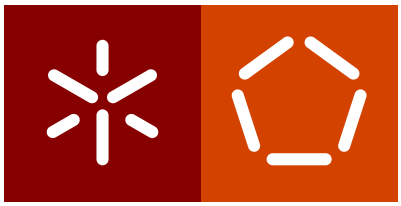
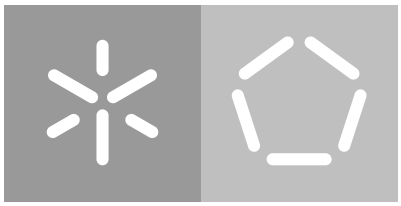


**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Rui Miguel Martins Ribeiro

# Common Infrastructure Provisioning

August 2017



**Universidade do Minho**
Escola de Engenharia
Departamento de Informática

Rui Miguel Martins Ribeiro

# Common Infrastructure Provisioning

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
**Professor António Luís Pinto Ferreira Sousa**

August 2017

## ABSTRACT

Provisioning commodity hardware used for scientific research while making it customizable and available for a large group of researchers is a process that requires automation. This dissertation describes the infrastructure, design and implementation of MOCAS and Bootler, an approach to management, allocation and provisioning of physical and virtual resources focused on enabling the users to remotely manage their nodes. MOCAS provides the necessary infrastructure and tools along with an appropriate web interface so researchers may lease bare metal resources and customize the full provisioning process, from installation to configuration without the need of specialized human-resources. Bootler, on the other hand, simplifies Virtual Machine (VM) life cycle management by providing a streamlined user interface and delegating VM scheduling to OpenStack. In this context, High-Assurance Software Laboratory (HASLab) researchers are now able to seemingly operate a 104 nodes (416 cores) commodity hardware cluster by leveraging the automation and abstractions these platforms provide.

## RESUMO

O aprovisionamento de hardware comum para ser utilizado por um vasto grupo investigadores no âmbito de investigação cientifica e ao mesmo tempo permitir personalização ao nível do sistema, é algo difícil de alcançar sem algum tipo de automação. Nesta dissertação descreve-se a infraestrutura, desenho e implementação de duas plataformas, MOCAS e Bootler, como proposta para gestão, alocação e aprovisionamento de sistemas físicos e virtuais, cujo foco principal é permitir que os utilizadores sejam capazes de gerir os seus próprios recursos remotamente. O MOCAS fornece toda a infraestrutura, bem como, um conjunto de ferramentas que se fazem acompanhar de uma interface web através da qual os investigadores podem, não só reservar recursos físicos, mas também, personalizar todo processo de aprovisionamento, desde a instalação até à configuração, sem necessidade de recorrer a recursos humanos especializados. Por outro lado, o Bootler, dinamiza a gestão do ciclo de vida de máquinas virtuais, para isso, recorre a uma interface web simplificada, através da qual se delega a instanciação dos recursos virtuais à plataforma OpenStack. Com recurso a estes processos de abstração e automação proporcionados por ambas plataformas, atualmente, os investigadores do HASLab têm a capacidade de operar de forma simplificada um cluster com 104 máquinas (416 *cores*) baseadas em hardware comum.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF LISTINGS

# 1

# INTRODUCTION

Research institutes make use of computing resources to run simulations in order to analyze, model or visualize datasets pertinent for a given research subject. These resources come with a cost in human-resources specialized in system administration which have to invest time managing cluster nodes. If we assume these resources are provisioned on a daily basis, because simulations require different software sets and clean slate machines, managing hundreds of resources manually becomes virtually impossible to maintain. To mitigate this problem we present two platforms, MOCAS and Bootler, an approach that not only helps automate provision tasks, but also, reduces specialized human-resources by shifting the provisioning responsibility and handling of bare metal or virtual resources to users with basic technical knowledge.

Managing a commodity hardware cluster comprised of more than one hundred heterogeneous machines dedicated to run computer science research simulations, where machines migrate from researcher to researcher as needed, is a task that can't be handled appropriately without automation. No automation means that common tasks such as installing an Operating System (OS) with required software and necessary configurations does not scale for large deployments Cons et al. (2000). For that same reason automation tools for system deployment, provisioning and configuration management are essential Tang et al. (2015) in dealing with infrastructure management. Nevertheless we can not expect researchers to be experts in system administration, or even, have them spending time with such side tasks.

In this type of environment, keeping a steady flow of ready and available computing resources is a requirement that is only met when three distinct tasks are automated. The first is system provisioning, second is resource allocation or lease to a given user, finally, the third, is enabling users to manage their own resources. Considering each simulation may require a clean slate environment, distinct system, software and configuration, being able to act upon assigned resources and streamline a given set of processes turns repetitive and time consuming tasks into optimized automated units of work.

This thesis describes the rationale, design and implementation of MOCAS and Bootler, two platforms created to simplify bare metal and VM provisioning, as well as managing VMs life cycle. By providing the user with the right tools and system abstractions we are able to migrate management and provisioning tasks to the users, providing them with

full autonomy to manage systems assigned to them. The shift of responsibilities reduces Information Technology (IT) staff tasks, while, at the same time, allowing users to have their systems ready without delay.

## 1.1 PROBLEM STATEMENT

Provisioning computing resources is a daily and common task for most System Administrators (SAs) or IT teams. Regardless of environment: small or medium business, enterprise, education or research laboratories. Each of them have one thing in common: resources to provide and manage. These resources come in different flavors, namely, hardware, virtualization or even appliances.

Let us settle on the meaning of provisioning, "the action of providing or supplying something for use". In the scope of this thesis we focus on the provisioning of computing resources, mainly hardware and virtualized machines with a base install of a given OS configured with a specific set of services or configurations.

For IT teams with infrastructure knowledge, specific training and the right tools, the process of installing and configuring is straightforward, but, for a less technical audience wanting to setup machines, services and re-roll previous configurations for testing purposes, it becomes a daunting task. Nevertheless, allocating an IT team for such a task could be a solution, but, it also means more expenses in manpower, which will increase with users demands.

The aforementioned problem can be mitigated by developing tools that allow oversight, pre-configuration and deployment by the IT team while allowing users to provision, configure and re-roll resources through seamless interaction with an uncluttered management interface that hides all the unnecessary setup and infrastructure knowledge.

Although tools like Cobbler or MaaS, which are covered in Chapter 2, provide similar functionality, we found that a modular homegrown system is easier to extend and maintain when specific edge cases arise.

## 1.2 OBJECTIVES

Computer science research demands physical and virtual resources in the form of computing power, storage and network. All these resources are volatile in regards to usage, when a researcher completes a given workload or simulation the resource becomes available for other uses, but, in a dirty state. Ideally, when a researcher inherits a resource there should be a way to easily make it pristine, because having previous configurations, services or data may disturb simulation results. In virtualized environments snapshots and base images en-

able us to return to previous states by simply discarding old data, but, when working with physical resources it becomes more complex and time consuming.

With the previous notions in mind we set to develop a platform that could aid in alleviating the burden of turning dirty resources into pristine ones. Solving the mentioned challenge is only part of the problem, since we also aim to develop a maintainable, decoupled and extensible platform in order to improve functionality over time.

From an infrastructure standpoint, we intend to build a modular system where each component is responsible for a single task or, several related tasks following a service oriented architecture (Rosen et al., 2012) where possible. To keep the infrastructure compatible we use proven and existing network services along with a set of stable libraries for development and service integration. As stated by (Smith, 2012) "throughout most of its life, software is being maintained", so, in order to alleviate the maintenance burden we intend to split the platform into manageable services, components and modules instead of following a monolithic approach that compromises maintainability.

Our top-level objective is to provide a frontend platform for easy resource allocation. The platform aims to enable on-demand creation and teardown of computational resources via a simplified web interface, but also, manage user access from an external location by identifying them with a set of credentials and enforcing Virtual Private Network (VPN) connection for resource access.

Ultimately, a user should have access to a set of pre-enabled services which will be described in this thesis. The main targets are a VPN service, allowing access to computational resources, virtual instances from a set of existing flavors and metal resources from a given pool. In the next sections we will succinctly describe each goal.

**Metal Allocation & Provisioning**

Provisioning bare metal machines from a pool of servers or commodity hardware was one of the first requirements for the platform. This feature will not only allow faster server deployments for new and old configurations, but also make it possible to quickly provision machines for testing purposes.

We break the process of provisioning in three steps: installation, configuration and environment setup, all of those, repetitive and time consuming tasks without the possibility of reuse when executed manually. Leveraging automated provisioning allows the system to quickly deliver reusable and ready to run resources without requiring much interaction or system knowledge from the user and, at the same time, reproducing a predictable and specific environment.

The process should be fully automated and unattended while keeping the user informed of machine state. When no existing remote control technologies are in place the system should rely and implement its own means to obtain that knowledge.

**Virtual Machine Provisioning**

There are plenty of technologies allowing VM management, orchestration and deployment (Armbrust et al., 2010; Wen et al., 2012; Barkat et al., 2014). Since our goal is to streamline allocation we use existing infrastructure tools as intermediaries, relying on them for VM and network management.

The platform should strive to simplify VM allocation only requesting the user to provide minimal information like hostname, flavor and credentials, while hiding virtual hardware details and network allocation requirements. Those would be sent as a template to the underlying VM manager.

This class of system relies on existing and legacy network technologies to fulfill its responsibilities, nevertheless it adds value by integrating a single Application Programming Interface (API), thus enabling support for integrating any management platform. A feature that would provide easy migration to other cloud platforms without the need to change our platform core.

**VPN Access**

Platform usage is much dependent on the setup, for our purposes having all computational resources in a private network is a requirement. Nevertheless, interaction with the User Interface (UI) may be done from a public network, only resources are private.

Providing visibility into the private network and allocated resources should be straightforward for most users.

The platform should provide the means for transparent VPN account creation upon registration. VPN management should also be available for each user in order to allow for account customization and configuration.

If the computational resources do not reside in a private network, this component should not interfere with the platform usage or setup.

**Hypervisor Independence**

The evolution of virtualization greatly revolves around the work on one piece of software, the hypervisor, also known as Virtual Machine Manager (VMM). It allows physical devices to share their hardware resources with VMs running as guests Sridhar (2009). In this sense a physical computer can be used to run multiple virtualized instances, each with its own OS and virtual hardware, CPU, memory, network, IO, all provided by the hypervisor.

Another important aspect about the hypervisor is the possibility to run guests without the need to modify their OS or applications. In all aspects the guests are unaware if the environment is virtualized or not, due to the hypervisor providing the same communication interface as a physical system.

It's important to note that there are several hypervisor categories, namely *Type 1* and *Type 2*. A *Type 1* hypervisor is implemented in bare metal, directly running over the hardware Sridhar (2009), while *Type 2* refers to having the hypervisor installed on top of the OS

like a normal piece of software. The clear difference between this two types of hypervisor is performance, since *Type 1* has a direct access to the hardware, while *Type 2* goes through an extra layer, the host OS.

In order to provide the ability of VM allocation regardless of hypervisor, the platform should provide or integrate a service that is able to communicate with a ubiquitous virtualization API. The hypervisor abstraction provided by it should enable us to build an extra VM provision component that communicates via an unified API.

Another goal with this implementation is providing a working system without the need of external platforms such as OpenStack OpenStack Project (2016b), which the infrastructure within HASLab currently depends on for serving VMs. Removing dependencies translates into less initial setup and infrastructure knowledge, on the other hand, reusing pre-existing dependencies facilitates integration. OpenStack is an open source cloud computing platform aimed at public and private clouds focused in providing Infrastructure as a Service (IaaS) Pepple (2011), since its inception in 2010 the project integrated multiple cloud services including object and block storage, networking, image and multiple others which are not relevant for the vanilla infrastructure.

Although our targets are Linux Linux Foundation (2016) and KVM Kernel Virtual Machine (2016), by using libvirt libvirt (2016) we get, with same API the ability to use other OSs and hypervisors, such as behyve BSD Hypervisor (2016) on FreeBSD FreeBSD Project (2016) or Hyper-V Microsoft Corporation (2016a) on Microsoft Windows Microsoft Corporation (2016b).

In the remainder of this dissertation, Chapter 2 goes through meaningful concepts and significant related work. Chapter 3 provides insight into the real world scenario that motivated the analysis, design and development of the platforms described in this contribution. The adopted approach regarding platform design is presented in Chapter 4, which is promptly followed by the implementation details in Chapter 5. Conclusion and ideas regarding future work follow in Chapter 6.

# 2

# STATE OF THE ART

Before the advent of virtualization systems provisioning was reserved for bare metal, namely, physical servers and desktops. Currently it expanded into virtualized environments and container based systems, turning provisioning into a commodity process, not only for production, testing and staging environments, but also, for local development environments. The importance of provisioning could be evaluated by merely looking at the variety of tools Cobbler Project (2016); Foreman Project (2016); Canonical (2016) developed to simplify the process, nevertheless, we can also observe real world usage for bigger Cons et al. (2000), smaller or individual Farrington (2016) deployments. We define provisioning as a process because it can be broken into three isolated and automated tasks, installation, configuration and setup.

Many tools, which are explored in this section, exist for automated systems installation and VM provisioning, each having different methods, strengths and weaknesses but usually targeting a single use case and sometimes compromising flexibility for convenience.

Building a platform for multiple use cases while targeting virtualization and infrastructure provision is not a novel approach. In fact, when properly configured, platforms like OpenStack OpenStack Project (2016b) or Foreman Foreman Project (2016) are able to provide many of the features we propose along this thesis, but currently, OpenStack most stable, mature and adopted features only target VMs. Although there is work being done regarding metal allocation in Project Ironic Kumar and Parashar (2010); OpenStack Project (2016a); Yamato et al. (2015) broad adoption is yet to be seen. On the other hand, Foreman features bare metal and VM lifecycle management, two of the most notable tasks we require, but lacks resources-to-user allocations.

In this section we present an overview on open source tools used for managed and unattended systems installs while providing an overview of most notable differences when compared to our proposal.

## 2.1 OPENSTACK

OpenStack is a collection of open source software projects that enterprises or cloud providers can use to setup and run their cloud infrastructure Wen et al. (2012). The project aims to build a cloud computing platform suitable for public and private clouds serving not only virtual resources, but also physical ones.

