Universidade de Aveiro
2018

Departamento de Electrónica,
Telecomunicações e Informática

NUNO MIGUEL DA SILVA DOMINGUES

REDES DEFINIDAS POR SOFTWARE E FUNÇÕES DE REDES VIRTUALIZADAS EM AMBIENTES COM RECURSOS RESTRITOS

SOFTWARE DEFINED NETWORKING AND NETWORK FUNCTION VIRTUALIZATION IN ENVIRONMENTS WITH CONSTRAINED RESOURCES

Universidade de Aveiro
2018

Departamento de Electrónica,
Telecomunicações e Informática

NUNO MIGUEL DA SILVA DOMINGUES

REDES DEFINIDAS POR SOFTWARE E FUNÇÕES DE REDES VIRTUALIZADAS EM AMBIENTES COM RECURSOS RESTRITOS

SOFTWARE DEFINED NETWORKING AND NETWORK FUNCTION VIRTUALIZATION IN ENVIRONMENTS WITH CONSTRAINED RESOURCES

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor Daniel Nunes Corujo, Investigador doutorado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e do Doutor Rui Luís Andrade Aguiar, Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

agradecimentos / acknowledgements

Agradeço aos meus orientadores, Doutor Daniel Corujo e Professor Doutor Rui Aguiar, pela sua orientação, ajuda e tempo dispensado ao longo desta dissertação, bem como a outros professores e colegas que contribuíram para a realização deste trabalho.

Finalmente agradeço à minha familía e aos meus amigos pelo apoio e incentivo que sempre me deram ao longo destes anos.

palavras-chave

SDN, NFV, Fog Computing, Dispositivos com recursos limitados, Raspberry Pi

resumo

Com tecnologias como SDN e NFV a impulsionar o desenvolvimento das redes da próxima geração, novos paradigmas como por exemplo, Fog Computing, apareceram na área de redes. Contudo, estas tecnologias têm estado associadas à infraestrutura das redes, como o datacenter. Para que estas tecnologias possam ser utilizadas, como por exemplo, num cenário de Fog Computing é necessário, então, estudar e desenvolver estas tecnologias para formar novos mecanismos de controlo e operação. Desta forma, um cenário de Fog Computing composto por dispositivos com recursos limitados, típicos neste tipo de situação, é desenvolvido, e, uma proposta de solução é apresentada. A solução consiste em adaptar uma VIM existente, OpenVIM, para este tipo de dispositivos, após a implementação da solução, onde um Raspberry Pi é utilizado para exemplificar este tipo de dispositvos. Testes são realizados para medir e comparar como estes dispositivos se comportam em comparação com dispositivos mais poderosos. Estes testes são compostos por testes de desempenho, focando o tempo de instanciação e consumo energético. Os resultados apresentam algumas limitações inerentes a este tipo de dispositivos resultantes dos seus recursos limitados, quando comparados com hardware com maior capacidade. Contudo, é possível verificar o potencial que este tipo de dispositivos podem apresentar no futuro próximo.

keywords SDN, NFV, Fog Computing, Resource-constrained devices, Raspberry Pi

Abstract

With technologies such as SDN and NFV pushing the the development of the next generation networks, new paradigms, such as Fog Computing, appeared in the network scene. However, these technologies have been associated with the network infrastructure, such as the datacenter. In order for these technologies to be used, for instance, in a Fog Computing scenario it is necessary to, therefore, study and develop these technologies to form new control and operation mechanisms. So, a Fog Computing scenario composed by resource-constrained devices, typical in these types of situations, was developed, and, a solution proposal is presented. The solution consists in customizing an existent VIM, OpenVIM, to this kind of devices, after the implementation of the solution, where a Raspberry Pi is used to exemplify this type of devices. Tests are done to measure and compare this devices to more powerful ones. The tests are comprised by benchmarks runs, focusing on instantiation times, and power consumption. The results show some drawbacks inherent to this kind of devices when compared to more powerful ones. However, it is possible to see the potential that this kind of devices might have in the near future.

Contents

List of Figures

List of Tables

List of Acronyms

AC	Alternate Current
AP	Access Point
API	Application Programming Interface
CAPEX	Capital Expenditure
COTS	commercial-off-the-self
CPU	Central Processing Unit
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DUT	Devices under test
EPA	Enhanced Placement Awareness
ETSI ISG	European Telecommunications Standards Institute Industry Specification Group
FFT	Fast Fourier Transform
flops	Floating Point Operations per Second
GUI	Graphical User Interface
I/O	Input/Output
IaaS	Infrastructure-as-a-Service
IoT	Internet of Things
ISC	Internet Systems Consortium
ISP	Internet Service Provider
LAN	Local Area Network
LXC	Linux Containers
MD-SAL	Model-Driven Service Abstraction Layer
MEC	Multi-access Edge Computing
MQTT	Message Queuing Telemetry Transport
MSS	Maximum Segment Size
NAT	Network Address Translation
NFV	Network Function Virtualization
NFV-MANO	NFV Management and Orchestration
NFVI	NFV Infrastructure
NIST	National Institute of Standards and Technology
ON.Lab	Open Networking Lab
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OPEX	Operational Expenditure
OS	Operating System
OSM	Open Source MANO
OvS	OpenvSwitch
PaaS	Platform-as-a-Service
PCIe	Peripheral Component Interconnect Express
PNF	Physical Network Function
RAN	Radio Access Network
RPi	Raspberry Pi
SaaS	Software-as-a-Service
SAL	Service Abstraction Layer
SBC	Single-Board Computer
SDN	Software Defined Networking

SDWN	Software Defined Wireless Networks
SRIOV	Single-root input/output virtualization
SSH	Secure Shell
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
UI	User Interface
VLAN	Virtual Local Area Network
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function

Chapter 1

Introduction

Technologies such as Software Defined Networking (SDN) and Network Function Virtualization (NFV) are here to stay, and with the next generation networks (5G) in sight, solutions are being designed and developed in which these technologies can be fully utilized. SDN, which brought to the table the decoupling of data and control planes in networks enabling a centralized view of the network, complemented with NFV, which brought the virtualization of network functions decoupling them from the proprietary hardware they used to run in, gave the opportunity for the community to walk away from the vendor lock-in problem, thus, propelling the rise of open source solutions. This meant that companies and researchers could now work together in developing these technologies, and while still not everything is disclosed and revealed, it still meant more people were now able to help its development.

Along with others paradigms that emerged due to the combination of the technologies mentioned above, Fog Computing appeared with the objective of enabling computing on the edges of the network. More and more devices are connecting to the Internet, therefore, infrastructures need to be constructed or modified in order to manage this recent growth of devices. Since some of these services have several requirements, such as low latency, the distance between the compute node and the infrastructure should be low as well, thus, the need of enabling computing near the edge, instead of just at the core of the network. Fog Computing intends to cooperate and compensate for the deficiencies of the Cloud in these types of scenarios. There are already applications which require both the Cloud Computing and the Fog Computing capabilities, moreover, the combination of these paradigms enables new services and applications to appear on the edge.

This dissertation rises from the need to explore mechanisms, such as SDN and NFV, in fields such as Fog Computing. These mechanisms have been more focused towards the infrastructure, such as the datacenter, so there is little support for devices on the edge of the networks, as the software designed ends up being too powerful to run. Therefore, this dissertation aims to explore scenarios where SDN and NFV technologies are applied in non-traditional equipments, such as Single-Board Computers (SBCs) (e.g., a Raspberry Pi (RPi)), which are devices that, when compared to normal computers and/or laptops, are more resource-constrained and, therefore, offer less computing power but are smaller and less expensive.

1.1 Motivation

The application of Information and Communication Technologies in a growing number of sectors of our society has been rising the technological requirements in the supply of networks and services, in both users and connected devices. Therefore, there are scenarios where not only a great number of services with different requirements exist, but also devices with different characteristics (i.e., from datacenters to microcontrollers), different capacities and connected by different types of access networks.

In order to expedite the control and organization of telecommunications networks, and face the necessities of controlling and operating networks in this type of environment, the next generation networks will capitalize in the usage of mechanisms such as SDN and NFV. However, so far, the focus of the application of these two new mechanisms is mainly associated with the operation of the infrastructure of the operator network, and the datacenter part, where processing capacity and energy efficiency are on a different scale when compared to resource-constrained devices. Conversely, if scenarios consisting of low resource devices, such as the ones involving Internet of Things (IoT), and aimed to use services with high requirements (such as video), it is necessary to develop and apply these mechanisms to operate in these environments.

In short, this dissertation aims to study, develop and validate new control mechanisms and network operations, using SDN and NFV, in low-power computing environments.

1.2 Objectives and Contributions

The objectives of this dissertation can be divided in two parts. In the first part, the goal is to define and design a scenario for the integration of low-power computing devices using technologies associated with 5G network control mechanisms (i.e., SDN and NFV). In the second part, the goal is to implement, configure and test the developed scenario in order to compare results with more powerful devices normally used in the usual scenarios. This dissertation is part of a pioneering national project in 5G networks (Mobilizador 5G, `http://5go.pt/`).

1.3 Document Structure

This document is structured in 5 chapters, where the first one is this introduction. The following chapters are listed below:

Chapter 2 - Concepts and Tools: A chapter where concepts are presented for a better understanding of this dissertation's contributions and a look is taken into tools recently developed and researched.

Chapter 3 - Framework and Implementations: Presenting the solution reached in order to solve the issue presented in this dissertation, the adjustments that had to be made and actual implementation.

Chapter 4 - Results and Analysis: Results obtained throughout this dissertation and their analysis.

Chapter 5 - Conclusion: In this final chapter the conclusions about the results are presented and future work is also discussed.

Chapter 2

Concepts and Tools

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 which are within the scope of this dissertation.

2.1 Concepts

The usage of technologies such as SDN and NFV drove the area to new paradigms. Initially, Cloud Computing appeared and by pushing the computing tasks to the Cloud led to a more efficient way to process data, since there was no need to invest on infrastructures if the processing was outsourced to Cloud Computing infrastructures. Therefore, computation time was reduced when compared to the low capabilities at edge. However, the increase of data consumption, data production, number of devices on the edge, and, the stagnation of bandwidth led to the appearance of new paradigms to solve these issues. So, Edge Computing was the answer to this issue, by enabling technologies that allowed computation at the edge of the network instead of at the centralized datacenter. Afterwards, other more niche paradigms appeared, with the name of Fog Computing and Multi-access Edge Computing (MEC) aiding this paradigm while answering their own issues [11].

2.1.1 Software Defined Networking

In legacy networks (non-virtualized networks), the data and control planes are coupled, the transmission of data has been dominated by the use of dedicated switches or routers to direct packets between servers or other connected devices. These switches consist of two planes: the data or forwarding plane, which handles the routing of data packets to its network destination; and the control plane, which creates the flow tables that determine how packets are sent to a destination [12]. In this approach, once the flow management (forwarding policy) has been set, modifying the configuration of the devices themselves is the only way to make changes. This proved restrictive for network operators who needed to scale their networks in response to changing traffic demands and increasing use of mobile devices [13]. Software Defined Networking (SDN) is a concept that proposes the decoupling of the data and control planes in networks detaching the connections and equipments through which packets flow (forwarding plane) from the control decisions made as to what should be done with those packets (control plane), turning the network into a programmable entity [14] and allowing the network to better adapt to more dynamic environments.

2.1.1.1 SDN Architecture

The SDN architecture (figure 2.1) allows for decoupling of the data and control planes as said above. This enables a centralized view of the network and an abstraction of the network infrastructure from the applications, which results in scalable, flexible networks that can adapt and react to the needs of the network administrator.

There are three main planes in this architecture:

- **Data plane** which is composed of network elements and is in charge of exposing their capabilities to the control plane.
- **Control plane** where decisions about where the traffic is sent to are decided and it is also the responsibility of the control plane to configure and manage the network as well as updating the flow table information. The control plane also translates the applications requirements and provides meaningful information to the applications.
- **Application plane** that consists of network applications that network administrators want to deploy in their networks.

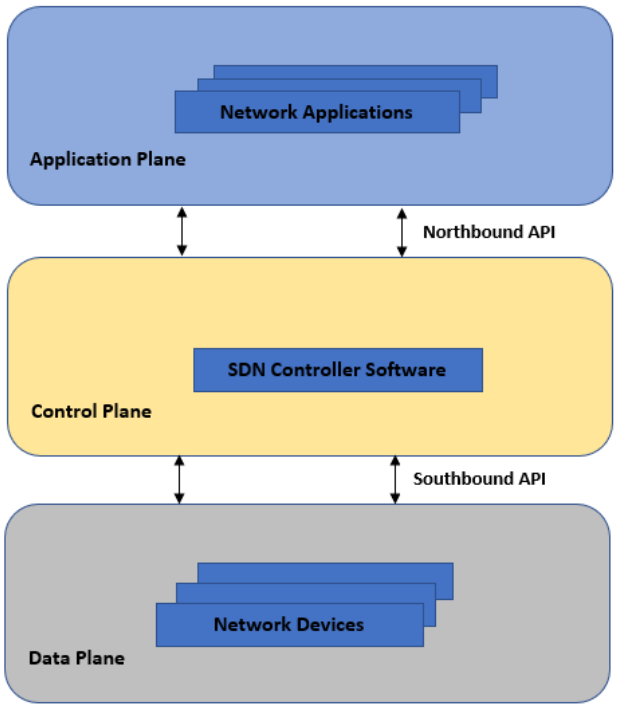

Figure 2.1: SDN architecture.

The interaction between the control and forwarding planes is enabled through a southbound Application Programming Interface (API). There are several interfaces, however the most prominent is OpenFlow, created by Open Networking Foundation (ONF)[1]. Between the control and application planes the communication is done through the northbound API which facilitates the orchestration and automation of network services running in the application plane.

2.1.1.2 SDN Benefits

With a centralized and programmable network, SDN provides several benefits to companies and network operators such as [15]:

- **Programmable:** Since the control plane is decoupled from the data plane, the network can be programmed or configured by the controller.

- **Centralized:** Network intelligence is logically centralized in the controller which keeps a global view of the network, which appears as a single, logical switch.

- **Reduce Capital Expenditure (CAPEX):** SDN potentially reduces the need to purchase dedicated networking hardware, supporting pay-as-you-grow models instead.

- **Reduce Operational Expenditure (OPEX):** Enables control of the network elements (e.g., hardware or software switches/routers) which are programmable, making it easier to design, deploy, manage and scale networks. Overall management time is reduced due to optimized serviced availability.

- **Agility and Flexibility:** Rapidly deploy new applications, services and infrastructure to meet changing goals and objectives.

- **Innovation:** It enables organizations to create new types of applications, services and business models that can offer new revenue streams and more value from the network.

All the points above illustrate why SDN has risen up in popularity and is now being utilized in the new generation networks.

2.1.1.3 SDN Controllers

As stated before, the separation of the data and control planes within the network, allowed the state of the network to be managed and controlled centrally. Residing in the control plane, the SDN controller has a complete view of the network and with that, it is capable of deciding where the traffic is directed, besides it also keeps an updated flow table.

Since SDN is still being heavily researched and developed, there is a large number of SDN controllers available for use, at different stages of development and deployment. Amongst them, OpenDaylight[2], ONOS[3], and, Floodlight[4] will be introduced, since these are the controllers supported by the software chosen for the framework presented later (section 3.1).

[1] `https://www.opennetworking.org`
[2] `https://www.opendaylight.org`
[3] `https://onosproject.org`
[4] `http://www.projectfloodlight.org/floodlight/`

OpenDaylight Founded in 2013, and now a member of LF Networking, OpenDaylight is an open-source project with a modular open platform for tailoring networks of any size and scale. The modular and multiprotocol permits downstream users and solution providers maximum flexibility when building a controller which fits their needs, by leveraging services created by others, writing and creating their own. Since OpenDaylight is implemented in Java, it can be ran on any hardware and operating system platform that supports Java.

Its latest release, named Fluorine[1], was released on August, 2018. It is the ninth release moving the project to a system of managed releases. The architecture of this release can be found on figure 2.2.

