp.com/rspec

[153] L. Torvalds. Git: free and open source distributed version control system. [Online]. Available: http://git-scm.com/

[154] PostRank. Goliath: open source non-blocking/asynchronous ruby web server framework. [Online]. Available: http://postrank-labs.github.com/goliath/

[155] B. Mizerany. Sinatra: A dsl for quickly creating web applications in ruby. [Online]. Available: http://www.sinatrarb.com/

[156] M.-A. Cournoyer. Thin: A fast and very simple ruby web server. [Online]. Available: http://code.macournoyer.com/thin/

[157] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, "Gridftp: Protocol extensions to ftp for the grid," *Global Grid ForumGFD-RP*, vol. 20, 2003.

[158] J. Shafer, S. Rixner, and A. Cox, "The hadoop distributed filesystem: Balancing portability and performance," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 122–133.

[159] K. Gilly, C. Juiz, and R. Puigjaner, "An up-to-date survey in web load balancing," *World Wide Web*, vol. 14, no. 2, pp. 105–131, 2011.

[160] Y. Teo and R. Ayani, "Comparison of load balancing strategies on cluster-based web servers," *Simulation*, vol. 77, no. 5-6, pp. 185–195, 2001.

[161] V. Ungureanu, B. Melamed, and M. Katehakis, "Effective load balancing for cluster-based servers employing job preemption," *Perform. Eval.*, vol. 65, no. 8, pp. 606–622, 2008.

[162] W. Tarreau. Haproxy: The reliable, high performance tcp/http load balancer. [Online]. Available: http://haproxy.1wt.eu/

[163] A. S. Foundation. Apache http server. [Online]. Available: http://httpd.apache.org/

[164] W. Reese, “Nginx: the high-performance web server and reverse proxy,” *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[165] V. Kaushal and M. Bala, “Autonomic fault tolerance using haproxy in cloud environment,” *International Journal of Advanced Engeneering Sciences and Technologies*, vol. 7, pp. 222–227, 2011.

[166] A. Bala and I. Chana, “Fault tolerance-challenges, techniques and implementation in cloud computing,” *International Journal of Computer Science Issues*, vol. 9, pp. 288–293, 2012.

[167] A. S. Foundation. Apachebench: Apache http server benchmarking tool. [Online]. Available: http://httpd.apache.org/docs/2.0/programs/ab.html

[168] O. Ben-Kiki, C. Evans, and B. Ingerson, “Yaml: Ain’t markup languageversion 1.1,” *Working Draft 2008-05*, vol. 11, 2001.

Appendix A

Installing and Configuring VISOR

In this appendix we provide a complete quick start guide to install and deploy VISOR. We will describe all the necessary installation and configuration procedures.

A.1 Deployment Environment

We will install VISOR by distributing its subsystems across three independent machines, with two host servers and one client. However, any other subsystems arrangement can be made by administrators (e.g. a subsystem per machine, all subsystems in the same machine). During the installation procedures we will always indicate in which machines a specific procedure should be reproduced. The deployment environment is pictured in Figure A.1.

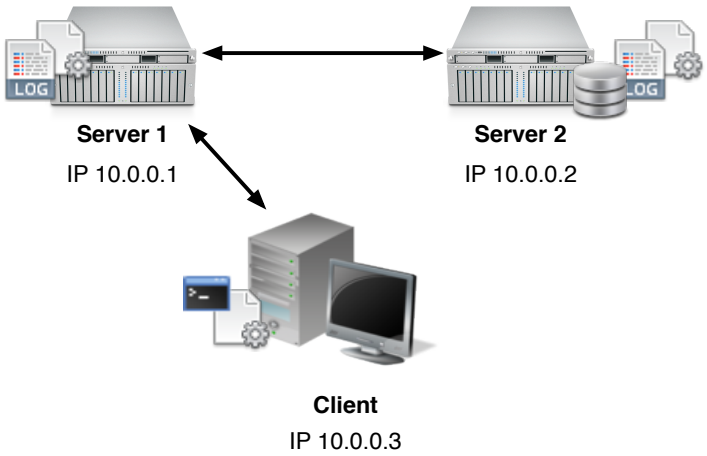

Figure A.1: The VISOR deployment environment with two servers and one client machines.

- **Server 1:** This machine will host the VISOR Image System (VIS), which is VISOR's core and client's front-end. The VIS server application will create a log file in which it will log operations. This machine will also comprise a VISOR configuration file, which will contain the necessary configuration options for customizing VIS.
- **Server 2:** This server will host both VISOR Meta System (VMS) and VISOR Auth System (VAS). Therefore, as they live in the same machine, they will use an underlying database to store both user accounts and image metadata. Both VMS and VAS will log to a local logging file. This server will host another VISOR configuration file which will contain the necessary parameters to configure both VMS and VAS.
- **Client:** The Client machine will host the VIS CLI, which will communicate with the VIS server hosted in Server 1. It will also contain a VISOR configuration file, including the necessary parameters to configure the VIS CLI.

A.2 Installing Dependencies

Before starting to install VISOR, we need to ensure that all required dependencies are properly installed and available in the deployment machines.

We will provide instructions tested on Ubuntu Server 12.04 LTS and Mac OSX Server 10.6 64-bit OSs. Instructions for installing these dependencies in other Unix-based OSs can be easily found among Internet resources.

A.2.1 Ruby

> ☞ These procedures should be reproduced in all machines (**Server 1**, **Server 2** and **Client**).