OpenStack follows a service oriented architecture comprised of multiple projects as depicted in Figure 1, each solving a specific problem. Most notably, and part of the base stack, compute (Nova), networking (Neutron), identity (Keystone) and other integration services. Bare metal provisioning is relatively new to OpenStack but, with Project Ironic OpenStack Project (2016a) it becomes a reality.

Project Ironic evolved in maturity but still lacks adoption, nevertheless it makes OpenStack the most complete IaaS offering, covering both aspects of provisioning, virtualization and bare metal. Architecture wise, the base system includes the following components:

**NOVA COMPUTE** In OpenStack, compute nodes are the physical resources that host guest VMs, run *Nova Compute*, a service that handles instances lifecycle executing tasks like spawning, scheduling and termination. The service is able to leverage a broad range of known hypervisors like KVM, QEMU, XEN, VMWare vSphere and Hyper-V, but also take advantage of container technology LCX.

**NEUTRON NETWORKING** The OpenStack network is managed by Neutron, a service that exposes an API for defining network connectivity and addressing in a cloud environment. The service handles the management and lifecycle of the virtual network infrastructure which includes all components such as routers, addresses, subnets and switches. The OpenStack network service is self contained, supporting multiple deployment approaches, standalone, collocated with the controller or leveraging high availability.

**KEYSTONE IDENTITY** The OpenStack identity service provides an entry point for the system regarding authentication, authorization and service discovery. Other services interact with it via a RESTful API Christensen (2009) in order to obtain access to the system or consult the service catalog, this approach abstracts communication interfaces to access the repositories where data is stored, simplifying inter service communication. For accessing repositories like relational databases or Lightweight Directory Access Protocol (LDAP) servers Keystone is equipped with drivers which understand the interface of a given repository rendering the identity service into a central hub that all other services may consult.

**GLANCE IMAGING** For handling OS images and snapshots OpenStack provides a service named Glance, like other services it operates via a RESTful API for managing disk or

server images which will later be served for compute instances. It also provides an additional metadata service in order to define image constraints and taxonomy on the resources it handles. This metadata is useful when compute nodes request an image with a set of characteristics for a given instance, since it allows for attribute filtering.

Glance is flexible enough to support multiple storage (Ceph, Swift, Local) repositories where images are dumped, but also multiple databases (MySQL, PostgreSQL, SQLite) that can be used for storing image characteristics in the form of metadata. Images can be saved in raw format or in more complex formats like qcow or qcow2.

CINDER BLOCK STORAGE By default, instances use the storage provided by the image file they boot from, this type of storage is volatile because once the instance is terminated the image file or files are also deleted. This may be solved by taking a snapshot of the instance prior to deletion, but Cinder provides a better way.

Cinder serves persistent block storage to instances allowing it to be used in multiples ways, provide a block for booting an instance, booting an instance from a previous snapshot or provide additional storage to an instance. Block storage may be attached or detached at any time and persisted after instance termination.

In the same line of Glance, Cinder supports and is able to manage multiple storage backends for serving blocks to instances, these are implemented as volume drivers. Being the most basic one the Logical Volume Manager (LVM). It can also integrate with Ceph, iSCSI and multiple others OpenStack Project (2017).

OpenStack is a big project comprised of many moving parts that require a stable setup, maintenance and monitoring. Due to its nature, the management interface is complex and exposes unnecessary clutter to the end-user. Our use case implies an unstable infrastructure based on commodity hardware, with that in mind we have less moving parts with simple configurations making debugging easier and, at the same time a more recoverable system.

Even if our target audience is not always the end-user, the interface should only present the absolute necessary for provisioning a resource. With that in mind our UI abstracts VM resources into pre-configured presets and automates network allocation for the user. These abstractions enables us to use it as a full solution or as a simple frontend for external APIs. While evaluating OpenStack as a complete solution to our problem, we verified that beyond the complex UI it doesn't handle resource leasing to users, regarding metal allocation we concluded that while Ironic matures and widens adoption our solution presents itself decoupled from other components and fulfills our objectives.

Nevertheless we found that OpenStack fits into our underlying architecture for several reasons, it's developed in Python, we already have it deployed in our infrastructure, it has broad adoption across industry and academia and although it provides many services, they are optional, which allows for maintaining a small required core.

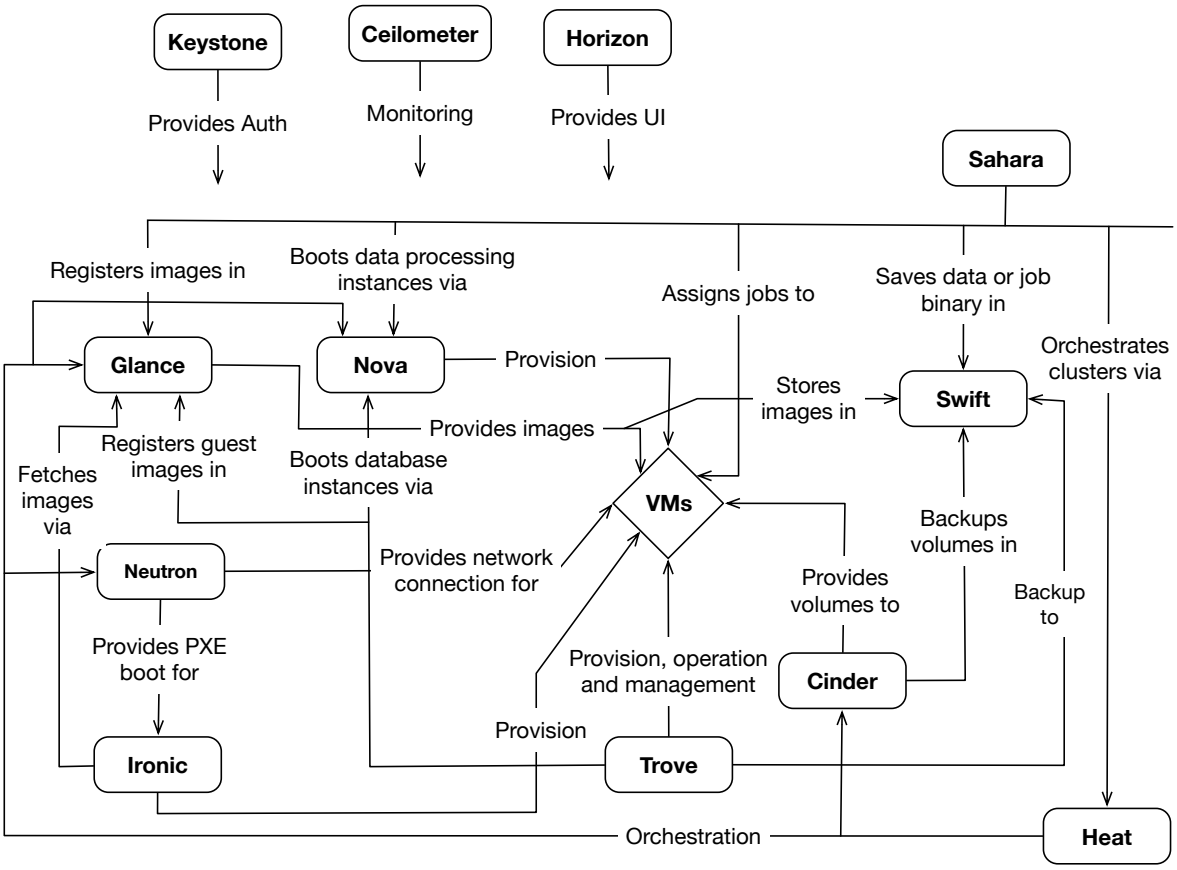



Figure 1.: OpenStack Conceptual Architecture

## 2.2 OPENNEBULA

OpenNebula started as a research project in 2005 and since its first public release in March 2008, it has evolved and matured not only as a platform, but also, as an open-source project (Wen et al., 2012). It presents itself as an efficient and scalable platform for VM management on large-scale distributed infrastructures with ease of integration provided by its plugin system which empowers companies to seemingly integrate it with their existing environments.

OpenNebula handles two main use cases, Data Center Virtualization Management and Cloud Management, the first enables server consolidation and integration with existing infrastructure making use of existing resources like computing, storage and networking. In this tier OpenNebula handles the hypervisors directly, supporting KVM, XEN and VMWare, having complete control over virtual and physical resources to provide elasticity and high availability. The second provides the usual IaaS experience enabling a multi-tenant cloud-like provisioning layer so users may create and manage virtual data centers or provide a public/private cloud features via a ready to use UI.

The cloud architecture in OpenNebula, as depicted in Figure 2, is defined by five basic components, front-end, virtualization hosts, storage, networks and user management:

**FRONT-END** Machine or host where the OpenNebula installation resides and the entry point for infrastructures management. The management portion integrates the management daemon and task scheduler. Serving the web interface is another service (sunstone-server) which provides cloud management via the web browser. It may still include optional advanced services for service provisioning (OneFlow), centralized monitoring agent (OneGate) and compatible APIs for storage, image and VM management (econe-*).

**VIRTUALIZATION HOSTS** Physical machines that will host the VMs. These hosts must have a hypervisor supported by OpenNebula virtualization Subsystem that is responsible of interacting with the hypervisor and manage VM life-cycle. The default configuration is ready for handling KVM out-of-the-box, but Xen and VMWare are also supported.

**STORAGE** OpenNebula presents three distinct storage types, namely system, image and file datastores. System Datastore holds VM base images which are used as templates for newly created VMs. These images may be simple base installs, snapshots or just filesystem links; Image DataStore is used for deployed VM images, when an instance is created, a base image is copied from the System Datastore into the Image Datastore and used as storage for the VM; File Datastores are used for kernels or ramdisks. Supported filesystems are file form images, LVM or Ceph volumes Weil et al. (2006).

**NETWORKING** Maintaining the trend from its competitors the network stack for OpenNebula presents two separate physical networks for service and instance communication. Service network handles communication between the services OpenNebula uses to function properly, front-end, monitoring hypervisor management traffic; The instance network provides communication between VMs deployed in different physical hosts; Administrators may configure additional network virtualization services or drivers, currently supported, simple firewall, 802.1Q, ebtables, Open vSwitch or VXLan.

**USER MANAGEMENT** The approach to user management in OpenNebula handle the common use case of authentication, authorization and accounting. For integration purposes it provides the ability to authenticate users from external sources like LDAP.

OpenNebula branches mainly into two different use cases respectively Data Center Virtualization Management and Cloud Management, one deals directly with the system and available hypervisors being completely in control of the infrastructure, the other works on top of an existing infrastructure management platform like VMWare vCenter or Amazon

AWS controlling provided resources. Regarding usage at the user interface level, management resembles OpenStack, the UI provides control over users, groups, instances lifecycle, images and permissions.

As a platform that does not enable metal provisioning and resource allocation or leasing to a given set of users we discarded it as a full solution for our problem, since those are operating requirements and covered by our platform. Evaluating it as a tool for managing virtual instances we appreciated the ease of deployment and the existence of less moving parts, but the XML-RPC API and lack of Python bindings discouraged its use, since OpenStack provides much friendlier integration scenarios for our use cases.

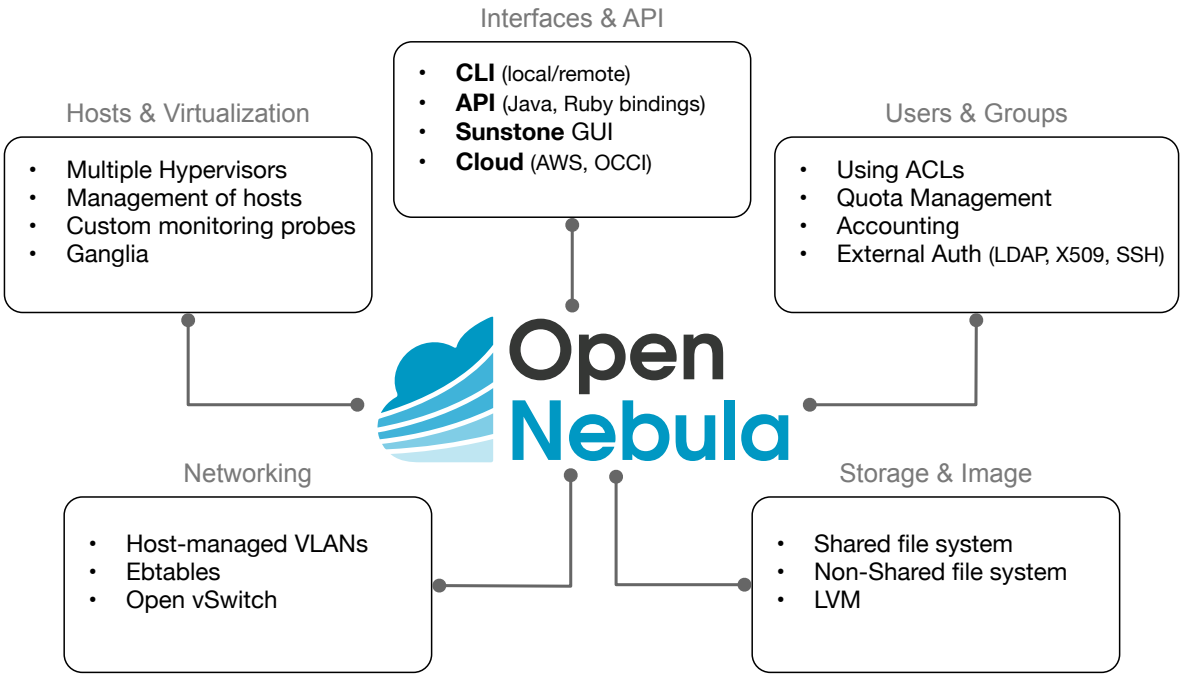



Figure 2.: OpenNebula Conceptual Architecture

## 2.3 CLOUDSTACK

CloudStack is an open source software platform written in Java designed to serve as IaaS. Like its competitors, CloudStack provides all the necessary tools for building public, private and hybrid clouds. Currently CloudStack is a project developed by Apache Software Foundation (ASF), although, its inception was in 2008 with Cloud.com and later acquired by Citrix which donated it to ASF (Barkat et al., 2014).

