JOÃO MANUEL
ALVES DE
MESQUITA BACELO

Integração de Funções de Rede Virtualizadas e Funções de Rede Físicas

Integration of Virtual Network Functions and Physical Network Functions

JOÃO MANUEL
ALVES DE
MESQUITA BACELO

Integração de Funções de Rede Virtualizadas e Funções de Rede Físicas

Integration of Virtual Network Functions and Physical Network Functions

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Eletrónica e Telecomunicações, realizada sob a orientação científica do Doutor Daniel Corujo, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Óscar Pereira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Este trabalho é financiado pela FCT/MEC através de fundos nacionais no âmbito do projeto PTDC/EEI-TEL/30685/2017

o júri / the jury

presidente / president

Professora Doutora Susana Isabel Barreto de Miranda Sargento
Professora catedrática da Universidade de Aveiro

vogais / examiners committee

Doutor Flávio Silva Meneses
System Developer na Skyline Communications

Professor Doutor Daniel Nunes Corujo
Professor auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

Agradeço ao Professor Doutor Daniel Corujo por toda a orientação e ajuda dadas no decorrer do trabalho. Agradeço aos meus colegas do Instituto de Telecomunicações pela paciência e auxílio, em particular ao Vítor Cunha pela sua disponibilidade. À minha família e namorada pelo apoio incondicional que sempre me deram dedico este trabalho. Esta dissertação foi feita com o apoio do Instituto de Telecomunicações, e no âmbito do projeto 5GCONTACT PTDC/EEI-TEL/30685/2017.

Palavras Chave

Funções de Rede Físicas, Funções de Rede Virtualizadas, Rede de testes, Virtualização, Rede sem fios.

Resumo

A Virtualização de Funções de Rede e as Redes Definidas por Software têm estado no centro da evolução das redes, prometendo uma forma mais flexível e eficiente de as gerenciar através da instanciação *on-demand* de Funções de Rede e da sua reconfiguração conforme o necessário. No entanto, à medida que novos mecanismos são desenvolvidos, é também necessário a realização de testes sobre estas tecnologias antes destas serem adotadas em implementações em contexto real. É aqui que esta dissertação contribui, propondo e avaliando uma arquitetura de sistema que integra um testbed físico sem fios, com um ambiente baseado em nuvem. Isto permite que os nós sem fios físicos se tornem parte do ambiente de nuvem, permitindo o seu uso e configuração como Funções de Rede Virtuais. Os resultados demonstraram a viabilidade do sistema, dada a capacidade da testbed em instanciar Funções de Rede virtuais e físicas quando requisitadas tanto nos nós sem fios físicos quanto no servidor OpenStack.

Keywords

Physical Network Function, Virtual Network Function, Testbed, Virtualization, Wireless.

Abstract

Network Functions Virtualization (NFV) and Software Defined Networking (SDN) have been in the center of network evolution, promising a more flexible and efficient way of managing networks through the on-demand instantiation of network functions (NFs) and reconfigurability of the network as necessary. Nevertheless, as new mechanisms are developed, such technologies require testing before their adoption into real-world deployments. This is where this dissertation contributes, by proposing and evaluating a system architecture that integrates a physical wireless testbed with a cloud-based environment. This allows physical wireless nodes to become part of the cloud environment, enabling its use and configuration as virtual NFs (VNFs). Results showcased the system feasibility, with the testbed being able to instantiate on-demand virtual and physical NFs, in the physical wireless nodes and in an OpenStack data-center.

Contents

List of Figures

List of Tables

Glossary

AMazING	Advanced Mobile wireless playGrouNd
API	Application Programming Interface
APT	Advanced Package Tool
BMC	Baseboard Management Controller
BOOTP	Bootstrap Protocol
CC	Cloud Computing
CLI	Command Line Interface
COTS	Commercial off-the-shelf
CPU	Central Process Unit
DHCP	Dynamic Host Configuration Protocol
DOS	Disk Operating System
DNS	Domain Name System
ETSI	European Telecommunications Standards Institute
GPU	Graphics Processing Unit
HA	High Availability
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
IP	Internet Protocol
IPMI	Intelligent Platform Management Interface
iSCSI	Internet Small Computer System Interface
IT	Information Technology
ITAV	Instituto de Telecomunicações at the University of Aveiro
M2M	Machine-to-Machine
MAAS	Metal As A Service
NAT	Network Address Translation
NF	Network Function
NFV MANO	Network Function Virtualization Management and Orchestration
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NFVO	Network Function Virtualization Orchestrator
NUMA	Non-Uniform Memory Access
NGI	Next Generation Internet
NGN	Next Generation Networking
NIC	Network Interface Controller
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform as a Service
PC	Portable Computer
PCI	Peripheral Component Interconnect
PNF	Physical Network Function
PXE	Preboot Execution Environment
RAM	Random Access Memory
REST	Representational State Transfer
RHEL	Red Hat Enterprise Linux
RPC	Remote Procedure Call
SaaS	Software as a Service
SDN	Software Defined Networking
SLA	Service-level Agreement
SSID	Service Set Identifier
TFTP	Trivial File Transfer Protocol
UI	User Interface
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function
VNFM	Virtual Network Function Manager
WN	Wireless Node

CHAPTER 1

Introduction

1.1 Motivation

In recent years, the typical usage of Internet resources has suffered several evolutions. From the typical use of one Portable Computer (PC) or mobile phone per person performing simple daily tasks, we reached to the point of millions of devices connected. Buzzwords such as the Internet of Things (IoT), Machine-to-Machine (M2M), Cloud Computing (CC), Virtualization among others have become part of the routine. The world is becoming more "remote" with everything being automated and online. In fact, Cisco's forecasts point to an increase of up to 29.3 billion devices connected to the Internet, up to 66% of the world's population will have Internet access, and up to 14.7 billion in M2M connections by 2023 [1]. This outbreak of new devices connecting to the network, and the associated massive new traffic, create an unparalleled strain in both network and storage, demanding new infrastructure approaches. Although traditional Internet Protocol (IP) networks are highly outspread, their configuration and maintenance is complex and hard to manage, setting a huge setback when the demand starts to increase. To cope with this problem Software Defined Networking (SDN) emerged as one of the most promising approaches to overcome the limitations of IP networks [2].

In this line, SDN [3] revolutionized traditional networks by decoupling the control and data planes. Traditional network equipment (e.g., switches, routers, and middlebox appliances) become simple forwarding elements without embedded control or software to make autonomous decisions. A logically-centralized control system became the "brain" of the networks. The use of SDN enables greater automation and programmability in the network and is often paired with Network Function Virtualization (NFV). The main idea of NFV is the decoupling of physical network equipment from the functions that run in them [4]. Thus, network functions Network Functions (NFs) (such as firewalls or load balancers) can be deployed in Virtual Machines (VMs) in cloud environments as Virtual Network Functions (VNFs) and/or in bare-metal devices as Physical Network Functions (PNFs). In this way, by leveraging virtualization technology, NFV offered a new way to design, deploy and manage networking

services, by decomposing them into a set of PNFs and VNFs.

SDN and NFV have been contributing to the improvement of cloud computing technologies. In this regard, cloud computing [5] is a model to access an on-demand network of shared configurable computing sources such as networks, servers, applications and services. Also, cloud computing offers compute, storage and networking as basic resources, while leveraging virtualization (through the use of SDN and NFV to accomplish its promises. In this line, the use of virtualization allows physical servers, storage and networking services to be partitioned on-demand by using software.

As every action has a reaction, the increase of virtualization driven projects and technologies will certainly have some impact whether in infrastructure, network, or society. The problem is that it is very difficult to predict what these effects might be, evidencing the need for platforms to simulate and/or experimentally evaluate the impact of these new technologies in real-world conditions. As such, it becomes necessary the creation and development of testbed platforms not only to accommodate the diversity of new applications, projects and upcoming technologies, but also to allow the careful observation of new developments in a controlled environment.

The Advanced Mobile wireless playGrouNd (AMazING) [6] testbed is an outdoor system that was deployed on the rooftop of Instituto de Telecomunicações at the University of Aveiro (ITAV). The testbed consists of 24 fixed wireless nodes forming a grid, in addition to a mobile node. Moreover, the ITAV has a local OpenStack cloud already deployed and running, which allows its exploration for the integration of the AMazING testbed in order to provide virtualization capabilities to the physical wireless nodes. The OpenStack cloud deployed in ITAV, like most clouds, only allows for VNFs to be instantiated in the hardware (i.e., servers) that belongs to it.

Therefore, this dissertation proposes, implements and evaluates a system architecture (for an experimental testbed) capable of managing the on-demand instantiation of PNFs in the wireless nodes and VNFs in OpenStack VMs. Thus adding the physical nodes as an extension to the existing cloud.

1.2 Objectives

The work developed in this dissertation aims to update the former AMazING testbed by building a system that takes advantage of the local cloud running in the Instituto de Telecomunicações at the University of Aveiro and extends it by connecting the AMazING testbed network to its infrastructure. To achieve this goal some tasks were performed and are enumerated as follows:

1. Deploy a functional OpenStack framework.
2. Deploy a system framework able to manage the wireless nodes (AMazING testbed) as NFs.
3. Aggregate both the cloud and wireless infrastructure in a single system framework manager.

4. The system needs to instantiate PNF and/or VNF on-demand.

1.3 Main Contributions

The work done on this dissertation allowed to design and test an architecture for a new testbed, and thus contributing to the process of modernization of the former AMazING testbed.

1.4 Document Structure

The document is structured as follows: Chapter 2 presents the background and the related work of wireless testbeds and their integration with NFV mechanisms. The proposed system architecture and its implementation is presented in Chapter 3. Chapter 4 assesses the proposed system and discusses its results. Finally, the document concludes in Chapter 5.

CHAPTER 2

State of the Art and Enabling Technologies

This chapter presents the main concepts that allow for a better understanding of this dissertation as well as some tools recently developed by third parties that are within the scope of this dissertation.

2.1 Cloud Computing

The National Institute of Standards and Technology (NIST) defined cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction." [5] In other words, the cloud enables anyone with an Internet connection to access IT resources on-demand. This model is composed of five essential characteristics, three service models, and four deployment models.

The five characteristics are:

- *On-demand self-service:* Computing capabilities can be unilaterally provisioned as needed by a consumer without requiring human interaction.
- *Broad network access:* Capabilities need to be available through the network and accessible through standard mechanisms.
- *Resource pooling:* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify a location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
- *Rapid elasticity:* Capabilities can be scaled (provisioned and released) at any time.

- *Measure service:* Resources must be metered so that cloud systems can automatically control and optimize resource usage.

The service models determine the abstraction level that the shared resources are made available. The three service models implemented in cloud computing are:

- *Software as a Service (SaaS):* The resources are presented to the consumer as an end-application. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, storage, or even individual application capabilities.
- *Platform as a Service (PaaS):* The consumer can deploy onto the cloud infrastructure consumer-created or acquired applications but, as in SaaS service model, does not manage or control the underlying infrastructure.
- *Infrastructure as a Service (IaaS):* The consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

There are four different deployment models used in cloud computing and they are:

- *Private cloud:* The cloud infrastructure is provisioned for exclusive use by a single organization and accessed by a private network connection, operating on servers managed either internally or by a third-party provider.
- *Community cloud:* The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.
- *Public cloud:* The public cloud refers to services provided by companies, which sell server resources (rather than dedicated physical servers) accessed over a public network such as the Internet.
- *Hybrid cloud:* This model is a composition of two or more distinct cloud infrastructures (private, community, or public).

Cloud computing was a breath of fresh air to the IT world and consequently to the enterprises by allowing massive developments in both infrastructure and economic spheres. By enhancing application scalability, operational flexibility, resource efficiency, agility improvements, among others, it allowed enterprises to avoid the upfront cost and complexity of owning and maintaining their own IT infrastructure.

The fundamental block of a cloud solution consists of virtualization, in which physical servers are shared by multiple tenants using virtual machines. The access to these hardware resources is managed according to the Service-level Agreement (SLA) [7].

Cloud computing applied to telecommunications requires dynamic computing and a high degree of automation to address rapidly changing demands. To do so, it relies on technologies such as SDN (section 2.2) and NFV (section 2.3).

2.2 Software Defined Networks

Traditional IP networks (non-virtualized networks), rely on the use of dedicated network devices such as routers, switches, and middle-boxes (i.e., devices that manipulate traffic for purposes other than packet forwarding, such as a firewall) to transmit data through the network. These devices are typically composed of two tightly coupled planes: 1) the control plane, which decides where the packets should be sent; 2) the data (or forwarding) plane, which is responsible for performing what was determined in the control plane and route the data packets to the destination. Thus, these devices use special algorithms, which are, in general, a set of rules implemented in dedicated hardware components, to control and monitor the data traffic in the network.

The network operators are responsible for configuring each network device individually using low-level and often vendor-specific commands. As a result, legacy networks management is quite challenging and thus error-prone. [3] To cope with these challenges, a new concept of "programmable network" called Software Defined Networking (SDN) emerged [8].

SDN proposes the decoupling of the network's data and control plane by moving it to a centralized entity named controller. The network devices mentioned above became just forwarding devices with no "decision" capabilities to execute actions. Therefore the use of a centralized software-based controller which can be directly programmed and managed turned the network into a programmable entity.

2.2.1 Software Defined Networks Architecture

As seen in section 2.2, SDN revolutionized the way of managing and configuring networks by decoupling the data and control planes, allowing a centralized view of the network and creating an abstraction between the network infrastructure and the applications. Thus, resulting in more scalable and flexible networks that respond quickly to the needs of the network administrator. Figure 2.1 summarizes the SDN architecture abstractions in form of a detailed, high-level schematic. Starting from the upper part of the figure and moving towards the bottom, the following planes are identified [9].

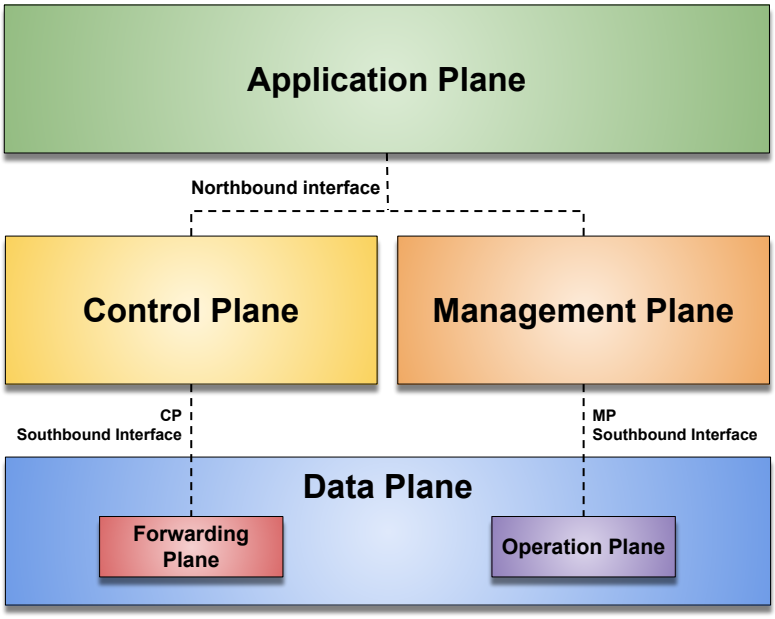


Figure 2.1: SDN architecture

- *Application plane:* Services and applications that define network behavior are located in this plane.
- *Control plane:* Responsible for deciding how packets should be forwarded by one or more network devices. The *Control Plane's* main job is to fine-tune the forwarding tables that reside in the *Forwarding Plane*, based on the network topology or external service requests. The *Control Plane* receives information from services in the *Application Plane* through the *Northbound Interface* (e.g., RESTful APIs).
- *Management plane:* Contrary to the *Control Plane*, the *Management Plane* focuses on the operation plane. It is responsible for monitoring, configuring, and maintain network devices.
- *Operation plane:* The *Operational Plane* is usually the end-point for managing services and applications. Its purpose is to manage the operational state of the network devices like the number of ports available, the status of each port, and the status of the device (i.e., active or inactive). It receives information from the Management Plane Southbound Interface and, the protocols used in this interface are vendor-specific;
- *Forwarding plane:* This plane is usually the end-point for control-plane services and applications. It is responsible for handling packets in the data plane based on the instructions provided by the control plane. Actions in this plane include forwarding, dropping, and changing the packet's headers. These actions are performed based on the rules provisioned by the Control Plane *Southbound Interface* (e.g., OpenFlow).

2.2.2 SDN Controller

As stated before, the separation of data and control planes allowed centralized control over the network as well as a new layer of abstraction between the network infrastructure and the applications.

The control plane (see Figure 2.1) accommodates the SDN controller. With a complete view over the network, the SDN controller manages flow control to the switches/routers "below" (via southbound APIs) and the applications and business logic "above" (via northbound APIs) to deploy intelligent networks. Two of the most well-known protocols used by SDN controllers to communicate with the switches/routers are OpenFlow [1] and open virtual switch database (OVSDB) [2].

There currently many SDN controller solutions such as OpenDaylight [3], ONOS [4], Floodlight [5] among others. [10]

2.3 Network Function Virtualization

The Network Function Virtualization (NFV) concept is based in virtualized network functions and migrating them from stand-alone boxes on dedicated hardware, to devices running on a cloud system [4].

A network service can be broken down into a set of network functions, which are then virtualized and executed on general-purpose servers or VMs. Thus, providing a higher degree of flexibility because NFVs can be dynamically instantiated, relocated, or destroyed, without the need to acquire and configure new and specific hardware [11].

NFV changed the way how network services are provisioned in comparison to traditional methods. In summary, these differences are as follows [12]:

- *Decoupling software from hardware:* As the network element is no longer a collection of integrated hardware and software entities, the evolution of both is independent of each other. This enables the software to progress separately from the hardware, and vice versa.
- *Flexible network function deployment:* The detachment of software from hardware helps reassign and share the infrastructure resources, thus together, hardware and software, can perform different functions at various times. The actual network function software instantiation can become more automated. Such automation leverages the different cloud and network technologies currently available. Also, this helps network operators deploy new network services faster over the same physical platform.
- *Dynamic scaling:* The decoupling of the functionality of the network function into instantiable software components provides greater flexibility to scale the actual VNF

[1] https://opennetworking.org/sdn-resources/customer-case-studies/openflow/
[2] https://www.openvswitch.org/
[3] https://www.opendaylight.org/
[4] https://opennetworking.org/onos/
[5] https://floodlight.atlassian.net/

performance in a more dynamic way and with finer granularity according to the actual traffic which, for instance, the network operator needs to provision capacity.