VISOR depends on the Ruby programming language [24], thus all machines used to host VISOR need to have the Ruby binaries installed. Since VISOR targets Unix systems, most up-to-date Linux and Mac OSX OSs are equipped with Ruby installed by default. However, VISOR requires Ruby to be at least in version 1.9.2. To ensure that host machines fulfil this requirement, users should open a terminal window and issue the following command ("prompt $>" indicates the terminal prompt position):

```
prompt $> ruby -v
```

If users' machines have Ruby installed, they should see a message displaying "ruby" followed by its version number. If receiving a "command not found" error, that machines do not have Ruby installed. If seeing a Ruby version lower than 1.9.2 or machines do not have Ruby installed at all, it should be installed as follows (depending on the used OS):

A.2.1.1 Ubuntu

```
prompt $> sudo apt-get update
prompt $> sudo apt-get install build-essential ruby1.9.3
```

A.2.1.2 Mac OSX

In Mac OSX, users should make sure that they have already installed Apple's Xcode (a developer library which should have come with Mac OSX installation disk) on machines before proceeding.

```
# First, install Homebrew, a free Mac OSX package manager
prompt $> /usr/bin/ruby -e "$(/usr/bin/curl -fsSL https://raw.github.com/mxcl\
/homebrew/master/Library/Contributions/install_homebrew.rb)"
# Now install Ruby with Homebrew
prompt $> brew install ruby
```

A.2.2 Database System

> ☞ These procedures should be reproduced in **Server 2**.

Since both VMS and VAS register data on a database, it is required to install a database system. Both VMS and VAS support MongoDB and MySQL databases, therefore it is user's responsibility to choose which one to install and use. Users can install either MongoDB or MySQL as follows:

A.2.2.1 Ubuntu

```
# Install MongoDB
prompt $> sudo apt-get install mongodb
# Or install MySQL
prompt $> sudo apt-get install mysql-server mysql-client libmysqlclient-dev
```

A.2.2.2 Mac OSX

```
# Install MongoDB
prompt $> brew install mongodb
# Or install MySQL
prompt $> brew install mysql
```

A.3 Configuring the VISOR Database

> ☞ These procedures should be reproduced in **Server 2**.

Now that all dependencies are satisfied, it is time to configure a database for VISOR. Users should follow these instructions if they have chosen either MongoDB or MySQL:

A.3.1 MongoDB

We just need to make sure that MongoDB was successfully installed, since MongoDB lets VISOR create a database automatically. Users should open a terminal window and type `mongo`:

```
prompt $> mongo
MongoDB shell version: 2.0.4
connecting to: test
```

If seeing something like the above output, MongoDB was successfully installed. Typing `exit` quits from the MongoDB shell. By default MongoDB does not have user's authentication enabled. For the sake of simplicity we will leave it that way. To configure an user account, one should follow the authentication tutorial in the MongoDB documentation [1].

[1] `http://www.mongodb.org/display/DOCS/Security+and+Authentication`

A.3.2 MySQL

If users have chosen to run VISOR backed by MySQL, they need to create and configure a database and an user account for it. To enter in the MySQL shell, the following command should be issued:

```
prompt $> mysql -u root
```

The following SQL queries should be used to create a database and an user account for VISOR. Users can provide a different database name, username (we will use ”visor” for both) and password (”passwd”), making sure to note those credentials as they will be further required:

```
CREATE DATABASE visor;
CREATE USER 'visor'@'localhost' IDENTIFIED BY 'passwd';
GRANT ALL PRIVILEGES ON *.* TO 'visor'@'localhost';
FLUSH PRIVILEGES;
```

If everything went without errors, we have already completed the database configurations (VISOR will handle tables creation). By typing `exit;` we will quit from the MySQL shell.

A.4 Installing VISOR

> ☞ From now on, all the presented commands are compatible with all popular Unix-based OSs, such as Ubuntu, Fedora, CentOS, RedHat, Mac OSX and others.

We have already prepared Server 1, Server 2 and Client machines to host VISOR. Thus, we can now download and install it. The VISOR service is currently distributed as a set of subsystems packaged in Ruby libraries, which are commonly known as *gems*. Therefore we will install each subsystem with a single command, downloading the required *gem* that will be automatically installed and configured.

A.4.1 VISOR Auth and Meta Systems

> ☞ These procedures should be reproduced in **Server 2**.

We will now install VAS and VMS subsystems in Server 2. To install these subsystems, users should issue the following command on a terminal window:

```
prompt $> sudo gem install visor-auth visor-meta
```

This command will automatically download, install and configure the last releases of the VAS and VMS from the Ruby gems on-line repository. During VAS and VMS installation, the VISOR Common System (VCS) will be automatically fetched and installed too (being `visor-common` gem), as all VISOR subsystems depend on it. After the installation completes, we will see a similar terminal output as the one below:

```
prompt $> sudo gem install visor-auth visor-meta
Successfully installed visor-common-0.0.2
***************************** VISOR *****************************
visor-auth was successfully installed!

Generate the VISOR configuration file for this machine (if not already done)
by running the 'visor-config' command.
*****************************************************************
Successfully installed visor-auth-0.0.2
***************************** VISOR *****************************
visor-meta was successfully installed!

Generate the VISOR configuration file for this machine (if not already done)
by running the 'visor-config' command.
*****************************************************************
Successfully installed visor-meta-0.0.2
prompt $>
```

As can be observed in the above output, both `visor-auth` and `visor-meta` were successfully installed, with `visor-common` being automatically installed prior to them. Both VAS and VMS display an informative message indicating that they were successfully installed, and that now the user should generate the VISOR configuration file for Server 2 machine.

A.4.1.1 Generating Server 2 Configuration File

To generate a template configuration file for the VAS and VMS host machine, the `visor-config` command should be used:

```
prompt $> visor-config

Generating VISOR configuration directories and files:

creating /Users/joaodrp/.visor...                  [DONE]
creating /Users/joaodrp/.visor/logs...             [DONE]
creating /Users/joaodrp/.visor/visor-config.yml... [DONE]

All configurations were successful. Now open and customize the VISOR
configuration file at /Users/joaodrp/.visor/visor-config.yml
prompt $>
```

As listed in the output above, the VISOR configuration file and directories were successfully generated. These include the YAML format [168] VISOR configuration file named *visor-config.yml*, the *logs/* directory to where both VAS and VMS servers will log, and the parent *.visor/* directory placed in the user's home folder, which in this case is */Users/joaodrp/*.