From an architectural standpoint the components don't diverge from competitors. From a bottom-up overview, as depicted in Figure 3, CloudStack integrates *primary storage* shared among *hosts*, also known as compute nodes, these run a given hypervisor. A *cluster* is

comprised from a set of identical hardware *hosts* sharing the same *primary storage*, this enables live migration from within the same cluster without disturbing services provided to the user.

PRIMARY STORAGE Storage associated with a cluster and shared among the hosts composing that cluster. It stores the virtual disks for all the VMs running on hosts belonging to the cluster. Although CloudStack supports adding multiple primary storage devices, the minimum operating requirement is the existence of at least one primary storage device.

HOSTS Also known as compute nodes, they represent the physical resources where the VMs will run. For that purpose, hosts are equipped with a hypervisor and connected to the primary storage device. A cluster is composed of multiple hosts running the same kind of hypervisor and one or more available primary storage devices.

CLUSTER A cluster provides a way to group hosts, it consists of one or more homogeneous hosts and a primary storage, forming a pool of resources sharing the same network subnet. Hosts belonging to the cluster must abide and operate according to the following rules, present identical hardware, run the same hypervisor, share the same network subnet and share the same primary storage. Conforming to these rules allows the cluster to perform live migrations without noticeable service interruption.

POD A pod logically groups together one or more clusters, meaning all hosts and primary storage devices inside the clusters are now in the same subnet and able to communicate with each other. Pods are logical constructs within CloudStack to which users have no insight.

ZONE A zone is composed of one or more pods and a secondary storage device. A benefit of organizing an infrastructure into zones is to provide isolation and redundancy since each zone may have its own power supply or network uplink. Also, zones can be organized into public or private clouds.

REGION A region is the largest available organizational unit within a CloudStack deployment. A region is composed of multiple zones making them perfect for providing fault tolerance and disaster recovery in large deployments.

NETWORKING CloudStack networking offers two network configuration scenarios, basic and advanced. Depending on service needs the basic setting provides guest isolation through a layer 3 switch while the advanced setting enables the creation of multiple networks, each with a given set of guest.

We found CloudStack to have an appealing architecture and component abstraction, the organizational approach of regions, zones, pods and clusters as depicted in Figure 3 facilitates the structure and management of virtual datacenters. Nevertheless it lacks support for two important features, provisioning and configuration of bare metal on hosts without an Intelligent Platform Management Interface (IPMI) interface and the ability to lease specific or a pool of bare metal resources to a given user. Our cluster of commodity hardware does not feature IPMI, not only that, but it does not incorporate any lifecycle management interface. The only feature for remote interaction is Wake on Lan (WoL), custom service agent or Secure Shell (SSH) access, limitations we address in our proposal.

Regarding features CloudStack and OpenStack are very similar as expected, any enterprise tier IaaS worth mentioning should provide a set of services and concepts so users can deploy these systems with the least possible constraints. Although CloudStack provides the functionality it lacks in modularity, thus placing some constraints on flexibility and extendability (Barkat et al., 2014). On the other hand OpenStack primes for being one of the most flexible, extensible and modular open source IaaS, as a consequence the deployment complexity increases with the introduction of more moving parts also turning high-availability into a harder challenge.

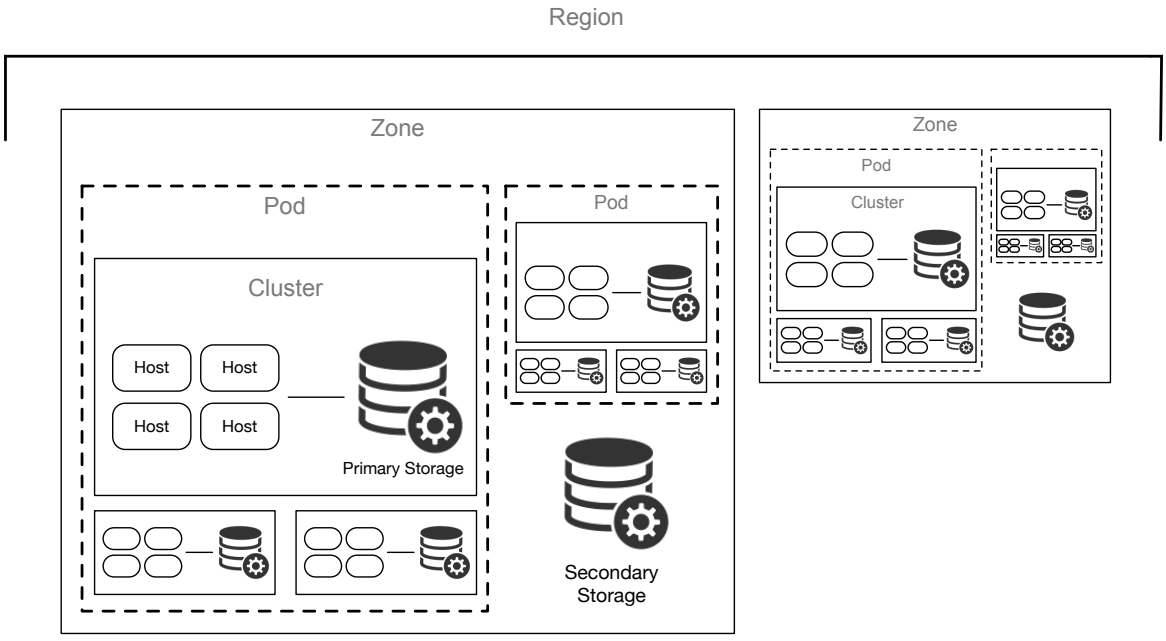



Figure 3.: CloudStak Components Organization Overview

## 2.4 COBBLER

Cobbler is a Linux installation server for automating, provisioning and configuration. The project goal is to provide automation, reproducibility and infrastructure management.

As an install, automation and management system, Cobbler is very capable. Using standard tools like Trivial File Transfer Protocol (TFTP) Sollins (1992) and Preboot Execution Environment (PXE) Droms (1997) it can handle any network bootable device, physical or virtual. With integration of configuration management tools, systems provisioned by Cobbler can also be configured according to a pre-made template.

Cobbler templates and profiles are managed from the provided command line tools or web interface. VM allocation is simplified by means of Koan, a client-side agent in charge of virtual resources (Pezzi et al., 2014).

From a management standpoint Cobbler fulfills most provisioning needs, tackles bare metal and virtualization via configurable templates and profiles for unattended installs, abstractions depicted in Figure 2.4. It also integrates with configuration management tools, getting systems configured according to a predefined state. Our proposed solution handles Cobbler use cases, but improves on separation of concerns by creating two distinct platforms able to work together or standalone. One platform handles bare metal provisioning while the other takes care of virtual machine allocation and both are able to work and integrate configuration management tools.

One issue that Cobbler doesn't handle is machine allocation management, one use case for our proposal is the possibility to lease resources to a given user whether the resource is virtual or physical. The lease is transparent to the user and only visible from a management standpoint, once a resource is leased, the owner becomes its administrator and may execute any task: unattended installs, cloning or configuration management.

The Cobbler architecture includes three major components:

**COBBLERD** The main Cobbler component that handles all the platform and API requests, abstracts access to database and storage repositories and handles resources lifecycle from boot, automated install, configuration and system events. Internally, logical components are abstracted into *distros* which represents an OS carrying information relative to kernel, initrd (initial ramdisk) and respective kernel options for booting. The *profiles* target operations for a specific distribution where automated install scripts and particular kernel parameters can be specified. The *system* represents a given host containing provisioning information including a profile and boot image. The *images* represent files to use for host booting, normally a kernel and initrd files.

**COBBLER WEB** User and management interface accessible via web browser for handling physical and virtual resources configuration. It simplifies Cobbler operation by hiding Command Line Interface (CLI) complexities and providing visual cues for every

logical Cobbler component such as *distros*, *profiles*, *systems* and *images*. Actions via CLI and Cobbler Web always target Cobblerd.

KOAN With the Koan client, Cobbler is able to provision VM instances remotely, given that Koan sever is running on the remote host. This client/server application expands Cobbler reach to virtual infrastructures.

Cobbler is a powerful and versatile tool for provisioning virtual and bare metal resources which also integrates configuration management capabilities. Apart from the web interface it serves power users by providing a CLI for issuing single or scripted commands. It also presents some disadvantages such as being monolithic, centering everything around Clobberd and require the installation of common network services, TFTP for example, as part of a base install. We believe common network services should be left in charge of the network infrastructure already in place.

Compared to our proposal, Cobbler is able to do more but does not cover a simplified architectural approach where concerns are spread across different services. It lacks the notion of resource pools and resource leasing, one of the simpler, but most important features in our proposal. Besides Koan there is no integration with third party IaaS platforms, a feature we currently support to leverage an on-premises OpenStack installation.

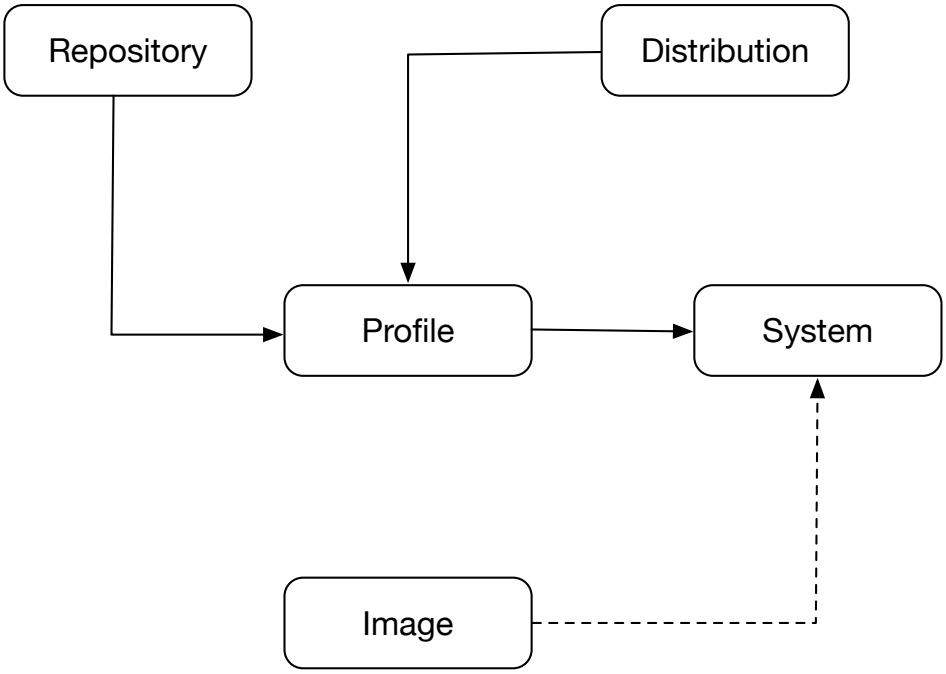



Figure 4.: Cobbler Components

## 2.5 CANONICAL MAAS

Metal as a Service (MaaS) is Ubuntu proposal for bare metal provisioning. The system started out as a Ubuntu centered automated install system, although, presently, supporting other OSs. Much like Cobbler it supports multiple management interfaces and templates for unattended installs.

The primary motivation for MaaS Chandrasekar and Gibson (2014) is working together with Juju Baxley et al. (2014), a configuration management system that leverages MaaS to provision resources for later configuration. Nevertheless, MaaS may be used standalone since it has no dependencies or hooks into Juju. A noteworthy highlight is MaaS ability to deploy images instead of starting a remote install of the OS, image based deployments greatly decreases deploy time, since no installation steps are performed.

MaaS delivers cloud like management, which is usually employed in virtualized environments to physical resources, presenting seamless bare metal provisioning, allocation, and, when used with Juju, configuration.

The architecture (Figure 5) of a MaaS deployment is abstracted into a set of concepts and terms that will be explored bellow:

NODE A node represents every networked equipment known by MaaS which includes controllers, machines and devices.

REGION CONTROLLER The Region Controller is responsible for providing a Representational State Transfer (REST) API and web interface for infrastructure and node management. It depends on PostgreSQL as a data repository for storing information regarding users, usage and keeping track of managed nodes. For correct operation it also depends on Domain Name System (DNS) for resolving nodes names and addresses.

[Rack Controller] Looking at a logical setup of the system we can observe the Region Controller as the data center manager having the Rack Controller below as a low level handler for a set of nodes grouped in a rack. In a given setup, one or many Rack Controllers can be deployed, they provide common network services like Dynamic Host Configuration Protocol (DHCP) Droms (1997) and TFTP for booting nodes and the ability to download OS images via Hyper Text Transfer Protocol (HTTP).

MACHINE MaaS handles provisioning of bare metal and VMs given they are configured for PXE booting. MaaS makes no distinction about the type of integrated machines and deploys to them without restrictions. One exception are the machines MaaS classifies as *devices*, these are seen as non-deployable artifacts kept only for inventory and infrastructure reference.

We find MaaS evolving fast and a viable option for metal and even VM provisioning, but we also find it lacks facilities to configure, template and manage boot scripts, preseeds and

resource-to-user allocation. MaaS is also targeted at computer savvy users while the user base for our platform is broader, ranging from end-users, researchers and professionals, in most cases expecting ready to run systems at the click of a button.