Although the virtualization of resources is the primal approach in the process of decoupling network functions from dedicated hardware, it is not the only one. Network operators could still develop NFs and run them on physical machines, but they need to ensure that these NFs can run on commodity servers. It is also possible to have hybrid scenarios where functions running on virtual resources co-exist with those running on physical resources.

2.3.1 Network Function Virtualization Architecture

The European Telecommunications Standards Institute (ETSI)[6] defined the NFV architectural framework in three main elements, the Network Function Virtualization Infrastructure (NFVI), the Virtual Network Function and Services and the Network Function Virtualization Management and Orchestration (NFV MANO), as shown in Figure 2.2.

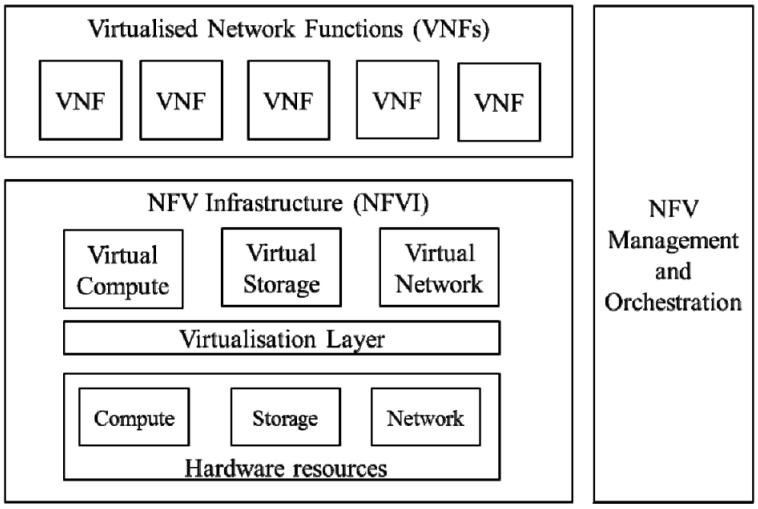


Figure 2.2: High Level NFV Framework

Source: [12]

2.3.1.1 Network Function Virtualization Infrastructure (NFVI)

The NFVI is a key element of the NFV architecture that describes the hardware and software components that build up the environment on which VNFs are deployed, managed and executed. The physical resources include computing, storage and network that provide processing, storage and connectivity to the VNFs. Unlike the traditional purpose-built hardware used to run NFs in legacy networks, the NFVI utilizes Commercial off-the-shelf (COTS) computing hardware. The virtual resources result of the abstraction of the hardware resources and the decoupling of the VNF software form the underlying hardware provided by the virtualization layer. Thus, guaranteeing a hardware-independent lifecycle for the VNFs

[6] www.etsi.org

and, therefore, the software can be deployed on different physical hardware resources. In this context, the OpenStack framework described in Section 2.4 acts as the NFVI of the proposed solution.

2.3.1.2 Virtual Network Functions and Services (VNFs)

A Network Function (NF) is a functional block within a network infrastructure that has well-defined external interfaces and well-defined functional behavior [13], such as firewalls, load balancers, DHCP servers etc.

Thus, NFs can be deployed in VMs in cloud environments as Virtual Network Functions (VNFs) and/or in bare-metal devices as Physical Network Functions (PNFs). The functional behavior, external operational interfaces and state of the NF are expected to be the same for PNFs and VNF.

A VNF can be deployed as a whole in a single VM or it can be composed of a set of functions that can be deployed over multiple VMs, in which case each VM hosts a single component of the VNF [12].

A service is, in general, implemented using one or multiple NFs combined. Nonetheless, in the user perspective, the services should have the same or better performance, whether running in PNFs or VNFs [14].

2.3.1.3 Network Function Virtualization Management and Orchestration (NFV MANO)

The NFV MANO handles the management and orchestration of all resources within the NFV framework, including compute, networking, storage, and VMs resources. The main focus of NFV MANO is to allow flexible onboarding, preventing the chaos that can be associated with the rapid spin-up of network components.

NFV MANO performs the orchestration and lifecycle management of the VNFs as well as the resources (physical and virtual) that support the infrastructure.

The NFV MANO is composed of three blocks: the Network Function Virtualization Orchestrator (NFVO), the Virtual Network Function Manager (VNFM) and the Virtualized Infrastructure Manager (VIM).

- *Network Function Virtualization Orchestrator:* The NFVO is responsible for the orchestration and lifecycle management of the NFVI and physical and/or software resources.
- *Virtual Network Function Manager:* The VNFM manages the VNF instance lifecycle, such as instantiation, update, query, scaling, termination, etc. A VNFM may be deployed per VNF or may serve multiple VNFs.
- *Virtualized Infrastructure Manager:* The VIM handles the interaction between the VNFs and the computing, storage and network resources. It is responsible for resource management like an inventory of software, VMs and resource allocation.

In this context, MAAS (see Section 2.5 can be classified as the NFV-MANO within the solution proposed. It manages and orchestrates the physical and virtual resources (made available by OpenStack) as well as the VNF and/or PNF that are deployed.

2.3.2 Virtual Network Functions

Virtual Network Functions are virtualized network services previously provided by proprietary and dedicated hardware. VNFs move individual network functions out of dedicated hardware devices into software that runs on generic hardware. Common services such as firewalls, DNS, caching, NAT, or virtualized routers, which can be deployed in virtual machines, are examples of VNFs.

2.3.3 Physical Network Functions

Unlike VNFs, Physical Network Function refers to a purpose built hardware box that provides specific networking function. Hardware routers, switches, firewalls, load balancers are examples of PNFs.

2.4 OpenStack

OpenStack is an open-source framework for managing, defining and use cloud resources. The official OpenStack website [7] defines it as: *"a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms. A dashboard is also available, giving administrators control while empowering their users to provision resources through a web interface."*

Typically to create a cloud computing environment, organizations resort to their existing virtualized infrastructure, using a well-established hypervisor such as VMware vSphere [8], Microsoft Hyper-V [9] or KVM [10]. But cloud computing offers more than just virtualization. Both public and private clouds also provide a high level of provisioning and lifecycle automation, user self-service, cost reporting and billing, orchestration and other features.

OpenStack features a "modular" architecture composed of a collection of open-source software modules that provide a framework to deploy and manage private and public cloud infrastructures. So, it arises as a "cloud operating system" that provisions and manages large pools of heterogeneous compute, storage and network resources.

The OpenStack enhanced the performing of these tasks, which usually required an IT administrator to do it, allowing these to be done through management dashboards and the OpenStack API.

[7] https://www.openstack.org/software/
[8] https://www.vmware.com/products/vsphere.html
[9] https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview
[10] https://www.linux-kvm.org/

As stated before, the OpenStack has a modular ecosystem broken up into services that allow users to plug and play components depending on their needs. The map depicted in Figure 2.3 gives an overview of the OpenStack ecosystem to see where those services fit and how they can work together. Due to the large number of services that compose the OpenStack framework, only those used in the course of this dissertation will be presented below.

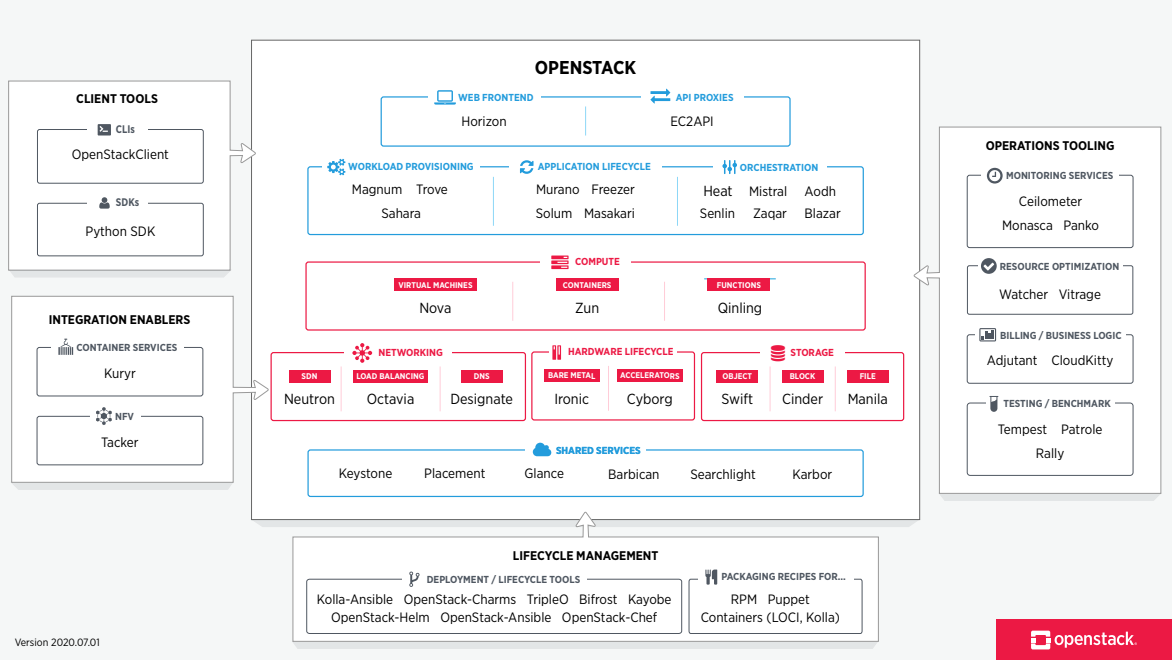


Figure 2.3: The OpenStack Services Ecosystem

Source: https://www.openstack.org/software/

- *Keystone:* The identity service, commonly known as Keystone, is responsible for managing all users and internal micro-services that constitute OpenStack. Each entity is assigned a username and password along with the information on who is allowed to do what.
- *Glance:* The Image service or Glance manages the virtual machine images allowing users to discover, register, and retrieve virtual machine images.
- *Nova:* Nova plays a vital role in the OpenStack architecture by interacting with the majority of services it acts as the manager of the cloud computing system. It manages the VMs lifecycle through drivers that communicate with the virtualization layer (i.e., with the hypervisor).
- *Neutron:* Neutron provides "networking as a service" between interface devices (e.g., vNICs) managed by other OpenStack services (e.g., nova). It allows users to create and manage network objects, such as networks, subnets and ports, which other OpenStack services can use through an API.
- *Horizon:* Horizon is a simple modular web interface, providing a user interface for cloud infrastructure management, allowing administrators and users to access a graphic interface easy to deploy.

- *Ironic:* Ironic is the bare-metal provisioning service in OpenStack. This service allows a customer to use hardware directly, deploying the workload (image) onto a real physical machine instead of a virtualized instance on a hypervisor. Ironic is triggered to launch a bare metal node by the Nova virtualization driver. With this Ironic *virt driver*, users can launch a bare metal server instance in the same way that they can currently launch a virtual machine (VM) instance.

2.5 Metal As A Service (MAAS)

Metal As A Service (MAAS) [15] is an open-source software from Canonical Inc, used for managing and monitoring bare-metal servers. Features like the automatic discovery of network devices, and zero-touch deployment of major Operating Systems (OSs) such as Ubuntu, CentOS, Windows, and Red Hat Enterprise Linux (RHEL) (Windows and RHEL require a Ubuntu Advantage [11] subscription to work correctly), made MAAS a popular cloud-native infrastructure management solution. MAAS supports several ways of interaction, from a powerful web-based interface, a command-line client, and an API allowing it to connect to external orchestration and modeling systems.

MAAS can be used to build the foundation on which cloud computing managing platforms like OpenStack run. Unlike OpenStack, MAAS does not add new layers between the OS (i.e., applications) and the hardware they run on.

However, MAAS facilitates the deployment process of OSs on physical servers as OpenStack does it with virtual machines.

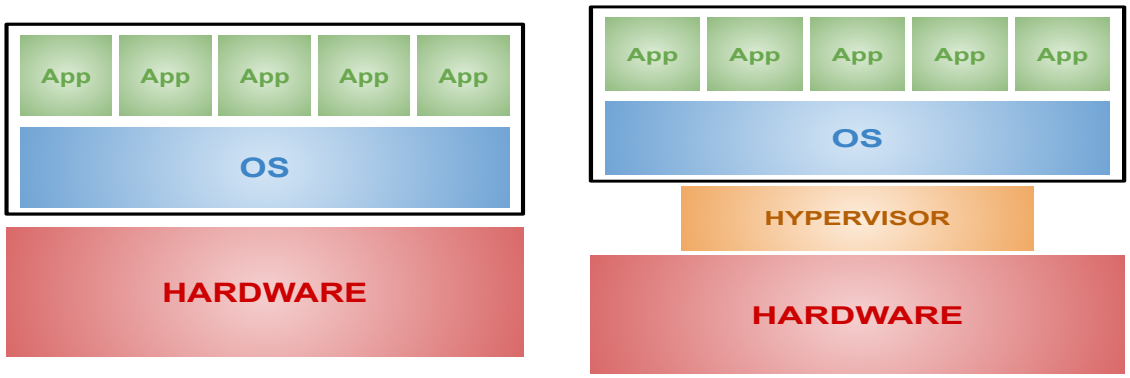
2.5.1 Bare-metal server

A bare-metal server is a physical machine designed to run services without interruptions for an extended period. In general, bare-metal servers are highly stable, durable, and reliable.

Unlike virtual machine servers that can be shared among multiple tenants through a hypervisor, bare-metal servers are dedicated to a single customer, meaning that they aren't shared between tenants.

A significant benefit of using bare-metal servers is the capability of accessing directly to the server and capitalize on all the underlying hardware. In other words, if someone provisions a virtual machine, it will get a guest OS sitting on top of a hypervisor sitting on top of physical hardware, as depicted in Figure 2.4b. Thus, it will only have access to the guest OS and the management interface used to create the VM. This does not happen with bare-metal servers because, as depicted in Figure 2.4a, the OS sits on top of the physical hardware giving the user full access to the underlying architecture, which increases the available options when creating a platform to host a service or application.

[11] https://ubuntu.com/advantage



(a) Traditional bare-metal server **(b)** Bare-metal server with hypervisor

Figure 2.4: Traditional and hypervisor bare-metal server comparison.

2.5.2 Supportive Cloud Tools

2.5.2.1 Curtin

Curtin [12] (short for Curt Installer) is a preseeding system developed by Canonical Inc., that applies customization during operating system (OS) image installation. System parameters such as the type of kernel, disk partitioning, network configuration, swap memory configuration, among others, can be preconfigured in a preseed file that will automatically configure images during the installation process. Thus, curtin allows for administrators to have their deployments equally set up all the time.

MAAS's internal deployment mechanism uses curtin to handle multiple operating system installations. To do this, it copies an image file from the Region controller to the hard disk and uses cloud-init for final configuration on the first boot. This image has to be in a specific format because MAAS interprets templates in lexical order by their filename.

The files used by curtin are located in the MAAS Region controller and when it needs to load a file, it chooses the one that is most specific to the machine and OS it is deploying. The process is based on the following priority:

1. {prefix}_{osystem}_{node_arch}_{node_subarch}_{release}_{node_name}
2. {prefix}_{osystem}_{node_arch}_{node_subarch}_{release}
3. {prefix}_{osystem}_{node_arch}_{node_subarch}
4. {prefix}_{osystem}_{node_arch}
5. {prefix}_{osystem}
6. {prefix}
7. 'Generic'

All user-data files use the same prefix "curtin_userdata", followed by the other specifications such as the node architecture and sub-architecture, release, and node name. These specifications follow the folder structure where MAAS stores images. Curtin will use the file

[12] https://curtin.readthedocs.io/en/latest/index.html

that suits the deployment starting from number 1 to the number 7 and applying a "Generic" file if there is no custom file defined.

So, for example, for CentOS, the MAAS folder structure is the following:

`/var/lib/maas/boot-resources/current/centos/amd64/generic/centos70/`

Thus, if it is necessary to create a custom user-data that will apply to all CentOS 7 installs on x86_64, it is required to create a file in `etc/maas/preseeds` called:

`curtin_userdata_centos_amd64_generic_centos70`

In the absence of a custom pressed file, MAAS uses a generic pressed file. This file simply includes the machine specifications defined by the user when it requests a deployment (i.e., the kernel to be installed, network configurations, and disk partitioning).

2.5.2.2 Cloud-init

Cloud-init [13] is the industry standard for early-stage initialization method for cross-platform cloud instances. It is used to specialize a generic operating system image at runtime by provisioning a set of configurations. Formerly developed by Canonical Inc. for configuring Ubuntu Linux running in Amazon EC2, it is now supported across all major public cloud providers, provisioning systems for private cloud infrastructure, and bare-metal installations.

The cloud-init package is installed in the operating system images supplied by most clouds. When started, cloud-init runs the commands in a sequence of modules to specialize the operating system installation for the intended purpose. This configuration comes from two sources:

1. Cloud provider-supplied metadata: This first configuration occurs before the beginning of the operating system installation. Several configurations can be applied in this stage and may involve setting up the network and storage devices to configuring SSH access keys and many other aspects of a system.
2. User-supply configuration: This happens after the machine has booted into the installed OS for the first time. The cloud-init will parse and process any optional user or vendor data that was passed to the instance.

The user can perform late configurations with custom scripts using cloud-init. These customizations are applied after the first boot when MAAS changes the machine's status to "Deployed".

Typically, the user-data custom files are written in YAML, but it is also possible to deploy Bash and Python files. After the script is written, the user needs to convert it to a base64 encoded file and then deploy it along with the machine over the API, as following:

`maas <user> machine deploy <system_id> user_data=<base64 encoded user-data>`

Customizations are per-instance, meaning that user-supplied scripts must be re-specified on redeployment. Thus, if a user needs its script to run on every machine boot, it needs to specify that in the user-data file, otherwise, it will only run one time on the first boot after the OS install.

Cloud-init customizations are the best way for MAAS users to customize their deployments.

[13] https://cloudinit.readthedocs.io/en/latest/index.html

2.5.2.3 Ephemeral image

MAAS uses an ephemeral image: a lightweight operating system, that allows for node controlling during the node deployment process. An ephemeral image consists of a kernel, a RAM disk, and a squashfs file-system that is booted over the network (i.e., PXE boot). Ephemeral images, use cloud-init to discover the node's hardware (e.g., number of CPUs, RAM, disk, etc.) and send that information to the MAAS Region controller.

2.5.3 MAAS Architecture

MAAS has a layered architecture as depicted in Figure 2.5 which allows for easy infrastructure coordination and integration. The two main elements in the MAAS architecture are the region controller (*regiond*), and rack controller (*rackd*).