A.4.1.2 Customizing Server 2 Configuration File

The generated configuration file should now be opened and customized. The full generated configuration file template is listed in Appendix C. Here we will only address the parts of the configuration file that should be customized within the VMS and VAS host machine. The remain parameters can be leaved with their default values.

Bind Host Users should change the host address to bind the VAS and VMS servers through the `bind_host` parameters (lines 6 and 13) to their Server 2 IP address (which in our case is 10.0.0.2):

```
1  ...
2  # ================================ VISOR Auth ================================
3  visor_auth:
4  ...
5      # Address and port to bind the server
6      bind_host: 10.0.0.2
7      bind_port: 4566
8  ...
9  # ================================ VISOR Meta ================================
10 visor_meta:
11 ...
12     # Address and port to bind the server
13     bind_host: 10.0.0.2
14     bind_port: 4567
15 ...
```

Backend Users should also customize the `backend` option for both VAS and VMS by uncomment and customizing the lines for using either MongoDB or MySQL, depending on the already chosen database system back in Section A.3:

- If users have chosen to use MongoDB, and considering that it is listening on its default host and port address (127.0.0.1:27017), with no authentication and using *visor* as the database name, the `backend` option for both VAS and VMS should be set as follows:

```
1  ...
2  # ============================== VISOR Auth ==============================
3  visor_auth:
4  ...
5      # Backend connection string (backend://user:pass@host:port/database)
6      backend: mongodb://:@127.0.0.1:27017/visor
7  ...
8  # ============================== VISOR Meta ==============================
9  visor_meta:
10 ...
11     # Backend connection string (backend://user:pass@host:port/database)
12     backend: mongodb://:@127.0.0.1:27017/visor
13 ...
```

- If users have chosen MySQL, and considering that it is listening on its default host and port address (127.0.0.1:3306), the `backend` option for both VAS and VMS should be set (with user's credentials previously obtained in Section A.3) as follows:

```
...
# ============================= VISOR Auth ==================================
visor_auth:
...
    # Backend connection string (backend://user:pass@host:port/database)
    backend: mysql://visor:passwd@127.0.0.1:3306/visor
...
# ============================= VISOR Meta ==================================
visor_meta:
...
    # Backend connection string (backend://user:pass@host:port/database)
    backend: mysql://visor:passwd@127.0.0.1:3306/visor
...
```

Users should make sure to provide the *username*, *password* and database *name* previously obtained in Section A.3, then saving the configuration file.

A.4.1.3 Starting VISOR Auth System

After completed all configurations, we can now launch the VAS server. Users should open a new terminal window (**keeping it open during the rest of this guide**) and use the following command:

```
prompt $> visor-auth start -d -f
[2012-06-14 13:04:15] INFO - Starting visor-auth at 10.0.0.2:4566
[2012-06-14 13:04:15] DEBUG - Configs /Users/joaodrp/.visor/visor-config.yml:
[2012-06-14 13:04:15] DEBUG - ************************************************
[2012-06-14 13:04:15] DEBUG - log_datetime_format: %Y-%m-%d %H:%M:%S
[2012-06-14 13:04:15] DEBUG - log_path: ~/.visor/logs
[2012-06-14 13:04:15] DEBUG - bind_host: 10.0.0.2
[2012-06-14 13:04:15] DEBUG - bind_port: 4566
[2012-06-14 13:04:15] DEBUG - backend: mongodb://:@127.0.0.1:27017/visor
[2012-06-14 13:04:15] DEBUG - log_file: visor-auth-server.log
[2012-06-14 13:04:15] DEBUG - log_level: INFO
[2012-06-14 13:04:15] DEBUG - ************************************************
[2012-06-14 13:04:15] DEBUG - Configurations passed from visor-auth CLI:
[2012-06-14 13:04:15] DEBUG - ************************************************
[2012-06-14 13:04:15] DEBUG - debug: true
[2012-06-14 13:04:15] DEBUG - foreground: true
[2012-06-14 13:04:15] DEBUG - ************************************************
```

In the above output we have started the VAS server in debug mode. We have also started it in foreground, therefore the process will remain yielding logging output to the terminal. If wanting to start it in background (daemon process), it can be done by omitting the `-f` flag:

```
prompt $> visor-auth start
Starting visor-auth at 10.0.0.2:4566
prompt $>
```

To stop the VAS when it was started as a daemon process, the `stop` command should be used:

```
prompt $> visor-auth stop
Stopping visor-auth with PID: 41466 Signal: INT
```

In this case, the VAS server process was running with the identifier (PID) 41466 and was killed using a system interrupt (INT). Passing the `-h` option to `visor-auth` displays an help message:

```
prompt $> visor-auth -h
Usage: visor-auth [OPTIONS] COMMAND

Commands:
     start        start the server
     stop         stop the server
     restart      restart the server
     status       current server status

Options:
    -c, --config FILE               Load a custom configuration file
    -a, --address HOST              Bind to HOST address
    -p, --port PORT                 Bind to PORT number
    -e, --env ENV                   Set execution environment
    -f, --foreground                Do not daemonize, run in foreground

Common options:
    -d, --debug                     Enable debugging
    -h, --help                      Show this help message
    -v, --version                   Show version
prompt $>
```

If users have stopped VAS during the above examples, they should open a terminal window (**keeping it open during the rest of this guide**) and start it again:

```
1 prompt $> visor-auth start -d -f
```

☞ All the above operations on how to manage the VAS server apply to all VISOR subsystems server's management. The only difference is the command name. For managing VIS it is `visor-image`, for VMS it is `visor-meta` and for VAS it is the `visor-auth` command.