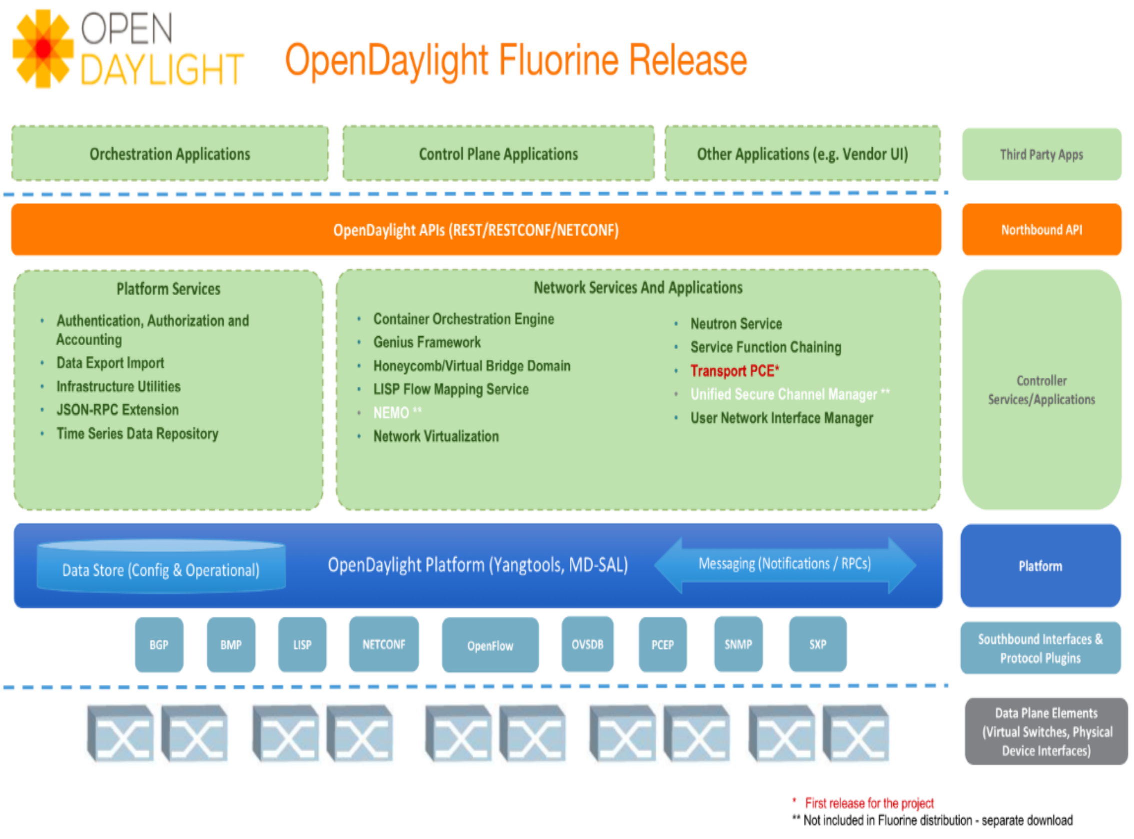
OPEN DAYLIGHT
OpenDaylight Fluorine Release
Orchestration Applications
Control Plane Applications
Other Applications (e.g. Vendor UI)
Third Party Apps
OpenDaylight APIs (REST/RESTCONF/NETCONF)
Northbound API
Platform Services
Authentication, Authorization and Accounting
Data Export Import
Infrastructure Utilities
JSON-RPC Extension
Time Series Data Repository
Network Services And Applications
Container Orchestration Engine
Genius Framework
Honeycomb/Virtual Bridge Domain
LISP Flow Mapping Service
NEMO **
Network Virtualization
Neutron Service
Service Function Chaining
Transport PCE*
Unified Secure Channel Manager **
User Network Interface Manager
Controller Services/Applications
Data Store (Config & Operational)
OpenDaylight Platform (Yangtools, MD-SAL)
Messaging (Notifications / RPCs)
Platform
BGP
BMP
LISP
NETCONF
OpenFlow
OVSDB
PCEP
SNMP
SXP
Southbound Interfaces & Protocol Plugins
Data Plane Elements (Virtual Switches, Physical Device Interfaces)
* First release for the project
** Not included in Fluorine distribution - separate download

Figure 2.2: OpenDaylight's Fluorine architecture [1].

OpenDaylight has three main layers. The northbound layer uses REST, RESTCONF or NETCONF in order to communicate between the third party apps and the core. The southbound layer has several protocols and control plane services, which can be individually selected or written, and packaged together in requirement to a specific use case, connecting to the data plane elements. The controller platform offers some platform services, as well as, network services and applications. This is managed by the Model-Driven Service Abstraction Layer (MD-SAL). In OpenDaylight, underlying network devices and network applications are all represented as objects, or models, whose interactions are processed within the Service Abstraction Layer (SAL)[16].

The SAL is a data exchange and adaptation mechanism between YANG models. These models give a generalized description of capabilities without needing to know the implementation details of the other. In SAL, models are defined by their roles in an interaction. A producer implements an API and provides its data, generates notifications and inserts data into SAL's storage, while a consumer utilizes the API and its data, receives notifications and gets data from providers, and reads data from SAL's storage. Inside SAL, consumer and producer are more accurate descriptions of interactions than southbound and northbound, which provide a network's engineer view. Using its data stores, SAL matches producers and consumers and exchanges information.

ONOS Founded by Open Networking Lab (ON.Lab), the Open Network Operating System (ONOS) is the open source SDN networking operating system for Service Provider networks architected for high performance, scale and availability[17]. As ONOS is a Network Operating System it is responsible for: managing finite resources in the name of resource consumers, isolating and protecting users from each other, providing useful abstractions that enable users to easily utilize services and resources, and, providing security from the external world[18].

In order to accomplish its goals, ONOS's architecture possesses the following features[19]:

- **High Availability and Resiliency:** High availability is required of service providers to avoid network downtime and there are mechanisms to support demanding networks ensuring reliable network connections.

- **Performance at Scale:** Built to provide the best performance possible, ONOS maintains its response time while adding new features.

- **Modular Software:** Software modularity makes it easy to develop, debug, maintain and upgrade ONOS as a software system.

- **Northbound Abstractions:** Eases the development of control, management, and configuration services.

- **Southbound Abstractions:** Insulates the core of ONOS from the details of different devices and protocols.

- **GUI Framework and Base UI:** Provides the view of the multi-layered network allowing the exploration of various elements, errors and state of the network.

- **YANG Tool-Chain:** Interacts with several network elements who support configurations modeled via YANG and allows ONOS's platform to dynamically extend its configuration capabilities.

ONOS believes there should be clear boundaries between subsystems, and therefore, it is partitioned into three main tiers[20]:

- protocol-aware network-facing modules interacting with the network;

- protocol-agnostic system core tracking and serving information about the network state;

- applications consuming and acting upon the information given by the core.

Figure 2.3 represents a summary of the various tiers of functionality included in the ONOS architecture. The Providers and Protocols layers include functionalities to communicate with the Network Elements and, with the help of southbound (provider) API, network-facing modules can also communicate to the core layer. The core layers, as mentioned above, track and serve information about the state of the network. Finally, the core can communicate with the applications via the northbound (consumer) API, which, also provides these applications with abstractions describing network components and properties.

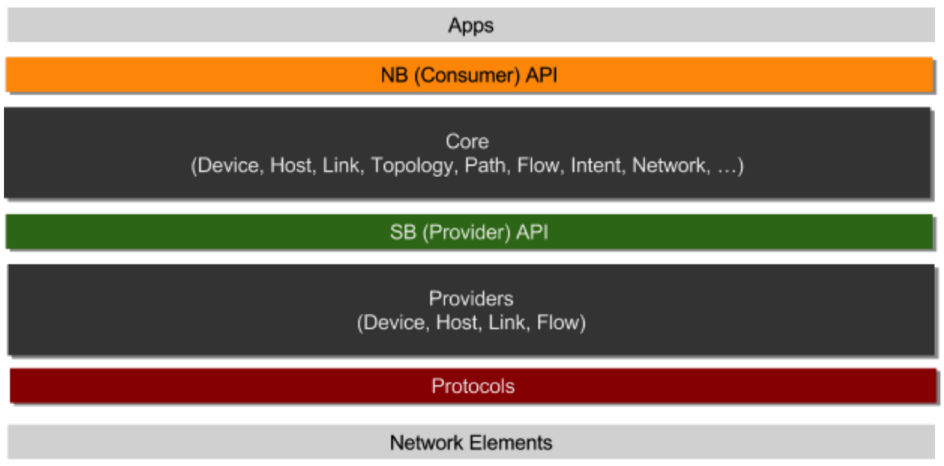
Apps
NB (Consumer) API
Core
(Device, Host, Link, Topology, Path, Flow, Intent, Network, ...)
SB (Provider) API
Providers
(Device, Host, Link, Flow)
Protocols
Network Elements

Figure 2.3: ONOS Layers [2].

Floodlight Floodlight is an open source SDN controller, supported by the community and Big Switch Networks developers. It is an Apache-licensed, Java-based OpenFlow Controller. The controller does a set of functionalities to control and inquire an OpenFlow network, and different user needs on the network are solved through applications running on top of it.

Figure 2.4 represents the architecture diagram of the floodlight controller, where it is possible to see that Floodlight is comprised of various modules that communicate via the east/westbound API and it can even be extended with new modules providing new functions. Developers can use any language that supports REST to build software for the application plane due to the REST compatible northbound API present[21].

Floodlight adopts a modular architecture to implement its controller features and some applications. Some of its modules are described below:

- **Device Manager:** Manages hosts/devices and where they are connected to the network;
- **Firewall:** Reactively inserts flows that either allow or deny packets from traversing the network, enforcing Access Control List rules on OpenFlow enabled switches;
- **Forwarding:** As the name implies, it forwards packets between two devices;
- **Learning Switch:** Replicates traditional learning switch behavior and associates MAC addresses with switch ports;
- **Packet Streamer:** Selectively streams Openflow packets exchanged between any switch and the controller to an observer;

- **PortDown Reconciliation:** Reconcile flows across a network when a port or link goes down;

- **Static Flow Entry Pusher:** Allows modules to insert static/proactive flows and groups identified by a unique string name per entry;

- **Thread Pool:** As implied, it provides threads to modules;

- **Topology Manager/Routing:** Manages the topology based on discovered links and connected switches;

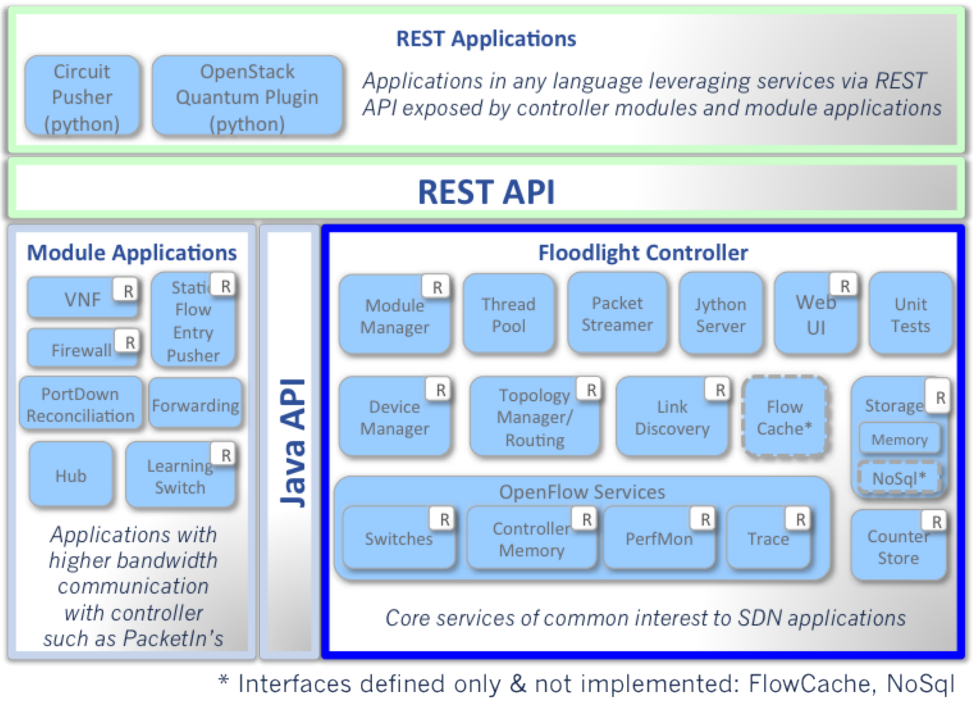

Figure 2.4: Floodlight Architecture Diagram [3].

2.1.2 Network Function Virtualization

In legacy networks, network function implementations are often strongly connected with the infrastructure they run on. Network Function Virtualization (NFV) is a concept that decouples network functions from the computation, storage, and networking resources they use [22] enabling them to be run on commercial-off-the-self (COTS) hardware be it inside a container or Virtual Machine (VM) bringing forth several benefits. The ability to launch a network function on-demand increases flexibility and agility. Furthermore, it removes the problem of vendor lock-in as there is no more need to buy proprietary hardware, which reduces CAPEX and OPEX in itself, since a general machine can be used to instantiate a network function and in the future be reused to instantiate a different one. It also allows for customization of services where a network function is designed to better fit different requirements.

2.1.2.1 ETSI-NFV Framework

In order to standardize work in the NFV field, the European Telecommunications Standards Institute Industry Specification Group (ETSI ISG)[5] created an high-level framework (figure 2.5) composed of three main working domains: NFV Infrastructure (NFVI), Virtual Network Functions (VNFs), and NFV Management and Orchestration[4].

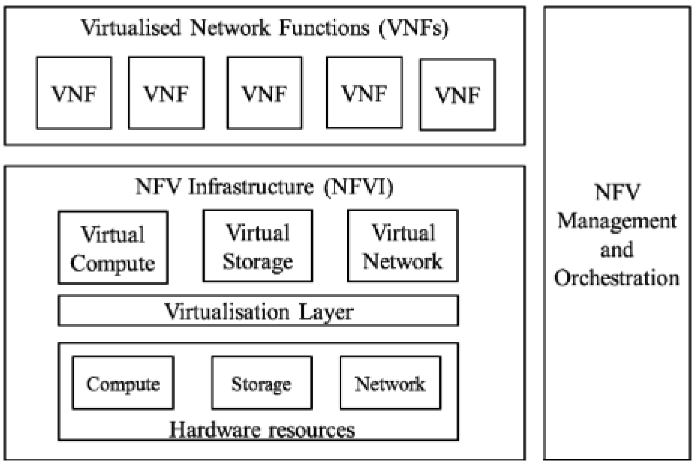

Figure 2.5: High-level NFV framework [4].

NFV Infrastructure: The NFVI features the totality of all hardware and software components in which the VNFs are deployed, managed and executed. From the VNF's point of view, the virtualization layer and the hardware resources look like a single entity providing the VNF with desired virtualized resources. The physical resources include computing, storage and network that supply processing, storage and connectivity to VNFs through the virtualization layer. The virtualization layer abstracts and logically partitions physical resources, enables the software that implements the VNF to use the underlying virtualized infrastructure and provides virtualized resources to the VNF, so that the latter can be executed [4].

Virtual Network Functions: A VNF is a virtualization of a network function in a legacy non-virtualized network. A Physical Network Function (PNF) and a VNF should have the same functional behaviour and external operational interfaces. The Element Management performs the management functionality for the VNFs [4].

NFV Management and Orchestration:
The NFV Management and Orchestration (NFV-MANO) represented in figure 2.6, is composed of three main blocks, they are the NFV Orchestrator, the VNF Manager and the Virtualized Infrastructure Manager (VIM). The NFV Orchestrator is tasked with orchestrating and managing of the NFV infrastructure and software resources, and realizing network services on NFVI. The VNF Manager is designated with VNF lifecycle management (e.g. instantiation, update, query, scaling, termination). Finally, the VIM encompasses the functionalities which are used to control

[5] `https://www.etsi.org`

and manage the interaction of a VNF with computing, storage and network resources, as well as their virtualization. In addition, performs resource management that is in charge of the inventory of software and hardware resources reserved to NFVI, allocation of virtualization enablers and management of infrastructure resource and allocation. It also offers operations for visibility into and management of the NFV infrastructure, as well as collecting various kinds of information intended for capacity planning, monitoring, optimization, etc.