2.5.3.1 Region Controller (*regiond*)

Region controllers are responsible for a single region or a data center. As shown in Figure 2.5, MAAS uses *fabrics* to accommodate subdivisions within a single region (e.g. multiple departments in a company). With the responsibility of dealing with operator requests, five components make the region controller: a REST API server and a web UI which provides user connectivity to the system, a PostgreSQL database that holds all the data for the MAAS environment (e.g., OS images, server details, and user credentials), DNS, and caching HTTP proxy which provides a way for its managed machines to use a proxy server when they need to access HTTP/HTTPS-based resources, such as the Ubuntu package archive.

2.5.3.2 Rack Controller (*rackd*)

The rack controller (*rackd*) provides local bare-metal with DHCP/BOOTP, TFTP, iSCSI, and HTTP services, stores operating system OS images in its local disk, and acts as a PXE server which is required for commissioning and deploying machines.

As defined in [16]: "Preboot Execution Environment(PXE) is a client-server interface that allows computers in a network to be booted from the server before deploying the obtained PC image in local and remote offices, for PXE enabled clients. PXE network boot is performed using client-server protocols like DHCP(Dynamic Host Configuration Protocol) and TFTP(Trivial File Transfer Protocol). PXE will be enabled by default on all computers."

As seen in Figure 2.5 each *fabric* has a rack controller attached to it.

2.5.3.3 Fabrics

A *fabric* connects VLANs. The operating principle of a VLAN is based on only allowing network connections between specific switch ports or specifically identified ports ("tagged" ports), which makes it impossible for two VLANs to communicate with each other. Thus, the use of fabrics allows these VLAN-to-VLAN connections, which are required in certain use

cases (e.g. allowing different departments within the same company to share data). MAAS creates a default fabric (‘*fabric-0*’) for each detected subnet during installation.

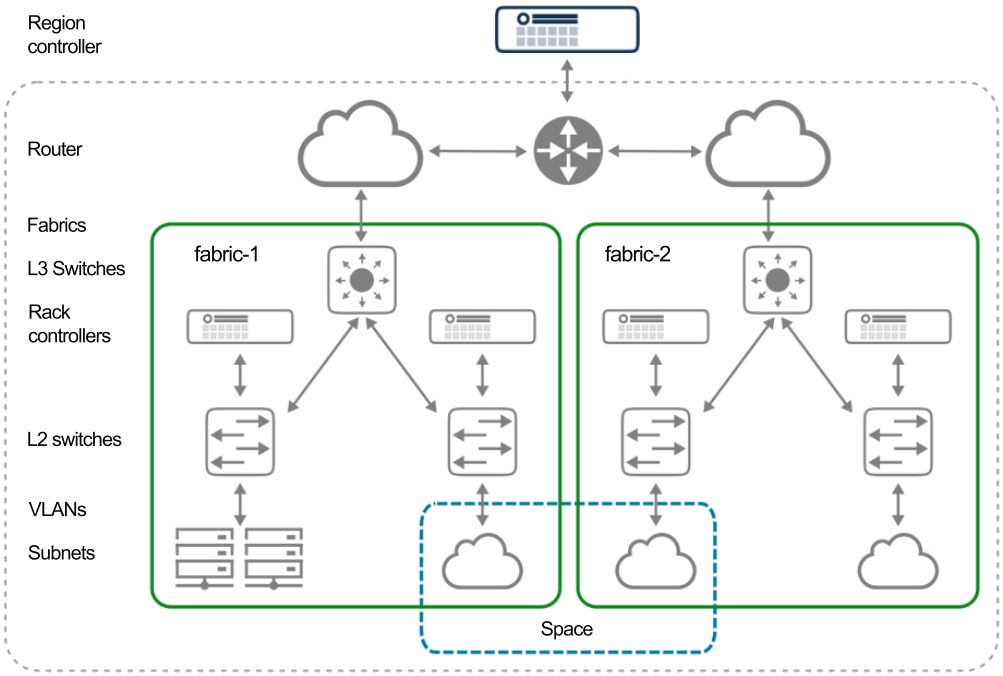


Figure 2.5: MAAS architecture overview (example)

Source: https://maas.io/docs/concepts-and-terms

2.5.4 Node lifecycle

When a new machine ("node") arrives at the MAAS network, it undergoes a series of events - from its enlistment to the point a user can deploy it.

Figure 2.6 illustrates the major events in a node lifecycle:

1. **New:** New machines can be enlisted automatically using PXE-boot or add manually by the administrator.
2. **Commissioning:** All relevant data such as RAM, CPUs disk, NICs, and accelerators like GPUs are listed to be used as constraints for machine selection.
3. **Ready:** After successful commissioning, a machine is considered "Ready" for MAAS to control it.
4. **Allocated:** At that point, nodes are Ready and can be allocated to users. Once Allocated, a node steps out from the available nodes and can’t be deployed from a different user.
5. **Deploying:** Users can request that MAAS to turn on a machine and install a complete OS without manual intervention.
6. **Releasing:** When the purpose of the node is achieved the user can release it. MAAS will wipe the disk and put it back on the available ("Ready") resource pool.

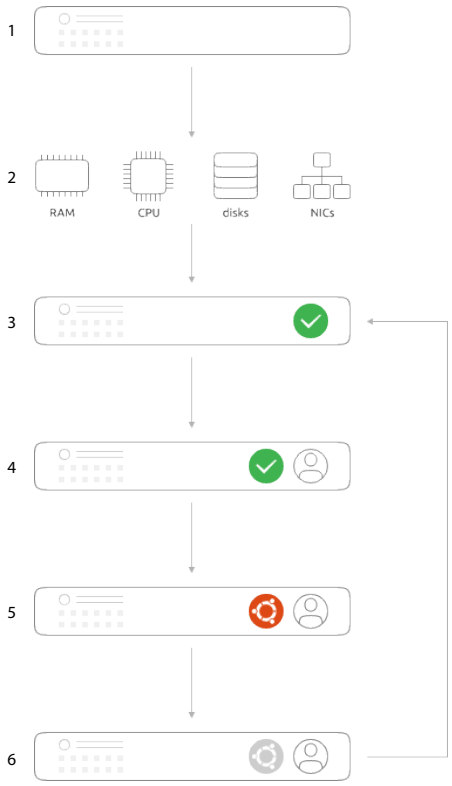


Figure 2.6: Node lifecycle

Source: https://maas.io/how-it-works

2.5.5 MAAS VM Hosting

MAAS allows for machines to be used as VM hosts. A VM host is a machine that can run virtual machines by dividing its resources (CPU cores, RAM, storage) among the number of VMs. Once MAAS has enlisted, commissioned, and acquired a newly-added machine, it can be deployed as a VM host. MAAS currently supports VM hosts and VMs created via libvirt [14]. Future MAAS releases are expected to support LXD [15] VMs and VM hosts as a beta feature.

Moreover, the MAAS release 2.7 introduced the Non-Uniform Memory Access (NUMA), feature. NUMA is a useful way of achieving high-efficiency computing, by pairing a CPU core with a very fast connection to RAM and PCI buses. A NUMA node is formed by a pair of a CPU and its dedicated RAM, which reduces memory access times, so the core won't spend a lot of time waiting for access to data in memory. By default, machines are assigned to a single NUMA node that contains all the machine's resources.

[14] https://ubuntu.com/server/docs/virtualization-libvirt
[15] https://linuxcontainers.org/lxd/introduction/

2.5.6 High Availability in MAAS

MAAS is a service that provides infrastructure coordination upon which cloud infrastructures depend so its availability is crucial.

High Availability (HA) in the region controller is achieved at the database level. The region controller will automatically switch gateways to ensure high availability of services to network segments in the event of a rack failure.

Enable HA in the rack controller level is a straightforward task. First, it is necessary to install multiple rack controllers to achieve real high availability. Once new rack controllers are installed, MAAS automatically identifies which rack controller is responsible for the BMC control (node power cycling) and sets up communication between rack controllers. As stated in Subsection 2.5.3.2, rack controllers are responsible for providing DHCP to the nodes. Therefore, it is necessary to enable DHCP HA by allowing primary and secondary DHCP instances to serve the same VLAN. This VLAN will afterward replicate all lease information between rack controllers.

2.5.7 MAAS Communication

MAAS has a very well defined communication pattern, from the UI/API through the region controller, to the rack controller, to the machines (and back). The addition of more controllers does not change the flow of communication.

2.5.7.1 Region and Rack controllers communication

There are several different ways in which the MAAS region and rack controllers interact. These may vary depending on the operation that the user requests. One of the usage examples is a machine commissioning process that consists of collecting information on its available resources (e.g., CPU, RAM, storage).

The example depicted in Figure 2.7, is a simple representative of the communication between rack and region controllers during a machine commissioning process. The description process was taken from [17]:

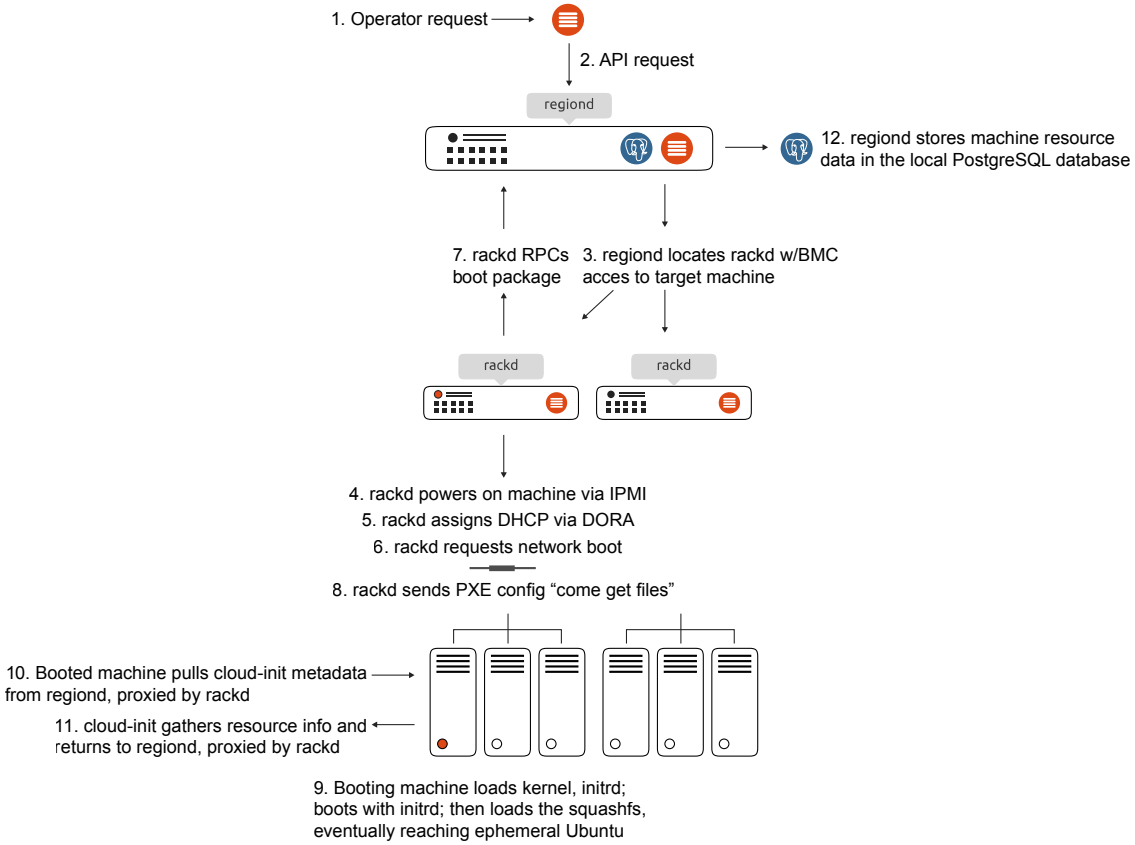


Figure 2.7: MAAS Communication Diagram

Source: https://maas.io/docs/snap/2.9/ui/maas-communication

1. An operator makes a request of MAAS, either via the Web UI or the API.
2. MAAS translates this to an API request to the region controller.
3. The region controller locates the rack controller that has BMC access to the machine in question, that is, the rack controller that can power on that machine.
4. That same rack controller powers on the machine via IPMI request.
5. The rack controller tasked with providing DHCP handles assigning an IP address to the machine via the DORA sequence (Discover, Offer, Request, Acknowledge). Note that this rack controller doesn't have to be the same one that powers on the machine.
6. The DHCP-managing rack controller inserts itself as the DHCP "next-server" and requests a network boot.
7. (Still) the same rack controller RPCs the region controller to get what's needed to boot an ephemeral Ubuntu kernel, namely the kernel, any kernel parameters, an *initrd* daemon, and a *squashfs* load.
8. That same rack controller transforms the RPC response from the region controller into a valid PXE config and tells the machine to come get its files.
9. The booting machine loads the kernel and *initrd*, boots with that *initrd*, and then loads the *squashfs*, eventually making its way up to an ephemeral Ubuntu instance.

10. The booted machine pulls *cloud-init* metadata from the region controller, *proxying* through the *rackd*.
11. *cloud-init* uses this metadata to gather resource information about the machine and pass it back to the region controller, again *proxied* by the *rackd*.
12. The region controller (*regiond* or "region daemon") stores this machine information in a postgres database that is accessible only to the *regiond*, making MAAS truly stateless with respect to machines.

2.5.7.2 Machines and Rack controller communication

MAAS sets up an internal DNS domain, which is not manageable by the user, and a unique DNS resource for each subnet that it manages. When a machine is booting, it uses the subnet DNS resource to determine which rack controller is available for communication. In the case of multiple rack controllers are available in the same subnet, MAAS uses a round-robin algorithm to balance the load across the controllers, ensuring that machines always have a rack controller. Thus, all communications between the machines and MAAS are proxied by the rack controllers.

2.6 Juju

Juju [18] is an open-source software from Canonical Inc. that leverages the deployment, configuration, management, maintenance, and scale of cloud applications on public clouds, as well as on physical servers. In short, Juju is a software solution that provides service orchestration.

Nowadays, current working environments are characterized by the use of multiple applications together. Even simple applications may require other applications to operate. For example, to model a complex system like OpenStack (see Section 2.4) requires multiple services/applications to be installed, configured, and connected to each other, which can be an intricate process.

Thus, Juju's modeling system leverages the ability to deploy and manage services quickly and more easily. Applications can be encapsulated in service definitions files called "*charms*", which can subsequently be instantiated to deploy a service in seconds. This process enables to quickly scale services up or down without disruption for the cloud environment.

2.6.1 Juju Workflow

The typical approach when deploying applications with Juju is summarized in the four steps illustrated in Figure 2.8. To describe the methodology it's going to be used OpenStack as an example.

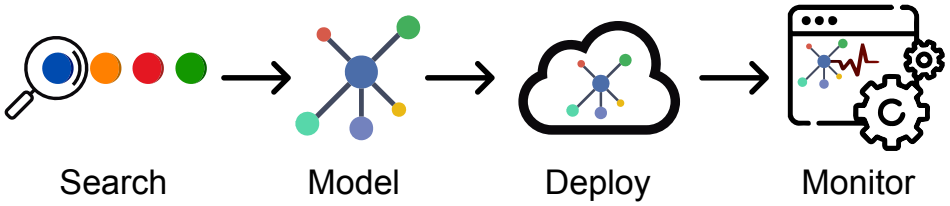


Figure 2.8: Juju Workflow Concept

OpenStack is composed of many services (e.g., Nova, Neutron, Horizon, Glance, etc.) that run and interact with each other. Thus, the first step is to "*Search*" for the appropriate charms that the project will need. A Juju charm, as mentioned before, contains all the instructions necessary for deploying and configuring application units as stated in [19]. There are a variety of charms for hundreds of popular cloud-oriented applications in the Juju charm store [16]. For example, there is a charm for each OpenStack service [17]. However, if there isn't a charm to a particular application, it can be easily developed because charms can be written in any language or configuration management scripting system. This allows for users that have existing scripts (e.g., for Puppet, Chef, Bash, etc.) to use it as a starting point for a new charm.

The next step is to build a "*Model*" of the service using the charms selected in the previous step. A simple way to describe a Model is as a workspace that enables the users to maintain an organized view of the service. After choosing the charms, users must create a Model for the service they want to deploy and built the charms relations within it. Using the OpenStack example, the Model [20] illustrated in Figure 2.9 contains all the needed charms as well as the relations between them.

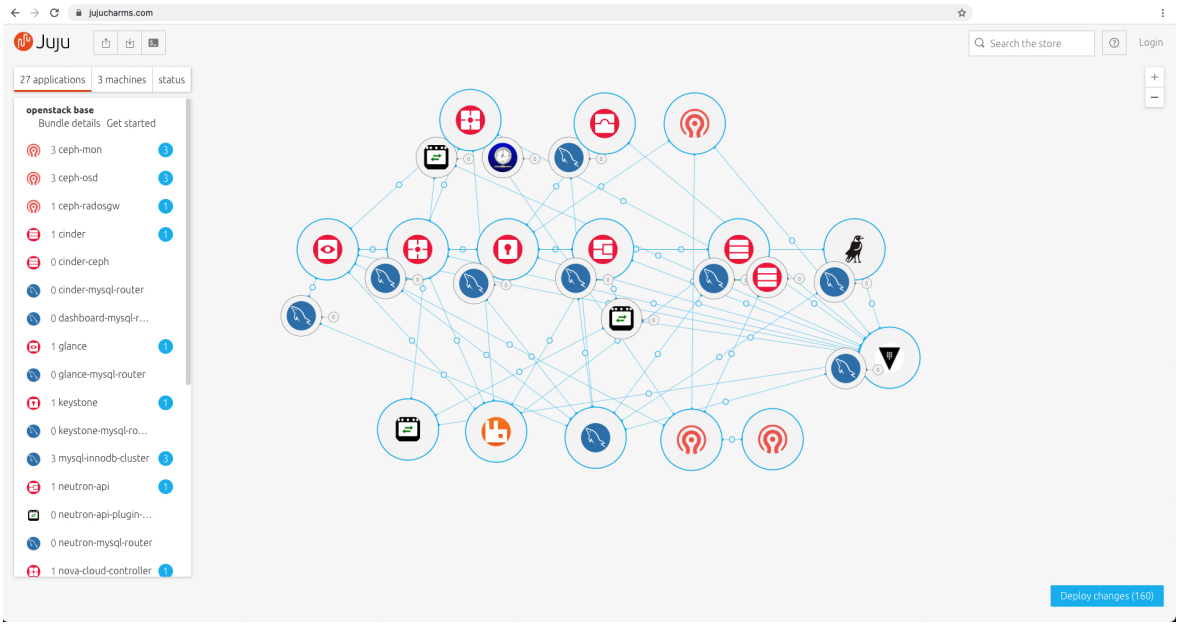


Figure 2.9: Juju OpenStack Model (Example)

[16] https://jaas.ai/store
[17] https://jaas.ai/u/openstack-charmers#charms

Once a model is built and functional, it becomes simple to export a model's definition as a bundle, then re-deploy that model in another host. A bundle is a feature that makes it possible to combine multiple charms and save them to reuse in another model, which enables the automation of a multi-charm solution.