A.4.1.4 Generating an User Account

In order to authenticate against VISOR, one should first create an user account. This is done in VAS, using the `visor-admin` command. On a new terminal window, users can see an help message on how to use `visor-admin` by calling it with the `-h` parameter:

```
prompt $> visor-admin -h
Usage: visor-admin <command> [options]

Commands:
     list           Show all registered users
     get            Show a specific user
     add            Register a new user
     update         Update an user
     delete         Delete an user
     clean          Delete all users
     help <cmd>     Show help message for one of the above commands

Options:
    -a, --access KEY                The user access key (username)
    -e, --email ADDRESS             The user email address
    -q, --query QUERY               HTTP query like string to filter results

Common options:
    -v, --verbose                   Enable verbose
    -h, --help                      Show this help message
    -V, --version                   Show version
prompt $>
```

It is also possible to ask for a detailed help message for a given command. For example, to know more about how to *add* a new user, the following command can be used:

```
prompt $> visor-admin help add
Usage: visor-admin add <ATTRIBUTES> [options]

Add a new user, providing its attributes.

The following attributes can be specified as key/value pairs:

  access_key: The wanted user access key (username)
       email: The user email address

Examples:
  $ visor-admin add access_key=foo email=foo@bar.com
prompt $>
```

We will follow the above example to add a new user account for user 'foo':

```
prompt $> visor-admin add access_key=foo email=foo@bar.com
Successfully added new user with access key 'foo'.

        ID: 8a65ab69-59b3-4efc-859a-200e6341786e
ACCESS_KEY: foo
SECRET_KEY: P1qGJkJqWNEwwpSyWbh4cUljxkxbdTwen6m/pwF2
     EMAIL: foo@bar.com
CREATED_AT: 2012-06-10 16:31:01 UTC
prompt $>
```

Users should make sure to note the generated user credential (access key and secret key) somewhere, as they will be further required to configure the Client machine.

A.4.1.5 Starting VISOR Meta System

To start the VMS, user should open a new terminal window (**keeping it open during the rest of this guide**) and use the `visor-meta` command:

```
prompt $> visor-meta start -d -f
```

Now we have finished both VAS and VMS configurations, and their servers are up and running.

A.4.2 VISOR Image System

> ☞ These procedures should be reproduced in **Server 1**.

We will now install the VIS subsystem. Users should open a terminal window on Server 1 and issue the following command:

```
prompt $> sudo gem install visor-image
```

This command will automatically download, install and configure the last releases of the VIS subsystem from the Ruby gems on-line repository. During VIS installation, and as for VAS and VMS installation, the VCS subsystem will be automatically downloaded and installed.

```
prompt $> sudo gem install visor-image
Successfully installed visor-common-0.0.2
****************************** VISOR ******************************
visor-image was successfully installed!

Generate the VISOR configuration file for this machine (if not already done)
by running the 'visor-config' command.
*******************************************************************
```

```
Successfully installed visor-image-0.0.2
prompt $>
```

As observed in the above output, `visor-common` and `visor-image` were successfully installed. VIS displays an informative message indicating that it was successfully installed and now the user should generate a VISOR configuration file for the Server 1 machine.

A.4.2.1 Generating Server 1 Configuration File

We need to generate a configuration file for Server 1 machine in order to customize the VIS. To generate a template configuration file (as done previously for Server 2 in Section A.4.1.1) the `visor-config` command should be used:

```
prompt $> visor-config
```

A.4.2.2 Customizing Server 1 Configuration File

The generated configuration file should now be opened and customized. Here we will only address the parts of the configuration file that should be customized within the VIS host machine.

Bind Host Users should change the host address to bind the VIS server (line 6) to their Server 1 IP address, which in our case is 10.0.0.1:

```
1 ...
2 # ================================ VISOR Image ================================
3 visor_image:
4 ...
5     # Address and port to bind the server
6     bind_host: 10.0.0.1
7     bind_port: 4568
8 ...
```

VISOR Meta and Auth Systems Location Since VIS needs to communicate with the VMS and VAS, users should indicate in the Server 1 configuration file what is the Server 2 IP address, and the ports where VMS and VAS servers are listening for incoming requests:

```
1  ...
2  # ================================ VISOR Auth ================================
3  visor_auth:
4  ...
5      # Address and port to bind the server
6      bind_host: 10.0.0.2
7      bind_port: 4566
8  ...
9  # ================================ VISOR Meta ================================
10 visor_meta:
```

```
11 ...
12     # Address and port to bind the server
13     bind_host: 10.0.0.2
14     bind_port: 4567
15 ...
```

In our case, Server 2 (which is the host of VMS and VAS) has the IP address 10.0.0.2. VMS and VAS were started in the default ports (4566 and 4567 respectively). Users should change the above addresses (lines 6 and 13) to their Server 2 real IP address. Equally, if they have deployed VMS and VAS in different ports, they should also change them (lines 7 and 14).

Storage Backends Besides the VIS server, it is also needed to pay attention to the image storage backends configuration. The output below contains the excerpt of the configuration file that should be addressed to customize the storage backends:

```
1  ...
2  # ============================ VISOR Image Backends ============================
3  visor_store:
4      # Default store (available: s3, lcs, cumulus, walrus, hdfs, file)
5      default: file
6      #
7      # FileSystem store backend (file) settings
8      #
9      file:
10         # Default directory to store image files in
11         directory: ~/VMs/
12     #
13     # Amazon S3 store backend (s3) settings
14     #
15     s3:
16         # The bucket to store images in, make sure it exists on S3
17         bucket:
18         # Access and secret key credentials, grab yours on your AWS account
19         access_key:
20         secret_key:
21     #
22     # Lunacloud LCS store backend (lcs) settings
23     #
24     lcs:
25         # The bucket to store images in, make sure it exists on LCS
26         bucket:
27         # Access and secret key credentials, grab yours within Lunacloud
28         access_key:
29         secret_key:
30     #
31     # Nimbus Cumulus store backend (cumulus) settings
32     #
```

```
33      cumulus:
34          # The Cumulus host address and port number
35          host:
36          port:
37          # The bucket to store images in, make sure it exists on Cumulus
38          bucket:
39          # Access and secret key credentials, grab yours within Nimbus
40          access_key:
41          secret_key:
42      #
43      # Eucalyptus Walrus store backend (walrus) settings
44      #
45      walrus:
46          # The Walrus host address and port number
47          host:
48          port:
49          # The bucket to store images in, make sure it exists on Walrus
50          bucket:
51          # Access and secret key credentials, grab yours within Eucalyptus
52          access_key:
53          secret_key:
54      #
55      # Apache Hadoop HDFS store backend (hdfs) settings
56      #
57      hdfs:
58          # The HDFS host address and port number
59          host:
60          port:
61          # The bucket to store images in
62          bucket:
63          # Access credentials, grab yours within Hadoop
64          username:
```

The configuration file contains configurations for all available storage backends, being the local filesystem, Amazon S3, Nimbus Cumulus, Eucalyptus Walrus, Lunacloud LCS and Hadoop HDFS. Users should fill the attributes of a given storage backend in order to be able to store and retrieve images from it. User's credentials should be obtained within each storage system.