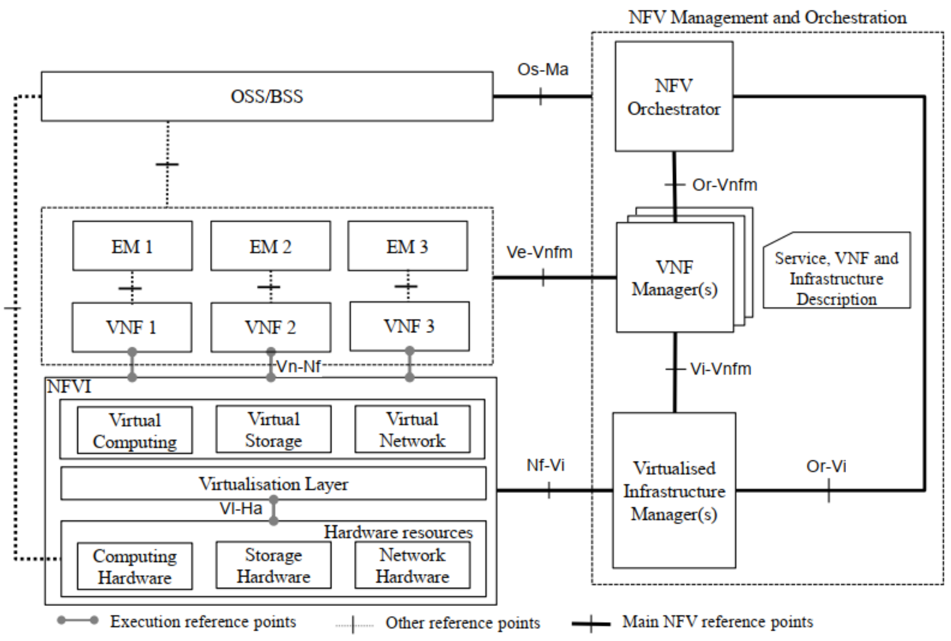

Figure 2.6: NFV architecture [4].

2.1.2.2 NFV Platforms

Now that NFV has been introduced, platforms where one can run virtualization services will be discussed. As with the SDN controllers, there are also several platforms available to run virtualization services. Two of them will be briefly discussed here: OpenStack and OpenVIM. OpenStack because it is, probably, the most popular one of these platforms, and, OpenVIM because it is the software chosen in this dissertation.

OpenStack Created in 2010 by the joint effort of NASA and Rackspace, who based this project on their previous work: NASA had a compute project which ended up being Nova, and Rackspace had a Cloud files project which ended up being Swift[23]. Nowadays, developed by the OpenStack Foundation, founded in 2012, OpenStack is an open-source Infrastructure-as-a-Service (IaaS) Cloud Computing software.

OpenStack software controls large pools of compute, storage, and, networking resources in the NFVI on a datacenter. It is comprised of multiple projects or modules developed semi-independently, each performing his part on a Cloud Computing service.

Figure 2.7 presents a logical architecture of OpenStack, where it is possible to see the common integrated services within OpenStack and how they interact with each other[24].

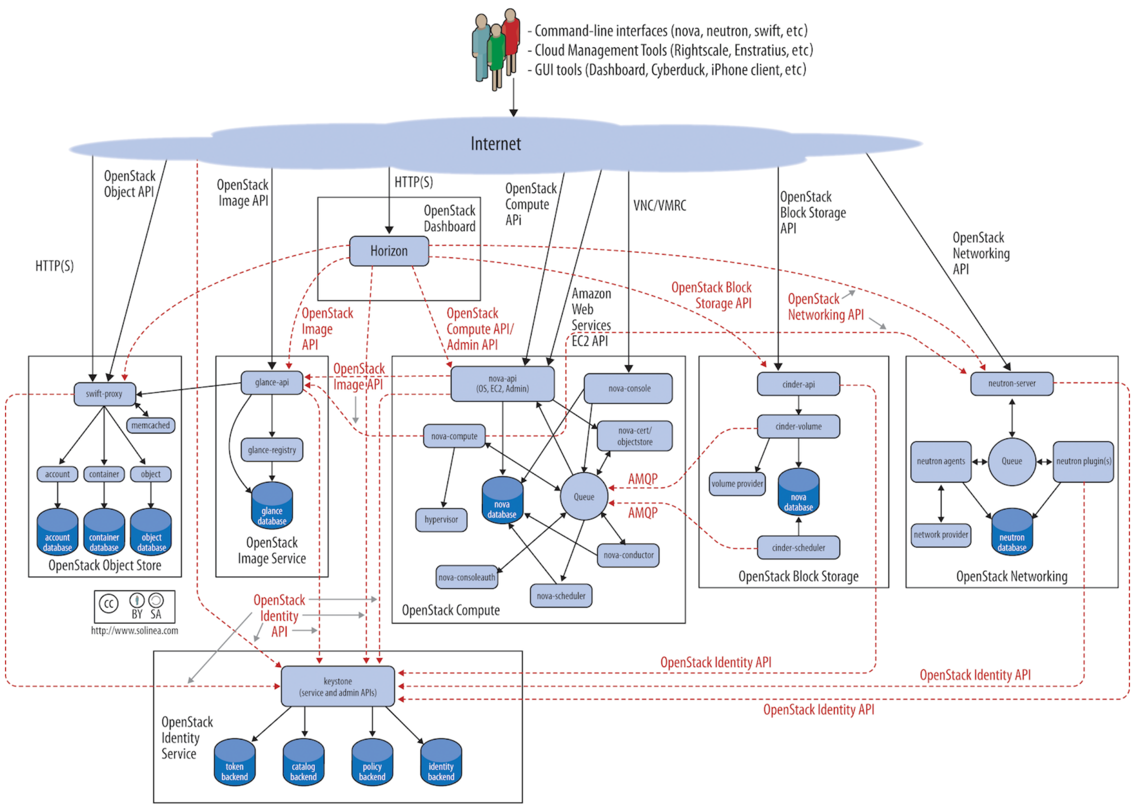

Figure 2.7: Openstack logical architecture [5].

- **Dashboard**, or Horizon, is a web application providing an User Interface (UI) where users can interact with the OpenStack framework. It can be used to implement most of the services, interacting with them through specific APIs;
- **Object Store**, or Swift, is a reliable, distributed object/blob store, which permits users to store and retrieve files;
- **Image Service**, or Glance, is a system for discovering, registering, and retrieving VM images, providing an efficient way to boot VMs. It can be configured to use Swift or a local file system as storage;
- **Compute**, or Nova, provides management and provision of VMs, or virtual servers, after interaction with the available hypervisors, giving a platform for Cloud management through its APIs;
- **Block Storage**, Cinder, provides persistent volumes to guest VMs, which are then managed by Nova. It is Cinder that allows through its API to manipulate volumes, volume types and volume snapshots;
- **Networking**, or Neutron, provides network connectivity as a service managed by other services (e.g., Nova). It also provides tasks that enable the deployment of VMs (e.g., Dynamic

Host Configuration Protocol (DHCP), Virtual Local Area Networks (VLANs), Domain Name System (DNS), etc.);

- **Identity Service**, or Keystone, provides authentication and authorization policy services applying them to users and services interactions.

OpenVIM Openvim[6] was originally an open source project providing a practical implementation of an ETSI MANO stack, where OpenVIM was the VIM implementation directly interfacing with the compute nodes and storage in the NFVI, and an OpenFlow controller. Nowadays, the project has contributed to the open source community project Open Source MANO (OSM)[7], hosted by ETSI, and is currently being maintained by OSM where it is currently on the third release[25].

OpenVIM has a simple architecture displayed in figure 2.8. It possesses 5 modes in which it can be used, these 5 modes are shown on table 2.1. The "normal" mode is the default mode, where the OpenFlow controller, switches and real hosts are needed. The "test" mode is used for testing the HTTP API and the database without connection to the host or to the OpenFlow controller. The "host only" mode is used when neither the OpenFlow controller nor the OpenFlow switch are provided, and, dataplane networks must be done manually (no Single-root input/output virtualization (SRIOV) connections, SR-IOV is a specification that makes a single Peripheral Component Interconnect Express (PCIe) physical device under a single root port to appear to be multiple separate physical devices to the hypervisor or the guest Operating System (OS)). The "OF only" mode is used to test new OpenFlow controllers support, no real VM deployments will be done, but the OpenFlow Controller will be used as in "normal" mode. Finally, the "develop" mode forces a Cloud-type deployment, where a bridge network instead of a real OpenFlow controller/dataplane networks is used, normal memory instead of hugepages is used, and, no CPU pinning is used, in other words, no Enhanced Placement Awareness (EPA) (EPA facilitates better informed decision making related to VM placement resulting in better performance and efficiency).

Mode	**Compute Hosts**	**OpenFlow Controller**	**Note**
Test	Fake	X	No real deployment. Just for API test.
Normal	Needed	Needed	Normal behaviour.
Host Only	Needed	X	No PT/SRIOV connections.
Develop	Needed	X	Force to Cloud type deployment without EPA.
OF Only	Fake	Needed	To test OpenFlow controller without need of compute hosts.

Table 2.1: Modes available in OpenVIM.

OpenVIM has a REST based northbound API alongside its main service (openvimd), which is responsible for maintaining the VMs, images and networks. Every functionality in this server has its own thread (HTTP Server, DHCP Controller, OpenFlow Controller, Database Management, and, Host). OpenVIM delegates the provision of the virtualization infrastructure to the host/compute node, therefore, each host needs to run libvirt and KVM[8]. When a compute node is added to

[6]`https://github.com/nfvlabs/openvim`
[7]`https://osm.etsi.org`
[8]`https://www.linux-kvm.org/page/Main_Page`

OpenVIM a Secure Shell (SSH) channel is created between the compute node and a host thread from OpenVIM for communication.

OpenVIM supports three OpenFlow controllers: Floodlight, ONOS, and, OpenDaylight. For overlay network management OpenVIM offers two options, precreated bridges at compute nodes which possess L2 connectivity simply named "bridge" or an OpenvSwitch (OvS) VXLAN tunnel named "ovs". If the first is chosen, then an external DHCP server needs to be provided for the management network, if it is the second that is chosen, then, OpenVIM launches a DHCP server in the OVS controller.

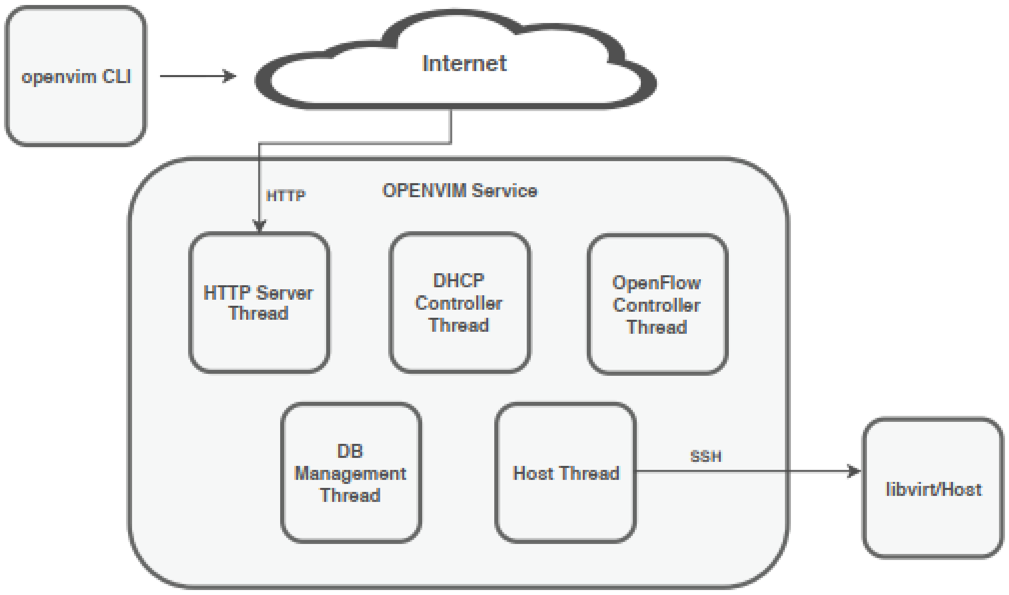

Figure 2.8: OpenVIM logical architecture [6].

2.1.3 Cloud Computing

According to [26], Cloud Computing is a blueprint for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources. This access can be rapidly provisioned and released with minimal management effort or service provider interaction. This blueprint is comprised 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 combined to serve a number of consumers using a multi-tenant model. Although, the consumer normally has no control or knowledge of the precise location of the provided resources, the consumer may be able to determine location at a higher level (e.g., country, state, or datacenter).

- **Rapid elasticity:** Capabilities can be scaled (provisioned and released) at any time.
- **Measure service:** Resources must be metered so that Cloud Computing systems can automatically control and optimize resource usage.

The service models determine the abstraction level that the shared resources are made available. Software-as-a-Service (SaaS) has the resources presented as an end-application, Platform-as-a-Service (PaaS) the consumer does not manage or control the underlying infrastructure, but has control over the deployed applications, finally, in IaaS the consumer is provisioned with computing resources where the consumer is able to deploy and run arbitrary software.

Lastly, the deployment models are the "private cloud", where the infrastructure is provisioned for the exclusive use by a single organization, the "community cloud", which is similar to the private Cloud, but is provisioned to several organizations that have shared concerns, the "public cloud", where the infrastructure is provisioned for open use by the general public and the "hybrid cloud", which is a composition of two or more Cloud Computing infrastructures, but are tied together by standardized or proprietary technology that enables migration of data and applications.

2.1.4 Edge Computing

Edge Computing stemmed due to the inefficient support from the Cloud Computing to some applications who required faster response times. Cloud Computing with its efficient, faster computation times due to the more powerful devices present in the datacenter is still an answer, however, data processing speed evolved, whilst, the bandwidth of a network came to stagnation and with more data generated at the edge, due to the growth of IoT, speed of data transportation became the bottleneck for the Cloud paradigm[11]. If data needs to be processed fast, the Cloud is an attractive idea, but, the response time would be too long, therefore, the data has to be processed at the edge to lower the response time and the network pressure. With IoT, attempting to connect everything together, the number of devices connecting to the Internet is growing as well, therefore, raw data production will grow to a phase where normal Cloud Computing will not be sufficient, and/or the bandwidth and resource usage would be to large[11]. The change from data consumer to "prosumer" (i.e., both producer and consumer) by the end devices also supports the need for another paradigm to exist before the Cloud. Thus, Edge Computing proposes the enabling of technologies to let computation be done closer to the end devices on behalf of the Cloud[11]. This way, response time can be lowered, important for time-restricted applications, which, needing lower latency felt the centralization of the data processing the most, and bandwidth on the datacenters can be alleviated by performing the processing closer to the devices. Inside this new paradigm, more specific ones appeared, like Fog Computing and MEC.

Fog Computing extends the Cloud Computing to the edge of the network. Ever since the rise of IoT there was a need of a new platform that better suited the needs of such deployments. As [27] stated, some motivating examples for a new Fog Computing architecture are edge location, location awareness, low latency, geographical distribution, large-scale sensor networks, very large number of nodes, support for mobility, real-time interactions, predominance of wireless access, heterogeneity, interoperability and federation and support for on-line analytic and interplay with the Cloud. By distributing small clusters of servers with limited computation and storage resources, the distribution of Fog Computing throughout the Edge is achieved. These clusters range from a single SBC (e.g., a RPi), to a cluster of RPis, a laptop, or even, a home server depending on the requirements[28]. Fog Computing is not only located at the edge, unlike Cloud Computing, which is more centralized, Fog Computing targets the services and applications with widely distributed deployments, like mobile nodes[29].

MEC can provide Cloud Computing capabilities at the edge of Radio Access Network (RAN) in close proximity to mobile users. It is currently being standardized by an ETSI ISG. Introduced in order to try to reduce the distance between the Cloud and the user, which hinders latency and therefore, response time, MEC is built on a virtualized platform and enables applications running at the edge and by giving RANs strong computing capabilities, intends to reduce latency, guarantee efficient network operation and service delivery, and provide an improved user experience. Therefore, MEC is characterized by low latency, proximity, high bandwidth and real-time insight into radio network information and location awareness, enabling the evolution to 5G, so the mobile network can be transformed into a programmable world[30].

Both Fog Computing and Multi-access Edge Computing are inserted within Mobile Edge Networks and although these two architectures are similar, there are some differences between them [31]. On one hand, Fog Computing intends to leverage Cloud Computing and Edge Computing architectures to enable end-to-end IoT scenarios, and as stated before, is driven from an IoT perspective to give distributed Cloud Computing to IoT, is an extension of the Cloud, but requires distributed computing and storage to accommodate the numerous devices present in an IoT scenario. It also requires near real time interaction and efficient communication between different edge nodes, and is located between devices and a datacenter. On the other hand, Multi-access Edge Computing intends to provide IT and Cloud Computing capabilities withing the RAN. It creates architecture framework and standards for APIs located at the edge, does not need to be an extension of Cloud Computing and it focuses more on enabling applications that need low latency. It does not need the same amount of efficiency and real time interaction as nodes in Fog Computing, since the mobility is mainly consisted of disassociating from one edge node and associating with a new one.