When the service Model is configured and running, the next step is to deploy it. Juju supports a wide variety of private and public clouds (e.g., Amazon AWS, Microsoft Azure, Google GCE, MAAS, etc.), as well as common servers. A Juju controller must be bootstrapped in a cloud before a model could be deployed. A Juju controller is the management node of a Juju cloud environment, whose main goal is keeping the state of all the models, applications, and machines in that environment.

The last step is to "*Monitor*" and maintain the services deployed. This can be done through the web UI as well as the CLI available.

2.7 Existing Test Infrastructures

The constant growth of networks demanded new ways to test what was being developed. The Advanced Mobile wireless playGrouNd (AMazING) [18] testbed is an outdoor system located on the rooftop of Instituto de Telecomunicações at the University of Aveiro. The testbed consists of 24 fixed nodes in addition to 1 mobile node as depicted in Figure 2.10. The nodes are approximately 8m apart from each neighbor and distributed over $1200m^2$.

This testbed was deployed to support researches on next-generation networks NGN. It provides users with full access to the node devices, which allows for a high degree of controllability for the experimenter, as well as, high reproducibility of the tests. The testbed has several support servers to provide processing power to analyze results and a redundant storage device that serves all files at its core.

[18] http://amazing.atnog.av.it.pt/

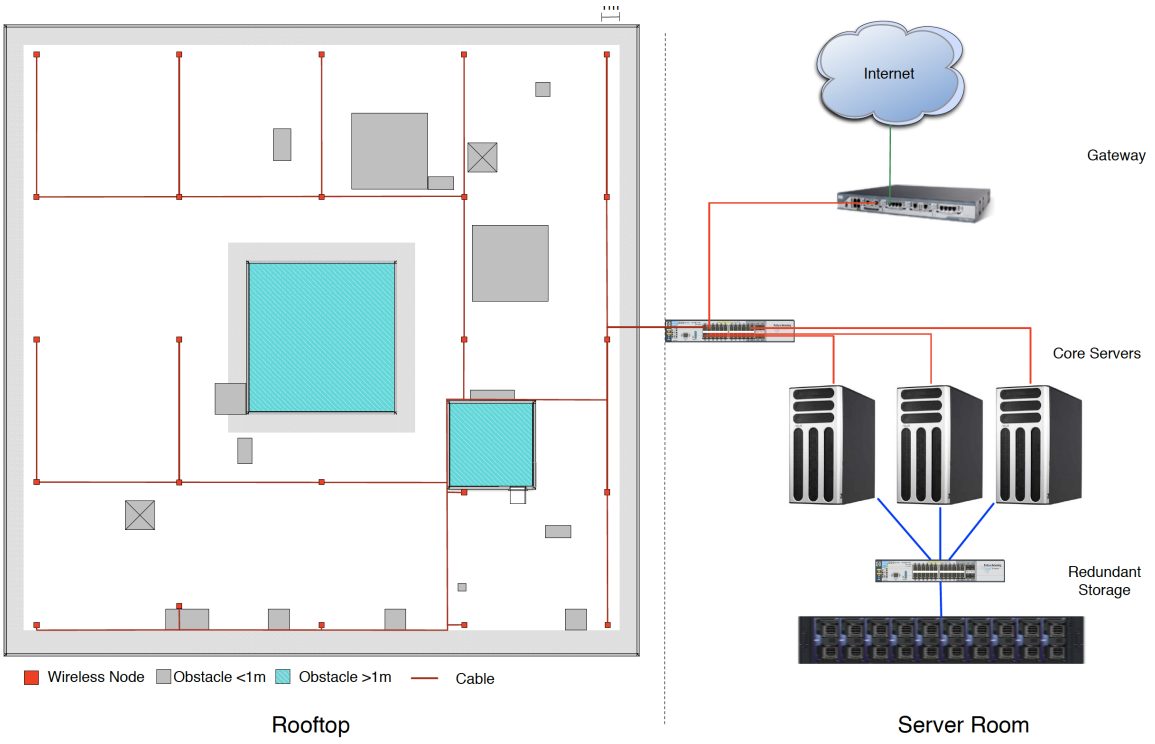


Figure 2.10: AMazING Testbed Overview

Source: [6]

The following examples describe the most advanced testbeds that can be found in the Fed4Fire+[19] project. As defined in [21]: "Fed4FIRE+ is a project under the European Union's Programme Horizon 2020, offering the largest federation worldwide of Next Generation Internet (NGI) testbeds, which provide open, accessible and reliable facilities supporting a wide variety of different research and innovation communities and initiatives in Europe, including the 5G PPP projects and initiatives." There are currently 18 testbeds federated with Fed4FIRE with different structures and implementations to attend different needs and goals.

An example is the IRIS[20] testbed which provides virtualized radio hardware, software virtualization, Cloud-RAN, NFV, and SDN technologies to support the experimental investigation of the interplay between future networks. The Figure 2.11 illustrates the Iris testbed architecture.

[19] https://www.fed4fire.eu/testbeds/
[20] http://iristestbed.eu/

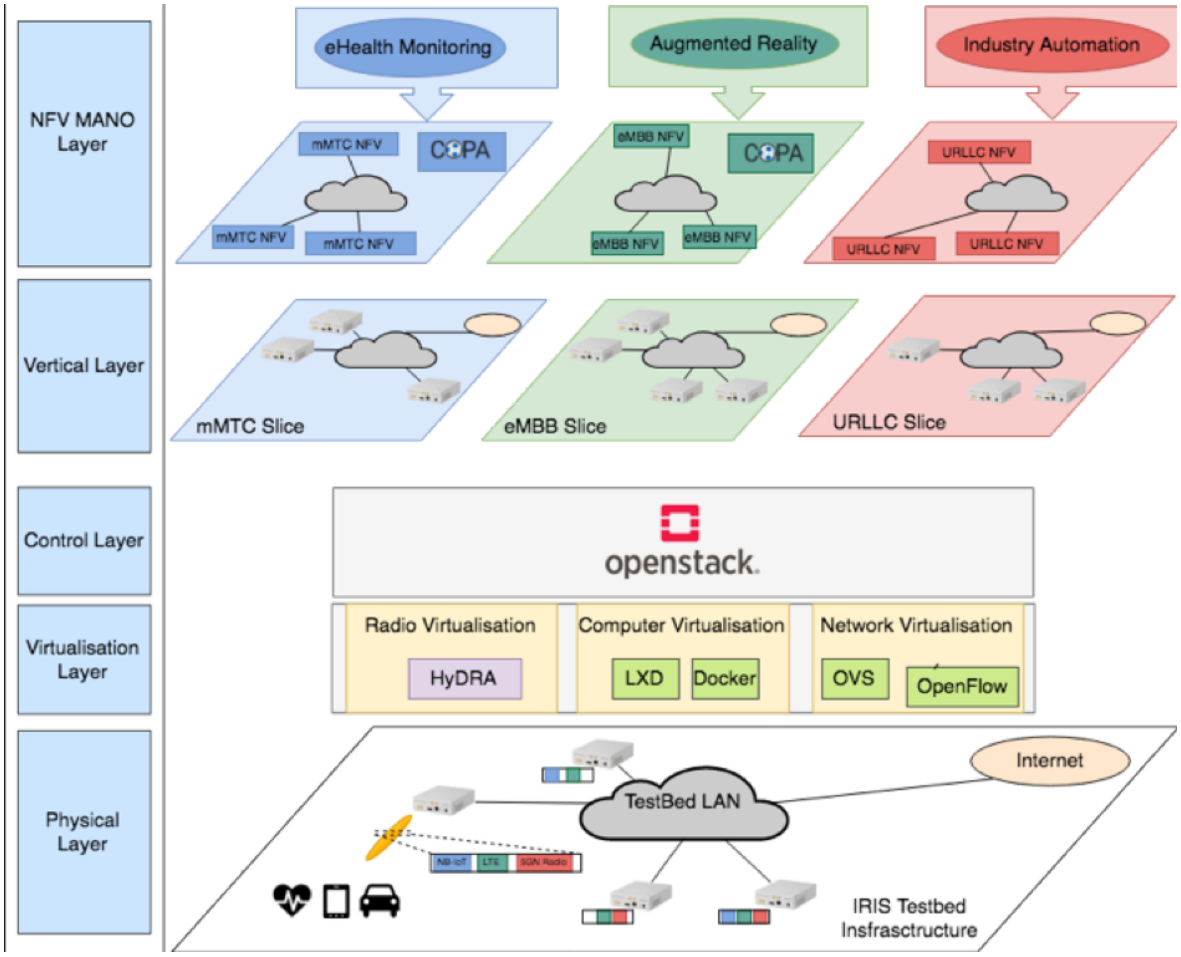


Figure 2.11: Iris Testbed Overview

Source: [22]

For instance, such a testbed as Iris proposes [23]:

> "The physical layer, at the bottom, represent the tangible resources including servers, switches, USRPs, and so forth, at the Iris testbed. The virtualisation and control layers in the middle are supported by software virtualisation technologies such as OpenStack to support cloud computing, OpenFlow to orchestrate and manage the USRPs and physical network equipment. The vertical and NFV MANO layers, are supported by the Open Source Network Function Virtualization (NFV) Management and Orchestration (MANO) (OSM) software stack. These elements interact with the physical and virtualisation layers to dynamically instantiate radio Experimental Vertical Instances (EVIs) at the Iris testbed."

Another example is the CityLab [24] testbed, which enables researchers to perform experiments in areas such as: cross-technology heterogeneous network, bare-metal outdoor networks, and smart city IoT networks. With currently 32 locations hosting the hardware, CityLab offers a close-to-real environment testbed with nodes installed in the streets. An interesting characteristic of this testbed is the use of PCEngines APU2C4 as the bare-metal node. However, it lacks virtual capabilities, not allowing users to combine virtual machines

and bare-metal. Figure 2.12 illustrates the CityLab architecture overview. According to [25], a data center of the University of Antwerp hosts an EmuLab-based experiment management system [26] to control a set of gateways (i.e., testbed nodes) distributed in the city of Antwerp, Belgium. Users use jFED [21] to access the experiment system.

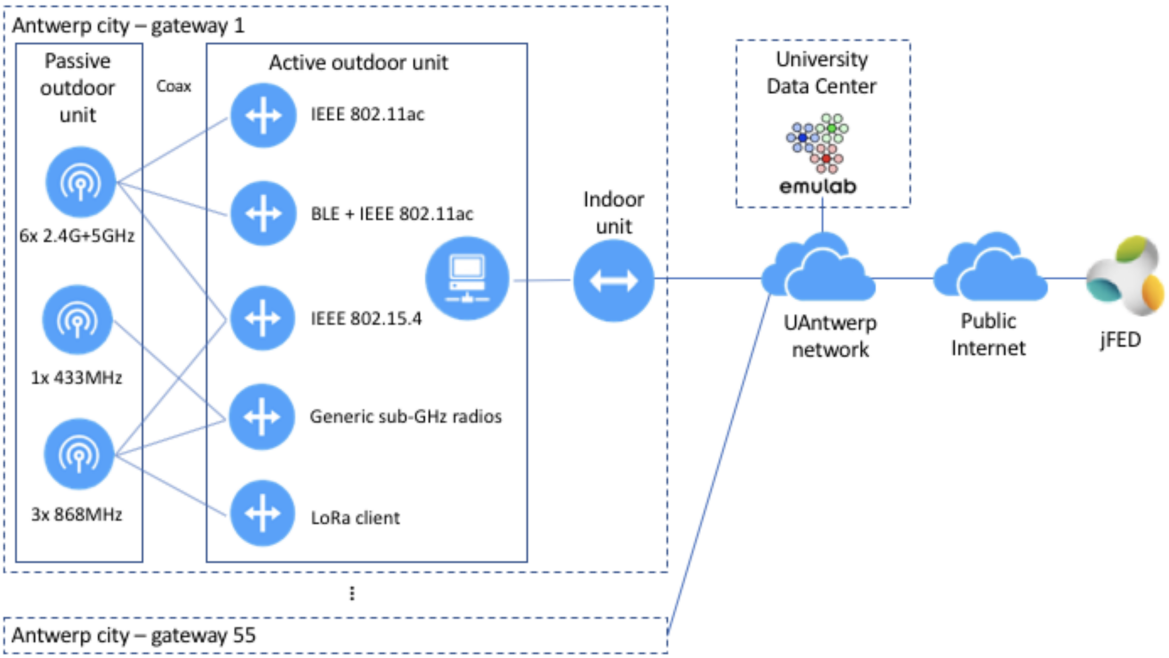


Figure 2.12: CityLab Architecture Overview

Source: [25]

The last example is the Network Implementation Testbed using Open Source platforms or NITOS [22] testbed. Currently running over 100 operational wireless nodes, this testbed focuses on supporting experimentation-based research in the wired and wireless networks area. The NITOS testbed is formed of an outdoor testbed that consists of powerful nodes that feature multiple wireless interfaces and allow for experimentation with heterogeneous (Wi-Fi, WiMAX, LTE) wireless technologies, and an indoor testbed that consists of 40 Icarus nodes and is deployed in an RF isolated environment. Figure 2.13 illustrates the NITOS architecture.

[21] https://jfed.ilabt.imec.be/
[22] https://nitlab.inf.uth.gr/NITlab/nitos

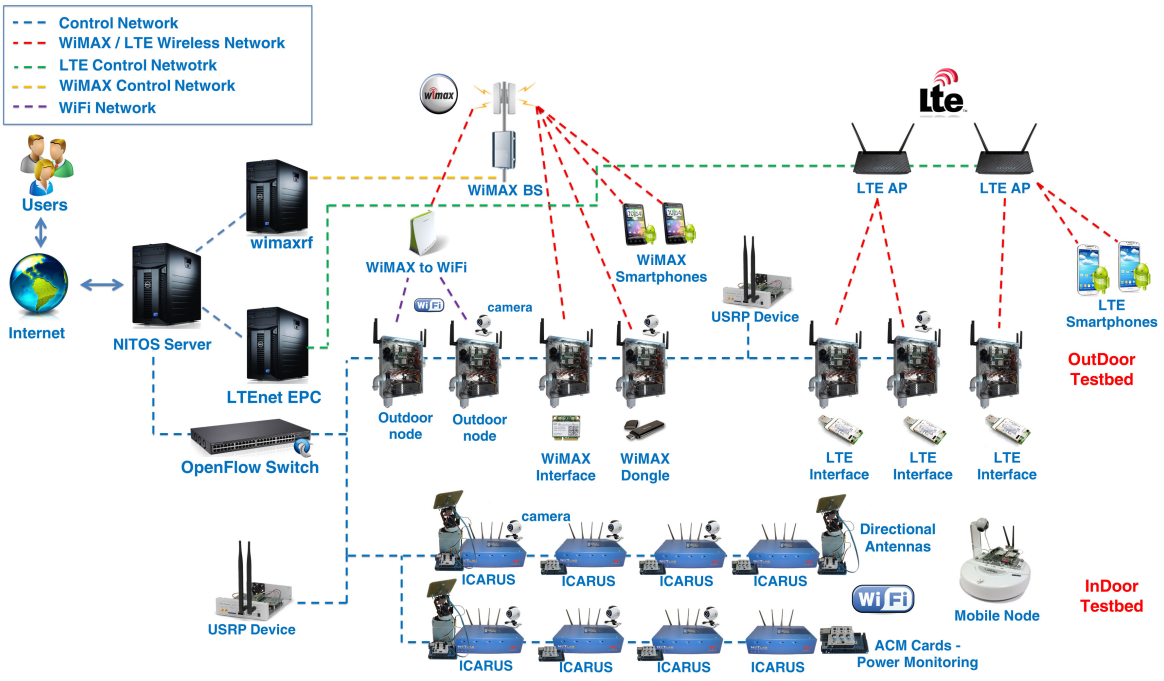


Figure 2.13: NITOS Facility Architecture

Source: [27]

According to [28] the main experimental components of NITOS are:

- A wireless experimentation testbed, which consists of 100 powerful nodes (some of them mobile), that feature multiple wireless interfaces and allow for experimentation with heterogeneous (Wi-Fi, WiMAX, LTE, Bluetooth) wireless technologies.
- A Cloud infrastructure, which consists of 7 HP blade servers and 2 rack-mounted ones providing 272 CPU cores, 800 Gb of Ram and 22TB of storage capacity, in total. The network connectivity is established via the usage of an HP 5400 series modular OpenFlow switch, which provides 10Gb Ethernet connectivity amongst the cluster's modules and 1Gb amongst the cluster and GEANT.
- A wireless sensor network testbed, consisting of a controllable testbed deployed in UTH's offices, a city-scale sensor network deployed in Volos city and a city-scale mobile sensing infrastructure that relies on bicycles of volunteer users. All sensor platforms are custom, developed by UTH, supporting Arduino firmware and exploit several wireless technologies for communication (ZigBee, Wi-Fi, LTE, Bluetooth, IR).
- A Software Defined Radio (SDR) testbed that consists of Universal Software Radio Peripheral (USRP) devices attached to the NITOS wireless nodes. USRPs allow the researcher to program a number of physical layer features (e.g. modulation), thereby enabling dedicated PHY layer or cross-layer research.

- A Software Defined Networking (SDN) testbed that consists of multiple OpenFlow technology enabled switches, connected to the NITOS nodes, thus enabling experimentation with switching and routing networking protocols. Experimentation using the OpenFlow technology can be combined with the wireless networking one, hence enabling the construction of more heterogeneous experimental scenarios.