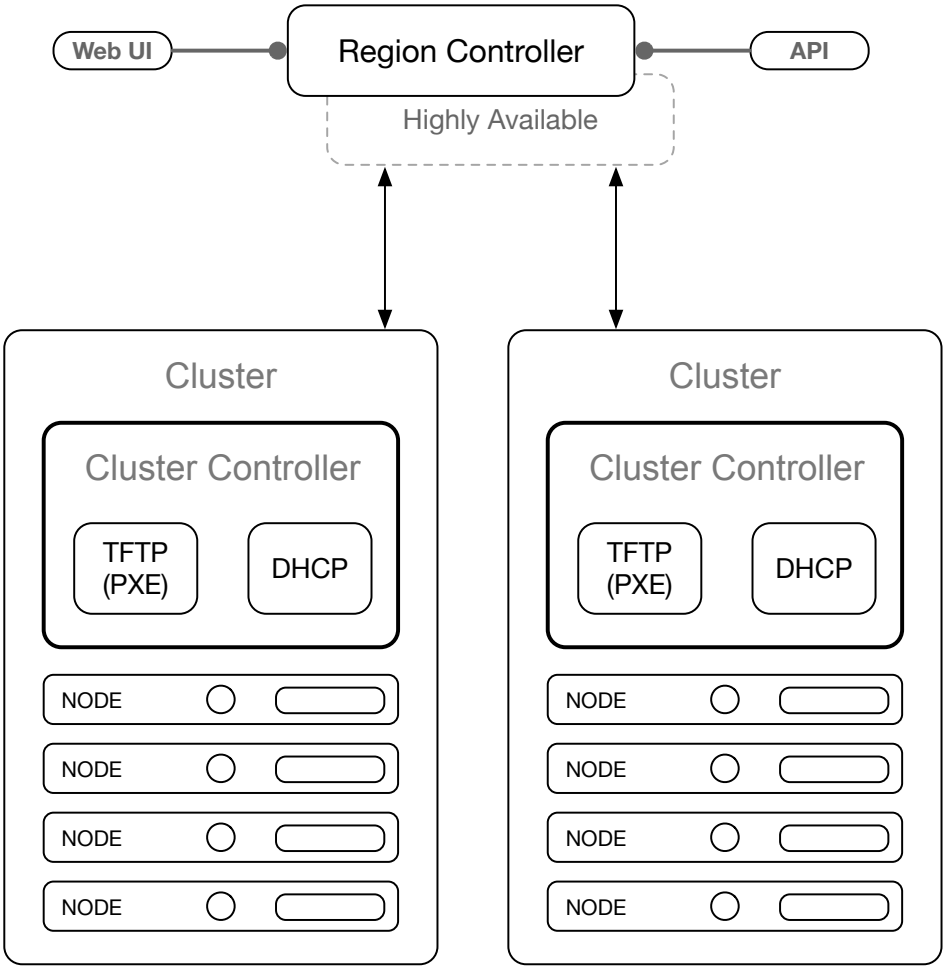



Figure 5.: MaaS Architecture Diagram

## 2.6 OPENVPN

A VPN enables secure point-to-point communication between multiple nodes in a otherwise insecure network. The secure communication is possible after establishing the connection and creating a secure tunnel, all traffic using the tunnel is encrypted. This tunneling technique enables secure dataflow between nodes and networks connected to the VPN.

OpenVPN is an open-source project that provides a user-space VPN solution based on Secure Sockets Layer (SSL)/Transport Layer Security (TLS) for session authentication and Internet Protocol Security (IPSec) Encapsulating Security Payload (ESP) for secure tunnel

transport. It is a highly portable solution, running on virtually any OS in regards to both, server and client.

We understand that OpenVPN is not a standard VPN protocol, nevertheless it relies on standards for its implementation and is present in most desktop and mobile platforms. Also, it provides all the versatility we need in a straightforward configuration file. Should the need arise it is also possible to package it for preconfigured client delivery.

## 2.7 RESTful API

Due to its simplicity and availability, web-services tend to use HTTP as transport for communication between client and server (Christensen, 2009). Running on top of HTTP is REST, a stateless protocol preconized by Roy Thomas doctoral dissertation (Fielding, 2000) where he describes a stateless, resource oriented protocol, suited for the modern web, mobile or any other kind of application needing remote request/response type communication.

In a service oriented architecture REST provides less communication overhead than a connection-oriented protocol, it does not expect state or protocol negotiation, just a simple HTTP request/response much like getting a webpage from the server. For using HTTP, REST is stuck with some limitations which are the know HTTP verbs, DELETE, GET, POST, PUT, although, since REST is resource oriented we can express every possible Create, Read, Update and Delete (CRUD) action upon a resource, for example:

DELETE destroy a given resource

GET show a resource or list resources

POST create a resource

PUT update a given resource

The limitation becomes an advantage by allowing us to use a dialect that has semantic meaning. Although the REST architecture does not have a standard, it has seen broad acceptance for web-services Mumbaikar et al. (2013). Our API is implemented to follow the rules where possible, so, assuming a resource is tied to an endpoint we are able to provide meaning to our routes/endpoints and define the responses accordingly.

Although REST does not enforce how the messages should be encoded the most common choices are Extensible Markup Language (XML) and JavaScript Object Notation (JSON). Both formats are known and provide many libraries able to parse them, although, JSON as some advantages for our use case. JSON doesn't use opening and closing tags, making it lighter regarding transfer, its native to JavaScript and easily encoded and decoded into an object or hash in most languages.

# 3

# USE CASE

The use case that motivated the creation of our platform comes from the need of supporting HASLab researchers in the process of allocation and setup of VMs and physical resources via a common point and click interface. Before we decided to build our own solution the laboratory was using MaaS for metal provision, Juju for service and configuration management and OpenStack for VM management, a group of tools previously described in Chapter 2. Although all of them have their strengths and weaknesses, they did not reveal to be a good fit for our specific needs.

In a research environment where everyone have their specific needs and configurations it becomes hard to maintain management applications that require command line interaction, or even, interfaces with many options. First observations revealed that researchers had difficulties reusing configurations and, in some cases, ended up overwriting them to suit individual needs, rendering them unusable for reuse at a later point in time.

The need for an uncluttered system that provided configuration sharing and re-usability as well as pre-configured settings became clear. After evaluating existing tools we decided to make use of what already works and developed the missing components and functionality that made sense for our use case. With that in mind we built the necessary tools and frontend that currently leverages OpenStack for VM management, although, not limited to it. Replaced MaaS and Juju with MOCAS for metal provisioning and privilege Ansible Red Hat, Inc. (2017) for configuration management, although we do not enforce its usage.

It's hard to create generic tools that fit all use cases, it's even harder to maintain that kind of tools long term due to a ever growing codebase or even when passing it from developer to developer. With that in mind we tackled the problem by creating multiple tools and modules where each one deals with its own set of problems and concerns. This is important for adding more functionality in the future in order to, at least try to fulfill most edge cases. While developing and maintaining in-house tools we have the freedom to focus on pressing needs and tight deadlines.

## 3.1 RESOURCES AVAILABILITY

HASLab hardware resources are divided into two distinct groups, a consumer hardware cluster composed of 104 nodes and 4 server grade machines. From the cluster, 20 nodes are assigned to a OpenStack installation with currently 18 nodes as compute service. Metal node installations are managed by MOCAS while OpenStack virtualized instances are handled by Bootler which may also act as frontend for MOCAS.

MOCAS is mostly used by the researchers at HASLab providing them an easy way to deploy any available Linux Distribution into cluster nodes. Most of the work involves distributed systems testing spread among multiple machines which must be in a pristine state before testing starts, since unexpected running services from previous users may create faulty statistical data. After allocating a machine the user is advised to mark it for provisioning in order to get a predictable system.

Any user with an institutional email should be able to register with the public facing platform, Bootler, and after completing email validation should successfully authenticate with the provided credentials during registration. Once authenticated the user is presented with the allocated physical resources and virtual flavors available for creation, this step also provides VPN access to the user. Being able to provide a registration system as an entry point for users affiliated with HASLab removes the administrative burden of handling account creation and access to resources, an edge case we found necessary to tackle and integrate into our systems that none of the evaluated tools provided as an option.

The system is able to serve and help researchers get work done without loosing time with system administration tasks while providing them access to tools available in our private network. At the same time freeing human resources from handling accounts and deployments, allowing them to focus on improving and creating services.

# 4

# DESIGNING MOCAS AND BOOTLER

The project described in this thesis is composed of multiple sub-projects and network services, each, with its own set of small interacting components. We may look at the services as the moving parts in the system, while components play a simpler role like a part in a puzzle.

Designing a provisioning system from scratch when there are so many options available (see Chapter 2) may look like we are duplicating existing tools. One might argument that extending existing platforms to cover edge cases is the best option, but observing previously described tools, we verify they overlap each other in several areas. In theory that happens because each of them was created to solve a specific problem and then evolved into a somewhat generic tool catering its developers expectations.

The reasoning for not following the extension path is that we want to avoid technical debt and future integration hurdles when upgrading to newer versions. We also understand that collaborative development via open source can mitigate the problem of extending with new features, but for that to happen ideas and long term goals need to be aligned in order to get pull requests merged.

This contribution tries to target a set of use cases not covered by existing solutions. From our analysis most platforms implemented some form of infrastructure management and provisioning for bare metal and VMs, some went further, implementing resource life cycle management and integration with configuration management platforms. Nevertheless neither of them handled user registration management or VPN integration to allow access into our private network from the exterior. Features not closely related to provisioning systems, but a valuable integration for us.

In an environment of heterogeneous hardware, ranging from servers to commodity clusters, resources mostly dedicated to research projects and continuously assigned to different researchers, the need for resource leasing is urgent. Analysis (see Chapter 2) revealed that currently available platforms UIs target systems administrators, not end-users. Problem we mitigate by delegating the ability to allocate, deallocate and observe resource usage to the user. This practice enables bare metal resource sharing between researchers without requests to administrators.

Cloud platforms, IaaS in particular are crucial in order to provide a cohesive and integrated management of shared resources (cpu, memory, storage, network). In that area our analysis inclined us towards including OpenStack in the architecture. It mitigates two problems, VM orchestration for simple standalone instances and IaaS, for a complete cloud experience. From our observations the first use case is the most common and the one we opted to simplify. In order to provide users with an intuitive interface for VM instances we completely abstracted the underlaying IaaS with a UI that only asks three questions and completely hides unnecessary details.

By hiding the underlaying IaaS infrastructure we stay platform agnostic with the ability to integrate OpenStack, other IaaS or even a custom solution. This approach expands the possibilities for extending our platform, situation that would not be feasible if we had to maintain a fork from any of the evaluated projects.

This chapter describes the full design and implementation of MOCAS, our contribution for bare metal lease and provisioning and Bootler, our proposal to simplify VMs creation and MOCAS integration with bare metal resource pools. The chapter also includes external services and how components are interconnected. Each section will focus on a single component, also describing the rationale behind the implementation. In some cases context about interacting services or components will be necessary for which an overview will be given.

We will not focus in the configuration part of the system since that particular topic is related to the OS where the system is being installed in.

## 4.1 Architecture Overview

Looking at the architecture from top to bottom as depicted in Figure 6, we have the Internet facing VPN acting as gateway into our system resources. OpenVPN permits routed connection from a client computer into our network via a secure tunnel. The VPN makes use of an existing OpenLDAP server acting as a centralized user repository.

At a next level we have MOCAS and Bootler (Figure 6) also leveraging OpenLDAP as a centralized user repository, with Bootler being the only component with write access to OpenLDAP (Figure 7), since account creation is one of its features. Also, only Bootler may be reached from the Internet without going through the VPN. This is necessary because account creation is done via Bootler and without an account, a user would not be able to authenticate.

The main components MOCAS and Bootler are both split into two parts, see Figure 10, the backend and frontend. The backend handles and dispatches operational tasks to the workers while providing a frontend API. The frontend allows the users to interact with the platform via the current web interface, nevertheless, with an API in place other frontend

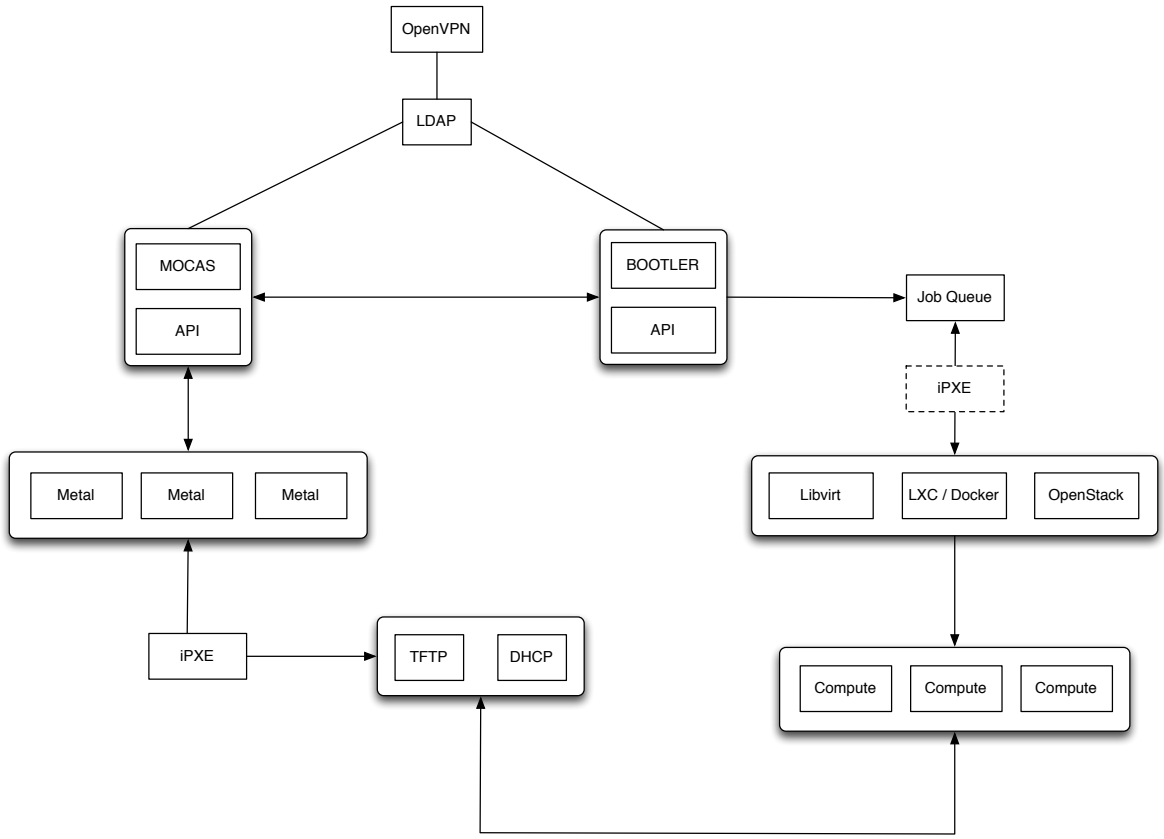



Figure 6.: Architecture Diagram

types or interactions may be developed. Although these two high level system components don't depend on each other to function, each of them is dependent at the usage level, without a frontend it's impossible to send actionable operations to the backend.

