laSalle ENG
Ramon Llull University

Escola Tècnica Superior d'Enginyeria
Electrònica i Informàtica La Salle

Treball Final de Màster

Màster Universitari en Enginyeria de Telecomunicació

Software Defined Networks (SDN) In Data Center Networks

Alumne

Pau Aragonès Sabaté

Professors Ponents

Joan Navarro

Rosa Maria Alsina

ACTA DE L'EXAMEN DEL TREBALL FI DE CARRERA

Reunit el Tribunal qualificador en el dia de la data, l'alumne

D. Pau Aragonès Sabaté

va exposar el seu Treball de Fi de Carrera, el qual va tractar sobre el tema següent:

Software Defined Networks (SDN) in Data Center Networks

Acabada l'exposició i contestades per part de l'alumne les objeccions formulades pels Srs. membres del tribunal, aquest valorà l'esmentat Treball amb la qualificació de

Barcelona,

VOCAL DEL TRIBUNAL

VOCAL DEL TRIBUNAL

PRESIDENT DEL TRIBUNAL

Abstract

Recently, new network services have emerged that have made the technologies we have been using for fifty years practically reach the limit of their capabilities. The appearance of Big Data, the introduction of cloud computing, applications in real time and the fact that modern communications are not exclusively between client and server, it is evident that a large amount of traffic is generated to the end user. To solve this increasing demand, temporary fixes have arisen, such as the creation of new protocols. However, these fixes do not solve the problem, and create the necessity of a global solution that faces the problems of modern networks and creates effective methods of communications.

Hence, a new concept is born: Software Defined Networking (SDN), which implements a new architecture that delegates control of network devices from external software, called a controller, based on a protocol created for this purpose called OpenFlow.

The general purpose of this project is the creation of a software environment that allow, by means of the integration of several elements, the implementation and testing of these networks.

The results obtained show which network is best in terms of performance and features. As the network grows in elements, the performance of the Software Defined Networks is superior to that of the traditional network.

These results prove that the SDN is the future networking architecture to implement in large datacenters as it introduces new features and improvements that the current business needs demands

Resumen

Recientemente, han surgido nuevos servicios de red que han hecho que las tecnologías que hemos estado usando durante cincuenta años prácticamente alcancen el límite de sus capacidades. La aparición de Big Data, la introducción de la computación en la nube, las aplicaciones en tiempo real y el hecho de que las comunicaciones modernas no son exclusivamente entre el cliente y el servidor, es evidente que se genera una gran cantidad de tráfico para el usuario final. Para resolver esta creciente demanda, han surgido soluciones temporales, como la creación de nuevos protocolos. Sin embargo, estas soluciones no resuelven el problema y crean la necesidad de una solución global que enfrente los problemas de las redes modernas y cree métodos efectivos de comunicación.

Por lo tanto, nace un nuevo concepto: Software Defined Networking (SDN), que implementa una nueva arquitectura que delega control de dispositivos de red desde un software externo, llamado controlador, basado en un protocolo creado para este propósito llamado OpenFlow.

El objetivo general de este proyecto es la creación de un entorno de software que permita, mediante la integración de varios elementos, la implementación y prueba de estas redes.

Los resultados obtenidos muestran qué arquitectura de red es mejor en términos de rendimiento y características. A medida que la red crece en elementos, el rendimiento de las Redes definidas por software es superior al de la red tradicional.

Estos resultados demuestran que SDN es la arquitectura de red futura para implementar en grandes centros de datos, ya que presenta nuevas características y mejoras que las necesidades empresariales actuales demandan.

Resum

Recentment, han sorgit nous serveis de xarxa que han fet que les tecnologies que hem estat utilitzant durant cinquanta anys pràcticament arriben al límit de les seves capacitats. L'aparició de Big Data, la introducció de la computació en el núvol, les aplicacions en temps real i el fet que les comunicacions modernes no són exclusivament entre el client i el servidor, és evident que es genera una gran quantitat de trànsit per al usuari final. Per resoldre aquesta creixent demanda, han sorgit solucions temporals, com la creació de nous protocols. No obstant això, aquestes solucions no resolen el problema i creen la necessitat d'una solució global que davant els problemes de les xarxes modernes i creu mètodes efectius de comunicació.

Per tant, neix un nou concepte: Software Defined Networking (SDN), que implementa una nova arquitectura que delega control de dispositius de xarxa des d'un programari extern, anomenat controlador, basat en un protocol creat per a aquest propòsit anomenat OpenFlow.

L'objectiu general d'aquest projecte és la creació d'un entorn de programari que permeti, mitjançant la integració de diversos elements, la implementació i prova d'aquestes xarxes.

Els resultats obtinguts mostren quina xarxa és millor en termes de rendiment i característiques. A mesura que la xarxa creix en elements, el rendiment de les Xarxes definides per programari és superior al de la xarxa tradicional.

Aquests resultats demostren que SDN és l'arquitectura de xarxa futura per implementar en grans centres de dades, ja que presenta noves característiques i millores que les necessitats empresarials actuals demanden..