2.8 Summary

This chapter provided the state of the art for the remainder of the dissertation. Some concepts and technologies that are important to the deployment of the described in Chapter 3 were defined. The cloud environment where the proposed system will reside, required to define Cloud Computing. As the work relies on the capacity to use network functions on the virtual and physical plane it was needed to define NFV. For SDN and NFV, both their architectures were described and detailed. OpenStack was also presented, as well as the most important projects of this software. This framework was used to create the cloud with which the system will work with. Then, both MAAS and Juju architectures were detailed as well as their key features that were important to the dissertation. MAAS acts as the main component, which is responsible for manage requests from clients and deploy either the VNFs or PNFs. Lastly, were given some examples of current testbeds.

CHAPTER 3

Scenario Description and Proposed Architecture

This chapter presents a description of the use-case scenario and the architecture design for testing the overall performance of the testbed implemented using the OpenStack, MAAS, and Juju technologies.

3.1 Scenario

The AMazING [6] testbed is an outdoor system that was deployed on the rooftop of ITAV. The testbed consists of 24 fixed wireless nodes forming a grid, in addition to a mobile node. In addition, the ITAV has a local OpenStack cloud already deployed and running, which allows its exploration for the integration of the AMazING testbed in order to provide virtualization capabilities to the physical wireless nodes.

Therefore, the system architecture proposal aims at the development of a testbed at Instituto de Telecomunicações at the University of Aveiro (ITAV), to automate wireless experiments while taking advantage of its local cloud infrastructure and enabling the physical wireless testbed to cope with virtualization research trends (e.g., develop and evaluate use cases where a PNF is dynamically instantiated for providing an alternative wireless access technology for a certain end-device).

3.2 Architecture

In this context, the system architecture was designed to resemble the reality of the existing system of ITAV, where a local OpenStack cloud is already deployed and running. As such, a dedicated OpenStack instantiation was used in parallel to the production OpenStack, allowing to minimize the allocation of resources and to avoid an invasive and destructive approach. OpenStack has a service that handles bare-metal servers (i.e., Ironic), as stated in Section 2.4.

However, Ironic needs to be connected to the other OpenStack services meaning that it would require stopping the cloud to set it up, which would be a very time-consuming process and would fail the non-evasive objective that was set at first.

Thus, MAAS was chosen to handle the bare-metal as well as the OpenStack VMs. MAAS was detailed in Section 2.5.

Figure 3.1 depicts the system architecture, where the *PC-OpenStack* acts as the cloud environment, which deploys the *MAAS-controller* and *Juju-controller*. Network entities are presented in the following subsections.

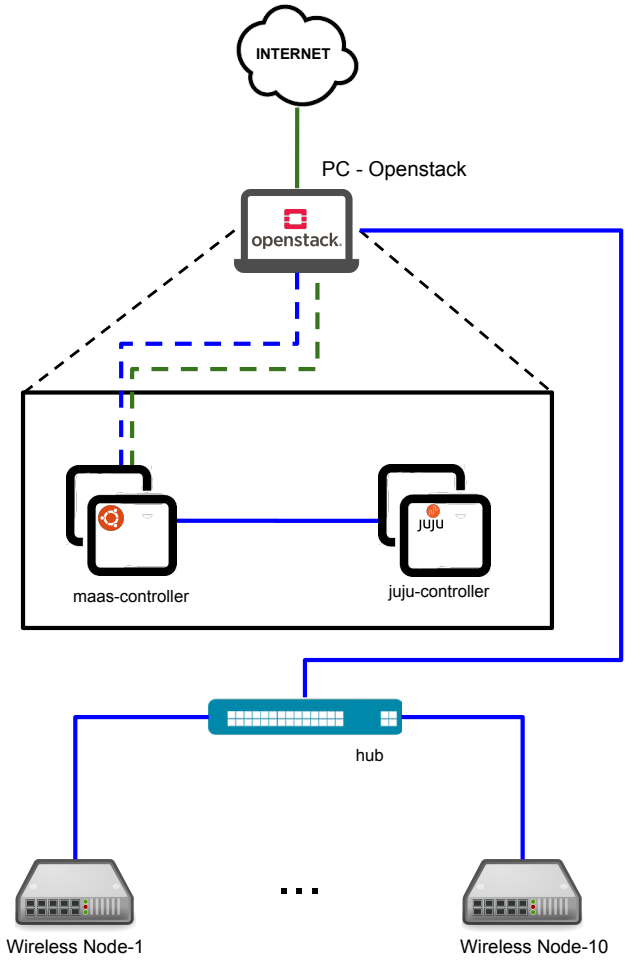


Figure 3.1: System architecture overview.

3.2.1 OpenStack Cloud Environment

The *PC-OpenStack* operates as a VIM running OpenStack, and consequently, it is responsible for managing the virtualized infrastructure of the of the system. The *PC-OpenStack* deploys both the *MAAS-controller* and *Juju-controller* as VMs, and provides the *MAAS-controller* with two physical network interfaces: one is connected to the Internet (green line in Figure 3.1), providing connectivity to all the other elements; the second provides connection to the physical wireless nodes (blue line in Figure 3.1).

3.2.1.1 PC-OpenStack specifications and configuration

The dedicated OpenStack Server deployment was installed and configured in a DELL XPS 13 (2016), with a Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz, 8GB of LPDDR3-SDRAM and 256GB of SSD storage. The machine is running Canonical Ubuntu 18.04 LTS desktop version.

As seen in Section 2.4, the OpenStack framework has a modular architecture composed of several services. First, it was necessary to install the following services for a minimal deployment of OpenStack Stein: Identity (keystone), Image (glance), Placement, Compute (nova), Networking (neutron), and Dashboard (horizon).

Then, using neutron, two networks were configured: a provider network (green line on Figure 3.1 - 192.168.1.0/24) to give Internet access to the instances; and a second, private network (blue line on Figure 3.1 - 10.10.10.0/24), to work as MAAS internal network to deploy and manage the wireless nodes as depicted in Figure 3.2.

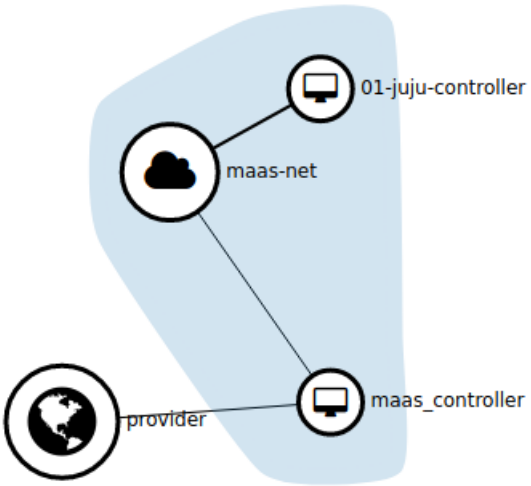


Figure 3.2: OpenStack network graph

The enlisting of new machines in MAAS is typically done by the combination of DHCP, TFTP, and PXE. If one machine connected to the MAAS network is configured to netboot, MAAS will automatically enlist it. So to enable this enlist process, two changes had to be done. First, OpenStack doesn't have PXE boot configured for its instances to use. The method described in OpenStack documents to allow PXE boot has two issues: (1) it is only described to enable the feature with the Ironic (bare-metal service), and (2) it requires a DHCP and/or TFTP servers to enable it. This does not fit the purpose of the project since it is imperative that MAAS has full control of the PXE boot process. Therefore, a new image was built using the method described in [29], which:

1) creates a small empty disk file and it DOS filesystem;
2) makes it bootable by syslinux;
3) install iPXE kernel;

4) make the iPXE kernel to load at bootup in the *syslinux.cfg* file.

The second change is related to the network. As stated before the machines (wireless nodes) have to be in the same physical network as the MAAS controller to be able to PXE boot. As such, the physical interface on PC-OpenStack (see hub on Figure 3.1) was bridged to the private network created on OpenStack, allowing all machines to go through the enlisting process. Figure 3.3 shows the bridge details: in orange the OpenStack private network (*maas-net*) in yellow the physical interface from the *PC OpenStack*.

```
atnog@atnog:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
brq1a6983fd-e9          8000.b273a7dc62c3       no              enxd037456cc6e9
                                                                tap330ec983-78
                                                                tap860663e0-38
brq6bb2f5ff-fe          8000.d037456cc5c6       no              enxd037456cc5c6
                                                                tap5db4e172-45
                                                                tap8956d623-79
                                                                vxlan-55
virbr0          8000.5254008085a6       yes             virbr0-nic
```

Figure 3.3: Network bridging in OpenStack

Finally, two VMs were deployed with the following specifications:

1. *maas-controller*
   - 2GB of RAM
   - 2 vCPU
   - 40GB of storage
   - OS: Ubuntu 18.04 LTS (server)
   - Interfaces:
     - ens3 (*provider*):
       - IP: 192.168.1.250/24
       - MAC: fa:16:3e:f2:7f:29
     - ens4 (*maas-net*):
       - IP: 10.10.10.199/24
       - MAC: fa:16:3e:e9:d1:5e
2. *01-juju-controller*
   - 3.5GB of RAM
   - 1 vCPU
   - 10GB of storage
   - OS: Custom PXE-boot image
   - Interfaces:
     - ens3 (*maas-net)*:
       - IP: 10.10.10.216/24
       - MAC: fa:16:3e:62:93:dc

3.2.2 MAAS Controller

In this proposal, the MAAS region and rack controller (see Subsection 2.5) were installed in the same machine. The machine used was a VM within the *PC-Openstack* named *maas-controller* (see Sub-subsection 3.2.1.1).

The first step was to install and integrate MAAS (version: 2.7.3 (8290-g.ebe2b9884)) on the VM using the Advanced Package Tool (APT) from Ubuntu. Once MAAS was installed, an administrator was created along with the login credentials using the MAAS CLI. From this point, a web UI was accessible to continue the configuration.

When the web UI is accessed for the first time, it prompts a different screens to the administrator in order to set several system options, including connectivity (e.g., DNS forwarder), image downloads, and authentication keys as depicted in Figure 3.4.

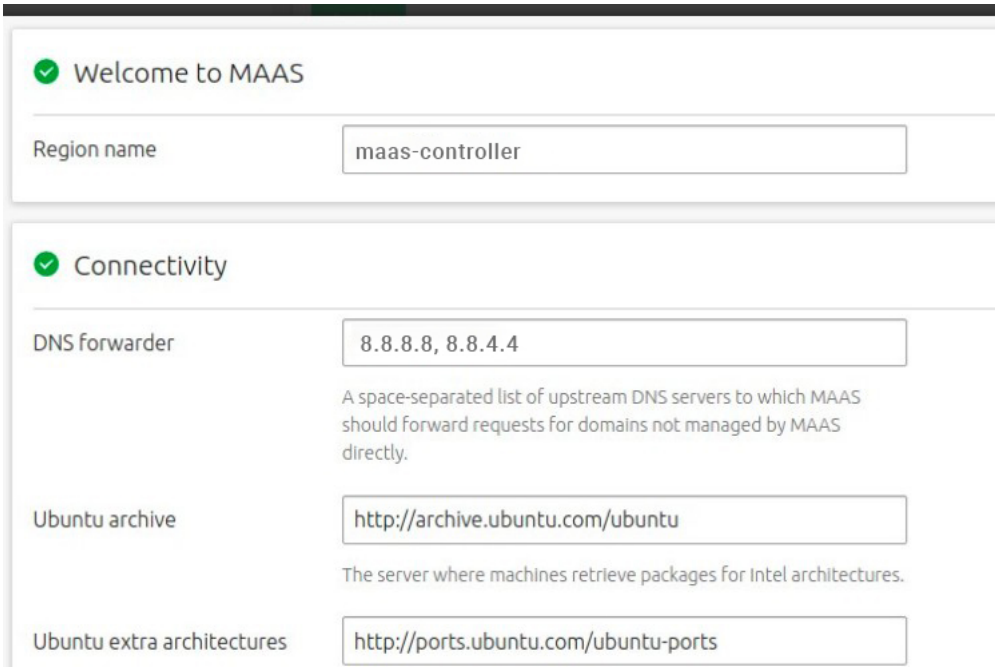


Figure 3.4: MAAS web UI first setup

3.2.2.1 MAAS Networking

As stated in Sub-subsection 3.2.1.1, the VM that runs the *maas-controller* is connected to the *provider* and *maas-net* networks. MAAS is aware of the networks to which its controller is connected and lists them in the *"Subnets"* tab on the web UI as shown in Figure 3.5.

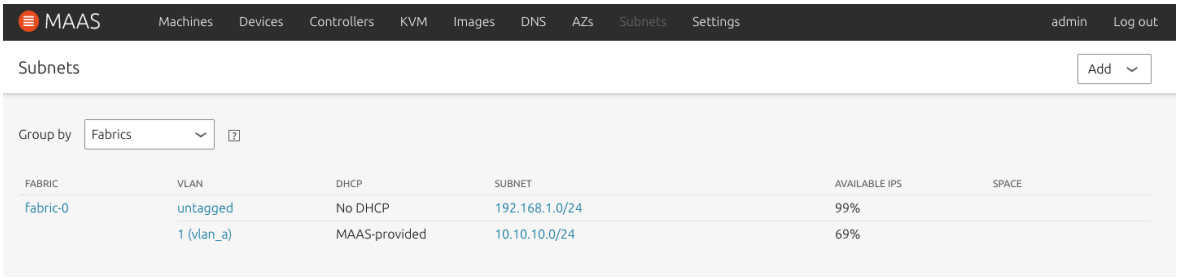


Figure 3.5: MAAS subnets

Then, the DHCP was enabled in the private subnet 10.10.10.0/24. The proposed architecture is not very complex when it comes to networks so, only the default fabric (*fabric-0*) was used.

The last step was the Wireless Nodes enlistment process. All nodes were connected directly to the MAAS controller as explained previously in Sub-subsection 3.2.1.1.

After the enlistment, the power management set up followed. MAAS provides an extensive list of power management options (e.g., IPMI, OpenBMC, etc.). However, in this case, this option was set up as Manual due to the lack of power management capabilities of the Wireless Nodes along with no external devices to facilitate it.

At last, all Wireless Nodes undergo the commissioning process. This process allows MAAS to collect relevant information about the Wireless Nodes such as the RAM, number of CPU cores, ethernet interfaces, among other data. The commissioning process is detailed in Figure 2.7. From this point, it was possible to command MAAS to acquire, test, deploy, and release the Wireless Nodes. Figure 3.6 shows the 10 Wireless Nodes and one virtual OpenStack VM who also went through the process described above.

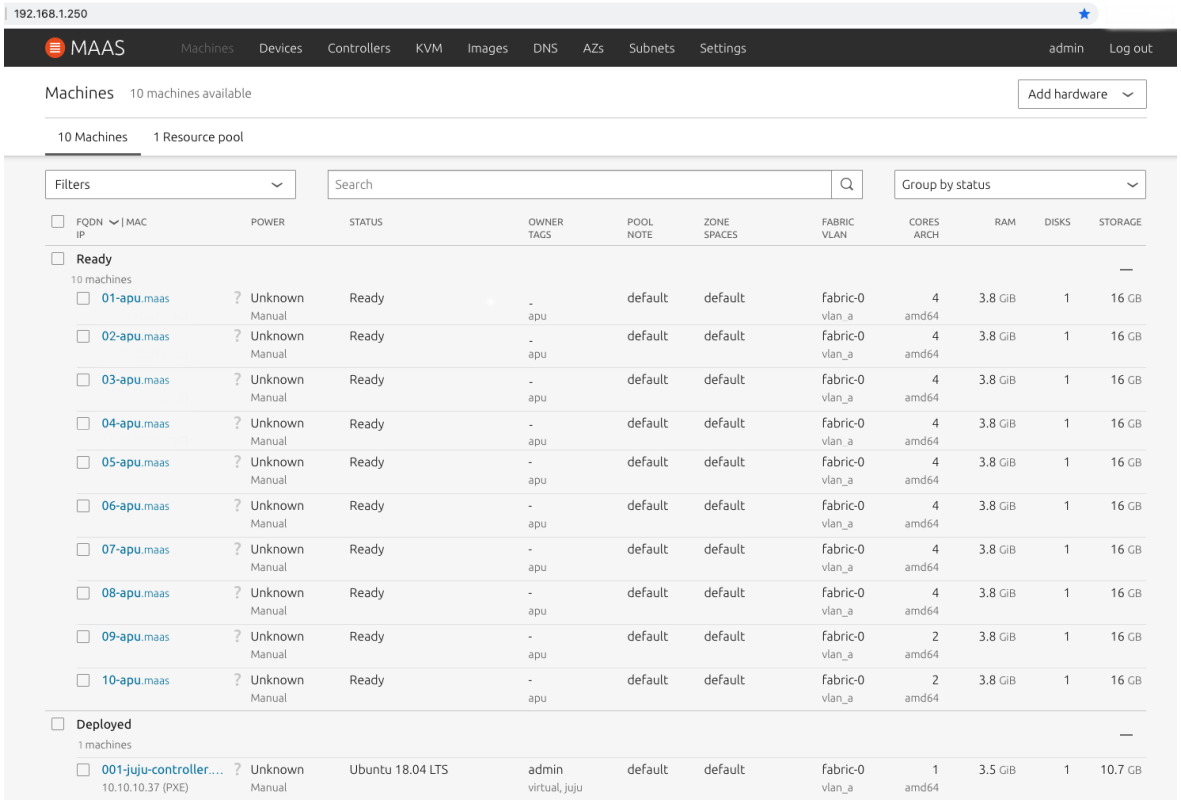


Figure 3.6: MAAS Wireless Nodes Ready

3.2.3 High-level message sequence

Once a wireless node has been enlisted and commissioned by MAAS, the following logical steps are required to deploy it. Depending on the end-use of the machines, the agent that triggers the deployment may vary. It can be directly triggered by MAAS, if the need is to install a base operating system and work with the machines manually. Alternatively, it can be triggered by an external agent such as Juju, if the job requires running complex, inter-related services, like a cloud. Either way, in both situations there is a "Client" that requests to the "MAAS Server" the instantiation of a "Wireless Node". Figure 3.7 depicts such procedure, which is described as follows.