2.2 Existing Experimental Works

There have been some active research made on resource-constrained devices, be it tests on general Central Processing Unit (CPU) performance, energy efficiency, etc., of SBCs, or in the developing fields of SDN and NFV for simple economic testbeds. Some research has, also, been made more specifically on Edge or Fog Computing scenarios, where, resource-constrained devices thrive.

2.2.1 Tests on Single-Board Computers

In [7] and [32], Kaup, Fabian, et al. evaluate the forwarding and computing performance of several SBCs over the last years and then derive models for performance and energy cost extending on previous studies done in [33]. The SBCs under analysis are Raspberry Pi 2 and 3, Odroid C1 and C2 and Cubieboard3 and a schematic of the power measurements is shown in figure 2.9. In the end, the authors are able to derive models mapping the system utilization to power, therefore, providing a method of calculating the power consumption of live systems providing VNFs at end-user premises. These experiments showcase one of the main strengths about SBCs, its power consumption, which, on this dissertation will also be calculated to gain a better understanding of the difference in power consumption between a SBC and a normal laptop running NFV services. The methods for power measurements differ between these experiments and this dissertation, as the power in the experiments is measured with the help of the USB dongles on the RPi, while, in this dissertation the power is calculated by measuring the current flowing from the outlet to the RPi's power source (section 4.3).

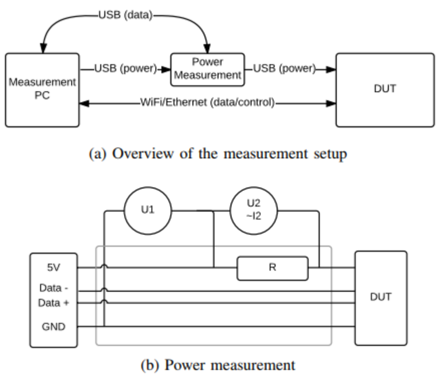

Figure 2.9: Overview of the power and measurement setup [7].

In [34], the author analyzes the deployment requirements of IoT gateways and evaluates containerized deployment with linux containers as a potential approach addressing them. Comparing two models of Raspberry Pi, RPi model B+ and RPi 2 model B, the author runs synthetic benchmarks for CPU, memory, disk, network, and Input/Output (I/O) performance and then application benchmarks for throughput and latency in order to isolate and understand the overhead introduced by the linux containers and Docker[9]. The results of the synthetic benchmarks show that the only relevant performance overhead introduced by Docker was found in the network I/O tests due to Network Address Translation (NAT), resulting on a performance degradation on the RPi B+. Additionally, the results of the application benchmarks indicate that using linux containers and Docker on the RPi B+ result in a significant performance degradation, supporting the results of synthetic benchmarks, and that NAT is not the only cause of performance decrease since disabling it did not allow to reach the performance of the native hardware. On the other hand, the RPi 2 showed more promising results, since, even though using Docker results in a measurable overhead in some occasions, the results are predictable and disabling NAT allows to reach performance comparable to the native hardware.

Analyzing the performance of the container (Docker) approach compared to a hypervisor-based virtualization (KVM) when running on devices typically used at the network edge, the authors in [35] run benchmarks on a Cubieboard2 to measure CPU, memory, disk I/O, and network I/O performance. First, a native (non-virtualized) performance was used as a base case to compute the virtualization overhead of both solutions. After running all the benchmarks stated previously, the results showed a better performance overall of Docker containers compared to

[9]`https://www.docker.com`

hypervisor virtual machines. In CPU benchmarks, Docker showed comparable results to native performance while KVM introduced relevant overhead. The disk benchmarks showed, once more, that native and container-based performances are similar, and KVM was significantly worse. In terms of memory benchmarks, although the difference between the results were small, Docker still outperformed KVM in every situation when compared to the native execution. Finally, the network benchmarks revealed that virtualized platforms were not as responsive as the native one, where Docker introduced a considerable overhead and KVM introduced an even higher one, concluding that while the hypervisor-based solution displayed a relevant overhead, the results of Docker were promising.

Both these articles test NFV scenarios in SBC devices, which, are of interest for this dissertation, as it compares the implementation of container-based virtualization and hypervisor-based virtualization. The implementation chosen on this dissertation was the hypervisor-based resorting to KVM and libvirt, which are the used by the software chosen.

2.2.2 Single-Board Computers as testbeds for SDN/NFV

Over the last few years researchers have utilized the low cost advantage of SBC to perform tests of technologies like SDN and NFV.

For example, in [36] the authors implement a realtime testbed for Software Defined Wireless Networks (SDWN) by using RPi as OpenFlow Switches. In order to provide an inexpensive and easy to implement testbed, OpenFlow switches are obtained by using RPi with WiFi dongles, where the software stack is based on OpenWRT[10] and OvS[11] is installed. A SDN controller is needed, in which OpenDaylight was used, furthermore, a traffic aware routing algorithm is implemented as a northbound service application in order to test and configure the tested environment.

Another one can be found in [8], where a SDN testbed with OvS based on RPi is implemented. In order to make a suitable testbed for evaluating small-scale SDN, the authors suggest a simple and cost-effective testbed using RPis. The testbed includes three network devices, the SDN controller, the SDN switch, and the host device, as shown in figure 2.10. For the SDN controller Floodlight is used while OvS is used on the switch device to create virtual interfaces, since the 1 Gbps Ethernet interface is not sufficient to process multiple connections individually. Afterwards, the maximum throughput is evaluated for different Maximum Segment Sizes (MSSs) where the testbed compared to the 1 Gbps net-FPGA showed similar results.

[10]`https://openwrt.org`
[11]`https://www.openvswitch.org`

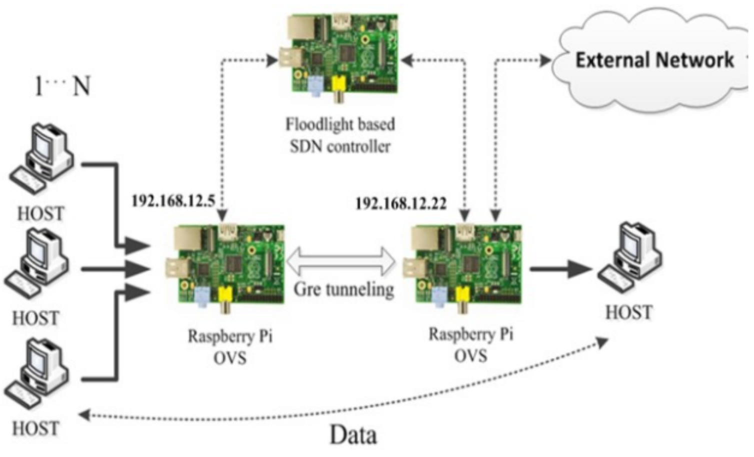

Figure 2.10: Network design of the Raspberry Pi based SDN testbed [8].

Similarly, in [9], the authors developed an OpenFlow testbed named Pi Stack Switch, where the Pi Stack Switch consists of four RPis installed with OvS. The reason these devices are tied is to expand the ethernet ports. The network architecture can be found in figure 2.11. To evaluate this testbed the topology change latency is measured, which is calculated by the sum of the link-up latency and link-down latency. The results showed that the network administrator recognized topology changes in less than one second.

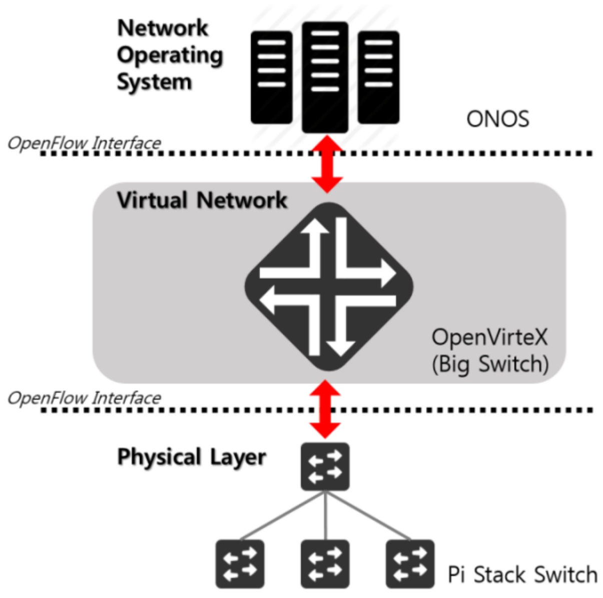

Figure 2.11: Network architecture of Pi Stack Switch [9].

Finally, in [37] and [38], RPis are used to create to build a cluster, which then is used in a testbed for Cloud Computing research.

All these testbeds showcase the ability to use SDN technology on SBCs, which, there is the possibility to implement in this dissertation with the software chosen offering various SDN controllers to implement.

2.2.3 Experiments on edge scenarios

Kempen, Alexandre et al., presented in [10] the design and implementation of MEC-ConPaaS, a mobile-edge Cloud platform, which aimed to support future research on edge Cloud applications. With the objective of creating an open-source mobile edge Cloud implementation which can be deployed over a campus or a city center to support real-word implementations, the authors claim that an easier way to deploy an experimental MEC testbed is to rely on SBC such as RPis and similar devices [10].

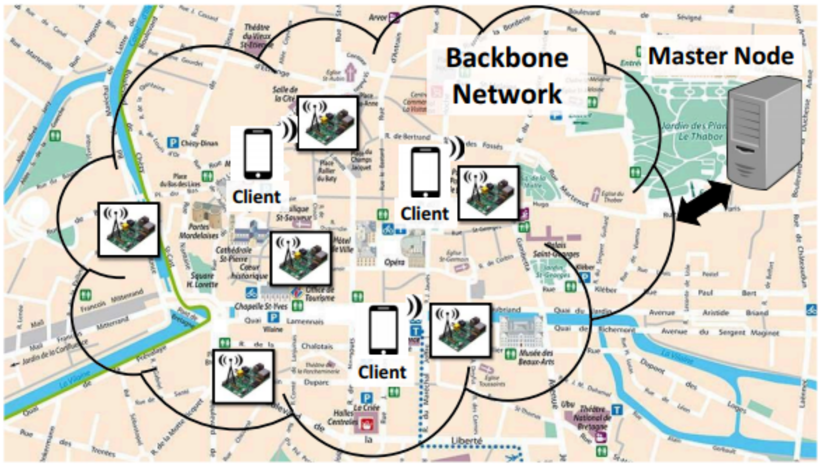

Figure 2.12: Geographical distribution of a MEC-ConPaaS deployment [10].

MEC-ConPaaS is a mobile edge Cloud platform designed to be physically deployed across a city or campus-sized geographical area, as shown in figure 2.12, and to execute single-user and multi-user edge applications where proximity between the Cloud Computing instances and their end users is important. Any system node acts both as an access point and a Cloud Computing server: every node has a Wi-Fi interface to which client nodes can connect, while, at the same time, a virtualized infrastructure run by the sames nodes can deploy applications in immediate proximity to the end users. A traditional server controls the system while acting as the entry point of the system.

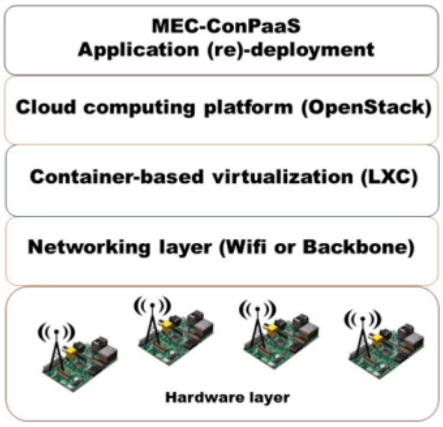
MEC-ConPaaS
Application (re)-deployment
Cloud computing platform (OpenStack)
Container-based virtualization (LXC)
Networking layer (Wifi or Backbone)
Hardware layer

Figure 2.13: System architecture of MEC-ConPaaS [10].

The system's architecture, represented in figure 2.13, features multiple layers. The hardware layer is composed of SBCs such as Raspberry Pis, these devices are equipped with wifi interfaces to directly communicate with users at the edge. A wired backbone network, or the leverage of the RPi's wifi interfaces ensures the interconnection between the devices. To provide elasticity and fault tolerance, applications are deployed inside a virtualized environment (Linux Containers (LXC)) hosted on the same single-board devices. These containers are orchestrated with OpenStack, while the ConPaaS layer eases the deployment of actual elastic applications.

Afterwards, the authors test the hardware to show that it is suitable enough to support Cloud Computing applications, describe the middleware layer and compare performances between the raspberries and a HP 820 G2 laptop concluding that the very fast development of other brands of single-board devices suggest that future generations of single-board computers will deliver better performance, while retaining similar cost, volume, and power consumption as the devices available nowadays [10]. This edge scenario differs from the scenario in this dissertation, since this is a MEC scenario, while this dissertation is aimed towards Fog Computing scenarios, but there are still similarities in the use of SBCs and NFV infrastructures.

In [39], the authors present a solution for slicing WLAN infrastructures, aiming to provide differentiated services on top of the same substrate through customized, isolated and independent digital building blocks. The authors then present a proof of concept realized over a real testbed and evaluate its feasibility, where they extend the capacity of a standard WLAN Access Point (AP) commonly used by Internet Service Providers (ISPs) (TP-Link) by attaching it to a SBC (Raspberry Pi). For the proof of concept evaluation, a network function responsible for enforcing bandwidth limits for two slices is implemented, one configured with 5 Mbps while the other was configured with 45 Mbps. Two virtual APs were set at the TP-Link to allow each client to connect to a different network, each one serving a different slice, while all the traffic arriving at the APs wireless interface was redirected to the VNF at the SBC where it was shaped before reaching the AP's WAN port. Afterwards, a performance evaluation was executed in order to perceive how the resource-constrained Fog node performed in the face of an increasing number of instantiated VNFs and a straightforward performance evaluation was also done in order to have a comparison between the Fog system and the Cloud. Finally, results showed that in the proof of concept evaluation the throughput values stayed within the limits of each slice. In the performance evaluation, results showed that the processing element can handle the same data rate for up to six chained VNFs, while the average delay of the Cloud was higher than the average delay of the Fog system. Being implemented on a Fog Computing scenario, this research utilizes slices to offer network services, where in this dissertation slices are not used, only possible VNFs can be provided.

The authors in [40] present a Fog-oriented framework for IoT applications based on the Kura framework and Docker-based containerization over resource-constrained RPi devices. The authors use Kura, an open-source framework for IoT applications, which provides a platform for building IoT gateways. After detecting some relevant weakness in the Kura architecture for Fog Computing exploitation, the authors add relevant extensions to nullify these weakness. These extensions consist on adding a Message Queuing Telemetry Transport (MQTT) broker on each gateway to collect sensed information at the gateway side, and, enabling cluster/mesh topologies for Kura gateways. In order to create Fog nodes on-the-fly, the authors resort to containerization-based virtualization by using Docker to create container-based applications/services. Afterwards, the authors tested a primary implementation to quantitatively analyze the scalability of the approach and its overhead on different filesystems available to Docker, which according to the authors, the performance was dependent on. The filesystems tested were AUFS, Device-mapper, and, OverlayFS, and the measurements made on the first test were the overhead on container start, I/O operations, CPU operations, and, total execution and on the second test, the execution time over multiple containers. The results show, for test one, that AUFS and OverlayFS have comparable results while Device-mapper had the worst performance, for test 2, AUFS and OverlayFS, again, outperform Device-mapper showing a linear dependency of execution time on the number of concurrently running containers. Finally, the authors conclude that the results for the scalability of multiple container execution on resource-constrained nodes are encouraging for further activities in the field and talk about some future work on this project. This work, whilst differing on the scenario, more directed towards IoT applications and tools used, Docker-based over hypervisor-based, showcases a scenario in Fog Computing, paradigm in which this dissertation is inserted.