- In line 5 it is defined the storage backend that VIS should use by default to store images. Line 11 describes the path to the folder where images should be saved when using the filesystem backend. This folder will be creation by the VIS server if it do not exists.

- For S3 and LCS, users need to provide the bucket name in which images should be stored, and their access and secret keys used to authenticate against S3 or LCS, respectively.

- Cumulus, Walrus and HDFS configurations are similar. Users should provide the host address and port where these storage services are listening in. For Cumulus and Walrus they should

also provide the access and secret key credentials. For HDFS users should provide their username in Hadoop.

A.4.2.3 Starting VISOR Image System

After customizing the VIS configuration file, users should open a new terminal window (**keeping it open during the rest of this guide**) and launch the VIS server with the `visor-image` command:

```
prompt $> visor-image start -d -f
[INFO] 2012-06-14 14:10:57 :: Starting visor-image at 10.0.0.1:4568
[DEBUG] 2012-06-14 14:10:57 :: Configs /Users/joaodrp/.visor/visor-config.yml:
[DEBUG] 2012-06-14 14:10:57 :: ************************************************
[DEBUG] 2012-06-14 14:10:57 :: log_datetime_format: %Y-%m-%d %H:%M:%S
[DEBUG] 2012-06-14 14:10:57 :: log_path: ~/.visor/logs
[DEBUG] 2012-06-14 14:10:57 :: bind_host: 10.0.0.1
[DEBUG] 2012-06-14 14:10:57 :: bind_port: 4568
[DEBUG] 2012-06-14 14:10:57 :: log_file: visor-api-server.log
[DEBUG] 2012-06-14 14:10:57 :: log_level: INFO
[DEBUG] 2012-06-14 14:10:57 :: ************************************************
[DEBUG] 2012-06-14 14:10:57 :: Configurations passed from visor-image CLI:
[DEBUG] 2012-06-14 14:10:57 :: ************************************************
[DEBUG] 2012-06-14 14:10:57 :: debug: true
[DEBUG] 2012-06-14 14:10:57 :: daemonize: false
[DEBUG] 2012-06-14 14:10:57 :: ************************************************
```

Now we have finished the VIS configurations and its server is up and running.

A.4.3 VISOR Client

> ☞ These procedures should be reproduced in **Client** machine.

The VIS subsystem contains the VISOR client tools, thus we need to install it on Client machine by simply issuing the following command:

```
prompt $> sudo gem install visor-image
```

A.4.3.1 Generating Client Configuration File

We need to generate a configuration file for Client machine in order to customize the VISOR client tools. To generate a template configuration file (as done previously for Server 2 in Section A.4.1.1) use the `visor-config` command:

```
prompt $> visor-config
```

A.4.3.2 Customizing Client Configuration File

The generated configuration file should now be opened and customized. Here we will only address the parts of the configuration file that should be customized within the Client machine. The remain parameters can be leaved with their default values.

Bind Host We need to indicate where does the VIS CLI can find the VIS server. Therefore users should indicate in the configuration file the host address and the port number where the VIS server is listening. In our case it is 10.0.0.1:4568. Users should customize these attributes accordingly to the IP address and port number that they have used to deploy the VIS server:

```
...
# ================================ VISOR Image ================================
visor_image:
...
    # Address and port to bind the server
    bind_host: 10.0.0.1
    bind_port: 4568
...
```

User Credentials Users should fill the *access_key* and *secret_key* parameters with the credentials obtained by them previously in Section A.4.1.4. In our case, the obtained credentials were the following (make sure to fill the configuration file with your own credentials):

```
# ===== Default always loaded configuration throughout VISOR sub-systems ======
...
    # VISOR access and secret key credentials (from visor-admin command)
    access_key: foo
    secret_key: P1qGJkJqWNEwwpSyWbh4cUljxkxbdTwen6m/pwF2
...
```

We have finished all VISOR installation procedures. VAS, VMS and VIS servers should now be up and running in order to proceed with the usage examples described in the next appendix.

Appendix B

Using VISOR

In this appendix we will present some examples on how to use VISOR to manage VM images, using its main client tool: a CLI named `visor`. This CLI was already installed in the Client machine, previously configured in Chapter A.

> ☞ To use VISOR, examples in this chapter should be reproduced in the **Client** machine, previously configured in Chapter A.

The full syntax of the CLI commands was described in detail in Section 4.2.9.2. To see an help message about the client CLI, the `visor` command should be used with the `-h` option:

```
prompt $> visor -h
Usage: visor <command> [options]

Commands:
     brief      Show brief metadata of all public and user's private images
     detail     Show detailed metadata of all public and user's private images
     head       Show an image detailed metadata
     get        Retrieve an image metadata and file
     add        Add a new image metadata and optionally upload its file
     update     Update an image metadata and/or upload its file
     delete     Delete an image metadata and its file
     help       Show help message for one of the above commands

Options:
    -a, --address HOST      Address of the VISOR Image System server
    -p, --port PORT         Port where the VISOR Image System server listens
    -q, --query QUERY       HTTP query like string to filter results
    -s, --sort ATTRIBUTE    Attribute to sort results (default: _id)
    -d, --dir DIRECTION     Direction to sort results (asc/desc) (default: asc)
    -f, --file IMAGE        Image file path to upload
    -S, --save DIRECTORY    Directory to save downloaded image (default: './')

Common options:
    -v, --verbose            Enable verbose
    -h, --help               Show this help message
    -V, --version            Show version
prompt $>
```

B.1 Assumptions

We need some VM images to register in VISOR. Therefore, we assume that users have downloaded and placed the following sample images inside their home folder in the Client machine:

- **Fedora-17-x86_64-Live-Desktop.iso:** Fedora Desktop 17 64-bit VM image [1].
- **CentOS-6.2-i386-LiveCD.iso:** CentOS 6.2 32-bit VM image [2].