Drilling down into MOCAS architecture, we reach the networked pool of hardware resources it manages. These resources are handled by MOCAS which uses common network services placed at the base of the infrastructure as depicted in Figure 6.

While MOCAS handles hardware resources, Bootler manages VM allocation by turning user frontend interactions into work units which in turn are queued and dispatched to workers for later execution. This approach promotes asynchronous execution of workloads, resulting in better user experience at the frontend level. The life cycle of VMs is then handled at the lower levels by the deployed infrastructure. Bootler can accommodate multiple scenarios such as private clouds, custom solutions or container based systems by integrating and implementing the appropriate communication interfaces.

MOCAS and Bootler are independent services each handling a different use case respectively bare metal provisioning and VM allocation. Nevertheless, Bootler also acts as a unified frontend for MOCAS enabling bare metal and VM handling from a single inter-

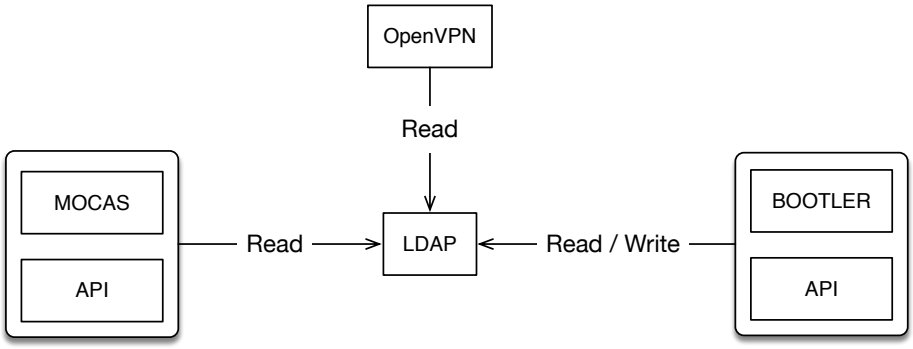



Figure 7.: LDAP Access

face. By enabling the feature flag that provides this functionality both platforms are able to communicate via a predefined API.

Workers are an important piece of the system since they handle all the long running tasks without blocking further requests. Web applications do their work by accepting a request, handling it and returning a response to the client. When done synchronously, the handling of a long running task blocks the request, leaving the client waiting for the response and unable to perform other operations. To mitigate this performance and usability issue, all long running tasks are dispatched to asynchronous workers, allowing the system to return a fast response and, at a later time, push the result of the operation after it completes, see Figure 8. We achieve these features by using a message queue connected to both, the client and the server, enabling us to push events or notifications only when they occur, thus leaving the browser interface free to perform other tasks and freeing web server resources with fast requests and making room for more clients.

At the lower levels of the implementation, we have the network services which support platform operation and boot automation. Noteworthy is the network boot services, mainly PXE and the alternative implementation *iPXE*, without it we wouldn't be able to automate system provisioning. *iPXE* provides us with the possibility to create scripts that control network booting, and although the functionality is limited to a series of commands it allows to deliver the right boot environment dynamically and on the fly. These scripts interact with MOCAS API (see Figure 9) in order to register nodes and also boot them.

## 4.2 Common Services

Before diving into the in depth implementation details and interactions follows, a brief description about the pillars supporting the platform. Being aware of the building blocks is fundamental to understand the next sections in this chapter.

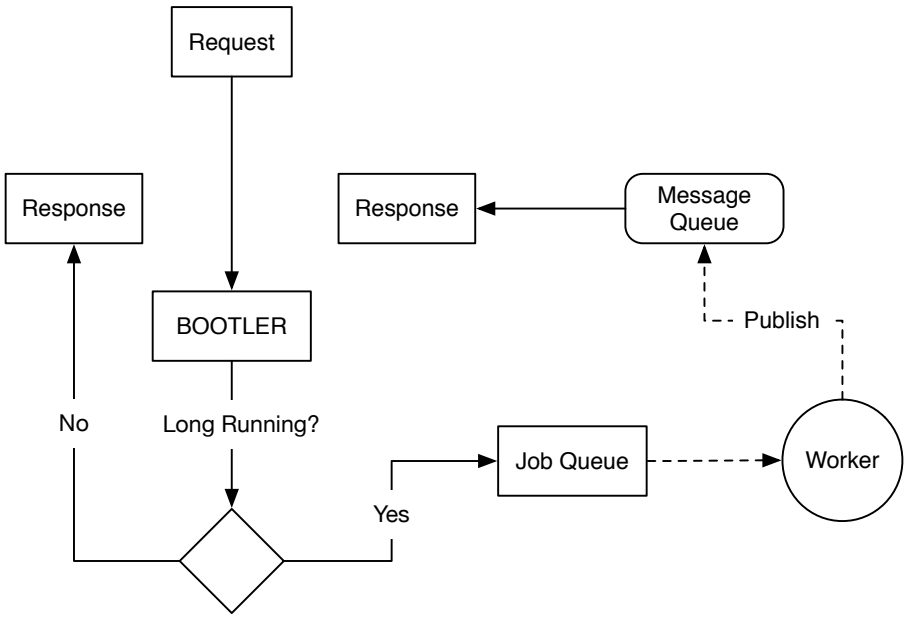



Figure 8.: Fast vs Slow Request Handling

**DHCP** The Dynamic Host Configuration Protocol grants lease on a set of network configurations to nodes. The most basic configuration would be an IP address, netmask, gateway and DNS.

**TFTP** The Trivial File Transfer Protocol is implemented on top of User Datagram Protocol (UDP) and allows nodes to *get* or *put* files onto remote hosts. It's mostly used for delivering files during early boot stages, in our case, boot images.

**PXE** Preboot eXecution Environment, is an environment to boot nodes remotely via the network interface independently of local storage devices.

**RESOURCE** Physical or virtual host with network presence, also used interchangeably as node.

**TEMPLATE** File or string with contents described in Jinja (Ronacher, 2011), grants reuse of configurations by means of expression evaluation and variable replacement.

**SYSTEM** Script/Template describing how a node should boot over the network.

Provisioning resources via network requires the availability of network services such as DHCP and TFTP. As such, our project leverages DHCP for network configuration and, since the installation files are delivered via the network, for TFTP discovery. After discovery the TFTP service delivers the initial boot files, which in turn will handle the unattended installation step.

The configuration of these services is independent from our platform, there are no constraints, which means that our project won't disrupt existing configurations. On the other hand if these services aren't already present they must be properly deployed and configured.

### 4.2.1 *PXE & iPXE*

Since our platform relies on network booting for use cases likes unattended install, cloning and restoring, one requirement is support for PXE booting. PXE enables systems to boot from the network card with no dependencies on data storage devices or currently installed OSs. PXE is also able to download images, allowing the boot process to continue into an installer, OS running from the network or the execution of a given tool.

Since the boot process is done by the network card the system can operate without local storage, thus booting and running completely from the network much like a *Thin Client* Sinclair and Merkow (1999).

While implementing our platform we found that PXE wasn't flexible enough for our needs, we had no way to send custom resource identification to our service endpoints or redirect request to custom Uniform Resource Identifiers (URIs). At this point we had two options, change the boot process so we didn't have to rely solely on PXE for delivering the images or extend PXE in order to integrate the needed features. While researching which path to follow we found *iPXE*, which after some experiments revealed itself as a perfect fit for boot interaction and image delivery while supporting custom boot scripts.

An open source effort to make PXE more flexible and feature complete is *iPXE*. It provides full PXE compatibility while adding the features we were missing, scripted boot process, HTTP boot, basic system and hardware information. Also, *iPXE* is bootable from the network or directly from hardware when the network card uses it as firmware. In order to avoid flashing new firmware in every network card, we opted on booting from the network, which facilitates the integration of bare metal resources.

Integrating *iPXE* scripts into our platform enabled us to glue and pick up where the boot process left off. Exchanging information during boot was crucial for resource registration, management and adequate image delivery.

### 4.2.2 *Cloning & Restoring*

Sometimes resources must keep their existing configuration. In this situations, system replication is easier than a clean install. With this in mind MOCAS handles the use case of restoring a system in two different ways, cloning and configuration management. Firstly we discuss cloning and approach configuration management in the next sections.

Cloning allows for the creation of resource images at a certain point in time where we make a block copy of the entire system. Depending on the system size, cloning it can become a lengthy process that uses a considerable amount of storage space. Nevertheless it's a necessary feature when an exact replica of the resource is required.

Cloning would not be complete without the ability to make a system restore from a saved image. Compared to an unattended install, a system restore is more demanding on the network, where a full system image is being steadily streamed to the client. The process is also destructive, since it wipes out partition information and overwrites any data previously present on the resource hard drive.

The restoring process makes no assumptions on the hardware present on the resource, this means that when we restore an image to different resources, there is no guarantee the previous system will be compatible with the new one.

For the cloning and restoring process we use Clonezilla in MOCAS templates. Clonezilla is an already proven disk cloning utility that is able to operate unattended in clone and restore mode. This tool integrates our boot scripts templates which will be discussed later in this thesis. Since executing the Clonezilla command is identical for all resources using the clone and restore use case, the use of templates turns a long command into a set of questions easily answered via the web interface.

### 4.2.3 *Unattended Install*

Installing and provisioning systems in the most user friendly way we can find is one of our main goals with Bootler and MOCAS. Providing unattended installs places some requirements on the OS installer, mainly, support for booting over the network, able to select install options from an external file and pulling install packages via the network.

For install automation we need a way to answer the installer queries, otherwise the installation would stop right after the first installer question or interaction. Each OS installer supporting an unattended work flow provides a way of feeding a script with answers for every possible installer question. By leveraging this functionality and providing adequate data to the installer we are able to successfully start and complete custom, but unattended installs.

Answers scripts come in different dialects, for Debian and its variants a *preseed* Hertzog and Mas (2014) is used, RedHat and RedHat clones use *kickstart* Cons et al. (2000) format. In them should exist a minimal set of answers that allows for an unattended installation, as exemplified in listings A.1 and A.2. This scripts are formatted in plain text, managed and stored in MOCAS and delivered to the installer when requested. They provide the necessary Domain Specific Language (DSL) for configuring and customizing parameters like partitions, network, users and any other option available via an interactive install.

Initiating the automated install is a multi step process, the target machine initiates the network boot process via PXE, after the request hits the DHCP server a response is delivered to the target with an *iPXE* image location, the target initiates a second pass booting from the network, but this time with *iPXE* which adds the ability to script the boot process. At this point, a request is made from the target to a TFTP server which delivers the *iPXE* image. Following, is a request to an API endpoint, in the request the target sends its Media Access Control (MAC) address and hostname, on the one hand, receiving the request the server checks if the target should boot locally or from the network. For a local boot, the response instructs the host to look for booting from the hard drive. On the other hand, if network boot is in order, the response sends instruction on where to download the network installer along with the unattended install script location, the formulated response is an *iPXE* boot script that will run on the target. Figure 9 depicts a visualization of the boot sequence.

The booting process for the automated install is always the same for the supported OSs. The platform supports any system that allows installation via the network and feeding an unattended install script. It adapts to the different OSs by administratively creating templates for customizing boot scripts and unattended scripts in order to fulfill edge cases and user defined steps.

### 4.2.4 *System Configuration*

Installing a system is only part of the provisioning process, in order to provide a custom and working setup we add a configuration step. In this step the user is able to provide custom scripts to run post-install and use them to configure users, public keys, packages or services. This approach is integrated into MOCAS by leveraging kickstart or preseed templates post-install directives. A process we intend to seamlessly integrate with future work.

By scripts we mean simple shell scripts or configuration managed by configuration management tools like Ansible Hochstein (2014), Puppet Varrette et al. (2014) or Chef Tang et al. (2015). This process should be run during the first boot in order to avoid chroot environments setup by OS installers, at this stage configuration agents or scripts can be executed and scheduled for execution on future boots.

In order to take a more versatile approach to this problem, the scripts should be stored in a version controlled repository with a tool such as git. This way, each user maintains their own scripts, thus moving this responsibility away from the platform. If the user wishes to run a script after a successful install there is only one additional step to take, tell the platform where the configuration repository is located.

Replicating previous configurations using configuration management tools is a process that can be easily executed by the user after a successful unattended install, nevertheless

we found that integrating this extra step in the process translates into a viable feature for an automated install platform.

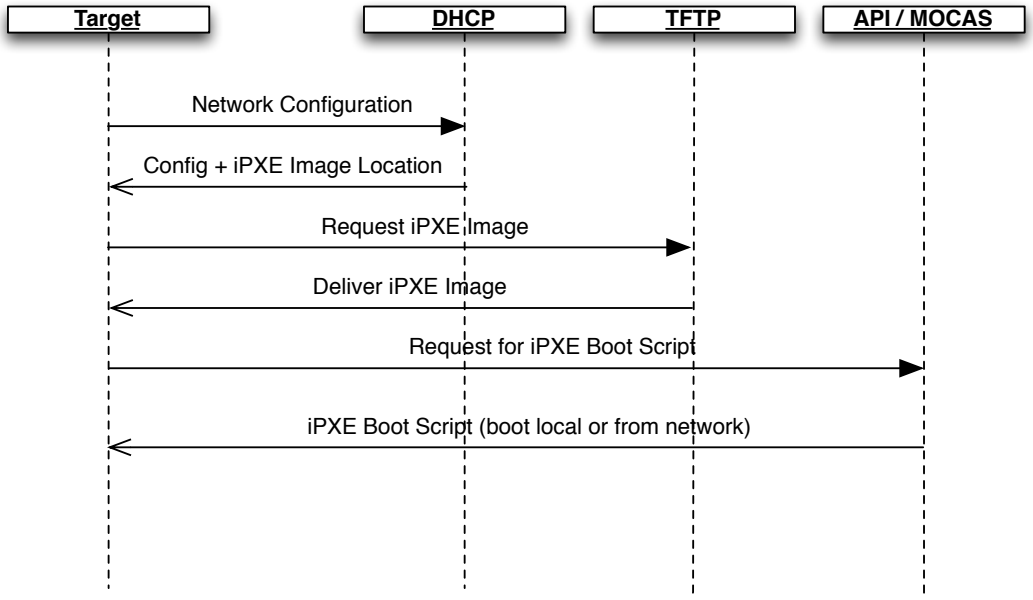



Figure 9.: Booting Sequence

### 4.2.5 *VPN Access*

Access to the private network from the exterior is a common practice for local and remote users. Since we are always connected, having access to remote resources from anywhere becomes a necessity. Remote access to an environment comes with some drawbacks, one is the configuration of a connection to the remote site, and in most cases, the installation of additional software to support it. The other is the security implications of issuing a remote connection with traffic going through an insecure medium, the public network.