In [6], the authors study the evolution of the edge computing by focusing on the component managing the resources, in other words, the VIM. By measuring the time overhead created when requests for VM provisioning and deprovisioning were sent and completed, the authors compare two different solutions, OpenStack and OpenVIM. This time overhead is measured using CloudBench and the test environment were implemented on a single host with the following configuration: Intel(R) Xeon(R) CPU E5-2623 v4 @ 2.60GHz with 32GB memory and 4TB storage and Ubuntu 16.04.1 Server operating system with KVM-enabled 4.4.0-31-generic Linux kernel[6]. After performing the tests, OpenVIM shows a better performance in all tests when compared to OpenStack, giving it advantage to resource-constrained enviroments, however, the authors conclude that OpenVIM lacks support and development from the community, as well as, lacking some important functionalities (e.g., authentication and security) and, with that, the authors eventually preferred OpenStack as the better choice. This work is similar to the work done in this dissertation, since NFV services are implemented using the same solution (OpenVIM), except in this work two different solutions doing the same implementation are compared and the hardware used for the virtualization services is a powerful computer and not a resource-constrained device, while in this dissertation the virtualization services are instantiated on resource-constrained devices, therefore, the comparison is done between hardware, a SBC and a laptop.

2.3 Chapter Overview

This chapter explained some important concepts discussed over this dissertation, these concepts being the technologies that galvanized the field (SDN and NFV), as well as, some of the paradigms that were born due to these technologies (Cloud computing, Edge computing, Fog computing, and, MEC), some platforms that implement these technologies are also addressed (SDN controllers and NFV solutions). Afterwards, a look into some tools that have been developed is taken, where it can be seen that resource-constrained devices have been used in a wide range of manners, where in some applications a small-cost device is required just to prove some sort of concept in which the device itself is not important (e.g., some testbeds) and others, where the object of the study is the device itself by seeing how can a concept be implemented (e.g., edge scenarios) in these sort of devices or how well theses devices fare against more powerful ones (e.g., tests on SBCs).

Chapter 3

Framework and Implementation

In this chapter, a framework is proposed to tackle the problem addressed by this dissertation. Furthermore, the actual implementation of the framework is presented and what compromises were made to accommodate the implementation.

3.1 Framework

3.1.1 Problem Statement

Currently, technologies such as SDN and NFV have been focusing more towards the infrastructure and the datacenter scenes. However, with paradigms such as Fog Computing growing in popularity, programs that realize functions such as, for example, a VIM, are not available or easy to implement in environments where low-power computing devices operate. Since such software was designed to accommodate the general use of what the demand of the community was, they ended up being too powerful to run in scenarios where low-power computing devices exist.

3.1.2 Solution

This dissertation aims to modify a lightweight VIM to be capable of running in scenarios where resource-constrained devices operate and then compare its performance to a more powerful implementation. This lightweight VIM will be able to be used in scenarios incorporated in Fog Computing, IoT, and others. An use case, at the moment, would be, for example, in a private Local Area Network (LAN) a personal laptop or computer running this lightweight VIM and instantiating VNFs in a single or multiple SBCs such as RPis to perform a needed function. This use case should be extendable to scenarios where the instantiation of one or more VNFs on one or more compute nodes, equipped with an ARM processor, occurs.

Having a VIM that was both lightweight and capable of running on ARM processors, as demanded by RPis, was not available, so an open-source lightweight VIM was modified to fit these two requirements. As shown in section 2.1.2.2, OpenVIM is a lightweight implementation of an NFV VIM. Utilizing its open source feature, the source code could now be modified so VMs could be launched in ARM processors, typical of resource-constrained devices.

A RPi serves as a compute node to the VIM and, as such, VMs can be instantiated on them. As mentioned in section 2.1.2.2, OpenVIM provides two options for overlay management network management: "bridge" and "ovs". The first was chosen, which, then led to the need of an external DHCP server for the management network, therefore, Internet Systems Consortium (ISC)'s DHCP[1] was used.

Finally, NAT was configured on instantiated VMs so that the VM could communicate to the Internet. An overview of the architecture of the solution is displayed in figure 3.1, there it can be seen the laptop running the modified OpenVIM and possible compute nodes connected to it.

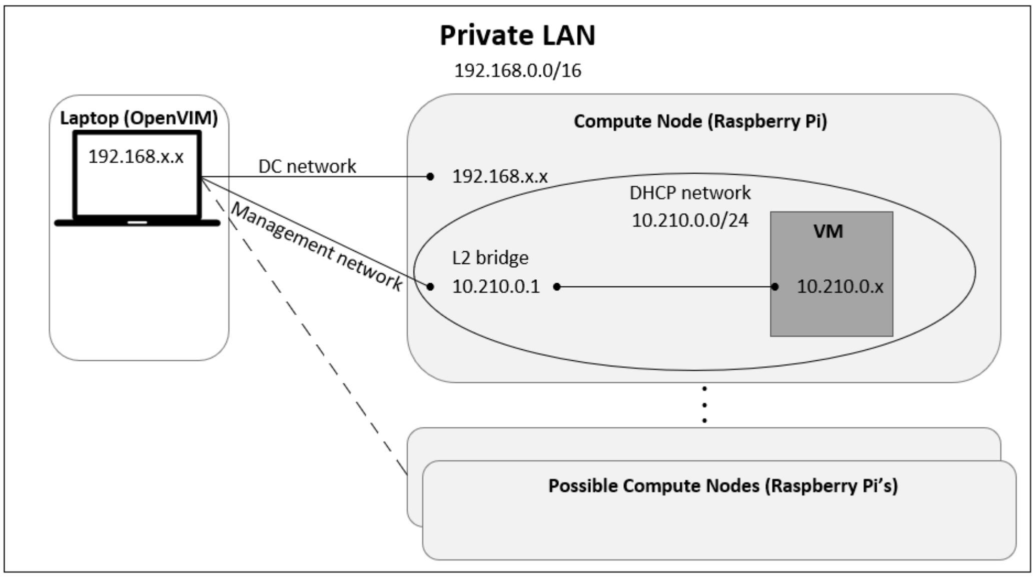

Figure 3.1: Overview of the solution proposed.

3.2 Implementation

In order to implement the solution above, 5 main steps were needed to be completed in order to launch a VM through OpenVIM on a compute node (RPi).

1. OpenVIM needs to be modified to create VMs in an ARM architecture;
2. Precreate a bridge with L2 connectivity in the compute node (RPi);
3. Create an external DHCP server to provide IP addresses;
4. Create description files for OpenVIM to instantiate a VM;
5. Configure NAT.

During these implementations, the configuration file of OpenVIM (`openvimd.cfg`) needs to be updated to avoid errors during the utilization of the program. Some of these updates correspond to: the DHCP implementation, since it is an external server, OpenVIM needs to know its parameters, and some others such as the host image path, or the path to SSH's keyfiles, or the bridge interfaces.

[1]`https://www.isc.org/downloads/dhcp/`

With all the previous steps completed and a compute node added to OpenVIM beforehand through the command line of the laptop, a VM can finally be instantiated successfully. OpenVIM creates a SSH channel between the laptop and the RPi and, remotely, instantiates a VM on the device.

3.2.1 OpenVIM with ARM processors

As stated before, the chosen tool to use as a VIM was OpenVIM, however when it is installed, OpenVIM does not have the ability to instantiate a VM on an ARM processor, more specifically on an "aarch64" architecture which is the architecture present on a RPi, only on AMD/Intel processors or "x86_64" architectures. Therefore, the first step is to change the source code so that VMs can be instantiated in this architecture. Afterwards, also related to OpenVIM, some configuration files have to be created to actually instantiate a VM, but this will be covered in section 3.2.4.

It is important to point out that, at a first glance, such changes may seem trivial. However, until these changes were effectively done and addressed, a thorough and careful analysis of the whole system of OpenVIM was done. Particularly considering the poor level of the development documentation, several trial and errors assessments were made, until finally the system was thoroughly comprehended.

3.2.1.1 Source Code

Before looking into the source files of OpenVIM, the expected change needed in the source code was where the VM were instantiated, since the code was designed for "x86_64" and not "aarch64", and after determining the part where the XML code for VM instantiation was generated, simply changing the architecture was required: `<type arch='aarch64' machine='virt-2.9'>hvm</type>`, where arch was previously `x86\_64` and `<emulator>/usr/bin/qemu-system-aarch64</emulator>`, which previously was `qemu-system-x86\_64`.

Simply changing these two lines made OpenVIM capable of launching VMs on a RPi. However, when instantiating there were errors from the rest of the XML code due to different architectures, more changes were made to instantiate a VM without errors. An example of the generated XML code after the changes for a VM instantiation is shown on appendix A1.

These changes were:

- **the code where the memory for each VM is calculated.** Since OpenVIM was designed for more powerful devices, the minimum memory designated was 1 GB, however this value is higher than what the RPi can provide, therefore, it was lowered so that it declared values lower (e.g., 524288 KB or approximately 524 MB);

- **the removal of hugepages.** OpenVIM was designed to run with 1 GB hugepages to optimize memory, but, again, since 1 GB is too much for a RPi, this part of the code was removed, despite slightly slowing the performance.

- **the OS type** was changed to `<type arch='aarch64' machine='virt-2.9'>hvm</type>`, the arch was mentioned above, the machine was also change to `virt-2.9` since the previous value was unsupported by the architecture/RPi.

- **the option for a loader and nvram were added** (`<loader readonly='yes' type='pflash'>/usr/share/qemu/aavmf-aarch64-code.bin</loader>` and `<nvram>/var/lib/libvirt/qemu/nvram/debian_VARS.fd</nvram>`), the first to support an UEFI image, the other to store variables for the UEFI firmware.
- **the CPU mode** to `<cpu mode='host-passthrough'>`. . . , since the previous mode was not supported by the hypervisor for a domain on aarch64.
- **the model type for the video device** was changed to `<model type='virtio'/>` due to not being supported as well.
- and **the ROM file** was specified to none (`<rom file=''/>`), since there was no definition before, and it lead to an error during boot due to the ROM file not being found.

The rest of the XML is the same as the original code generates. The rest of the unchanged options on the XML are described briefly below:

- the `VCPU` option defines the number of virtual CPUs allocated to the VM, and the `cputune` option tunes this allocation. This is generated by the code according to the description file of the VM to be created;
- the `NUMA` option is to provide details for the performance of a NUMA host. This is generated by the code according to the description file of the VM to be created;
- the `features` are options that hypervisors accept. The features added correspond to physical address extension (`pae`), power management (`acpi`), and, usage of programmable IRQ management (`apic`);
- the `clock` option is to define the VM time, the option 'utc' synchronizes to UTC;
- the `poweroff`, `restart`, and, `crash` are options to specify what the VM should do when these events happen;
- the rest of the options are to define devices provided to the VM:
  - the `serial` and `console`, are used to define a serial and console communication between the host and the VM;
  - the `disk` is used to declare the disk configuration, in this case, is the image chosen, therefore, these values come from the description file for the image provided to the VM;
  - the `interface` is used to declare the network configuration, in this case, the network interface (bridge) used to host the VM, the configuration comes from the description file of the network provided to the VM.

3.2.2 Bridge

As stated before, a precreated bridge at the computed node had to be created beforehand. The hardware used during this implementation was a RPi installed with openSUSE (table 4.1), therefore, openSUSE offers two methods to create a bridge: through the command line terminal, or through a Graphical User Interface (GUI) supplied by YaST. Despite which method is utilized the important part is that the bridge ends up correctly set up. First, a VLAN of the ethernet interface was created, and then, the bridge itself was created. This resulted in two configuration files called `ifcfg-vlan9` and `ifcfg-br0`, respectively, these configurations files are shown below:

ifcfg-vlan9:

```
BOOTPROTO='none'
BROADCAST=''
ETHERDEVICE='eth0'
ETHTOOL_OPTIONS=''
IPADDR=''
MTU=''
NAME=''
NETMASK=''
NETWORK=''
PREFIXLEN=''
REMOTE_IPADDR=''
STARTMODE='auto'
VLAN_ID='9'
```

The only important parameters are the ones filled, in this case, the VLAN number (`VLAN_ID`), the interface of the VLAN (`etherdevice`), the startmode, so that the VLAN starts automatically, and no protocol is used to give an address to the vlan due to being connected to a bridge (`BOOTPROTO='none'`).

ifcfg-br0:

```
BOOTPROTO='static'
BRIDGE='yes'
BRIDGE_FORWARDDELAY='0'
BRIDGE_PORTS='vlan9'
BRIDGE_STP='off'
BROADCAST=''
ETHTOOL_OPTIONS=''
IPADDR='10.210.0.1/24'
MTU=''
NAME=''
NETWORK=''
GATEWAY='10.210.0.1'
DNS1='193.136.92.73'
DNS2='193.136.92.74'
REMOTE_IPADDR=''
```

```
STARTMODE='auto'
```

Once again, the important parameters are the ones filled.

- the `BOOTPROTO` is now static, because the IP address is given statically;
- the `bridge`, `forward_delay`, `ports`, and, `stp` declare whether this interface is a bridge and its delay, ports, and whether the Spanning Tree Protocol (STP) is activated or not;
- the `ipaddr` specifies the IP address and the mask for the bridge interface network;
- the `gateway` specifies the gateway of this network;
- the `dns` specifies the DNS, this is an optional parameter, but it was used since the VM was having Internet connection problems;
- the `startmode` is once again set to auto to start automatically.

3.2.3 DHCP

The bridge configuration led to the creation of an external DHCP server, the service used ended up being ISC's DHCP. To configure a DHCP server, the dhcpd.conf file needs to be edited to declare the configurations of said server. The parameters should be consistent with the previous configurations. An example of a configuration can be found below:

```
option broadcast-address 10.210.0.255;
option subnet-mask 255.255.255.0;
option domain-name-servers 193.136.92.73, 193.136.92.74;
option routers 10.210.0.1;

subnet 10.210.0.0 netmask 255.255.255.0 {
  range 10.210.0.4 10.210.0.254;
}
```

The parameters, respectively, represent the IP address used for broadcast, the mask of the network, the DNS addresses, the gateway, the network and mask, and, the range of IPs available for the DHCP server to give. Afterwards, a script is created on the compute node (RPi) on a directory accessible from PATH, so that, OpenVIM can retrieve the IP address leased from the DHCP server.

```
#!/bin/bash
awk '
 ($1=="lease" && $3=="{"){ lease=$2; active="no"; found="no" }
 ($1=="binding" && $2=="state" && $3=="active;"){ active="yes" }
 ($1=="hardware" && $2=="ethernet" && $3==tolower("'$1';")){ found="yes" }
 ($1=="client-hostname"){ name=$2 }
 ($1=="}"){ if (active=="yes" && found=="yes"){ target_lease=lease; target_name=
name}}
 END{printf("%s", target_lease)} #print target_name
' /var/lib/dhcp/db/dhcpd.leases
```

3.2.4 Description Files

OpenVIM accepts two languages for this kind of files, JSON and YAML, and both languages can define the same parameters.

3.2.4.1 Compute Node

This file will describe the RPi that will be added to OpenVIM as a compute node to have VMs instantiated on it. An example of a JSON file can be found on Appendix A2.

The parameters are:

- `ip_name`, which is the IP address of the compute node;
- `user`, which is the username to which OpenVIM will SSH into to create the VM;
- `numa` description, that has the number of cores, interfaces (e.g., data plane interfaces) and memory available to implement VMs, in other words, the resources available in the host;
- `ranking`, which symbolizes how powerful the host's processor is, in this case it was given the lowest value of 100;
- `name`, which is the name of host, that can be used instead of its ID given by OpenVIM upon being added to the database.

3.2.4.2 Flavour

A flavour defines the compute, memory, network, and storage capacity of an instance. An example of a YAML file can be found on Appendix A3. It is noteworthy, that, some of the descriptions on this file can be overwritten with other files later implemented.

3.2.4.3 Image

The image is the file which the VM will be run in. OpenVIM provides two methods to deliver an image to a VM, either the image is provided locally on a directory on the RPi or an URL link to the image is provided and the image is then downloaded during the VM instantiation. Two examples for each method can be found on Appendices A4 and A5, respectively. The implementation is similar between both, where the variable `"path"` either receives an URL or a directory path to an image file. The metadata provided on this file is optional, meaning its not necessary for the image to be added to the OpenVIM database.