B.2 Help Message

For displaying a detailed help message for a specific command, we can use the *help* command, followed by a specific command name for which we want to see a help message:

```
prompt $> visor help add
Usage: visor add <ATTRIBUTES> [options]

Add new metadata and optionally upload the image file.
The following attributes can be specified as key/value pairs:

        name: The image name
architecture: The Image operating system architecture (available: i386 x86_64)
      access: If the image is public or private (available: public private)
      format: The image format (available: iso vhd vdi vmdk ami aki ari)
        type: The image type (available: kernel ramdisk machine)
       store: The storage backend (s3 lcs walrus cumulus hdfs http file)
    location: The location URI of the already somewhere stored image

Any other custom image property can be passed too as additional key/value pairs.

Provide the --file option with the path to the image to be uploaded and the
'store' attribute, defining the store where the image should be uploaded to.
prompt $>
```

B.3 Register an Image

B.3.1 Metadata Only

For registering only image metadata, without uploading or referencing an image file, users should use the command *add*, providing to it the image metadata as a set of key/value pairs arguments in any number, separated between them with a single space:

[1] http://download.fedoraproject.org/pub/fedora/linux/releases/17/Live/x86_64/Fedora-17-x86_64-Live-Desktop.iso

[2] http://mirrors.arsc.edu/centos/6.2/isos/i386/CentOS-6.2-i386-LiveCD.iso

```
prompt $> visor add name='CentOS 6.2' architecture='i386' format='iso' \
access='private'
Successfully added new metadata with ID 7583d669-8a65-41f1-b8ae-eb34ff6b322f.

         _ID: 7583d669-8a65-41f1-b8ae-eb34ff6b322f
         URI: http://10.0.0.1:4568/images/7583d669-8a65-41f1-b8ae-eb34ff6b322f
        NAME: CentOS 6.2
ARCHITECTURE: i386
      ACCESS: private
      STATUS: locked
      FORMAT: iso
  CREATED_AT: 2012-06-15 21:01:21 +0100
       OWNER: foo
prompt $>
```

As can be seen in the above example, we have registered the metadata of the CentOS 6.2 VM image. We have set its access permission to ”private”, thus only user ”foo” can see and modify it. Status is automatically set to ”locked”, since we have not uploaded or referenced its image file but only registered its metadata.

B.3.2 Upload Image

For registering and uploading an image file, users can issue the command *add*, providing to it the image metadata as a set of key/value pairs arguments, and the *--file* option, followed by the VM image file path:

```
prompt $> visor add name='Fedora Desktop 17' architecture='x86_64' \
format='iso' store='file' --file '~/Fedora-17-x86_64-Live-Desktop.iso'

Adding new metadata and uploading file...
Successfully added new image with ID e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.

         _ID: e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
         URI: http://10.0.0.1:4568/images/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
        NAME: Fedora Desktop 17
ARCHITECTURE: x86_64
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 676331520
       STORE: file
    LOCATION: file:///home/joaodrp/VMs/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.iso
  CREATED_AT: 2012-06-15 21:03:32 +0100
    CHECKSUM: 330dcb53f253acdf76431cecca0fefe7
       OWNER: foo
```

```
 UPLOADED_AT: 2012-06-15 21:03:50 +0100
prompt $>
```

B.3.3 Reference Image Location

If users want to reference an already somewhere stored image file, it can be done by including the *store* and *location* attributes, with the latter being set to the VM image file URI:

```
prompt $> visor add name='Ubuntu 12.04 Server' architecture='x86_64' \
format='iso' store='http' \
location='http://releases.ubuntu.com/12.04/ubuntu-12.04-desktop-amd64.iso'

Adding new metadata and uploading file...
Successfully added new metadata with ID edfa919a-0415-4d26-b54d-ae78ffc4dc79.

         _ID: edfa919a-0415-4d26-b54d-ae78ffc4dc79
         URI: http://10.0.0.1:4568/images/edfa919a-0415-4d26-b54d-ae78ffc4dc79
        NAME: Ubuntu 12.04 Server
ARCHITECTURE: x86_64
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 732213248
       STORE: http
    LOCATION: http://releases.ubuntu.com/12.04/ubuntu-12.04-desktop-amd64.iso
  CREATED_AT: 2012-06-15 21:05:20 +0100
    CHECKSUM: 140f3-2ba4b000-4be8328106940
       OWNER: foo
prompt $>
```

In the above example we have registered an Ubuntu Server 12.04 64-bit VM image, by referencing its location through a HTTP URL. As can be observed, VISOR was able to locate that image file and find its size and checksum through the URL resource HTTP headers.