A common way to overcome the security problem is using a VPN. A VPN is a network technology that allows the creation of a secure connection between two remote locations. It may be used over a public network such as the Internet or within a private corporate network.

By creating a logical and encrypted channel of communication between two nodes all traffic is kept private. When discussing security concerns we must be aware that even when encryption is used, data may get leaked, factors like vulnerable software, poor credentials or weak certificates can cripple data encryption. So, using a VPN doesn't solve the problem, it simply mitigates it. Nevertheless, we found that using a VPN was our best option to seamlessly route encrypted traffic between endpoints and through the public network.

OpenVPN is an SSL VPN which uses TLS for authentication and key exchange mechanism. Its compatibility with most systems, be it desktop or mobile and the possibility to issue a connection into an endpoint in most network environments makes it a good candidate for solving our remote connectivity problem. We choose OpenVPN because it allowed us to run multiple configurations side by side on distinct ports, thus enabling the use of different authentication schemes per configuration.

On the VPN authentication side we are able to choose between multiple options such as keypairs and certificates, user and password or pre-shared key. From the given options we opted for a combination of pre-shared key along with the common user and password scheme. For the user credentials backend we privilege LDAP since it integrates nicely in heterogeneous environments giving us the chance to centralize authentication into a single point. The reason we opted-out keypairs and certificates and followed the credentials route was the need to use the same credentials for all the services provided by our platform.

The authentication via LDAP, in our case OpenLDAP is initiated by OpenVPN via a small bridge developed in Go Go Programming Language (2017) configured within the *auth-user-pass-verify* directive. The script is fed with the authenticating user credentials which in turn are validated by OpenLDAP.

# 5

# IMPLEMENTING MOCAS AND BOOTLER

Optimizing a process or procedure is a task that requires analysis and observation in order to completely understand all the steps involved. With that in mind, during the design phase we focused on the interactions between the user and the system while performing bare metal provisioning and VM management. From those observations we where able to design a system, graphical interface and supporting components, that, we believe, abstract and simplify the act of provisioning physical and virtual resources.

While designing we focused on one level of detail, our top level platform. Stepping into implementation we quickly realized that what we describe as common network services, presented in Section 4.2, would require specific configuration in order to support the full provisioning life cycle. With that in mind we not only developed the platform, but also setup and created the necessary *glue* that allows MOCAS and Bootler to integrate and make use of the underlaying network infrastructure. Consequently, we will not broadly discuss network configuration in detail, but instead highlight network services as needed for implementation context.

This chapter discusses our approach for implementing MOCAS and Bootler while describing the interaction between smaller components and underlaying network infrastructure. Section 5.1 presents the rationale behind MOCAS implementation while extensively describing how each concept is observed and handled inside the platform. Bootler implementation in Section 5.2 focus on distinguishing the handling of physical vs virtual resources, how it shares responsibilities with MOCAS via a communication API and the dispatching of VM handling to OpenStack. Sections 5.3 to 5.5 discuss the importance of handling some tasks asynchronously and why it is a requirement for web applications that have to deal with long running tasks. Also, how notifications originating from the server find their way to the client is another covered topic. Section 5.6 provides insight on why we choose Jinja as template engine, along with the security problems user created templates bring and how they can be mitigated. Finally, Section 5.7 provides insight into the initial REST API provided by MOCAS and Bootler how it can be use to integrate the two platforms.

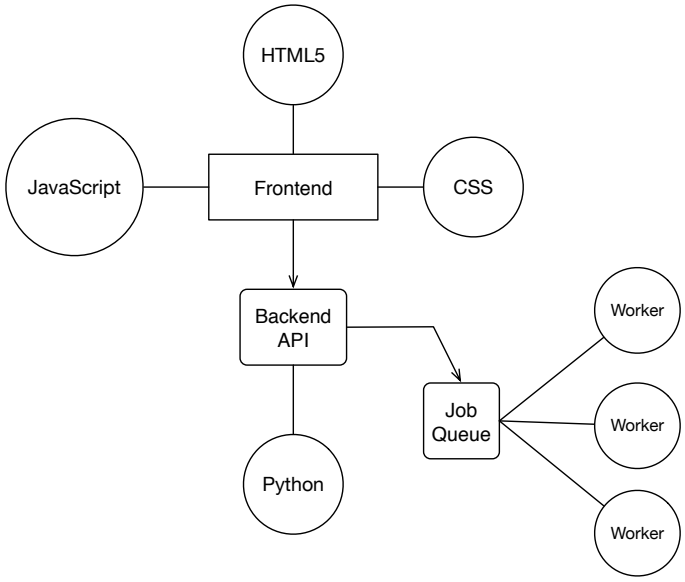



Figure 10.: MOCAS And Bootler Overview

## 5.1 MOCAS

Bare metal or VM provisioning is a lengthy and time consuming process when done repeatedly. Going through all the steps of an OS installer, updating packages and installing application are steps that should be automated when handling multiple machines. In a research environment where distributed systems simulations are constantly being run, optimizing and simplifying provisioning is a valuable resource. MOCAS provides the means for automated provisioning of multiple Linux Distributions, BSD Flavors where automation is possible and image replication for Windows systems.

The platform is web based, API driven, developed for ease of use targeting multiple audiences, allowing less technical users to manage the provisioning process without any technical knowledge of the underlaying infrastructure or on how these systems are supposed to work.

MOCAS is composed of a web interface and API and its cornerstone is metal resources allocation. Providing two distinct access interfaces enables MOCAS to serve users, services and resource generated calls with independence between API and user interface.

As an allocation tool, MOCAS is managed via its web interface which enables users to handle resource allocation, system scripts and templates. As an admin user we are able to get a full overview of available resources, elevate privileges, allocate and free resources, etc.

The first step of integration is resource registration. Resources are registered by calling a specific endpoint and providing the necessary information, IP address, MAC and hostname.

With it, the resource is made available for use. To use a resource, MOCAS needs input on which system to boot and which template to use.

A system is a template which after processing transforms into a valid *iPXE* script. This script describes the boot process, directives and parameters of how a resource should boot, but also, where to look for the necessary assets to proceed with the installation. Listing A.4 presents the necessary instructions set on how to use Debian network installer for an unattended install.

In MOCAS a system is created with the Jinja Jinja2 template engine. By using templates we are able to provide customization by leveraging the use of variables. With this approach, variables containing information about a given resource are provided to the template, allowing configuration and customization of the process.

In order to proceed with the provisioning, a system may need to call an unattended install template which can also be created in MOCAS. The unattended template does not expect any special format, it's treated as a simple text file, it may come in the form of a preseed, kickstart, shell script or any other type of installer answer file. If an unattended template is not necessary we may leave it empty.

A MOCAS template follows the same rules as the MOCAS system, it's described with the same template engine and is also able to consume injected variables, from which, it builds and customizes the unattended install script. Nevertheless, a MOCAS template must follow the rules of the installer that will consume it.

The creation of system and unattended templates enables resource boot customization pinpointing from which system and template it should boot from. Although the described customization is possible, MOCAS already provides ready to use sane defaults for Debian, Ubuntu, CentOS and image cloning.

Resource customization is also an important part of MOCAS, besides defining from which system and unattended template to boot, the definition of extra variables is also possible. These variables, expressed in JSON, will be made available for use in templates via the *vars* keyword.

Template rendering is provided by Jinja, a Python template engine that provides sandboxed execution of templates, a feature we deem necessary when rendering untrusted user created templates. Untrusted users create a well known threat model Scholte et al. (2012), input from unknown sources into a given system may make it vulnerable to information leaks, data loss or compromise. In order to exploit the system, a malicious user could create a template with the ability to run Python code, the code would run on the server hosting MOCAS having access to the file system and application data with the same privileges as the user running the MOCAS daemon.

Leveraging Jinja sandbox mode mitigates the problem of running untrusted code on the server by disabling the interpretation of Python code coming from any MOCAS created template.

Resources will only boot from the system script when they are *marked for clean install*, a user operation that indicates MOCAS should deliver the system script during the resource next boot. If a resource is not marked it will boot locally by default. Any MOCAS action will only take place during next boot, this means that the user should start or restart the resource. To remotely start resources MOCAS currently provides WoL.

Assuming required services, see Section 4.2, are in place, correctly configured and resources are setup to boot from the network (Section 4.2.1) MOCAS will handle them. Upon boot the resource will identify or register itself by calling `/<mac>?hostname=<hostname>` endpoint, receive the previously configured system template and execute it. By calling `/<mac>/unattended` endpoint MOCAS will feed the unattended template thus continuing the boot process, installation or whatever the boot script is setup to do.

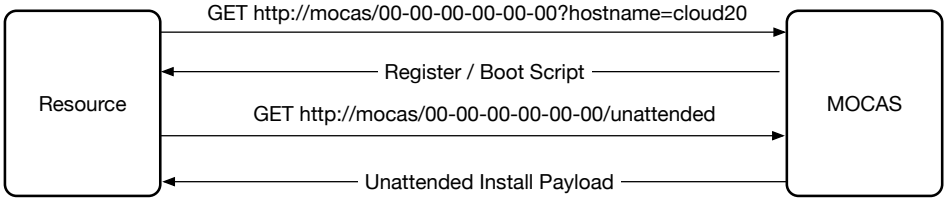



Figure 11.: MOCAS / Resource Interaction

## 5.2 BOOTLER

The entry point for metal and VM management is Bootler, a web interface and API that leverages MOCAS (Section 5.1), for metal deployments, OpenStack (Section 2.1) and libvirt Hat (2012); Bolte et al. (2010) for virtualization. Libvirt provides a predictable local and remote interface to manage hypervisors and containers via a single API, an approach that perfectly aligns with our goals for Bootler.

With the multiple choices within the virtualization field regarding hypervisors or IaaS platforms we found that every product comes with a different management interface. Also, given the complexities of a management platform, these interfaces present functionalities that consumer type users do not care or need. Bootler deals with this problem by presenting itself as a common interface for managing concrete objects like bare metal resources or VMs instead of revealing the underlaying infrastructure.

A key point for developing Bootler was the ability to interact with different APIs from a single interface while maintaining the possibility of integrating new services at later time, always using the same API when interfacing with our platform.

By acting as a gateway for communicating with other services and external APIs, Bootler is able to provide a stable and predictable interface for building applications targeting service automation and presenting desktop, web and mobile user interfaces. Bootler is usable via the web interface for provisioning bare metal machine or virtualized resources. Also, since our use case requires external access to a private network the platform also provides the means for creating VPN accounts so user can connect and use resources outside the network.

First time Bootler users must create an account to be able to use the management interface, during account creation a LDAP entry is created, which later can be used for VPN access. Once logged in the user can start allocating resources.

**Managing Metal Resources**

Physical hardware resources may be served from a predefined pool of available bare metal machines or issued from direct allocation by user action. In the first case, when the user requests a resource the system looks for unallocated hardware, when available locks it for the given user. The second option allows a user or administrator to directly allocate a resource to another user, there is only one caveat, users may only allocate to others, the resources they already own.

After being allocated the resource only becomes available when the user releases it, until then it stays marked as allocated. In an allocated state one of two actions may be applied to a resource, it can be unallocated which presents the resource free for allocation, or it can be directly allocated to a given user.

Direct allocation allows attribution of resources without showing them as available in the resources pool. This can be useful when a user means to borrow a resource without first marking it as available, thus skipping the pool. Although borrowing might be useful, it can also become a problem, if every user skips the pool there will never be available resources for allocation. It's a feature that will need some experimentation to get acquainted with user behavior and resource availability.

Eventually part of this process may need a revision, initial usage observation reveals that users don't care about releasing hardware, thus, making the availability pool scarce on unused resources. Another behavior that can cripple the availability pool is having users allocate resources to other members instead of freeing them. A quota or credit system has also been considered, but we decided to postpone its trials at a later date when we are able to gather more usage behavior.

Once a user owns a resource, the available configuration allows customization of parameters such as preferred network interface, creation of additional variables made available for templates, which system to boot and template to use as entry file for unattended installs. Regarding actionable options, a user is able to mark the resource for a clean install on the next boot, undo the previous action and issue a WoL command.

**Managing Virtual Resources**

Virtual hardware resources served until the computational nodes get saturated, a quota defined by the number of available nodes and set of processor cores. When allocating virtual resources we identified two limitations, one of them is the maximum number of cores per VM, which is a system level configuration setting for the platform where we take in account the number of cores per physical node. In the following example, if all our nodes have four, six or eight cores we would provide multiple tiers for one, two and four cores. The second limitation is maximum number of available cores which, of course, will be dependent on the number of virtual cores per physical node and number of cores allocated per virtual machine.

Virtual resources are presented in pre-configured flavors with a given capacity for cpu, memory and storage. User configuration is available only for hostname, operating system image and public key used for VM authentication via SSH. Network configuration is delegated to the hypervisor management stack, OpenStack, libvirt or other.

As described earlier in this section, Bootler is a simplified frontend for managing virtual and physical resources, which means, resources scheduling and lifecycle management happens in the lower levels of the infrastructure. Currently we leverage OpenStack for the task, nevertheless, support for other APIs can be developed as an additional module for Bootler. This feature enables Bootler to extend its reach into other platforms while maintaining the core API intact.

## 5.3 JOB QUEUING

Considering inter-service communication is done via REST API its valid to assume synchronous requests will block application flow, to avoid this issue we have two options, asynchronous requests or request delegation to a worker process. In our case, services are actual web-services following a request/response pattern leaving the connection as soon as all data arrives, following this pattern the request life-cycle should end after the client terminates the connection, thus not giving the chance of calling an asynchronous callback with a delayed job response. For that reason we found the second option more manageable and opted for a job queue able to run work units without delaying client response.