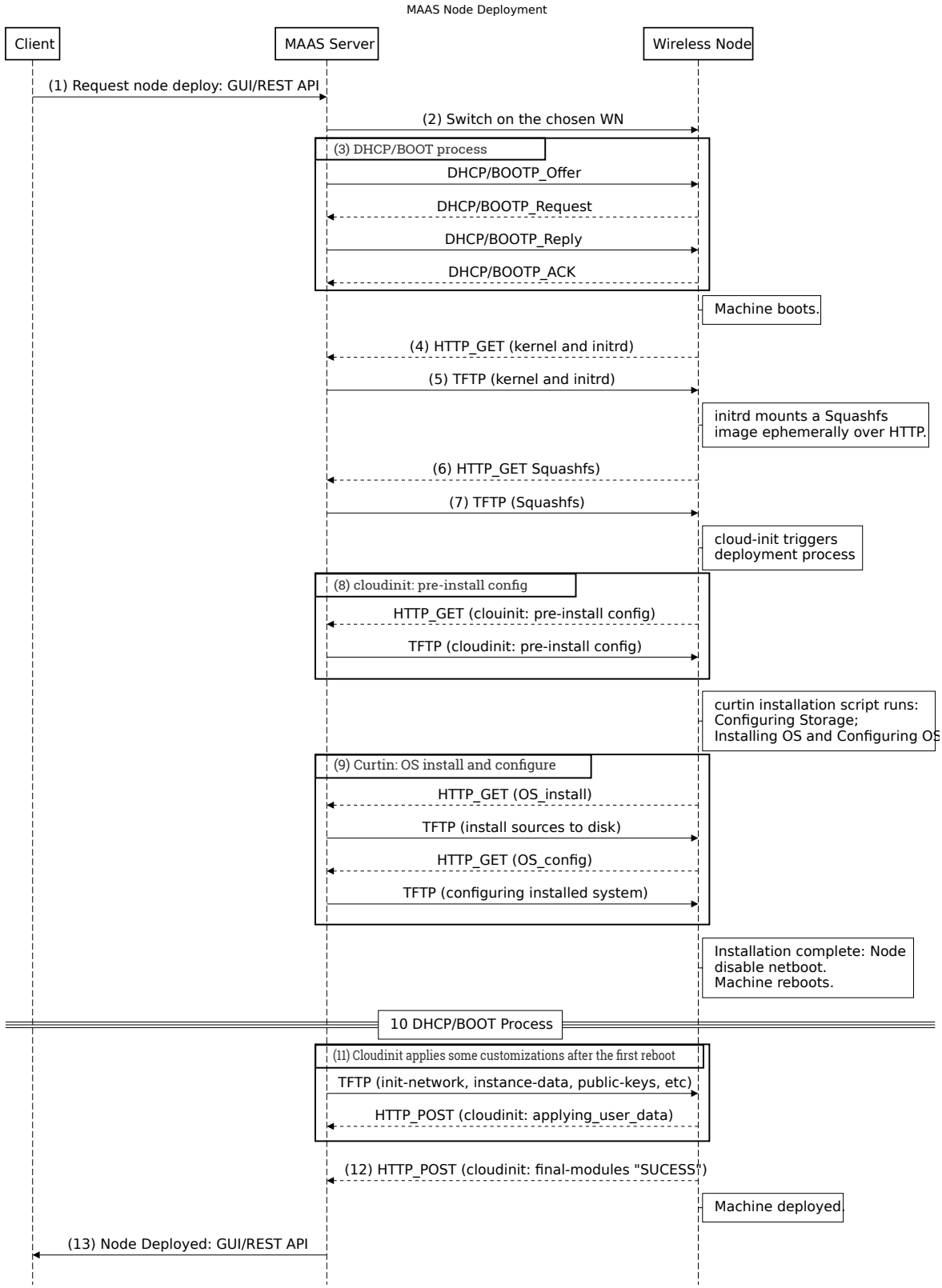


Figure 3.7: High-level message sequence for a MAAS node deployment.

The procedure starts with a request from the "Client" to the "MAAS Server" using a REST API (Figure 3.7, message 1), which will power up a Wireless Node with the specified

constraints provided by the Client (Figure 3.7, message 2). The DHCP server (that can be managed by MAAS or external) is contacted, and the Bootstrap Protocol (BOOTP) procedure occurs (Figure 3.7, message 3). Then, a kernel and *initrd* are received by the Wireless Node over TFTP (Figure 3.7, message 4 and 5), and it boots up. Initrd mounts a SquashFS image ephemerally over HTTP (Figure 3.7, message 6 and 7). The *cloud-init* configuration file is loaded to the wireless node (Figure 3.7, message 8), which triggers the OS installation process. During the OS image installation, the *curtin* installation script starts and applies customizations (Figure 3.7, message 9) at the image level, such as adding and updating package repositories, configure network interfaces, re-order booting options which result in the system will boot from the same device that it booted to run curtin, (for MAAS this will be a network device). Curtin allows for administrators to customize their deployments to have identical setups all the time. When the installation and configuration of the OS are complete, the wireless node reboots, triggering the DHCP/BOOTP again, but this time the machine will boot from its own hard disk (Figure 3.7, message 10). At this stage, the *cloud-init* script starts running which is used by users to customize their deployment immediately after instantiation (Figure 3.7, message 11). Finally, the Wireless Node acknowledges the MAAS server that the *cloud-init* script ran successfully (Figure 3.7, message 12) and MAAS notifies the Client that the wireless node is deployed and ready to be used (Figure 3.7, message 13).

3.2.4 Juju-Controller

It's necessary to review two aspects in order to have a broader understanding of the given explanation. The OpenStack VM named *01-juju-controller* has a diskless image and has been commissioned in MAAS. Therefore, it is going to be used during the Juju installation to bootstrap a Juju controller in the MAAS environment.

First, the Juju software was installed in the OpenStack VM named maas-controller using snap [1]. With the software installed, it was necessary to configure a few prerequisites first, such as Clouds, Credentials, and Controllers.

As stated before in Sub-section 2.8, Juju works on top of many different types of clouds and has built-in support for MAAS. Thus, a new MAAS cloud named *maas-01* was added to the Juju environment as depicted in Figure 3.8. Along with the type and name of the cloud, the MAAS API endpoint URL was also provided.

[1] https://snapcraft.io/

```
ubuntu@maas-controller:~$ juju add-cloud

Cloud Types
  lxd
  maas
  manual
  openstack
  vsphere

Select cloud type: maas

Enter a name for your maas cloud: maas-01

Enter the API endpoint url: http://192.168.1.250:5240/MAAS/

Cloud "maas-01" successfully added

You will need to add credentials for this cloud (`juju add-credential maas-01`)
before creating a controller (`juju bootstrap maas-01`).
```

Figure 3.8: Adding clouds in Juju

As the output in Figure 3.8 refers, it is necessary to add the MAAS server credentials before Juju can interact with the MAAS API. These credentials can be obtained using the MAAS CLI or the MAAS web UI. Figure 3.9 shows the window prompted when a new credential is being added. In this case, a new credential named *maas-01-creds* is being added to *maas-01* cloud. By default the auth-type is "oauth1" but the user can choose other auth-type, such as "access-key", "userpass", "jsonfile", and others. The maas-auth field refers to the MAAS API key that is not displayed on the screen. To validate/list the added credentials the second command was executed.

```
ubuntu@maas-controller:~$ juju add-credential maas-01
Enter credential name: maas-01_creds

Using auth-type "oauth1".

Enter maas-oauth:

Credential "maas-01_creds" added locally for cloud "maas-01".

ubuntu@maas-controller:~$ juju credentials
Cloud    Credentials
maas-01  maas-01_creds
```

Figure 3.9: Adding credentials in Juju

After adding a cloud and its associated credentials, the following step was bootstrapping the Juju controller. To ensure that Juju deploys the controller on the *01-juju-controller* VM

that was added to MAAS, a *"juju" Tag* was attached to the machine that Juju can use as a deployment constraint as depicted in the green rectangle in Figure 3.10.

Moreover, Figure 3.10 shows the *01-juju-controller* machine details. The default tab, "Machine summary", presents a series of blocks. The first block gives an overview of the machine's current status: *Deployed* and running 18.04 LTS Bionic Beaver. The following block, details the *CPU* (1 core, amd64/generic), *Memory* (3.5GiB), *Storage* (10.7GB), and *Tag* characteristics (virtual, juju) of the machine. Since the *01-juju-controller* is an OpenStack VM, the *System* block describes the *Vendor* as OpenStack Foundation, *Product* as OpenStack Nova and give details on the *Version* and *Serial*. The *Mainboard* details the mainboard information (Unknown since it's a VM).

The *1 NUMA Node* block, gives information on the NUMA node of the machine (see Subsection 2.5.5). The *Network* block details the interfaces of the machine and how they are connected.

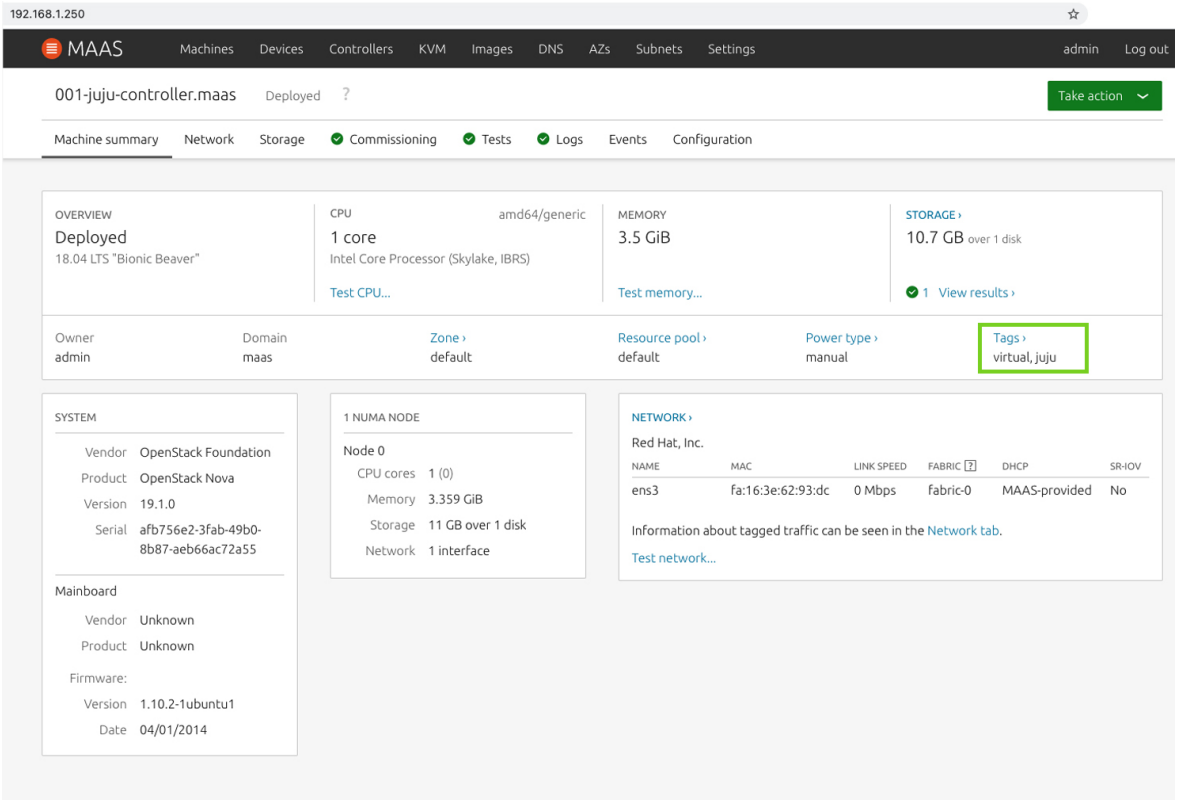


Figure 3.10: 01-juju-controller Tagging in MAAS

To bootstrap the controller on MAAS the following command depicted in Figure 3.11 was executed. This will bootstrap a controller named *juju-01* to a machine with the tag *juju* on the *maas-01* cloud.

```
ubuntu@maas-controller:~$ juju bootstrap --constraints tags=juju maas-01 juju-01
Creating Juju controller "juju-01" on maas-01
Looking for packaged Juju agent version 2.6.8 for amd64
Launching controller instance(s) on maas-01...
 - eppk7s (arch=amd64 mem=4G cores=1)
Installing Juju agent on bootstrap instance
Fetching Juju GUI 2.15.0
```

Figure 3.11: Juju controller bootstrap on MAAS

A few minutes later the terminal will return the deployment status as shown in Figure 3.12. When a controller is bootstrapped two models are automatically added, *controller* and *default*. The controller model is used for internal Juju management and is not intended for general workloads. The default model can be used to deploy any supported software but is typically used for experimentation purposes.

```
 Waiting for address
 Attempting to connect to 10.10.10.37:22
 Connected to 10.10.10.37
 Running machine configuration script...
 Bootstrap agent now started
 Contacting Juju controller at 10.10.10.37 to verify accessibility...

 Bootstrap complete, controller "juju-01" now is available
 Controller machines are in the "controller" model
 Initial model "default" added

ubuntu@maas-controller:~$ juju controllers

 Use --refresh option with this command to see the latest information.

 Controller  Model    User   Access     Cloud/Region  Models  Nodes    HA  Version
 juju-01*    default  admin  superuser  maas-01            2      1  none  2.6.8
```

Figure 3.12: Juju controller bootstrap on MAAS (2)

Besides the API, the Juju controller has a web interface that facilitates experiments with Juju's modeling and automation capabilities. Once the controller bootstrapped, it enabled access to the web UI as shown in Figure 3.13.

```
ubuntu@maas-controller:~$ juju gui

 GUI 2.15.0 for model "admin/default" is enabled at:
   https://10.10.10.37:17070/gui/u/admin/default
 Your login credential is:
   username: admin
   password: 85faa8d324ce26300d8aa6b43b8fd794
```

Figure 3.13: Enable Juju GUI

3.2.5 Wireless Node

In this proposal, a Wireless Node (WN) is a bare-metal programmable device that allows the deployment of either VNFs or PNFs. These devices play an instrumental role in the system architecture due to their wireless capabilities, which allows to expand the cloud-based environment NFs (e.g., firewalls) to PNFs located at the edge of the network (e.g., wireless access points), with the added benefit of being remotely controlled and instantiated on-demand. In this line, the wireless nodes are connected to control and data networks (in Figure 3.1, green and blue lines) for providing management actions and for handling the data experimentation, respectively.

The wireless node grid is composed by a dedicated subset of the same nodes existing in AMazING, namely ten PC Engines APU devices (two APU1C4 and eight APU2C4). The APU1C4 (Figure 3.14a) are equipped with a 2 cores AMD G-T40E @ 1GHz CPU, 4GB RAM, 16GB storage, three Realtek RTL8111E Gigabit Ethernet Controller, two Compex WLE200NX wireless cards, and two Antsmadb antennas. Similarly, the APU2c4 (Figure 3.14b) are equipped with a 4 cores AMD GX-412TC CPU @ 1.2GHz CPU, 4GB RAM, 16GB storage, three Intel(R) Ethernet Controller I210-AT, two Compex WLE600VX wireless cards, and two Antsmadb antennas.

(a) APU1C2 **(b)** APU2C4

Figure 3.14: Wireless Nodes used.

A firmware update was made since the one installed on the APUs was outdated. After that,

the APUs were configured for booting from PXE in order to allow MAAS to proceed with the enlistment. The information on the manufacturing site was not sufficient to understand what changed between versions. However, the update had a significant impact on results, as described in Chapter 4.

3.3 Summary

This chapter addressed the designed framework, which can be attached to the existing OpenStack cloud running in ITAV and act as a testbed. The proposed architecture was tested in Chapter 4 to understand the feasibility to deploy PNFs in physical nodes (i.e., APUs) as well as VNFs in OpenStack VMs on-demand. These tests were performed with requests directly by the user on MAAS and/or by using the Juju framework.

CHAPTER 4

Evaluation

In order to assess the feasibility of the implementation presented in the previous section, as well as its capability to be used as a wireless testbed for performing wireless experiments in academic environments, several tests were performed. In this line, the implemented system was evaluated in terms of instantiation, booting and deployment delays. Also, as presented above, the proposed system envisions the integration of physical wireless nodes in cloud-based environments, with the system enabling such nodes to be seen as PNFs which are flexible enough to accommodate different network functions through the on-demand deployment of VNFs.

The values were measured through the MAAS which tracks necessary events of each machine, allowing to collect the times of the deployment phase of the APUs (i.e., wireless nodes), notably from its starting, the APU "Performing PXE boot", until the APU was "Deployed". The experiments were repeated 10 times, with this section presenting the measurements average and a confidence interval of 95%.

4.1 System evaluation and results

4.1.1 Deployment of Operating System

The OS deployment measures the time since a user requested an APU deployment from the MAAS server to deploy an OS, until the moment that this deployment is finished and the APU is fully accessible. Four Linux distributions (all server editions) were deployed and evaluated for comparison, namely: Ubuntu 18.04, Ubuntu 16.04, CentOS 7, and Ubuntu Core.

The following tables compare the results of the instantiation/deployment delay for the evaluated OSs considering the different number of wireless nodes (i.e., APUs). In the first test, no configuration or update was made to the APUs. At the time, the firmware was v4.10.0.1 (release date 10/09/2019). Table 4.1 summarizes the data obtained on this test.

In an attempt to improve the obtained results, a firmware update to v4.12.0.2 (release date 28/06/2020) was made as mentioned in Sub-section 3.2.5. The improvement, when compared

Table 4.1: Average instantiation time by OS with APU firmware v4.10.0.1 (in minutes).

	Operating System		
No. of Wireless Nodes Deployed	Ubuntu 18.04	Ubuntu 16.04	CentOS 7
1	10.21 ± 0.03	9.58 ± 0.04	11.25 ± 0.04
5	10.18 ± 0.03	9.68 ± 0.04	11.33 ± 0.04
10	10.20 ± 0.04	9.68 ± 0.05	11.30 ± 0.05

to the results shown in Table 4.2, was clear, with an average decrease of approximately 19%, 11%, and 22% on the deployment time of Ubuntu 18.04, Ubuntu 16.04, and CentOS7, respectively. Thus, the remaining tests were carried out with the new firmware version.

Table 4.2 details the average instantiation time by OS with the new firmware update. The first point that stands up in the obtained results is that, as far as can be ascertained, with the number of available nodes to simultaneously instantiate and the characteristics of the infrastructure, the increase of the number of nodes had no significant impact on the overall instantiation time. Regarding the time difference between operating systems, the Ubuntu Core [1] was the fastest to instantiate, which was expected since it is defined as a very lightweight OS target especially for IoT deployments.

Table 4.2: Average instantiation time by OS (in minutes).

	Operating System			
No. of Wireless Nodes Deployed	Ubuntu 18.04	Ubuntu 16.04	CentOS 7	Ubuntu Core
1	8.34 ± 0.03	7.65 ± 0.04	8.78 ± 0.05	5.38 ± 0.04
5	8.36 ± 0.03	7.68 ± 0.02	8.81 ± 0.05	5.41 ± 0.04
10	8.37 ± 0.03	7.69 ± 0.02	8.82 ± 0.03	5.42 ± 0.03

Figure 4.1 displays in more detail the average time per deployment stage of a single Wireless Node. This process was detailed in Sub-section 3.2.3.