B.4 Retrieve Image Metadata

B.4.1 Metadata Only

For retrieving an image metadata only, without the need to also download its file, users can use the *head* command, providing the image ID as first argument. The produced output is similar to that received when the image was registered in Section B.3.3.

```
prompt $> visor head e5fe8ea5-4704-48f1-905a-f5747cf8ba5e

         _ID: e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
         URI: http://10.0.0.1:4568/images/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
```

```
        NAME: Fedora Desktop 17
ARCHITECTURE: x86_64
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 676331520
       STORE: file
    LOCATION: file:///home/joaodrp/VMs/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.iso
  CREATED_AT: 2012-06-15 21:03:32 +0100
    CHECKSUM: 330dcb53f253acdf76431cecca0fefe7
       OWNER: foo
 UPLOADED_AT: 2012-06-15 21:03:50 +0100
```

B.4.2 Brief Metadata

For requesting the brief metadata of all public and user's private images, one can use the *brief* command:

```
prompt $> visor brief
Found 3 image records...
ID            NAME                  ARCHITECTURE  TYPE  FORMAT  STORE  SIZE
------------  --------------------  ------------  ----  ------  -----  ---------
e5fe8ea5...   Fedora Desktop 17     x86_64        -     iso     file   676331520
edfa919a...   Ubuntu 12.04 Server   x86_64        -     iso     http   732213248
7583d669...   CentOS 6.2            i386          -     iso     -      -
```

B.4.3 Detailed Metadata

For requesting the detailed metadata of all public and user's private images, one can use the *detail* command:

```
prompt $> visor detail
Found 3 image records...
--------------------------------------------------------------------------------
         _ID: e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
         URI: http://10.0.0.1:4568/images/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
        NAME: Fedora Desktop 17
ARCHITECTURE: x86_64
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 676331520
       STORE: file
    LOCATION: file:///home/joaodrp/VMs/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.iso
  CREATED_AT: 2012-06-15 21:03:32 +0100
    CHECKSUM: 330dcb53f253acdf76431cecca0fefe7
```

```
       OWNER: foo
 UPLOADED_AT: 2012-06-15 21:03:50 +0100
--------------------------------------------------------------------------------
         _ID: edfa919a-0415-4d26-b54d-ae78ffc4dc79
         URI: http://10.0.0.1:4568/images/edfa919a-0415-4d26-b54d-ae78ffc4dc79
        NAME: Ubuntu 12.04 Server
ARCHITECTURE: x86_64
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 732213248
       STORE: http
    LOCATION: http://releases.ubuntu.com/12.04/ubuntu-12.04-desktop-amd64.iso
  CREATED_AT: 2012-06-15 21:05:20 +0100
    CHECKSUM: 140f3-2ba4b000-4be8328106940
       OWNER: foo
--------------------------------------------------------------------------------
         _ID: 7583d669-8a65-41f1-b8ae-eb34ff6b322f
         URI: http://10.0.0.1:4568/images/7583d669-8a65-41f1-b8ae-eb34ff6b322f
        NAME: CentOS 6.2
ARCHITECTURE: i386
      ACCESS: private
      STATUS: locked
      FORMAT: iso
  CREATED_AT: 2012-06-15 21:01:21 +0100
       OWNER: foo
```

B.4.4 Filtering Results

It is also possible to filter results based in some query string. Such query string should conform to the HTTP query string format [37]. Thus, for example, if we want to get brief metatada of all 64-bit images stored in the HTTP backend only, we would do it as follows:

```
prompt $> visor brief --query 'architecture=x86_64&store=http'
Found 1 image records...
ID           NAME                  ARCHITECTURE  TYPE  FORMAT  STORE  SIZE
-----------  --------------------  ------------  ----  ------  -----  ---------
edfa919a...  Ubuntu 12.04 Server   x86_64        -     iso     http   732213248
```

B.5 Retrieve an Image

The ability to download an image file along with its metadata is exposed through the *get* command, providing to it the image ID string as first argument. If we do not want to save the image in the current directory, it is possible to provide the *--save* option, followed by the path to where we want to download the image.

```
prompt $> visor get e5fe8ea5-4704-48f1-905a-f5747cf8ba5e

         _ID: e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
         URI: http://10.0.0.1:4568/images/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e
        NAME: Fedora Desktop 17
ARCHITECTURE: x86_64
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 676331520
       STORE: file
    LOCATION: file:///home/joaodrp/VMs/e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.iso
  CREATED_AT: 2012-06-15 21:03:32 +0100
  UPDATED_AT: 2012-06-15 21:07:14 +0100
    CHECKSUM: 330dcb53f253acdf76431cecca0fefe7
       OWNER: foo
 UPLOADED_AT: 2012-06-15 21:03:50 +0100

Downloading image e5fe8ea5-4704-48f1-905a-f5747cf8ba5e...      | ETA:  --:--:--
Progress:      100% |=========================================| Time:   0:00:16
```

B.6 Update an Image

B.6.1 Metadata Only

For updating an image metadata, users can issue the command *update*, providing the image ID string as first argument, followed by any number of key/value pairs to update metadata. If wanting to receive the already updated metadata, the *-v* option should be passed:

```
prompt $>visor update edfa919a-0415-4d26-b54d-ae78ffc4dc79 name='Ubuntu 12.04' \
architecture='i386' -v

Successfully updated image edfa919a-0415-4d26-b54d-ae78ffc4dc79.

         _ID: edfa919a-0415-4d26-b54d-ae78ffc4dc79
         URI: http://10.0.0.1:4568/images/edfa919a-0415-4d26-b54d-ae78ffc4dc79
        NAME: Ubuntu 12.04
ARCHITECTURE: i386
      ACCESS: public
      STATUS: available
      FORMAT: iso
        SIZE: 732213248
       STORE: http
    LOCATION: http://releases.ubuntu.com/12.04/ubuntu-12.04-desktop-amd64.iso
  CREATED_AT: 2012-06-15 21:05:20 +0100
```

```
  UPDATED_AT: 2012-06-15 21:10:36 +0100
    CHECKSUM: 140f3-2ba4b000-4be8328106940
       OWNER: foo
```