All requests regarding service and infrastructure communication expected to fail or take more than a user tolerable time to wait, which according to (Nah, 2004) is around two to four seconds, are delegated to a worker queue for delayed execution. The measure is implemented in order to keep the interface responsive allowing the user to execute other operations while the job is in execution. Also, in a web application context, if the user migrates to another page he still gets notified when the task is ready. The user may actually

wait more than the tolerable time for a task to complete, but gets notified that the system is working on the request.

## 5.4 WORKERS

Job Queues look and act on tasks as units of work, where each task is assigned to a worker. Within Bootler these tasks are described as workers, where each worker class or method is designed to fulfill a single task.

Take the task of creating a VM as example, after the user submits the request a worker class or method is injected into the distributed task queue Celery Celery Project (2017) for later execution, as depicted in Figure 12. This type of dispatching for long running jobs is what enables fast connections from the client to Bootler and solves the problem of client disconnections during job processing.

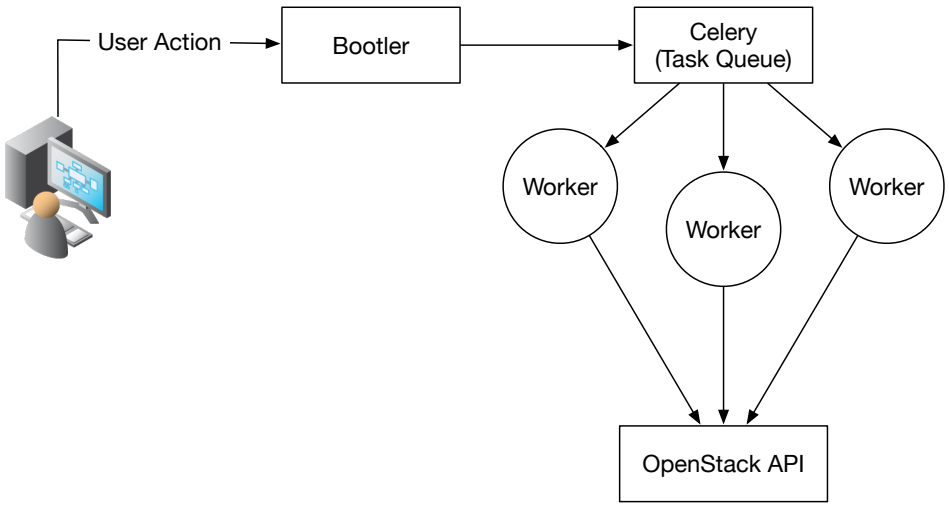



Figure 12.: Bootler VM Tasks Dispatching

Nevertheless this approach poses the problem of handling client notifications when a task finishes. We identified two scenarios, the user is connected but a notification never arrives and the user closes the client browser while the job is processing. In the first scenario, if some unexpected event happens to the connection, the state of the system won't be reflected on the user interface. In this case the client would display the correct state after a page reload. The second scenario wouldn't even pose a problem, since the next time a client connects, the state would be displayed correctly. In any of the situations the job would not be affected, since it would already be in the queue or even processing, only the client view of system state could be inconsistent and easily solved with a page reload.

The HTTP protocol is, by nature, stateless connection response, it wasn't designed for persistent connections. In order to handle near real-time state between client and server

we leverage WebSockets Lubbers and Greco (2010) which are designed for persistent connections and two-way communications. Via this connection we are able to notify the client about state changes on the server. So, when workers finish a task they publish a message to Redis Redis Project (2017) which implements the publish-subscribe pattern Eugster et al. (2003), the subscriber receives the payload and delivers it to the client.

With the presented approach we are able to notify the client and maintain coherent state between the server and client browser without any side effects, even if the client disconnects or closes the browser.

## 5.5 INTERFACE NOTIFICATIONS

To use the platform, by default, clients use the web interface, here a client is the browser to which we render our user interface. In order to notify the user about system events and keep them informed about the state of previous interactions we display textual notifications. Notifications occur at two stages, when the user submits an interaction:

**IMMEDIATE NOTIFICATIONS** When a user submits a request to the system, for example, "create a new instance", these generates a client notification completely independent from the server which indicates work has been sent to the server.

**SYNCHRONOUS SERVER SIDE NOTIFICATIONS** When a user request triggers a synchronous server action a notification is presented to the user after the request finishes indicating the state of execution.

**ASYNCHRONOUS SERVER SIDE NOTIFICATION** Triggering asynchronous actions which get deferred for execution at a later time generate an immediate notification. After the worker responsible for the task concludes, the server pushes a notification to client with success or error. At this time the client may or may not be available, either way the notification is expired but the resulting system alterations are persisted.

Queuing jobs for deferred execution raises a problem with client notifications, we inform the client that work is being done, but there is no knowledge about service state, namely if it has failed, is currently running or if it has already completed. To mitigate this issue we use a message queue between workers and clients using publish-subscribe pattern which allows to handle notifications on the client side concerning any kind of event triggered as depicted in Figure 13.

The client is able to get server side notifications by keeping a WebSocket connection open to the message queue. WebSockets provide better latency and produce less communication overhead than connection polling (Pimentel and Nickerson, 2012), since we intend to receive

notifications instantly and can't predict when they will occur, maintaining a communication channel open scales better than polling.

The queue is setup so that each client listens on a channel bound to the currently logged in user, enabling users logged in different machines with the same username to get notifications. The queue transports all kinds of messages, it's up to the client to display them as success, error, warning or other kind of message. Since each message transports a hint about the content it's easy for the client to decipher and display the meaning accordingly.

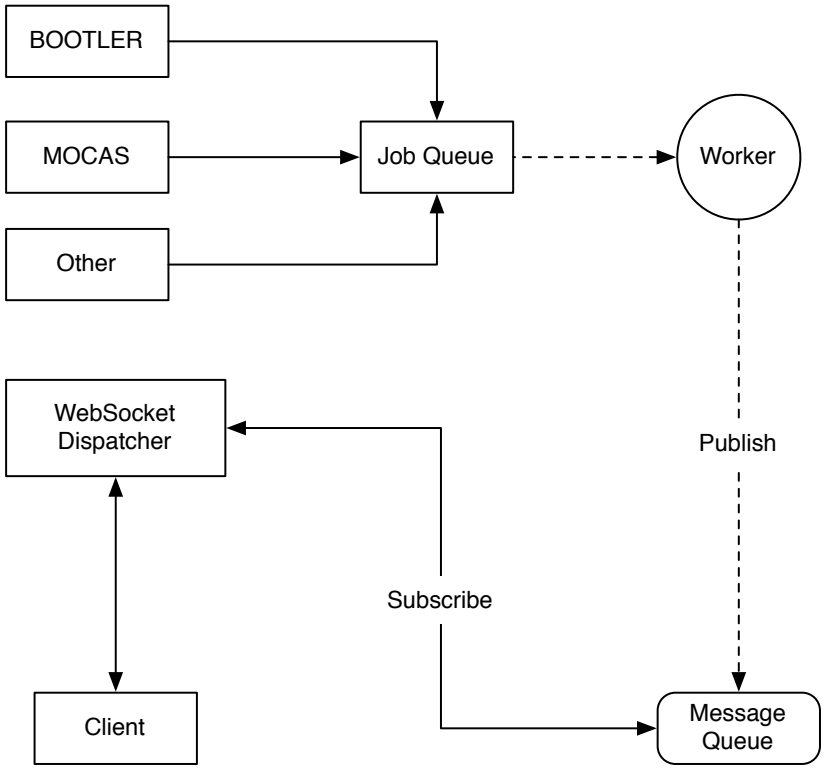



Figure 13.: Deferred Execution & Client Notifications

## 5.6 TEMPLATE ENGINE

Dynamic generation of configurations and scripts is a very important part of the platform which enables users to adjust settings according to the environment they work on, at the same time fulfilling specific needs and overcoming edge cases without altering the platform core.

Jinja templates introduce concepts like variables, conditions, loops and code execution, turning a simple template into a script with logic components. By focusing on enabling customization via templates instead of hard-coding concepts we give the user ability to manipulate the system, thus avoiding edge cases that would require code alterations.

Most backend development is centered around Python and Flask web micro-framework which integrates Jinja templates out of the box. There is a security concern around user facing templates, since they are evaluated server side a malicious user can easily write a template to introspect system configuration, thus exposing possibly sensitive data such as logins, passwords, file contents or database records.

As described in Section 5.1 We mitigate this problem by limiting the template context to only access the necessary data for building the template or script, removing access to code evaluation that could compromise the system and configuration objects. Also, leveraging Jinja sandbox mode mitigates the problem of running untrusted code on the server by disabling the interpretation of Python code coming from any MOCAS created template.

The adopted approach provides flexibility while mitigating security concerns regarding templates. It allows the use of Jinja Template Engine features while blocking the evaluation of possibly malicious and unexpected code injection into the system.

## 5.7 RESTful API Design

As described in Section 2.7, API communication between the platforms described in this thesis is done following the REST guidelines. This common approach of communication between web applications and web services provides the ability to express an API semantically by defining resources and acting upon them. Resources provide a way of describing business logic objects as well as ways to act upon them via predefined endpoints.

We can break an endpoint into three components, HTTP method, host and path. The method must be GET, POST, PUT, PATCH or DELETE; hostname defines the host where the request is directed; finally, the path indicates the mapping for a given resource action. Depicted in Figure 14 is an example of a request with each component highlighted.

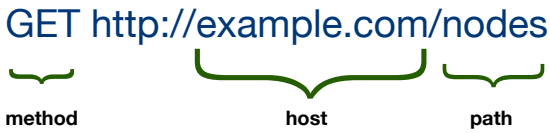



Figure 14.: Endpoint Components

### 5.7.1 *API Request / Response*

Communication with the platform using the API is done through the network via HTTP, leveraging its capabilities and giving meaning to the *verbs* like DELETE, GET, POST, PUT as described in section 2.7. One problem with these *verbs*, is that they need to be dispatched

to the appropriate endpoint. A route may be described as a way to map or dispatch a given request to the right endpoint. One might define an architecture where the routes are given by unique strings which in turn creates a one-to-one mapping to a given endpoint, which is not the case for our platform, since all routes are composed of two elements, an HTTP *verb* and a string.

For the sake of example consider the following routes:

1. POST - /resources
2. GET - /resources

At first analysis we might consider the endpoint or action would be the same for both routes, they point to the same URI. Nevertheless, the dispatcher looks at both parts of the route, the method or *verb* and the route, applying meaning to the pair. After evaluating the meaning of the method and observing the route it's clear the first issues a resource creation, but, the second requests retrieval of a collection of resources.

The evaluation needs to be made at the application level and never in the web server or application server. Request routing is an application concern which is important in terms of ubiquity, the application should not depend on any web or application server.

The interaction between client and server regarding payload is done via JSON. A request from the client to the server must define the content type as *application/json* and the respective body in JSON, the format expected by the application. The response from the server to the client should also follow a set of rules. The body of the response should be JSON and the HTTP status code should match the result of the request.

Requests resulting in success must return a status code of 200, while errors must return a 422 status code. These codes are important for error handling on the client, they enable, without further analysis, correct handling of a response. If necessary an error response may include more information about the error which can be useful for further parsing, but also, for correct client notification.

### 5.7.2 *MOCAS API*

For the API definition, depicted in Table 1, this component presents only one resource, the *host*. In the system it represents a resource or node able to be allocated, configured and provisioned by a given user.

### 5.7.3 *Bootler API*

The API definition for Bootler, depicted in Table 2, covers instance allocation and metal provisioning, thus, defining two main resources *instance* and *host*. The *instance* resource

| Description | Endpoint | Method |
|---|---|---|
| Mark Host | /hosts/:id/mark | POST |
| Change Boot Options | /hosts/:id | PUT |
| Get Resource | /hosts/:id | GET |
| Get Resources | /hosts | GET |

Table 1.: MOCAS API Endpoints

handles VM management from creation to deletion, while *hosts* takes care of metal allocation, requests that will be redirected to MOCAS. As described in section 5.2, Bootler leverages another platform, MOCAS to handle physical machine handling concerns.

| Description | Endpoint | Method |
|---|---|---|
| Start Instance | /instances/:id/start | POST |
| Stop Instance | /instances/:id/stop | POST |
| Destroy Instance | /instances | DELETE |
| Create Instance | /instances | POST |
| Get Instance | /instances/:id | GET |
| Get Instances | /instances | GET |
| Mark Host | /hosts/:id/mark | POST |
| Change Boot Options | /hosts/:id | PUT |
| Get Resource | /hosts/:id | GET |
| Get Resources | /hosts | GET |

Table 2.: Bootler API Endpoints

# 6

# CONCLUSION

In this thesis we described our rationale behind the design and implementation approach to MOCAS and Bootler, two platforms for respectively, bare metal or VM provisioning and VM life cycle management. With this tools and implementations we provide insight on how to approach the management of a heterogeneous research cluster by automating tasks and transferring the provisioning responsibility to the user.

MOCAS provides the necessary tools and exposes a user interface which abstracts the technicalities behind system provisioning. With it, resources are allocated to users, which in turn are able to provision bare metal computing resources autonomously via a web interface. The pre-made templates cover most use cases, but the system also provides the means for power-users to create custom templates and scripts, making it useful for a broader audience.

Bootler is a simpler tool with less functionality, it abstracts VM life cycle management complexities by providing a streamlined interface in which the user answers just the necessary questions in order to be able to boot and access a VM. Nevertheless, it handles all the backend complexities in order to instantiate resources, task accomplished by communicating with a lower level infrastructure, currently OpenStack (but not limited to it).

Observing the usage of these tools within a research laboratory, namely HASLab, demonstrated that researchers, after having resources allocated are able to manage and recycle them as needed without requesting support from the IT staff. The availability of automation provided by MOCAS, pre-made templates and the ability to provision a system with only a few interactions provides a viable path for a sustainable computing resource growth without increasing the need of specialized human resources. Interestingly, no further provisioning templates where created, but instead, the users requested for addition or alteration, which means that our user base finds usefulness in the point and click nature of the platform, but prefers to delegate the more involved process of template creation. Nevertheless, the described behavior aligns with our contribution objective of simplifying infrastructure provisioning.

Instancing VMs within Bootler while delegating scheduling, node management and network allocation to the underlaying infrastructure proved to be a good approach. It allowed

us to focus on user interface abstraction and on tackling the problem of turning a multi-step interaction into a single-step.