3.2.4.4 Network

The network also needs to be described, this file was created after the Bridge and DHCP configuration, so that all parameters could be filled correctly. As stated before the method for overlay management network chosen was the "bridge" method, which differs from the "ovs" method. A sample code for a network YAML file can be found on appendix A6. In this file:

- `name` defines the name of the network, which can be used instead of the ID of the network given after being added to the database;

- `type` defines the type of network, there are four options: two for data plane networks, and two for control plane networks;
- `shared` defines if the network is external or not, in other words, if there is more than one tenant, whether all of them can use the network or not;
- `cidr` defines the IP of the network;
- `enable_dhcp` defines whether DHCP is needed or not;
- `dhcp first and last ip` defines the range of IPs of the DHCP server;
- `provider` defines which interface is providing this network, in this case, a bridge named br0;
- `dns` defines the DNS addresses for the VMs.
- `routes` defines the default route for the instantiated VMs.

3.2.4.5 Virtual Machine/Server

Finally, the file description for implementation of a VM or, as OpenVIM names it, server is created. Once again, an example can be found on Appendix A7. In this file:

- `hostId` defines the host where the VM will be instantiated, although, this is an optional parameter, since OpenVIM automatically searches available hosts to implement the VM that can accommodate the VM requirements, nevertheless a declaration overwrites this search.
- `name` defines the name of the VM, which can be used instead of the ID given by OpenVIM after being added to the database;
- `imageRef` defines the image to be used for the VM;
- `flavorRef` defines the flavour to be used for the VM;
- `start` defines whether the VM should be started upon being added to OpenVIM or not, other options include 'no', where only the resources for the VM are reserved, and 'pause', where the resources for the VM are reserved and the VM is launched on pause mode;
- `extended` defines parameters for the VM like the number of cores allocated, the interfaces, and the memory. Some of these parameters can overwrite the definitions declared previously on the flavour description;
- `networks` defines the networks where the VM is instantiated.

3.2.5 NAT

To enable NAT for VMs to be able to connect to the Internet, IPv4 forwarding needs to be enabled and then, the bridge and its interfaces need to be added to the firewall so that these interfaces can be masqueraded by the firewall and traffic prerouted and forwarded to them. In openSUSE, these changes can be made in the YaST's GUI or directly in the firewall configuration file located in `/etc/sysconfig/SuSEfirewall2` and IPv4 forwarding can be enabled by typing in the command line terminal: `sysctl -w net.ipv4.ip_forward=1` or `echo 1 > /proc/sys/net/ipv4/ip_forward`.

3.3 Chapter Overview

In this chapter, the problem that this dissertation attempted to solve is stated, as well as, a solution to said problem, which corresponds to the lack of programs to implement functions needed in Fog Computing environments like a VIM. Therefore, this work attempts to modify an already lightweight VIM, OpenVIM, to function in resource-constrained environments, which are characterized by the usage of SBCs like RPis which use ARM processors.

In the second part of the chapter, since OpenVIM was not designed to run in this architecture, changes to the source code made to accommodate the new architecture and to implement VMs in these scenarios are explained. Afterwards, the various implementations made are described, in order for OpenVIM to instantiate a VM on a compute node (RPi). This second part results from a thorough assessment of the underlying operation mechanisms from OpenVIM which, due to poor documentation, required deep analysis of the source code for its comprehension.

Chapter 4

Results and Analysis

In order to have an idea of how a VM performs in a low-resources environment, some tests were realized to get some quantifiable results. The tests realized were CPU benchmarks on a RPi, along with the time needed for a instantiated VM to perform a ping to another computer on the network and the Internet. Finally, the power consumption of a RPi was measured during a VM instantiation. All these tests were performed on a laptop to serve as reference for comparison with the RPi.

4.1 Performance Benchmarks

The CPU performance benchmarks were realized on a laptop computer for reference against a Raspberry Pi 3 Model B using the Phoronix Test Suite[1] software. The specifications of the Devices under tests (DUTs) can be found in table 4.1. The chosen tests were two versions of the SciMark2, the ANSI C version and the Java version, which are benchmarks for scientific and numerical computing developed by programmers at the National Institute of Standards and Technology (NIST), comprised of[41]:

- **Fast Fourier Transform (FFT)** where a 1D forward transform of complex numbers is performed.
- **Jacobi Successive Over-relaxation** where an algorithm assigns each cell with the average weight of its four nearest neighbours.
- **Monte Carlo** where the value of Pi is estimated by approximating the area of a circle.
- **Sparse Matrix Multiply** computes a matrix-vector multiply with a sparse matrix held in compress row format.
- **dense LU matrix factorization** calculates the LU factorization using partial pivoting.
- **Composite** where more than one of the previous tests is run.

The Phoronix Test Suite software offers the option of comparing benchmarks performances against other people's benchmarks that are posted on their website[2], however, this feature will

[1]`https://www.phoronix-test-suite.com`
[2]`https://openbenchmarking.org`

not be utilized since both the reference device and the device under scope of this test are available to be tested directly.

	Dell Inspiron 3251	**Raspberry Pi 3 Model B**	**Raspberry Pi 3 Model B VM**
Machine Type	Laptop	Single-board	Single-board
Operating System	Ubuntu 16.04	openSUSE 42.3	Debian 9.5
Kernel	4.15.0-36-generic (x86_64)	4.4.155-68-default (aarch64)	4.9.0-7-arm64 (aarch64)
CPU	Intel Core i5-3337U @ 1.80 GHz (2 Cores)	Quad Core 1.2 GHz Broadcom BCM2837 64bit	VM
Memory	2 GB	1 GB	512 MB
Network	1 Gbps	100 Mbps	100 Mbps
Storage	150 GB HDD Drive	16 GB microSD card	2 GB QEMU HDD

Table 4.1: Hardware specifications of the DUTs.

The two tests were ran several times in order to obtain 10 samples and calculate the mean and the standard deviation. The benchmarks were ran, firstly, on a laptop as referred above, then on a RPi and finally, on a VM instantiated on the RPi to first see the differences between a normal laptop and a SBC, in this case a RPi, and then to see the differences between the performance of a normal RPi and its VM. The results are displayed on tables 4.2 and 4.3 and then on bar graphics on figures 4.1 and 4.2. The results are given in Floating Point Operations per Second (flops) and despite the software in some cases giving a higher number of decimal digits all the results were rounded up to 2 decimal digits.

	PC - Ubuntu	**RPi - openSUSE**	**RPi - Debian (VM)**
Composite	368.90 ± 7.85 Mflops	23.67 ± 2.98 Mflops	24.98 ± 2.56 Mflops
Monte Carlo	86.29 ± 1.19 Mflops	10.60 ± 0.64 Mflops	11.83 ± 1.64 Mflops
Fast Fourier Transform	94.75 ± 3.21 Mflops	10.16 ± 1.06 Mflops	9.37 ± 1.03 Mflops
Sparse Matrix Multiply	390.00 ± 13.00 Mflops	21.65 ± 3.79 Mflops	21.33 ± 4.00 Mflops
Dense LU Matrix Factorization	515.65 ± 13.38 Mflops	32.80 ± 5.14 Mflops	31.26 ± 3.28 Mflops
Jacobi Successive Over-Relaxation	757.07 ± 12.58 Mflops	44.49 ± 7.96 Mflops	50.70 ± 7.68 Mflops

Table 4.2: C-SciMark2 benchmarks results for the different devices.

	PC - Ubuntu	**RPi - openSUSE**	**RPi - Debian (VM)**
Composite	1342.71 ± 6.42 Mflops	84.40 ± 6.92 Mflops	99.42 ± 5.17 Mflops
Monte Carlo	619.90 ± 2.60 Mflops	47.30 ± 5.76 Mflops	61.25 ± 15.07 Mflops
Fast Fourier Transform	814.33 ± 24.99 Mflops	58.85 ± 19.86 Mflops	66.45 ± 15.82 Mflops
Sparse Matrix Multiply	1238.44 ± 29.18 Mflops	70.40 ± 14.93 Mflops	81.42 ± 21.45 Mflops
Dense LU Matrix Factorization	3022.64 ± 15.62 Mflops	83.70 ± 9.70 Mflops	157.42 ± 23.40 Mflops
Jacobi Successive Over-Relaxation	1018.21 ± 1.77 Mflops	156.41 ± 20.66 Mflops	165.97 ± 44.92 Mflops

Table 4.3: Java-Scimark2 benchmarks results for the different devices.

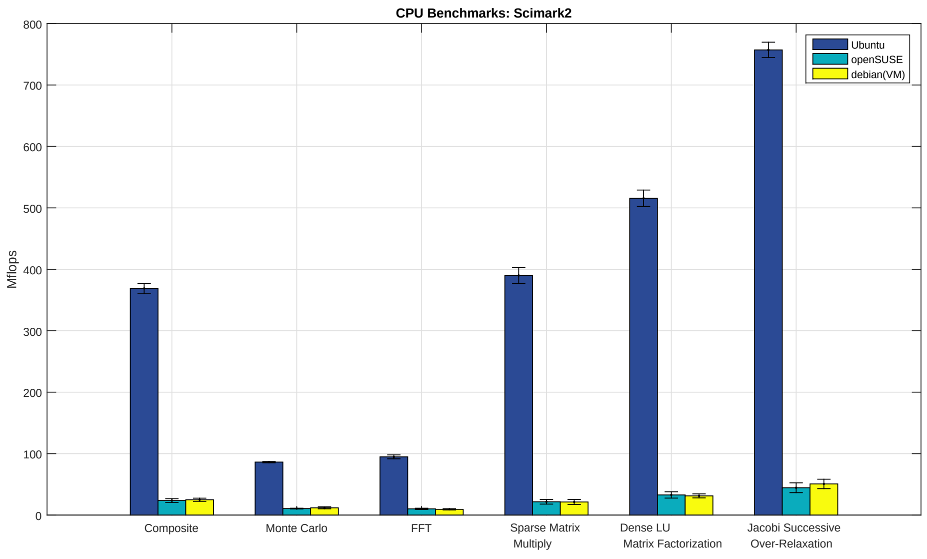

Figure 4.1: CPU Scimark2 benchmarks.

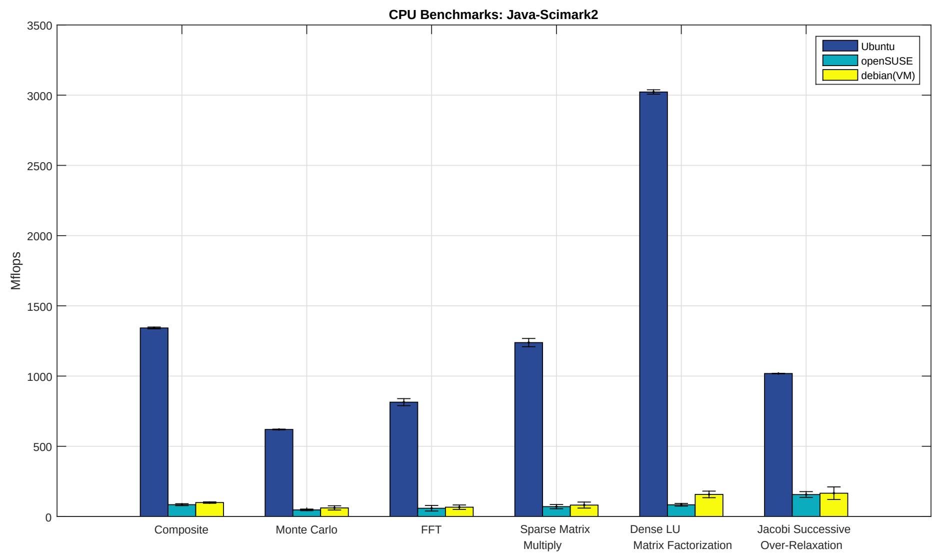

Figure 4.2: CPU Java-Scimark2 benchmarks.

The results above show, as expected, that the RPi is still far away from the results of a regular laptop available on the market. With performances varying between 6 and 18 times higher, every test was dominated by the computer, as the bar graphics in figure 4.1 and 4.2 show. Comparing the results of table 4.2 and table 4.3, for the ANSI C benchmarks, the performance of the RPi with and without a VM is fairly similar where the values obtained for both cases are almost identical. However, the Java benchmarks performed higher than the ANSI C benchmarks, therefore, at least in these tests, the Java language is able to obtain higher flops than the ANSI C language. These tables also indicate that the standard deviation is higher for the laptop tests, however, it can be explained by the fact that since the mean values are higher, the deviation is higher as well, since the laptop has more stable results than the RPi (lower percentage standard deviation). Although it is not visible on table 4.2, table 4.3 shows better performance for the runs on the VM when compared to the runs on the RPi. Since the hardware is the same on both cases, a possible explanation can be the different OSs in which the tests were performed, leading to believe that, at least for the Java benchmarks, the Debian OS leads to higher flops compared to the openSUSE OS. The usage of different OSs occurs because compatible images with OpenVIM of the OS installed on the RPi were not readily available to be used to instantiate the VM.

Finally, it should be said that despite having lower results, devices such as the RPi can still be used in several scenarios, such as scenarios where there is no need for high processing power. That is because the RPi is still highly cost-effective considering its performance and, despite the difference in Mflops, depending on the computer, its price can be at least ten times lower.

4.2 Instantiation Time

With the objective of knowing how long it took for a VM to be ready after instantiation, a small test divided in two phases was devised to obtain this value.

The first phase of the test is composed of a Transmission Control Protocol (TCP) socket opened between a server and a client (the RPi) exchanging commands between them as depicted in figure 4.3, starting by the command for instantiation of the VM and finishing by a ping from the VM to the server and to the internet. Meanwhile, in between the first and the last command the client is in a loop trying to open a SSH channel to the newly VM and when it is possible to do so, the loop finishes and the next command is to send a packet from the VM to the server. The second phase is the same as the first, except now, two VMs are instantiated instead of one, allowing observation of how scalability affects the DUTs in terms of booting time since two VMs are being instantiated simultaneously, depicted in figure 4.4.

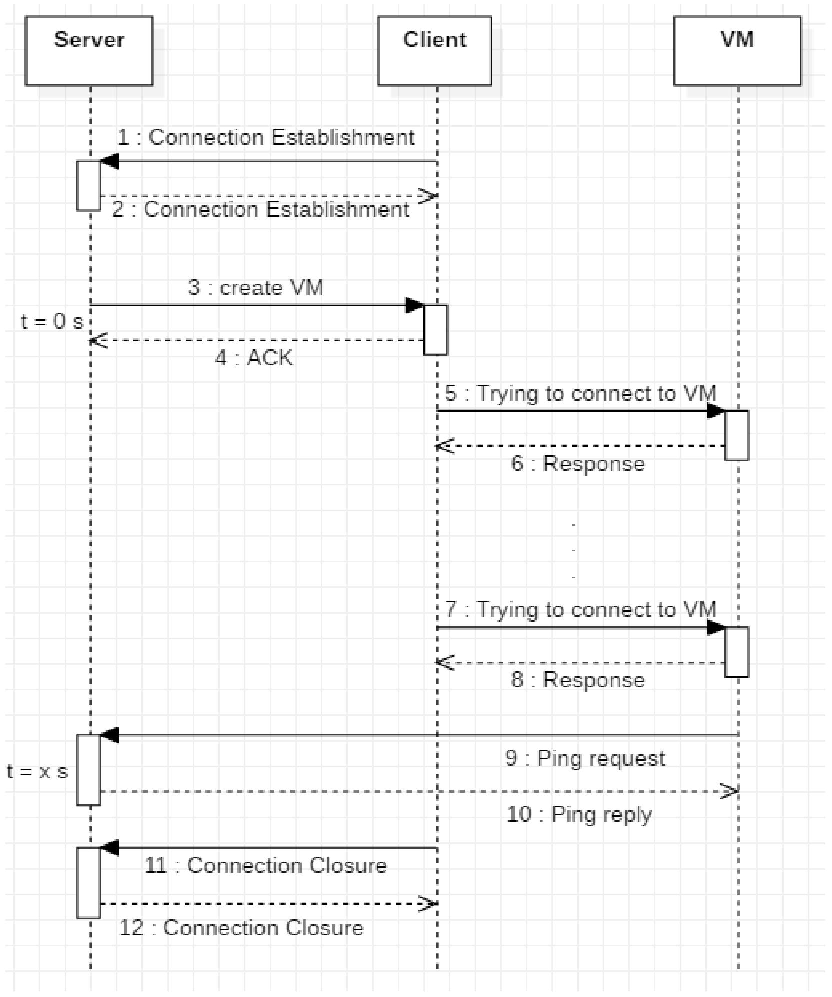

Figure 4.3: Sequence diagram of the messages exchanged during the first phase of the test.