B.6.2 Upload or Reference Image

If users want to upload or reference an image file to a registered metadata, it can be done by providing the *--file* option, or the *location* attribute, as done for the *add* command (Section B.3).

```
prompt $>visor update 7583d669-8a65-41f1-b8ae-eb34ff6b322f store='file' \
format='iso' --file '~/CentOS-6.2-i386-LiveCD.iso' -v

Updating metadata and uploading file...
Successfully updated and uploaded image 7583d669-8a65-41f1-b8ae-eb34ff6b322f.

         _ID: 7583d669-8a65-41f1-b8ae-eb34ff6b322f
         URI: http://10.0.0.1:4568/images/7583d669-8a65-41f1-b8ae-eb34ff6b322f
        NAME: CentOS 6.2
ARCHITECTURE: i386
      ACCESS: private
      STATUS: available
      FORMAT: iso
        SIZE: 729808896
       STORE: file
    LOCATION: file:///home/joaodrp/VMs/7583d669-8a65-41f1-b8ae-eb34ff6b322f.iso
  CREATED_AT: 2012-06-15 21:01:21 +0100
  UPDATED_AT: 2012-06-15 21:12:27 +0100
    CHECKSUM: 1b8441b6f4556be61c16d9750da42b3f
       OWNER: foo
prompt $>
```

B.7 Delete an Image

To receive as response the already deleted image metadata, the `-v` option should be used in the following *delete* command examples.

B.7.1 Delete a Single Image

To remove an image metadata along with its file (if any), we can use the *delete* command, followed by the image ID provided as its first argument:

```
prompt $> visor delete 7583d669-8a65-41f1-b8ae-eb34ff6b322f
```

```
Successfully deleted image 7583d669-8a65-41f1-b8ae-eb34ff6b322f.
prompt $>
```

B.7.2 Delete Multiple Images

It is also possible to remove more than one image at the same time, providing a set of IDs separated by a single space:

```
prompt $> visor delete e5fe8ea5-4704-48f1-905a-f5747cf8ba5e \
edfa919a-0415-4d26-b54d-ae78ffc4dc79

Successfully deleted image e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.
Successfully deleted image edfa919a-0415-4d26-b54d-ae78ffc4dc79.
prompt $>
```

It is also possible to remove images that match a given query with the *--query* option. The images removed in the example above, could have also been removed using a query to match 64-bit (x86_64) images, as they were the only ones in the repository with that architecture:

```
prompt $> visor delete --query 'architecture=x86_64'

Successfully deleted image e5fe8ea5-4704-48f1-905a-f5747cf8ba5e.
Successfully deleted image edfa919a-0415-4d26-b54d-ae78ffc4dc79.
prompt $>
```

Appendix C

VISOR Configuration File Template

This appendix lists the YAML-based [168] VISOR configuration file template, which is generated at VISOR subsystems installation time, using the `visor-config` command. Fields with empty values are those which should be necessarily set by users when needed.

```yaml
# ===== Default always loaded configuration throughout VISOR sub-systems ======
default: &default
    # Set the default log date time format
    log_datetime_format: "%Y-%m-%d %H:%M:%S"
    # Set the default log files directory path
    log_path: ~/.visor/logs
    # VISOR access and secret key credentials (from visor-admin command)
    access_key:
    secret_key:

# ================================ VISOR Auth ================================
visor_auth:
    # Merge default configurations
    <<: *default
    # Address and port to bind the server
    bind_host: 0.0.0.0
    bind_port: 4566
    # Backend connection string (backend://user:pass@host:port/database)
    #backend: mongodb://<user>:<password>@<host>:27017/visor
    #backend: mysql://<user>:<password>@<host>:3306/visor
    # Log file name (empty for STDOUT)
    log_file: visor-auth-server.log
    # Log level to start logging events (available: DEBUG, INFO)
    log_level: INFO

# ================================ VISOR Meta ================================
visor_meta:
    # Merge default configurations
    <<: *default
    # Address and port to bind the server
    bind_host: 0.0.0.0
    bind_port: 4567
    # Backend connection string (backend://user:pass@host:port/database)
    #backend: mongodb://<user>:<password>@<host>:27017/visor
    #backend: mysql://<user>:<password>@<host>:3306/visor
    # Log file name (empty for STDOUT)
```

```
    log_file: visor-meta-server.log
    # Log level to start logging events (available: DEBUG, INFO)
    log_level: INFO

# ================================ VISOR Image ================================
visor_image:
    # Merge default configurations
    <<: *default
    # Address and port to bind the server
    bind_host: 0.0.0.0
    bind_port: 4568
    # Log file name (empty for STDOUT)
    log_file: visor-api-server.log
    # Log level to start logging events (available: DEBUG, INFO)
    log_level: INFO

# ========================== VISOR Image Backends ============================
visor_store:
    # Default store (available: s3, lcs, cumulus, walrus, hdfs, file)
    default: file
    #
    # FileSystem store backend (file) settings
    #
    file:
        # Default directory to store image files in
        directory: ~/VMs/
    #
    # Amazon S3 store backend (s3) settings
    #
    s3:
        # The bucket to store images in, make sure it exists on S3
        bucket:
        # Access and secret key credentials, grab yours on your AWS account
        access_key:
        secret_key:
    #
    # Lunacloud LCS store backend (lcs) settings
    #
    lcs:
        # The bucket to store images in, make sure it exists on LCS
        bucket:
        # Access and secret key credentials, grab yours within Lunacloud
        access_key:
        secret_key:
    #
    # Nimbus Cumulus store backend (cumulus) settings
    #
```

```
    cumulus:
        # The Cumulus host address and port number
        host:
        port:
        # The bucket to store images in, make sure it exists on Cumulus
        bucket:
        # Access and secret key credentials, grab yours within Nimbus
        access_key:
        secret_key:
    #
    # Eucalyptus Walrus store backend (walrus) settings
    #
    walrus:
        # The Walrus host address and port number
        host:
        port:
        # The bucket to store images in, make sure it exists on Walrus
        bucket:
        # Access and secret key credentials, grab yours within Eucalyptus
        access_key:
        secret_key:
    #
    # Apache Hadoop HDFS store backend (hdfs) settings
    #
    hdfs:
        # The HDFS host address and port number
        host:
        port:
        # The bucket to store images in
        bucket:
        # Access credentials, grab yours within Hadoop
        username:
```