## 6.1 FUTURE WORK

This dissertation and respective implementations are a work in progress which leave room for improvement in multiple areas, such as: extracting account management, support for containers and API authentication mechanisms.

Account management could be removed from Bootler and implemented as a self contained application. This approach would centralize credential handling and make it independent from Bootler. Explicit support for containers within Bootler would enable users to manage three resources, physical, virtualized and containerized from a single platform. Redefining the API and integrating it with its own authentication mechanism could completely decouple backend and frontend responsibilities, enabling third parties to explore the API and build their own user experience.

# BIBLIOGRAPHY


Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

Amine Barkat, Alysson Diniz dos Santos, and Thi Thao Nguyen Ho. Open stack and cloud stack: Open source solutions for building public and private clouds. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pages 429–436. IEEE, 2014.

Kent Baxley, JD la Rosa, and Mark Wenning. Deploying workloads with juju and maas in ubuntu 14.04 lts, 2014.

Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 574–579. European Design and Automation Association, 2010.

BSD Hypervisor. behyve - bsd hypervisor, 2016. URL http://bhyve.org/.

Canonical. Maas: Metal as a service, 2016. URL http://maas.io/.

Celery Project. Distributed task queue, 2017. URL http://www.celeryproject.org.

Ashok Chandrasekar and Garth Gibson. A comparative study of baremetal provisioning frameworks. *Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep. CMU-PDL-14-109*, 2014.

Jason H Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 627–634. ACM, 2009.

Cobbler Project. Cobbler - linux install and update server, 2016. URL http://cobbler.github.io/.

Lionel Cons, German Cancio, Philippe Defert, Mark Olive, Ignacio Reguero, and Cedric Rossi. Automating linux installations at cern. 2000.

Ralph Droms. Dynamic host configuration protocol. 1997.

Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

Will Farrington. Github: Introducing boxen, 2016. URL https://github.com/blog/1345-introducing-boxen.

Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

Foreman Project. Foreman: Provision from anywhere, 2016. URL https://www.theforeman.org.

FreeBSD Project. The freebsd project, 2016. URL https://www.freebsd.org/.

Go Programming Language. The go programming language, 2017. URL https://golang.org.

Red Hat. libvirt: The virtualization api, 2012.

Raphaël Hertzog and Roland Mas. *The Debian Administrator's Handbook, Debian Wheezy from Discovery to Mastery*. Lulu. com, 2014.

Lorin Hochstein. *Ansible: Up and Running*. " O'Reilly Media, Inc.", 2014.

Jinja2. Jinja the python template engine. URL http://jinja.pocoo.org.

Kernel Virtual Machine. Kernel virtual machine, 2016. URL http://www.linux-kvm.org.

Rakesh Kumar and Bhanu Bhushan Parashar. Dynamic resource allocation and management using openstack. *Nova*, 1:21, 2010.

libvirt. libvirt: The virtualization api, 2016. URL http://libvirt.org/.

Linux Foundation. The linux foundation, 2016. URL https://www.linuxfoundation.org/.

Peter Lubbers and Frank Greco. Html5 web sockets: A quantum leap in scalability for the web. *SOA World Magazine*, (1), 2010.

Microsoft Corporation. Microsoft hyper-v, 2016a. URL https://www.microsoft.com/en-us/cloud-platform/virtualization.

Microsoft Corporation. Microsoft, 2016b. URL https://www.microsoft.com/.

Snehal Mumbaikar, Puja Padiya, et al. Web services based on soap and rest principles. *International Journal of Scientific and Research Publications*, 3(5), 2013.

Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

OpenStack Project. Ironic: Bare-metal provisioning, 2016a. URL http://www.openstack.org/software/releases/mitaka/components/ironic/.

OpenStack Project. Openstack open source cloud computing software, 2016b. URL http://www.openstack.org/.

OpenStack Project. Openstack: Cinder support matrix, 2017. URL https://wiki.openstack.org/wiki/CinderSupportMatrix.

Ken Pepple. *Deploying openstack*. " O'Reilly Media, Inc.", 2011.

Michele Pezzi, M Favaro, D Gregori, PP Ricci, and V Sapunenko. Testing an open source installation and server provisioning tool for the infn cnaf tierl storage system. In *Journal of Physics: Conference Series*, volume 513, page 032075. IOP Publishing, 2014.

Victoria Pimentel and Bradford G Nickerson. Communicating and displaying real-time data with websocket. *Internet Computing, IEEE*, 16(4):45–53, 2012.

Red Hat, Inc. Ansible - it automation, 2017. URL https://www.ansible.com.

Redis Project. Redis - in-memory data structure store, 2017. URL https://redis.io.

Armin Ronacher. Welcome— jinja2 (the python template engine), 2011.

Michael Rosen, Boris Lublinsky, Kevin T Smith, and Marc J Balcer. *Applied SOA: service-oriented architecture and design strategies*. John Wiley & Sons, 2012.

Theodoor Scholte, Davide Balzarotti, and Engin Kirda. Have things changed now? an empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344–356, 2012.

Joseph T Sinclair and Mark Merkow. *Thin Clients clearly explained*. Morgan Kaufmann Publishers Inc., 1999.

Dennis D Smith. *Designing maintainable software*. Springer Science & Business Media, 2012.

K Sollins. The tftp protocol (revision 2). 1992.

T Sridhar. Cloud computinga primer part 1: Models and technologies. *The Internet Protocol Journal*, 12(3):2–19, 2009.

Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343. ACM, 2015.

Sébastien Varrette, Pascal Bouvry, Hyacinthe Cartiaux, and Fotis Georgatos. Management of an academic hpc cluster: The ul experience. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 959–967. IEEE, 2014.

Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

Xiaolong Wen, Genqiang Gu, Qingchun Li, Yun Gao, and Xuejie Zhang. Comparison of open-source cloud management platforms: Openstack and opennebula. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*, pages 2457–2461. IEEE, 2012.

Yoji Yamato, Yukihisa Nishizawa, Masahito Muroi, and Kentaro Tanaka. Development of resource management server for production iaas services based on openstack. *Journal of information processing*, 23(1):58–66, 2015.

# GLOSSARY

**API** Application Programming Interface. 1, 4, 5, 7, 8, 10, 11, 14, 16, 18, 22, 24, 28, 31, 32, 34–36, 40, 41
**ASF** Apache Software Foundation. 1, 11

**CLI** Command Line Interface. 1, 14, 15
**CRUD** Create, Read, Update and Delete. 1, 18
**CSS** Cascading Style Sheets. 1

**DHCP** Dynamic Host Configuration Protocol. 1, 16, 25, 28
**DNS** Domain Name System. 1, 16, 25
**DSL** Domain Specific Language. 1, 27

**ESP** Encapsulating Security Payload. 1, 17

**FCT** Fundao para a Ciłncia e Tecnologia. 1

**GUI** Guided User Interface. 1

**HASLab** High-Assurance Software Laboratory. 1
**HPC** High Performance Computing. 1
**HTML** Hypertext Markup Language. 1
**HTTP** Hyper Text Transfer Protocol. 1, 16, 18, 26, 37, 40, 41

**IaaS** Infrastructure as a Service. 1, 5, 7, 9, 11, 13, 15, 22, 34
**INESC TEC** Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciłncia. 1
**IPMI** Intelligent Platform Management Interface. 1, 12
**IPSec** Internet Protocol Security. 1, 17
**IT** Information Technology. 1, 2, 43

**JSON** JavaScript Object Notation. 1, 18, 33, 41

**LDAP** Lightweight Directory Access Protocol. 1, 7, 30, 35
**LVM** Logical Volume Manager. 1, 8

**MaaS** Metal as a Service. 1, 15, 16, 19
**MAC** Media Access Control. 1, 28, 32

**OS** Operating System. 1, 2, 4, 5, 7, 14–17, 22, 26–28, 32

**PPTP** Point-to-Point Tunneling Protocol. 1
**PXE** Preboot Execution Environment. 1, 13, 16, 24, 26, 28

**REST** Representational State Transfer. 1, 16, 18, 31, 36, 40

**SA** System Administrator. 1, 2
**SSH** Secure Shell. 1, 13, 36
**SSL** Secure Sockets Layer. 1, 17, 30

**TFTP** Trivial File Transfer Protocol. 1, 13, 15, 16, 25, 28
**TLS** Transport Layer Security. 1, 17, 30

**UDP** User Datagram Protocol. 1, 25
**UI** User Interface. 1, 4, 8–10, 21, 22
**URI** Uniform Resource Identifier. 1, 26, 41
**UX** User Experience. 1, 20

**VM** Virtual Machine. 1, 4–12, 14, 16, 19, 21–23, 31, 32, 34, 36, 37, 42, 43
**VMM** Virtual Machine Manager. 1, 4
**VPN** Virtual Private Network. 1, 3, 4, 16, 17, 20–22, 29, 30, 35

**WoL** Wake on Lan. 1, 13, 34, 35

**XML** Extensible Markup Language. 1, 18

# A

# LISTINGS

Listing A.1: Kickstart Unattended Script

```
#version=RHEL7
auth --enableshadow --passalgo=sha512
url --url="http://mirror.centos.org/centos/7/os/x86_64"
firstboot --enable
ignoredisk --only-use={{ vars.disk|default('sda') }}
keyboard --vckeymap=pt-latin1 --xlayouts='pt'
lang en_US.UTF-8
eula --agreed
reboot
network  --hostname={{ host.name }}
rootpw --iscrypted $6$...
user --name=gsd --password=$6$... --iscrypted --groups=wheel
timezone Europe/Lisbon --isUtc --nontp
bootloader --location=mbr
clearpart --all --initlabel --drives={{ vars.disk|default('sda') }}
part /boot --fstype="xfs" --size=512
part pv.28 --fstype="lvmpv" --size=1 --grow
volgroup {{ host.name }} pv.28
logvol / --fstype="xfs" --size=4096 --grow --name=root --vgname={{ host.name
    }}
logvol swap  --fstype="swap" --recommended --name=swap --vgname={{ host.name
    }}
%packages
@core
sudo
vim-common
ntp
%end
%post
mkdir /home/gsd/.ssh
touch /home/gsd/authorized_keys
chmod 700 /home/gsd/.ssh
chmod 600 /home/gsd/.ssh/authorized_keys
```

```
chown -R gsd: /home/gsd/.ssh
echo "ssh-rsa ..." >> /home/gsd/.ssh/authorized_keys
%end
```

Listing A.2: Preseed Unattended Script

```
d-i debian-installer/locale string en_US
d-i console-setup/ask_detect boolean false
d-i debian-installer/keymap select pt
d-i keymap select pt-latin9
d-i keyboard-configuration/layoutcode string jp
d-i keyboard-configuration/modelcode pt-latin9
d-i keyboard-configuration/xkb-keymap select pt-latin9
d-i mirror/country string manual
d-i mirror/http/hostname  string mirrors.up.pt
d-i mirror/http/directory string /ubuntu
d-i mirror/http/proxy string
d-i clock-setup/utc boolean true
d-i time/zone string Europe/Lisbon
d-i clock-setup/ntp boolean true
d-i passwd/user-fullname string user Grupo de Sistemas Distribuidos
d-i passwd/username string gsd
d-i passwd/user-password password 123456
d-i passwd/user-password-again password 123456
d-i user-setup/allow-password-weak boolean true
d-i user-setup/encrypt-home boolean false
d-i partman-auto/method string lvm
d-i partman/eacommand string debconf-set partman-auto/disk `list-devices disk
    | head -n1`
d-i partman-auto-lvm/guided_size string max
d-i partman-auto-lvm/new_vg_name string system
d-i partman-auto/choose_recipe select atomic
d-i partman/default_filesystem string ext4
d-i partman-lvm/device_remove_lvm boolean true
d-i partman-lvm/device_remove_lvm_span boolean true
d-i partman-md/device_remove_md boolean true
d-i partman-lvm/confirm boolean true
d-i partman-lvm/confirm_nooverwrite boolean true
d-i partman-partitioning/confirm_write_new_label boolean true
d-i partman/choose_partition select Finish
d-i partman/confirm_nooverwrite boolean true
d-i partman-basicmethods/method_only boolean false
d-i partman/confirm boolean true
tasksel tasksel/first multiselect
d-i pkgsel/include string openssh-server vim-nox ethtool openntpd
d-i pkgsel/upgrade select none
d-i pkgsel/update-policy select none
```

```
d-i grub-installer/only_debian boolean true
d-i debian-installer/add-kernel-opts string net.ifnames=0 biosdevname=0
d-i finish-install/reboot_in_progress note
d-i preseed/late_command string \
    echo "gsd ALL=(ALL) NOPASSWD: ALL" > /target/etc/sudoers.d/gsd;sync ; \
    chmod 0440 /target/etc/sudoers.d/gsd ; \
    wget http://mocas.lsd.di.uminho.pt/maas-new.sh -O /target/root/maas.sh ; \
    in-target /bin/sh /root/maas.sh {{ email }}; \
    true
```

Listing A.3: iPXE Embedded Firmware Script

```
#!ipxe

:try_net0
isset ${net0/mac} && dhcp net0 || goto try_net1
set mac_address ${net0/mac:hexhyp}
goto chainboot

:try_net1
isset ${net1/mac} && dhcp net1 || goto try_net2
set mac_address ${net1/mac:hexhyp}
goto chainboot

:try_net2
isset ${net2/mac} && dhcp net2 || goto exit
set mac_address ${net2/mac:hexhyp}
goto chainboot

:chainboot
chain http://mocas.lsd.di.uminho.pt/boot/${mac_address}?hostname=${hostname}

:exit
exit
```

Listing A.4: MOCAS System

```
#!ipxe

set base-url {{ base_url }}

kernel ${base-url}/images/xenial-64/linux
initrd ${base-url}/images/xenial-64/initrd.gz
imgargs linux auto=true url=${base-url}/boot/{{ mac.mac_address }}/unattended
    netcfg/choose_interface={{ host.interface }} hostname={{ host.name }} net.
    ifnames=0 biosdevname=0
boot
```

Listing A.5: JSON Sample

```
{
    "block_device": "vda",
    "iface": "eth1",
    "public_key": "...",
    "user": "gsd"
}
```