It can be seen that the average time spent in the first two stages (i.e., "Performing PXE Boot" and "Loading ephemeral") by all of the OSs was similar. These results were expected since these tasks are not related to a specific OS, and the hardware used has the same characteristics. On the "Configuring storage" stage, the time for Ubuntu Core stands out from remaining OSs. MAAS has a *pressed* configuration file for each OS, that *curtin* uses to perform configurations. As the Ubuntu 18.04, Ubuntu 16.04, and CentOS 7 are storaged natively in the MAAS server, their *pressed* files already existed. But Ubuntu Core was loaded to the MAAS server as a custom image, and no *pressed* file was created to it. These led to a higher storage configuration time because MAAS used the default *pressed* file for OS that it is not aware of. On the "Installing OS" step, it can be seen that Ubuntu 18.04 is the OS that took the longest time to install its OS on the node, and as expected Ubuntu Core was the fastest due to be incredibly light. Given its characteristics, Ubuntu Core was,

[1] https://ubuntu.com/core

again, the fastest on the "Configuring OS" stage. Ubuntu 18.04 spent 11.46% and 22.85% when compared to Ubuntu 16.04 and CentOS 7, to configure the OS. Finally, on the "Reboot" stage, the CentOS 7 took the longest time to perform the first reboot, due to *cloud-init* post-installation configuration that took longer than the other OS. Observing Figure 4.1 it is clear that Ubuntu Core had the second slowest result, which happened because MAAS uses *cloud-init* to customize node deployments after the first reboot (as stated in subsection 3.2.3). *Cloud-init* uses *apt* or *yum* commands depending if it is Ubuntu 18.04/16.04 or CentOS 7. But Ubuntu Core uses *snap*[2] to install apps. So, as in the previous "Configure storage" stage, MAAS uses a default *cloud-init* configuration file that is *apt* based, which introduces the delay on the reboot.

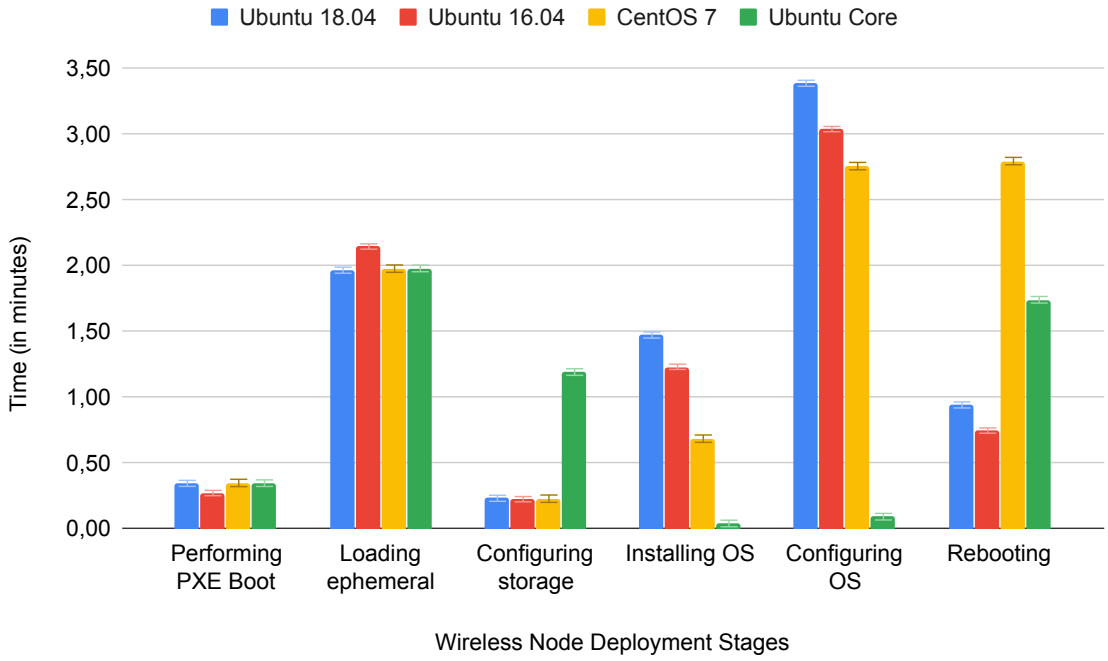


Figure 4.1: Average time by deployment stage.

4.1.2 Rebooting from Operating System

The time that a single machine took to reboot was measured for the different OS, as shown in Table 4.3. After the first reboot during the deployment stage, it was expected that the reboot time of the machines should decrease because of all the main configurations were made. *Cloud-init* will continue to perform some minor reboot customizations such as "reading and applying user-data", "configuring ssh-import", among others, maintaining both MAAS server and the Wireless Node aware of each other. As explained in subsection 4.1.1, Ubuntu Core doesn't work with this configuration method resulting in a higher reboot delay.

[2]https://snapcraft.io/

Table 4.3: Average reboot time by OS. (in seconds)

	Operating System			
No. of Wireless Nodes Deployed	Ubuntu 18.04	Ubuntu 16.04	CentOS 7	Ubuntu Core
1	38.37 ± 0.01	32.37 ± 0.01	32.55 ± 0.02	68.26 ± 0.02

4.1.3 Juju charm deployment

As presented in sub-section 3.2.3, the instantiation request can be made by different clients, notably the MAAS or Juju. In this context, the time since Juju requests an APU until it's fully deployed was measured. The charm used was magpie[3] which verifies the ICMP, DNS, MTU, and RX/TX speed between itself and any peer units deployed.

Three stages are shown in Table 4.4:

1. "OS Install", which refers to the average time of an OS deployment;
2. "Juju Charm install", which is defined as the average time to install the magpie charm on an APU;
3. "Charm Fully Deploy", which is the time until the charm return the information.

Results show that deploying a charm in a Wireless Node using Juju had significantly increased the total instantiation time.

Drawing a parallel between Figure 4.1, and this scenario, Juju acts as the Client, which will request a machine from MAAS. Here, the charm took approximately 6 minutes for an instantiation request of 5 wireless nodes, which was the maximum amount that we were able to simultaneously instantiate since the amount of RAM demanded by Juju-controller (Figure 3.1) increases as the number of wireless nodes increases. For example, while the Juju-controller consumed 80% of RAM for completing the request of 5 wireless nodes, for the request of 10 wireless nodes the Juju-controller was not able to complete the task due to the lack of available RAM. Notwithstanding, the MAAS server behaved as expected and the average results presented itself within the same range of the previous test.

Table 4.4: Average time to deploy a Juju Charm. (in minutes)

	Deployment Phases			
No. of Wireless Nodes Deployed	OS Install (Ubuntu 18.04)	Juju Charm Install (magpie)	Charm Fully Deployed	Total Time
5	8.31 ± 0.03	5.78 ± 0.05	13.83 ± 0.08	27.91 ± 0.12

4.1.4 Deployment of OpenStack VM

This test verifies the MAAS capability to handle OpenStack VMs as nodes and assess its the deployment times. In this context, an OpenStack VM with 4 vCPU, 4GB RAM, and 16GB storage (similar characteristics as the bare-metal APU) was used to deploy all the OS stated before. Results are presented in Table 4.5 and show that VMs have on average

[3]https://jaas.ai/u/admcleod/magpie/45

about 70% lower deployment times than the Wireless Nodes, which was expected due to its architecture [30]. Also, we were not able to deploy the Ubuntu Core in the OpenStack VM due to an endless freeze upon its reboot. Nevertheless, this experiment demonstrates the capability of the implemented system to instantiate both physical and virtual network functions (i.e., PNFs and VNFs) in the wireless nodes (i.e., APUs) and as OpenStack VMs, respectively.

Table 4.5: Average time to deploy a OpenStack VM. (in minutes)

	Operating System			
No. of OpenStack VM Deployed	Ubuntu 18.04	Ubuntu 16.04	CentOS 7	Ubuntu Core
1	2.90 ± 0.01	2.40 ± 0.01	2.50 ± 0.01	-

It is necessary to note that there is a caveat to using the PXE-boot image [29] mentioned in Subsection 3.2.1.1. This approach does not result in the mounting of the PXE-boot image as an additional booting device. Instead, OpenStack writes the image contents directly to the VM disk and then boots it. This behavior leads to a challenge when the MAAS user releases a previously used VM. Because the first VM installation will overwrite the disk, there is no longer a PXE-boot image available to reprovision that machine in MAAS, thus failing to deploy the OS unless further action is taken upon releasing the VM. Unlike the bare-metal hardware, where it is mandatory to have PXE boot capabilities to repurpose those resources, the cloud paradigm usually solves this problem by terminating the VM and launching a new one. However, that cloud approach is not ideal for this dissertation's purposes. Adding a newly launched VM to MAAS would require undergoing the enlistment and commissioning process, which is time-consuming. The devised solution is to go to the OpenStack environment after releasing the VM and re-building the disk with the PXE-boot image. This way, the VM will now boot once again in PXE mode, and MAAS can then use it for the on-demand deployment. However, it is important to clarify that it's not mandatory to previously commission the VMs in the MAAS server. They can be commissioned on-demand without the need to hold resources with idle VMs.

4.1.5 Firewall deployment

This test aimed to simulate a simple firewall instantiation (i.e., PNF) on an APU and understand the reaction time since a user starts the deployment until the firewall was up. Three bash scripts were written for this test:

- The *start-machines* is a script (see Appendix A.1) to help the user deploy MAAS machines with custom scripts.
- The *firewall-pnf* script (see Appendix A.2) uses *iptables* to drop packets coming from APU-1 to the machine deployed.
- The *pinger* script (see Appendix A.3) receives the machine ID from the user, gets the IP from that machine and waits until the APU-2 is deployed to start pinging the IP until the firewall is up.

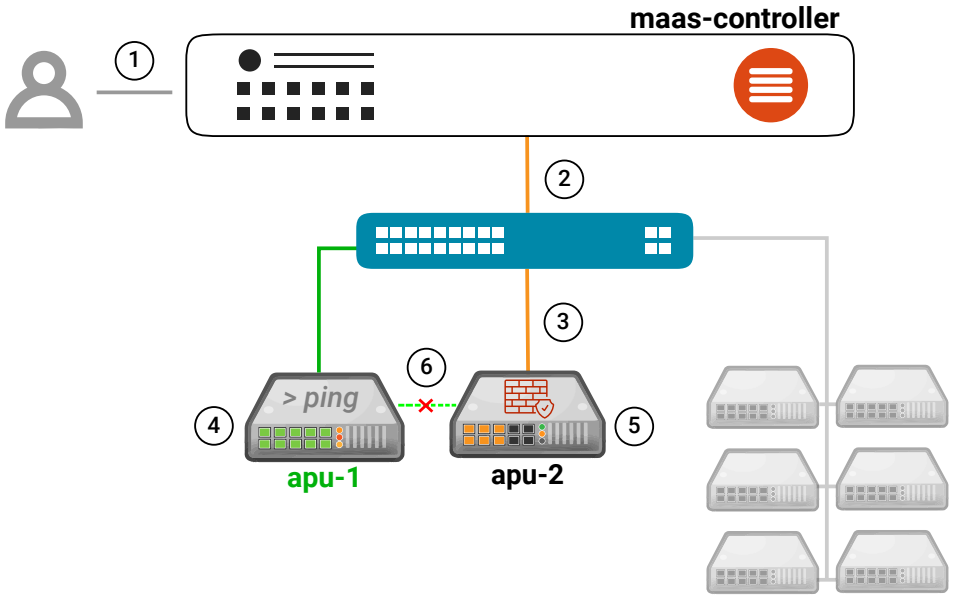


Figure 4.2: Firewall (PNF) implementation overview.

The test is illustrated in Figure 4.2:

1. Using the command line, the user runs the *start-machine* script to deploy a new machine with the *firewall-pnf* script. User also starts the *pinger* script on APU-1 (already deployed and running Ubuntu 18.04).
2. MAAS starts the deployment.
3. Machine boots up and start installing OS.
4. The *pinger* script starts.
5. With the OS installation concluded, the *firewall-pnf* script runs immediately.
6. The connection between APU-1 and APU-2 is lost. The ping stops. The firewall is up.

As shown in Table 4.6, the time since the user queries MAAS to deploy the machine and the packets loss from the APU-1 was, on average, similar to that obtained in Table 4.2 on the average deployment time for one node. In fact, the script runned so quickly making the firewall up that the APU-1 didn't show any successful ping.

Table 4.6: Average time to deploy the firewall PNF (in minutes)

No. of Wireless Nodes Deployed	Ubuntu 18.04 + firewall script	Ubuntu 18.04 + firewall script + 60sec delay
1	8.35 ± 0.03	9.36 ± 0.03

Therefore, a second test was performed, this time introducing a 60-second delay in triggering the firewall script. This delay was introduced to:

1) understand if there was a viable connection between the two machines;

2) verify the success of the firewall blocking the packets.

The results on Table 4.6 shows an increase of 60-seconds, in average, of the time the ping remained active. Therefore, it was proved that: 1) there was a connection between the two machines, and 2) the firewall was able to block the packets coming from the APU-1.

4.1.6 Wireless Access Point deployment

The scope of this test was to test the wireless capabilities of the wireless nodes. The test was divided into two parts. First, MAAS was used to instantiate an APU and deploy the "access-point" script measuring the time it takes for an external device (i.e., laptop) to connect to the network, and using *iperf* [4] to measure the bandwidth between the devices. Then, the test was repeated with a manual install of the OS in the APU to assess if there's any difference in the bandwidth.

Three bash scripts were written for this test:

- The *access-point* script (see Appendix A.4) is used to configure the wireless access point. It installs and configures a daemon software that enables a network interface card to act as an access point (i.e., hostapd [5]), and a DHCP server (i.e., isc-dhcp-server [6]) services, configures the wireless subnet and the wireless interface from the APU.
- The *access-device* script (see Appendix A.5) repeatedly tries to connect the laptop to the SSID of the network that the wireless node will create. When it is successful, it returns the time it took to do it.
- The *iperf-test* script (see Appendix A.6) runs an *iperf* command ten times and save the result into a text document.

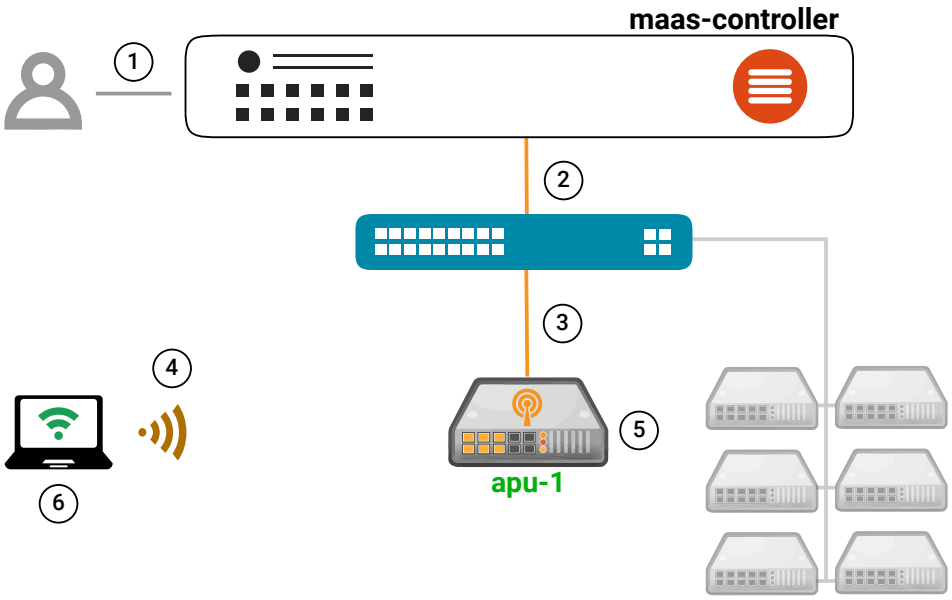


Figure 4.3: Wireless Access Point (PNF) implementation overview.

Figure 4.3 illustrates the test:

[4] https://iperf.fr/
[5] http://w1.fi/hostapd/
[6] https://www.isc.org/dhcp/

1. Using the command line, the user runs the *start-machine* script to deploy a new machine with the *access-point* script.
2. MAAS starts the deployment.
3. Machine boots up and start installing OS.
4. User starts *access-device* script on the external device.
5. With the OS installation concluded, the *access-point* script runs and the machine becomes an wireless access point.
6. The external device connects to wireless access point. The user starts the *iperf-test* script.

Table 4.7 shows the average time since the APU starts deploying until the external device successfully connects to the wireless access point.

Table 4.7: Average time to deploy the Wireless Access Point. (in minutes)

	Operating System
No. of Wireless Nodes Deployed	Ubuntu 18.04 + *access-point* script
1	9.48 ± 0.04

Thus, the average deployment was 14% higher compared to the obtained in Table 4.2 for the deployment of one node. These results were expected because the script needs to install two packages (hostapd and isc-dhcp-server), configure them, and start up the access point. On top of that, the external device needs to discover and connect to the wireless network which depending on the device it can vary a little.

Regarding the *iperf* test, Table 4.8 shows the results of the average throughput in Mbit/sec on both MAAS deployment and manual deployment. Therefore, the results demonstrate that the use of MAAS to deploy the APUs does not cause loss of network performance.

Table 4.8: Average throughput in iperf tests. (in Mbits/sec)

	APU deployed by MAAS	APU deployed manually
Iperf Between	Ubuntu 18.04 + *access-point* script	Ubuntu 18.04 + *access-point* script
External device (Laptop)	19.53 ± 0.12	19.52 ± 0.12

4.2 Summary

This chapter presents the results of the tests and results of the proposed testbed for metrics such as instantiation times, network function suitability, and physical and virtual nodes management. With these experiments, it was possible to understand if the presented solution meets the objectives initially set, which will be discussed in the following chapter.

CHAPTER 5

Conclusions

5.1 Conclusions

This dissertation proposed a system architecture that aims the integration of an existing physical wireless testbed (i.e., the AMazING testbed) with a cloud-based environment at the Instituto de Telecomunicações at the University of Aveiro.