As stated previously, figure 4.3 depicts an abbreviated diagram comprised of the sequence of messages exchanged between the server and client. The first two messages labeled as "Connection

Establishment" are an abbreviation of the three-way handshake performed when a TCP connection is established in order to synchronize the client and the server. The next pair of messages (3 and 4) is the server sending the command to the client to create the VM and the respective response (ACK), this marks the initial point of the instantiation time. Following this, a loop starts trying to open a SSH channel between the client and the newly created VM and breaks when a connection is finally established represented in messages 5 to 8. Afterwards, the following pair of messages (9 and 10) refer to the ping requested from the VM to the server, marking the end point of the instantiation time, and the response from the server. Finally, the messages 11 and 12 are an abbreviation of the termination of the TCP session.

While this packet exchange is happening, a network packet analyzer, Wireshark[3], is running on the background of the server catching the packets that arrive and leave the server. Afterwards, it is possible to review the packet exchange and measure the time period between the instantiation command and the first ping to the server to obtain an estimated value for the time needed for a virtual machine to be ready. In this evaluation, 15 exchanges were realized and then a mean value was obtained as well as a standard deviation value and, although Wireshark has 9 decimal digits in its time column, the results shown are rounded up to only 4 decimal digits. By using the filter feature of Wireshark with the filter "tcp.port == (port number) || icmp", it is possible to filter the unwanted packets showing only the relevant packets, since the port filtered only has the socket traffic catching all the TCP exchange while "icmp" catches the ping packets.

Computing the values obtained in the first phase, it is possible to compute the mean value and the standard deviation obtaining a mean value of 82,4484 seconds and a standard deviation of 5,2060 seconds. Analyzing these results, the time that a VM needs to perform a command is rather long, however, it should be noted that most of this time is due to the boot of the OS (Debian) which is a rather heavy OS and lighter operating systems would reduce this time. To establish a reference, a similar method to the one explained before was used on a laptop to obtain samples and calculate the same values for comparison. For the laptop, a mean value of 92,4290 seconds was obtained, as well as a standard deviation of 1,5047 seconds.

Comparing instantiation times, the RPi shows a faster booting time, by approximately 10 seconds, when compared a normal laptop, which for a device such as a RPi is really important, since the benchmark tests were in favor of the more powerful laptop.

Afterwards, the second phase of the test is performed to evaluate the scalability of the RPi when the number of VMs increased, the same test, shown in figure 4.4 was made except now the number of VMs launched are two. This means that 2 VMs are instantiated at the same time, and then the time that it takes each one to issue a ping command to the server is measured. The main differences are, now, two SSH channels need to open, one for each VM, and, the TCP socket has to be kept open until both VMs boot. Figure 4.4 is merely an example, since the VM that is instantiated first might not be the one that finishes first during the actual test. On this test 4 samples were taken, and for the RPi, VM1 had a mean booting time of 101,3184 seconds with a standard deviation value of 3,1914 seconds, the VM2 had a mean value of 91,8872 seconds and a standard deviation value of 4,0515 seconds. Comparing these 2 values with the value taken from a single instantiation it is possible to observe that the instantiation times are slower by around 9 seconds for one VM and 18 seconds for the other. The same test is, once again, ran on the

[3]`https://www.wireshark.org`

laptop for reference. The values obtained are, for VM1, a mean value of 115,8727 seconds and a standard deviation of 2,8902 seconds, and, for VM2, a mean value of 122,6502 seconds and a standard deviation of 4,4458 seconds, resulting in a difference of around 23 seconds for VM1 and 30 seconds for VM2.

Comparing these results, once again the results are in favor of the RPi, since it still boots faster and both times degraded less than the laptop's. Since the laptop's disk is faster than the RPi's disk, a possible explanation for the better results is the RPi's VM/image being better tuned for the device when compared to the VM/image used for the laptop. However, further scalability would prove difficult for the RPi due to lower RAM (table 4.1), since its 1 GB is already in the higher end of RAM available to the RPi models and more VMs would cap this value, whilst, the laptop's RAM is very low for the current market.

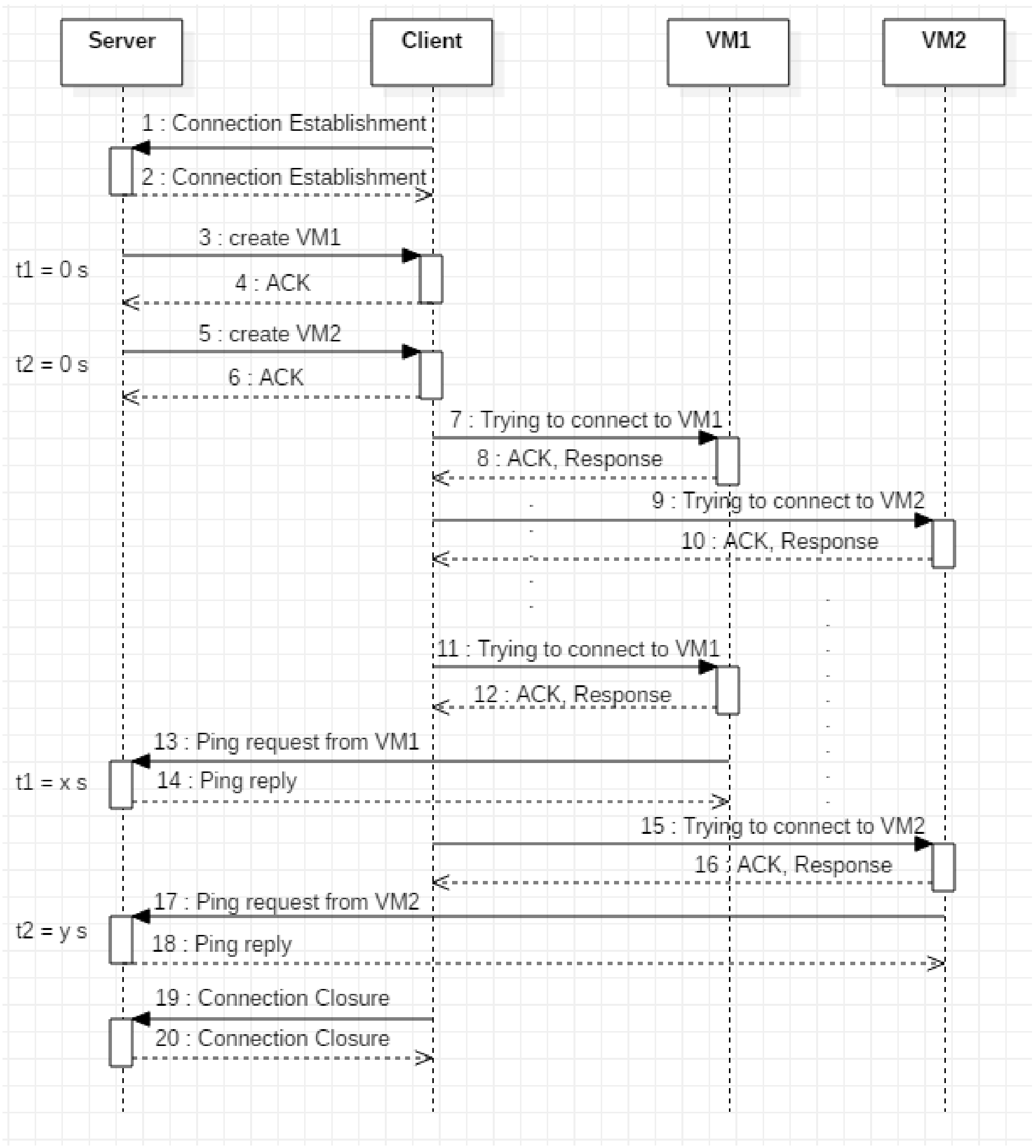

Figure 4.4: Sequence diagram of the messages exchanged during the second phase of the test.

4.3 Power Consumption

In order to better understand the impact of a VM on a RPi, a look into the energetic impact is taken, thus, a small circuit was devised to measure the power consumption of a RPi before, during, and after the instantiation of a VM and a laptop for comparison. The circuit is displayed in figure 4.5 and it features a multimeter (Fluke 77 Series II) in series between the Alternate Current (AC) voltage source and the RPi's power supply (DUT) measuring the alternate current passing through the voltage source and the DUT. By measuring the alternate current and having measured the voltage beforehand, it is possible to calculate the power consumption.

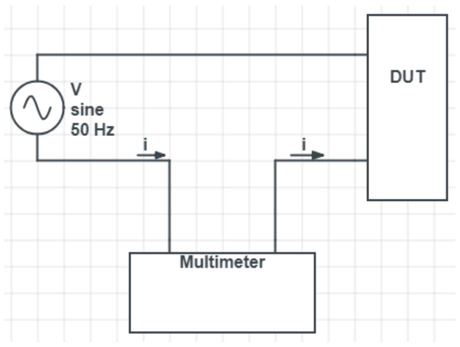

Figure 4.5: Block diagram for the power consumption measurement.

The voltage measurement before starting the test was 220.4 V, this value is obtained by putting the plugs of the multimeter in both ends of the AC voltage source and reading the value on the multimeter.

After these preparations, the measurement of the current of the RPi began where each mensuration for a total of 5 performed lasted for 200 seconds with measures being taken every 10 seconds. The first 30 seconds is simply the RPi before the VM instantiation, to have an idea of the idle power when nothing is running on the RPi. Then, approximately, on the 30 second mark the VM is instantiated to see how a RPi performs during the instantiation. Finally, knowing from before (section 4.2) that it takes approximately 80 seconds for a VM to be ready, 140 seconds pass to evaluate the power consumption after the VM stabilizes and then, on the 170 seconds mark a continuous ping command is issued from the VM in order to evaluate how a command from a VM affects the power consumption. For the reference laptop, the procedure and number of measurements is the same as the RPi.

The power consumption is then obtained by multiplying the samples obtained with the measure voltage, as equation 4.1 shows. Having obtained the power consumption for each measurement, the mean value in each point is calculated and then graphically shown in figures 4.6 and 4.7.

$$\begin{aligned} P(s) &= V_{rms} * I_{rms}(s) \qquad (4.1) \\ P(s) &= 220.4 * I_{rms}(s) * 10^{-3} \end{aligned}$$

Since the multimeter was set in the mA scale, that has to be taken into account when computing the power hence the 10^{-3} in equation 4.1.

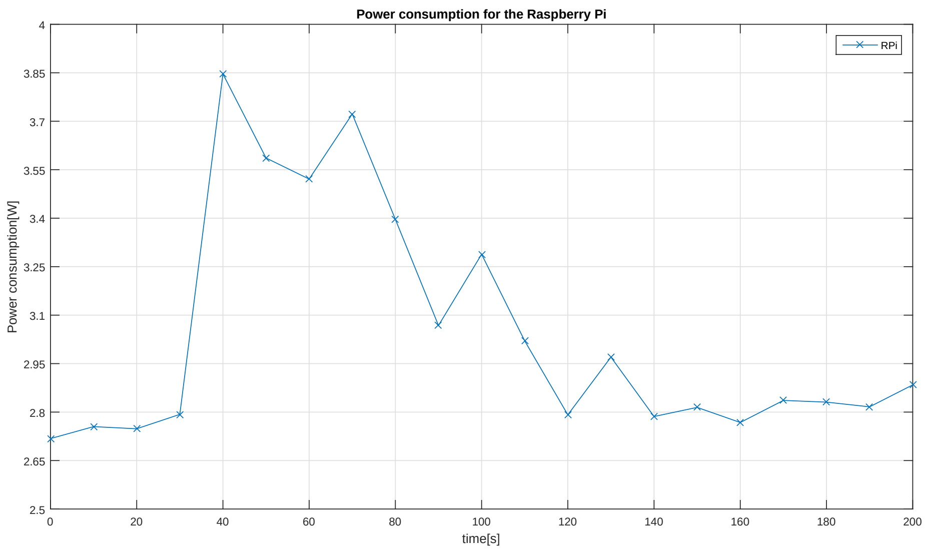

Figure 4.6: Power consumption during a VM instantiation for a RPi.

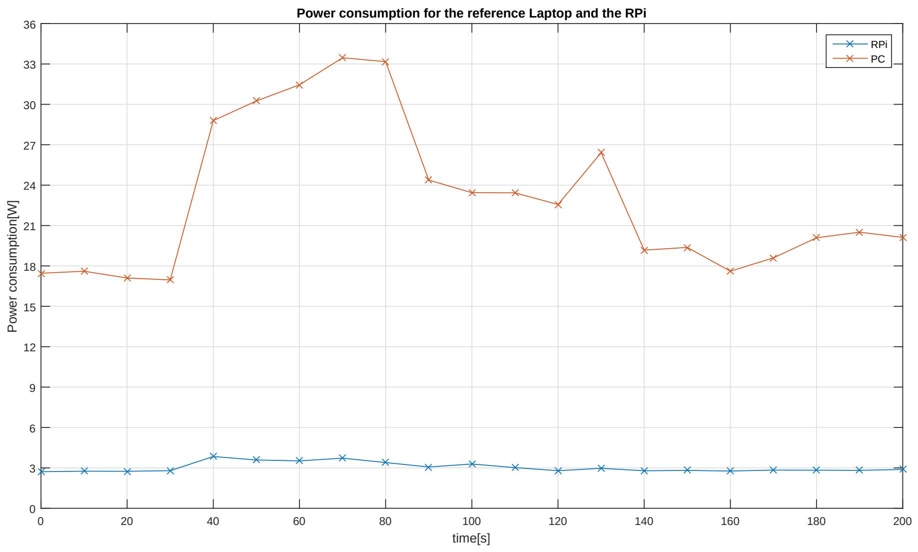

Figure 4.7: Power consumption during a VM instantiation for a RPi and a laptop.

Analyzing the RPi graphic on figure 4.6, the first 30 seconds of idle power settles around 2.75 W when the RPi is inactive. Past 30 seconds it is possible to clearly see the spike of power consumption right after the VM instantiation with values almost reaching 4 W. After the instantiation, the power consumption starts slowly decreasing with sudden increases in between skewing towards a value slightly above the idle power close to 2.80 W. Finally, at 170 seconds a ping command is issued from the VM to continuously ping another host on the same network and the graphic shows the power consumption slightly above the value measured before the issued ping at around 2.90

W.

On the other hand, the laptop graphic on figure 4.7 shows that for the first 30 seconds the computer has an idle power of around 17 W, which then spikes after the 30 second mark (the VM instantiation) reaching values close to 34 W. At the 80 second mark, the power starts decreasing ending up with values ranging 22 to 24 W. Just before the 130 second mark, the power consumption spikes again, this time reaching values above 26 W, and then the VM is finally instantiated. Similarly to the RPi graphic, the power consumption after the VM finishes the instantiation decreases settling around 19 W, which is a little higher than the idle power obtained from the early measurements. At the 170 second mark, a ping command is issued on the VM and it can be seen that the power reaches values around 20 W.

Observing the graphic on figure 4.7, which also shows the RPi power consumption on the same scale of the laptop's, the VM instantiation on the RPi is almost unnoticeable in comparison, and as expected, the laptop consumes more power than the RPi with values of 6 to 8 times higher by doing the same action. It should be noted that there were some programs running on the background on the laptop during the measurement, however, these programs should not draw much power due to being on the background and the values would still be higher than the RPi. This can be seen in the graphic, when the VM stopped pulling resources from the laptop the difference of power was just around 2 W, meaning that one or two programs on the background with less resource dependency as a VM would not draw much power. Comparing both figure 4.6 and figure 4.7, both devices show a similar behaviour during the experiment. Due to having a larger range of values in the same measuring unit, the laptop has more sensibility to changes, as can be seen when a ping command is issued, where, in the RPi, the changes are barely seen, but in the laptop they are more visible. During this experiment, it was noted that power consumption on the reference laptop was highly susceptible to text being printed on the display while in the RPi it was not that clear. A parallel between power consumption and CPU usage can be made here, since when CPU usage is high the power consumption is also high as stated before.

These values show another one of SBC's strengths, which is, its low power consumption. For a device that despite offering a lower CPU performance has such low values for power consumption while having similar times to instantiate a VM is a great advantage when looking towards scenarios where continuous activity is required, making it a possible viable economic approach when providing services to costumers.

4.4 Chapter Overview

In this chapter, tests were performed to draw a comparison between the performance of a RPi and a normal laptop used as reference. The results showed that whilst the laptop was clearly superior in a straight CPU benchmark, the RPi achieved faster times when launching a VM and as expected drew less power than its counterpart, however, even though the number of VMs increased and the launching times were still faster in the VM its lower memory thwarts larger scalability. In the end, these results still showcase the promise of using SBC's in these types of scenarios.