Achieving the primary outcomes of this dissertation did not come without some setbacks during its execution. Initially, we used the existing OpenStack cloud to run the MAAS controller VM. This choice proved to be a challenge due to the intricate security layers of the production OpenStack, which did not allow the MAAS controller VM to reply to the APU PXE requests. Deploying an OpenStack environment in the laptop proved to be a great challenge on its own. Installing all the services and making them work correctly was a time-consuming, challenging, yet very educational process. Nonetheless, a functional OpenStack framework was successfully deployed, thus achieving the first proposed objective.

In this context, the proposed system architecture described in Chapter 3 proved capable of connecting the wireless node grid (AMazING testbed) with the OpenStack framework and managing the on-demand instantiation of both physical and virtual NFs in physical wireless nodes and/or in OpenStack VMs, respectively. These functionalities fulfill the requirements initially set as the main objectives.

The proposed system architecture was experimentally evaluated in Chapter 4, with results demonstrating the feasibility of the proposal while avoiding an evasive approach to our in-house cloud during the integration of the physical wireless nodes. Also, results showed the system was able to instantiate on-demand PNFs and VNFs in the wireless nodes and in the VMs, with the measurements evidencing a high instantiation delay on the wireless nodes compared to the VMs. On the other hand, the results showed that the number of nodes being deployed at the same time does not have a negative impact on the overall instantiation time on every test.

When using Juju to deploy charms a significant and unworkable increase in the instantiation

time becomes evident. In this case, the increase in the number of wireless nodes being deployed had a direct impact on the performance of the system.

The firewall and access point tests allowed to assess the feasibility of the proposal when a more "real" use case was presented to it. It also proves that the use of the MAAS to instantiate services on the wireless nodes does not decrease the processing and network performance.

5.2 Future Work

As future work, some improvements must be made in order to make the testbed more usable in a real environment, such as:

- Add an external power management system to be able to power up the APU automatically.
- Integration with an authentication service to add security.
- Development of a user interface to make the use of the testbed more user-friendly.
- Currently, a restructure of architecture is taking place in IT and the use of the testbed proposal will be added at a later stage.

References

[1] Cisco, "Cisco Annual Internet Report (2018–2023)", *Cisco*, pp. 1–41, 2020. [Online]. Available: http://grs.cisco.com/grsx/cust/grsCustomerSurvey.html?SurveyCode=4153%7B%5C&%7Dad%7B%5C_%7Did=US-BN-SEC-M-CISCOASECURITYRPT-ENT%7B%5C&%7DKeyCode=000112137.

[2] N. Mckeown, *How sdn will shape networking*, [YouTube video], Accessed Jul. 15, 2020, Oct. 2011. [Online]. Available: https://www.youtube.com/watch?v=c9-K5O_qYgA.

[3] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015, ISSN: 15582256. DOI: 10.1109/JPROC.2014.2371999. arXiv: 1406.0440.

[4] M. B. Chiosi, D. Clarke, P. Willis, A. Reid CenturyLink, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, J. Benitez, U. Michel, H. Damker KDDI, K. Ogaki, T. Matsuzaki NTT, M. Fukui, K. Shimano, D. Delisle, Q. Loudier, C. Kolias, I. Guardini, E. Demaria, R. Minerva, A. Manzalini, D. López, F. Javier Ramón Salguero, F. Ruhl, and P. Sen, "Network Functions Virtualisation", Tech. Rep. [Online]. Available: http://portal.etsi.org/NFV/NFV%7B%5C_%7DWhite%7B%5C_%7DPaper.pdf.

[5] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology", Tech. Rep. DOI: 10.6028/NIST.SP.800-145.

[6] J. P. Barraca, D. Gomes, and R. L. Aguiar, "AMazING-Advance Mobile wireless playGrouNd", Tech. Rep.

[7] P. Patel, A. H. Ranabahu, A. P. Sheth, P. Patel, A. H. Ranabahu, A. P. Sheth, A. Ranabahu, and A. Sheth, "Service Level Agreement in Cloud Computing", Tech. Rep., 2009.

[8] B. A. A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks", *IEEE Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[9] S. Denazis, J. Hadi, S. Mojatatu, N. D. Meyer, and B. O. Koufopavlou, "RFC 7426 - Software-Defined Networking ... SDN—: Layers and Architecture Terminology", 2015, ISSN: 2070-1721. [Online]. Available: http://www.rfc-editor.org/info/rfc7426..

[10] O. Salman, I. H. Elhajj, A. Kayssi, and A. Chehab, "Sdn controllers: A comparative study", in *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, 2016, pp. 1–6. DOI: 10.1109/MELCON.2016.7495430.

[11] M. Richart, J. Baliosian, J. Serrat, and J. Gorricho, "Resource slicing in virtual wireless networks: A survey", *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 462–476, 2016.

[12] "Network Functions Virtualisation (NFV); Architectural Framework Group Specification", Tech. Rep., 2013. [Online]. Available: http://portal.etsi.org/chaircor/ETSI%7B%5C_%7Dsupport.asp.

[13] "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV Group Specification", Tech. Rep., 2013. [Online]. Available: http://portal.etsi.org/chaircor/ETSI%7B%5C_%7Dsupport.asp.

[14] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges", *IEEE Communications Surveys and Tutorials*, vol. 18, no. 1, pp. 236–262, 2016, ISSN: 1553877X. DOI: 10.1109/COMST.2015.2477041. arXiv: 1509.07675.

[15] *MAAS | Metal as a Service*. [Online]. Available: https://maas.io/ (visited on 09/15/2020).

[16] *What is PXE? | PXE Boot Server - ManageEngine OS Deployer*. [Online]. Available: https://www.manageengine.com/products/os-deployer/pxe-preboot-execution-environment.html (visited on 12/26/2020).

[17] *MAAS | MAAS communication (snap/2.9/UI)*. [Online]. Available: https://maas.io/docs/snap/2.9/ui/maas-communication (visited on 12/17/2020).

[18] *Juju - The simplest way to deploy and maintain applications in the cloud*. [Online]. Available: https://juju.is/ (visited on 07/15/2020).

[19] *JUJU | Concepts and terms*. [Online]. Available: https://juju.is/docs/concepts-and-terms (visited on 12/17/2020).

[20] *openstack base | Juju*. [Online]. Available: https://jaas.ai/openstack-base/bundle/70 (visited on 12/17/2020).

[21] *About Fed4Fire+ - FED4FIRE+*. [Online]. Available: https://www.fed4fire.eu/the-project/ (visited on 12/17/2020).

[22] *IRIS - FED4FIRE+*. [Online]. Available: https://www.fed4fire.eu/testbeds/iris/%7B%5C#%7D1574352122286-039e2221-2315 (visited on 12/17/2020).

[23] *Iris Testbed | Home*. [Online]. Available: https://iris-testbed.connectcentre.ie/ (visited on 12/17/2020).

[24] J. Struye, B. Braem, S. Latré, and J. Marquez-Barja, "CityLab: A Flexible Large-scale Multi-technology Wireless Smartcity Testbed", Tech. Rep. [Online]. Available: https://doc.lab.cityofthings.eu.

[25] J. Struye, B. Braem, S. Latré, and J. Marquez-Barja, "The citylab testbed — large-scale multi-technology wireless experimentation in a city environment: Neural network-based interference prediction in a smart city", in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2018, pp. 529–534. DOI: 10.1109/INFCOMW.2018.8407018.

[26] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks", Tech. Rep. [Online]. Available: www.flux.utah.eduwww.netbed,org.

[27] *IRIS - FED4FIRE+*. [Online]. Available: https://www.fed4fire.eu/testbeds/iris/%7B%5C#%7D1574352122286-039e2221-2315 (visited on 12/17/2020).

[28] *NITOS - NITlab - Network Implementation Testbed Laboratory*. [Online]. Available: https://nitlab.inf.uth.gr/NITlab/nitos (visited on 12/20/2020).

[29] *Create PXE-Boot image for Openstack | Kimi Zhang*. [Online]. Available: https://kimizhang.wordpress.com/2013/08/26/create-pxe-boot-image-for-openstack/ (visited on 05/04/2020).

[30] C. G. Kominos, N. Seyvet, and K. Vandikas, "Bare-metal, virtual machines and containers in openstack", in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, 2017, pp. 36–43.

APPENDIX A

Bash Scripts

A.1 start-machines bash script

This script facilitates the deployment of new machines with custom cloud-init user-data scripts through the MAAS CLI. First, it logs into the MAAS region controller using the Profile (i.e., username) and API key. Then, MAAS will display the Hostname, System ID, Status, and Tags for all machines that are "Ready" to be used. The user chooses the machine by writing the System ID and passes the name of the script to be deployed. Finally, MAAS converts the script with a base64 encoded and deploys the machine.

```
#!/bin/bash
echo "Logging into MAAS"
PROFILE=admin
API_KEY_FILE=/home/ubuntu/admin_key
API_SERVER=192.168.1.250:5240
MAAS_URL=http://$API_SERVER/MAAS/api/2.0
maas login $PROFILE $MAAS_URL - < $API_KEY_FILE >/dev/null
echo "Log in success"
echo -en '\n'
echo "Listing Machines"
maas admin machines read | jq -r '(["HOSTNAME","SYSID","STATUS", "TAGS"] | (.,
↪  map(length*"-"))), (.[] | (select(.status_name=="Ready") | [.hostname, .system_id,
↪  .status_name, .tag_names[0]])) | @tsv' | column -t | (sed$
echo -en '\n'
echo "Choose machine ID to deploy"
read machine_id
echo -en '\n'
echo "Choose Script:"
read script
echo -en '\n'
echo "Deploying machine $machine_id with the script $script"
maas admin machine deploy $machine_id user_data=$(base64 -w0 ./$script) >/dev/null
echo "Deployment started"
```

A.2 firewall-pnf bash script

This script blocks incoming ICMP (i.e., ping) packets from a specific IP to mimic a firewall behavior. In this case, the incoming ICMP packets from the device with the IP - 10.10.10.38 (i.e., APU-1) are blocked. The counter added at the beginning delays by 50 seconds the ICMP blocking.

```
#!/bin/bash
#-------------------------------------------------------------#
valid=true                  # This is a forced delay
count=1                     # to validate the existence
while [ $valid ]            # of a connection between
do                          # the machine (receiver) and the
echo -ne "$count "          # machine sending the pings
if [ $count -eq 50 ];
then                        # The 50sec was an arbitrary
break                       # number. It could be any
fi                          # numeber.
sleep 1
((count++))
done
#-------------------------------------------------------------#

# Iptables command to block incoming pings from the APU-1
sudo iptables -A INPUT -p icmp --icmp-type echo-request -s 10.10.10.38 -j REJECT
```

A.3 pinger bash script

This script queries the user on the System ID of the machine being deployed. Then, gets the IP address of that machine and SSH into the APU-1 to start pinging the machine being deployed.

```
#!/bin/bash
#Logging into MAAS
PROFILE=admin
API_KEY_FILE=/home/ubuntu/admin_key
API_SERVER=192.168.1.250:5240
MAAS_URL=http://$API_SERVER/MAAS/api/2.0
maas login $PROFILE $MAAS_URL - < $API_KEY_FILE >/dev/null

# Query user for the MachineID
echo "Starting"
echo "Enter SystemID"
read SYSTEM_ID

# Get the IP from the machine
ping_ip = $(maas $PROFILE interfaces read $SYSTEM_ID | jq -r '.[] | (.links[].ip_address)')
↪  >/dev/null

# SSH to APU-1 and starts sending ping to the IP of the machine
```

```
ssh ubutu@10.10.10.38 PING_IP=$ping_ip 'bash -s' <<'ENDSSH'

count=1
echo "Starting Ping Requests"
start=`date +%s`
while ping -q -c 1 $PING_IP >/dev/null
do
echo -ne "$count  "
((count++))
sleep 1
done
end=`date +%s`
echo "Ping Error"
echo Execution time was `expr $end - $start` seconds.

ENDSSH
```

A.4 access-point bash script

This script installs and configures a daemon software that enables a network interface card to act as an access point. First install hostapd deamon software to enable the wlp1s0 network interface card to act as an access point and authentication server. The flow chart in Figure A.1 summarizes the process:

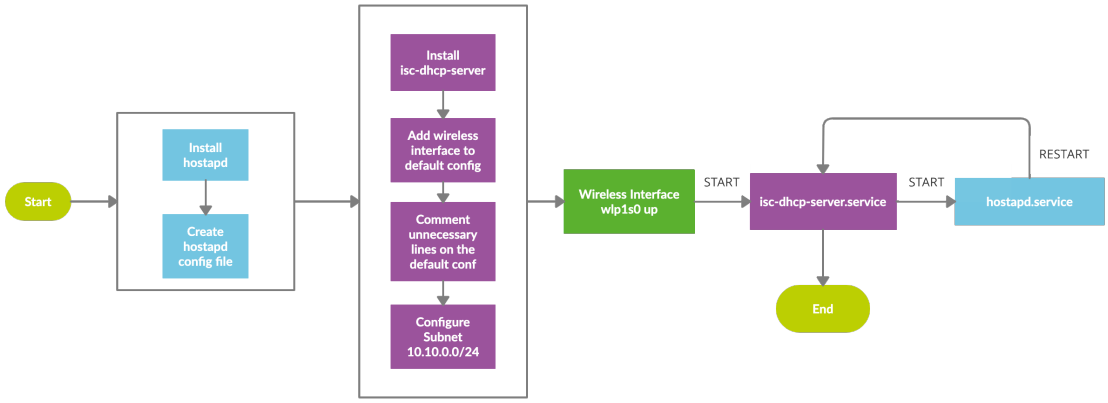


Figure A.1: Access Point Bash Script Flow Chart

First, it installs and creates a configuration file for the hostapd daemon software to enable the wlp1s0 network interface card to act as an access point and authentication server. Then, it installs the isc-dhcp-server daemon, adds the wireless network interface (i.e., wlp1s0) to the default isc-dhcp-server configuration file, comment unnecessary lines within the default configuration file and configures the 10.10.0.0/24 subnet. Next, puts the wireless interface wlp1s0 up, starts the isc-dhcp-server service, starts the hostapd service, and restart the isc-dhcp-server service to update the hostapd.

```
#!/bin/bash
# Install hostapd
sudo apt install hostapd -y

# hostadp config file create
mkdir ~/conf
cd ~/conf
cat > hostapd.conf << EOF
interface=wlp1s0
driver=nl80211
ssid=apu_app
hw_mode=g
channel=1
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
#wpa=3                      # ignore wpa
#wpa_passphrase=12345678    # doesn't need password
#wpa_key_mgmt=WPA-PSK
#wpa_pairwise=TKIP
rsn_pairwise=CCMP
EOF
cd ~

# install isc-dhcp-server
sudo apt install -y isc-dhcp-server
sudo service isc-dhcp-server stop

# add wireless interface to default isc-dhcp-server config file
sudo sed -i 's/INTERFACESv4=""/INTERFACESv4="wlp1s0"/' /etc/default/isc-dhcp-server
sudo sed -i 's/INTERFACESv6=""/INTERFACESv6="wlp1s0"/' /etc/default/isc-dhcp-server

# comment unnecessary lines on /etc/dhcp/dhcpd.conf
sudo sed -i '10,14 s/^/#/' /etc/dhcp/dhcpd.conf
sudo sed -i '20,21 s/^/#/' /etc/dhcp/dhcpd.conf

# configure subnet
sudo tee -a /etc/dhcp/dhcpd.conf > /dev/null <<EOT
subnet 10.10.0.0 netmask 255.255.255.0 {
        range 10.10.0.2 10.10.0.16;
        option domain-name-servers 8.8.8.8, 8.8.4.4;
        option routers 10.10.0.1;
}
EOT

# configure wireless interface -> wlp1s0
sudo ip addr add 10.10.0.1/24 dev wlp1s0
sudo ip link set wlp1s0 up

# start hostapd deamon && isc-dhcp-server
```

```
sudo service isc-dhcp-server start
sudo hostapd -B ~/conf/hostapd.conf
sudo service isc-dhcp-server restart
```

A.5 access-device bash script

This script tries to access the subnet with the "apu_app" SSID, and measures the time until succeeds. As depicted in Figure A.2, the script starts by putting up the wireless interface of the device and starting a time counter. Then, check if the interface is connected to a network: 1) if not, tries to access the network with "apu_app" SSID, 2) if yes, request an IP from the DHCP server and stops to measure the time.

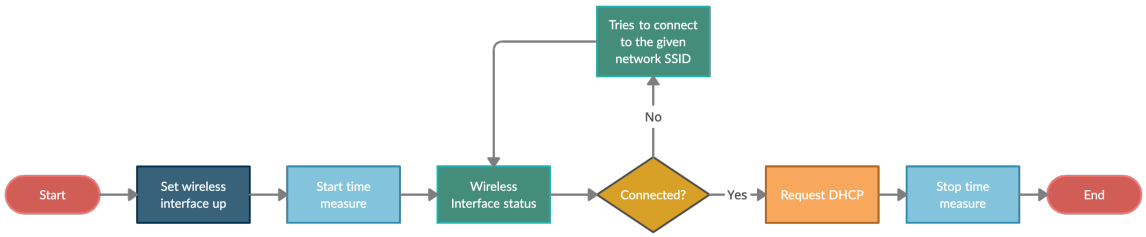


Figure A.2: Access Device Bash Script Flow Chart

```
#!/bin/bash
echo "Trying to access network"
sudo ip link set wlp3s0 up                          # Wireless device interface up
start=`date +%s`                                    # Start measuring the time
var2="Not connected." >> /dev/null                  # Compare string

while true; do
status=$(sudo iw dev wlp3s0 link) >> /dev/null      # Check the wireless interface status
sudo iw dev wlp3s0 connect -w apu_app >> /dev/null  # Tries to connect to the SSID "apu_app"

if [[ "$status" != "$var2" ]]; then                 # If success perform a DHCP request
sudo dhclient wlp3s0                                # if not, check status and tries again
break
fi
done

end=`date +%s`                                      #Stops measuring time
echo "Took `expr $end - $start` seconds to access the network."
```

A.6 iperf-test bash script

This script performs ten iperf3 measurements and saves each measure in a separate file. To test the network connectivity the command iperf3 -i 10 -w 1M -t 60 -c 10.10.0.1, which instructs iperf3 run the test for 60 seconds (-t), report the statistics every 10 seconds (-i), and use a TCP window of 1M (-w).

```
#|/bin/bash

for i in {1..10}
do
echo "IPERF $i"
iperf3 -i 10 -w 1M -t 60 -c 10.10.0.1  | tee ~/iperf_tests/results/t3/tcp_client$i.txt
echo ""
echo "IPERF $i done"
echo ""
done
```