Chapter 5

Conclusion

In this dissertation, new control and operation mechanisms are studied in order to develop the new paradigms, such as Fog Computing. This new paradigm, as stated before, locates itself in the edge of networks, which means, away from the datacenters and the core networks, in other words, devices in Fog Computing scenarios which require low latency will need new solutions to take the place of these datacenters.

Therefore, in chapter 3, a VIM, OpenVIM, is installed on a Fog Computing scenario to provide VMs to compute nodes (RPis). Then the implementation is presented, where changes on the source code of the software were made to accommodate the new aarch64 architecture and the other elements required for the framework are created, such as, a bridge, a DHCP server, and, description files for the instantiation of VMs. After successfully implementing the framework proposed, in chapter 4, tests were performed on the device, a RPi, to understand how it was placed among the usual, more powerful, devices that run this type of services to better understand its performance and potential problems.

The first test, CPU performance, compared both the RPi and a VM instantiated inside it against a regular laptop through a CPU benchmarking software, Phoronix Test Suite, and showed that the laptop had values 6 to 18 times higher, demonstrating that the RPi still lacks the processing speed to match against a laptop, while the RPi with and without a VM showed similar times, demonstrating that the VM possessed the same processing capabilities of the RPi. The next test, the instantiation time, compared the time it took VMs to boot on a laptop and a RPi. Divided in two phases, in the first, a single VM is launched on the DUTs, then two VM are launched simultaneously. Resorting to a TCP socket and a network packet analyzer, commands are exchanged between a server and the client (the DUT) and the time needed for a boot is measured. In the first phase, the RPi instantiated faster than the laptop with times of approximately, 82 s and 92 s, respectively. In the seconds phase, where two VMs are instantiated at the same time, the RPi, once again, had faster times than the laptop, with values of, approximately, 92 s and 101 s, compared to the values of the laptop of, approximately, 116 s and 123s, however, RPi's low memory value stops scalability to much larger values. The third test compares the power consumption of both a laptop and a RPi during 200 seconds. During that time, the power is measured when the device is idled and during the instantiation of a VM with a ping command being issued in the end. Their behaviour during the measurement is similar, both graphics spiked after the instantiation of the VM and then slowly decreased to values slightly above the idle power. As expected, the power consumption of the RPi is lower when compared to the laptop, with values 6 to 8 times lower than the laptop.

These results exemplify the state in which SBCs currently stand. On one hand, its hardware is still one step below the normal equipments used to deploy these services, as the CPU performance shows. On the other hand, its low power consumption combined with its low cost and, unexpected, similar instantiation times demonstrate the reason, and the promise, of why these devices are appearing in these scenarios.

5.1 Contributions

This work contributed to the outcome of an ongoing national project, "Mobilizador 5G" (`http://5go.pt/`) where virtualization mechanisms were studied in human-to-machine environments.

5.2 Future Work

Although this dissertation obtained positive results, to have a VIM provide virtual resources to Fog Computing nodes, there is still interesting work to realize.

- The increase in compute nodes, as this work only utilized one RPi for compute nodes, despite being possible to add more to the VIM.
- The implementation of a SDN controller will bring its benefits for these networks, which, OpenVIM already supports.
- Depending on how the technology evolves it might be a better idea to swap to a more powerful, stable, and, updated VIM like OpenStack, since, whilst, OpenVIM is under the OSM project, the more popular and updated more frequently VIM at the moment appears to be OpenStack.
- Finally, to test this scenario on others SBCs, the SBC chosen in this dissertation was a RPi, however, this is not the only SBC on the market, so further studies might be interesting to find if there are better choices in the SBC market for a device more suitable for these scenarios.

Appendices

A1 Sample XML code after source code changes.

```
<domain type='kvm'>
  <name>vm-debian</name>
  <uuid>c6218e9c-c010-11e8-8cc6-000c2998c4d4</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>1</vcpu>
  <cputune>
    <vcpupin vcpu='0' cpuset='0'/>
  </cputune>
  <numatune>
    <memory mode='strict' nodeset='0'/>
  </numatune>
  <os>
    <type arch='aarch64' machine='virt-2.9'>hvm</type>
    <boot dev='hd'/>
    <loader readonly='yes' type='pflash'>/usr/share/qemu/aavmf-aarch64-code.bin<
/loader>
    <nvram>/var/lib/libvirt/qemu/nvram/debian_VARS.fd</nvram>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <cpu mode='host-passthrough'>
    <model fallback='allow'/>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>preserve</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <video>
      <model type='virtio'/>
    </video>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2' cache='writethrough'/>
      <source file='/opt/VNF/images/debian-9.5.0-openstack-arm64.inc.qcow2'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <interface type='bridge'>
      <source bridge='br0'/>
      <model type='virtio'/>
      <mac address='52:50:22:e7:aa:23'/>
      <rom file=''/>
    </interface>
  </devices>
</domain>
```

A2 Sample code for a compute node JSON file.

```
{
    "host":
    {
        "ip_name": "192.168.1.82",
        "user": "root",
        "name": "host"
    },
    "host-data":
    {
        "name": "host",
        "user": "root",
        "ip_name": "192.168.1.82",
        "ranking": 100,
        "numas":
        [
            {
                "cores":
                [
                    {
                        "core_id": 0,
                        "thread_id": 0
                    },
                    {
                        "core_id": 0,
                        "thread_id": 1
                    },
                    {
                        "core_id": 1,
                        "thread_id": 2
                    },
                    {
                        "core_id": 1,
                        "thread_id": 3
                    }
                ],
                "interfaces":
                [
                    {
                        "source_name": "eth0",
                        "Mbps": 1000,
                        "pci": "0000:00:02.0",
                        "switch_port": "port2",
                        "switch_dpid": "00:00:00:00:00:00:00:01",
                        "sriovs":
                        [
                            {
                                "mac": "a6:12:47:bd:2e:03",
                                "pci": "0000:00:00.1",
                                "source_name": 0,
                                "vlan": 1
                            }
                        ],
                        "mac": "B8:27:EB:AC:D5:68"
                    },
                    {
                        "source_name": "br0",
                        "Mbps": 1000,
                        "pci": "0000:00:01.0",
                        "switch_port": "port1",
                        "switch_dpid": "00:00:00:00:00:00:00:01",
                        "sriovs":
                        [
                            {
                                "mac": "a6:23:58:ce:3f:14",
                                "pci": "0000:00:00.1",
                                "source_name": 0,
                                "vlan": 1
                            }
                        ],
                        "mac": "FE:C7:D5:7A:FD:DF"
                    }
                ],
                "numa_socket": 0,
                "hugepages": 2,
                "memory": 1
            }
        ]
    }
}
```

A3 Sample code for a flavour YAML file.

```
flavor:
  disk: "1"
  name: myflavor
  description: personal flavor
  ram: 512
  vcpus: 4
  extended:
    processor_ranking: 100
    numas:
    - memory: 1
      cores: 4
      interfaces:
      - name: xe0
        bandwidth: 100 Mbps
        dedicated: "no"
```

A4 Sample code for a local image YAML file.

```
{
    "image":
    {
        "path": "http://cdimage.debian.org/cdimage/openstack/archive/9.5.0/
debian-9.5.0-openstack-arm64.qcow2",
        "metadata":
        {
            "architecture": "aarch64",
            "use_incremental": "no",
            "vpci": "0000:00:02.0",
            "os_distro": "debian",
            "os_type": "linux",
            "os_version": "9.5.0"
        },
        "name": "debian-9.5.0-link",
        "description": "debian,9.5.0,arm64,link"
    }
}
```

A5 Sample code for an URL based image JSON file.

```
image:
    name:         debian-9.5.0-path
    description:  debian,9.5.0,arm64,local
    path:         /opt/VNF/images/debian-9.5.0-openstack-arm64.qcow2
```

A6 Sample code for a network YAML file.

```
network:
    name: net2
    type: bridge_man
    shared: True
    cidr: 10.210.0.0/24
    enable_dhcp: True
    dhcp_first_ip: 10.210.0.4
    dhcp_last_ip: 10.210.0.254
    provider:physical: bridge:br0
    dns:
    -   193.136.92.73
    -   193.136.92.74
    routes:
      default: 10.210.0.1
```

A7 Sample code for a VM JSON file.

```
{
    "server":
    {
        "hostId": "40b2fad0-58a4-11e8-ae14-000c2998c4d4",
        "name": "vm",
        "imageRef": "0aa47536-c0b4-11e8-bdf0-000c2998c4d4",
        "flavorRef": "23afdb8e-22d3-11e4-94c0-52540030594e",
        "start": "yes",
        "extended":
        {
            "processor-ranking": 100,
            "numas":
            [
                {
                    "cores": 1,
                    "interfaces":
                    [
                        {
                            "dedicated": "no",
                            "bandwidth": "100 Mbps",
                            "name": "eth0"
                        }
                    ],
                    "memory": 1
                }
            ]
        },
        "networks":
        [
            {
                "uuid": "b06e0bca-c010-11e8-8cc6-000c2998c4d4"
            }
        ],
        "description": "debian arm vm in a rpi with local qcow2 image"
    }
}
```

Bibliography

[1] OpenDaylight, "Fluorine - OpenDaylight." `https://www.opendaylight.org/what-we-do/current-release/fluorine`. [accessed November 5, 2018].

[2] ONOS, "Features - ONOS." `https://wiki.onosproject.org/display/ONOS/System+Components`. [accessed November 6, 2018].

[3] Floodlight, "The Controller - Floodlight Controller - Project Floodlight." `https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/1343548/The+Controller`. [accessed November 6, 2018].

[4] ETSI, "Network Functions Virtualisation (NFV); Architectural Framework." `https://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf`, 2014.

[5] OpenStack, "OpenStack Docs: Design." `https://docs.openstack.org/arch-design/design.html`. [accessed November 7, 2018].

[6] T. Sechkova, M. Paolino, and D. Raho, "Virtualized Infrastructure Managers for Edge Computing: OpenVIM and OpenStack Comparison," *2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, June 2018.

[7] F. Kaup, S. Hacker, E. Mentzendorff, C. Meurisch, and D. Hausheer, "Energy Models for NFV and Service Provisioning on Fog Nodes," *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018.

[8] H. Kim, J. Kim, and Y.-B. Ko, "Developing a Cost-Effective OpenFlow Testbed for Small-Scale Software Defined Networking," *16th International Conference on Advanced Communication Technology*, February 2014.

[9] S. Han and S. Lee, "Implementing SDN and Network-Hypervisor based Programmable Network using Pi Stack Switch," *2015 International Conference on Information and Communication Technology Convergence (ICTC)*, October 2015.

[10] A. V. Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, "MEC-ConPaaS: An Experimental Single-Board Based Mobile Edge Cloud," *2017 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, April 2017.

[11] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, p. 637âĂŞ646, October 2016.

[12] K. Kirkpatrick, "Software-defined networking," *Communications of the ACM*, vol. 56, pp. 16–19, September 2013.

[13] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, pp. 36–43, July 2013.

[14] V. Cunha, "Service Function Chaining for NFV in Cloud Environments," Master's thesis, Universidade de Aveiro, 2015.

[15] "What's Software-Defined Networking (SDN)?." `https://www.sdxcentral.com/sdn/definitions/what-the-definition-of-software-defined-networking-sdn/`, 2018. [accessed October 1, 2018].

[16] OpenDaylight, "Platform Overview - OpenDaylight." `https://www.opendaylight.org/what-we-do/odl-platform-overview`. [accessed November 5, 2018].

[17] ONOS, "ONOS Kingfisher Release Performance." `https://wiki.onosproject.org/download/attachments/13994369/Whitepaper-%20ONOS%20Kingfisher%20release%20performance.pdf?api=v2`, 2017. White paper.

[18] ONOS, "Whitepaper - ONOS - final." `http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf`, 2014. White paper.

[19] ONOS, "Features - ONOS." `https://onosproject.org/features/`. [accessed November 6, 2018].

[20] ONOS, "ONOS : An Overview - ONOS - Wiki." `https://wiki.onosproject.org/display/ONOS/ONOS+%3A+An+Overview`. [accessed November 6, 2018].

[21] L. V. Morales, A. F. Murillo, and S. J. Rueda, "Extending the Floodlight Controller," *2015 IEEE 14th International Symposium on Network Computing and Applications*, September 2015.

[22] ETSI, "Network Functions Virtualisation (NFV); Management and Orchestration." `https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf`, 2014.

[23] X. Wen, G. Gu, Q. Li, Y. Gao, and X. Zhang, "Comparison of open-source cloud management platforms: OpenStack and OpenNebula," *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*, May 2012.

[24] T. Rosado and J. Bernardino, "An Overview of Openstack Architecture," *Proceedings of the 18th International Database Engineering and Applications Symposium on - IDEAS 14*, 2014.

[25] O. ETSI, "OpenVIM installation (Release THREE) - OSM Public Wiki." `https://osm.etsi.org/wikipub/index.php/OpenVIM_installation_(Release_THREE)`. [accessed November 6, 2018].

[26] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," tech. rep., Nat. Inst. Standards Technol. (NIST), Gaithersburg, MD, USA, 2011. [Online] Available: `https://csrc.nist.gov/publications/detail/sp/800-145/final`.

[27] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog Computing and Its Role in the Internet of Things," *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC 12*, pp. 13–16, 2012.

[28] F. Jalali, S. Khodadustan, C. Gray, K. Hinton, and F. Suits, "Greening IoT with Fog: A Survey," *2017 IEEE International Conference on Edge Computing (EDGE)*, June 2017.

[29] M. Aazam and E.-N. Huh, "Fog Computing and Smart Gateway Based Communication for Cloud of Things," *2014 International Conference on Future Internet of Things and Cloud*, August 2014.

[30] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, *"Mobile Edge Computing - A Key Technology Towards 5G"*, vol. 11. Sophia Antipolis, France: European Telecommunications Standards Institute, September 2015. White paper.

[31] G. I. Klas, "Fog Computing and Mobile Edge Cloud Gain Momentum: Open Fog Consortium, ETSI MEC and Cloudlets." `http://yucianga.info/wp-content/uploads/2015/11/15-11-22-Fog-computing-and-mobile-edge-cloud-gain-momentum-%E2%80%93-Open-Fog-Consortium-ETSI-MEC-Cloudlets-v1.pdf`, November 2015. [accessed October 22, 2018].

[32] F. Kaup1, S. Hacker, E. Mentzendorff, C. Meurisch, and D. Hausheer, "The Progress of the Energy-Efficiency of Single-board Computers," tech. rep., Ottovon-Guericke-University Magdeburg, Networks and Distributed Systems Lab, Tech. Rep. NetSys-TR-2018-01. [Online] Available: `http://www.netsys.ovgu.de/netsys_media/Publications/NetSys_TR_2018_01.pdf`.

[33] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi," *39th Annual IEEE Conference on Local Computer Networks*, September 2014.

[34] A. Krylovskiy, "Internet of Things Gateways Meet Linux Containers: Performance Evaluation and Discussion," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, December 2015.

[35] F. Ramalho and A. Neto, "Virtualization at the Network Edge: A Performance Comparison," *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, June 2016.

[36] M. Ariman, G. Secinti, M. Erel, and B. Canberk, "Software Defined Wireless Network Testbed using Raspberry Pi OF Switches with Routing Add-on," *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, November 2015.

[37] F. P. Tso, D. R. White, S. Jouet, J. Singer, and D. P. Pezaros, “The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures,” *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, July 2013.

[38] P. Abrahamsson, S. Helmer, N. Phaphoom, L. Nicolodi, N. Preda, L. Miori, M. Angriman, J. Rikkilä, X. Wang, K. Hamily, and S. Bugoloni, “Affordable and Energy-Efficient Cloud Computing Clusters: The Bolzano Raspberry Pi Cloud Cluster Experiment,” *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, December 2013.

[39] M. S. Carmo, S. Jardim, A. V. Neto, R. Aguiar, and D. Corujo, “Towards Fog-Based Slice-Defined WLAN Infrastructures to Cope with Future 5G Use Cases,” *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, 2017.

[40] P. Bellavista and A. Zanni, “Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi,” *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN 17*, 2017.

[41] “SciMark Test Profile.” `https://openbenchmarking.org/test/pts/scimark2`. [accessed October 15, 2018].