Appreciations

I would like to thank everyone who has been involved in my thesis or has been suffering my stress during this time. An special appreciation to my family, my sister, my colleagues at work and my friends.

Content

1 Introduction 1

1.1 Background 1

1.2 Overview and Goals 1

1.3 Project Structure 2

2 State of the art 5

2.1 Limitations of traditional networking 5

2.2 SDN architecture 7

2.2.1 Application Layer 8

2.2.2 Control Layer 8

2.2.3 Infrastructure layer 9

2.3 OpenFlow Protocol 10

2.3.1 OpenFlow Switch 12

2.4 Open vSwitch 14

2.5 The controller 17

2.5.1 OpenDayLight (ODL) 19

2.6 Benefits of Software Defined Networks 22

2.7 SDN implementations 23

3 Implementation 27

3.1 Mininet 27

3.2 Environment Setup 29

3.3 OpenDayLight controller setup 29

3.3.1 Controller access and management 31

3.4 Network Design 34

3.4.1 SDN network architecture design 36

3.4.2 Standard network architecture design 38
3.5 Connecting Mininet with OpenDayLight controller 39
3.6 Standard network simulation 44
4 Results 45
4.1 Throughput 45
4.2 Latency 46
5 Conclusions 51
5.1 Future lines of work 52
6 Budget 53
7 References 55
8 Annexes 57
8.1 SDN network architecture code 57
8.2 Legacy Network architecture code 59

Acronyms

API: Application Programming Interface.

GUI: Graphical User Interface.

ICMP: Internet Control Message Protocol.

IP: Internet Protocol.

ODL: OpenDayLight.

OF: OpenFlow.

OSGi: Open Services Gateway Initiative

OvS: Open vSwitch.

REST: Representational State Transfer.

RTT: Round-trip time

SDN: Software Defined Network.

TCP: Transfer Control Protocol.

VM: Virtual Machine.

UI: User Interface.

1 Introduction

1.1 Background

Throughout my studies in Telecommunications Engineering, I developed an interest in network architecture, as I see them as one of the fundamental pillars in this field. For this reason, I was very interested in developing my Master's Thesis about this theme.

We live in a world in constant change due to the technological advances that introduces new concepts, new challenges and new ways of solving today's problems in a manner that it would not have been imaginable few years ago. Thus, individuals and enterprises must adapt these changes in order to stay up to date and be competitive and make sure they are not stalled in a technology no longer in use or deprecated.

Enterprises must face huge revolutionary new technologies that defies the conception of IT infrastructure that was hard to imagine in the past decades. The introduction of Cloud technologies, high intensive consumption of infrastructure capabilities from new trends in the IT sector such as Artificial Intelligence or Machine Learning, and the new decentralized architectures proposed in technologies like Blockchain, all these factors make the traditional and current IT infrastructure in enterprises not ideal and insufficient to adapt to these changes.

Network architecture is currently experiencing a massive innovation due to the appearance of *the* Software Defined Networking (**SDN**) that promises the introduction of many new features to the immediate future. I presented this topic to the GRITS research group and Professor Joan Navarro accepted to be my advisor for this thesis.

1.2 Overview and Goals

This thesis is developed at the *Universitat Ramón Llull – La Salle* at the *GRITS* department, focused in the field of network architecture.

The purpose of this project is to design and implement a Software Defined Network (SDN) in a data center environment and evaluate its performance by means of extensive simulations.

There are different implementations of SDN deployed nowadays due to most of these implementations are open source. The most known implementation is Open Daylight controller software in conjunction with Open vSwitch virtual switch software, as it is actively developed and supported by The Linux Foundation.

The project's main goals are:

1. Implement a network architecture using the SDN protocol OpenFlow and the SDN controller OpenDaylight.
2. Compare a traditional network architecture and a SDN architecture in terms of features and performance.

1.3 Project Structure

The structure of this thesis has been designed in the following order. Firstly, some theoretical concepts have been explained in order to understand all the basic concepts of network architectures and the features the new Software Defined Networks introduce. In order to accomplish this sections, some related technical papers have been read and analysed for obtaining this information and presenting the state of the art of the subject.

Once explained the main theoretical concepts, a practical part is developed so that the theoretical concepts can be applied and evaluated. Two network architectures are proposed, a traditional network architecture and a Software Define Network architecture,. These two networks are designed to be most similar in terms of number of network elements, hosts and links between the devices.

After the design of both networks, a set of tools used for evaluating network performance and simulation are implemented for evaluation and comparison between them. A pair of network metrics are taken into account for comparing the performance of both networks for presenting the results of the evaluation.

Finally, a conclusion of this project is developed taken into account the results obtained in this thesis and some future lines of work are proposed for further investigation of the subject.

2 State of the art

Software Defined Networking (SDN) is an emerging network architecture that gives the network control to an application called controller. This architecture allows splitting the control management and the data management for achieving networks that are programmable, flexible and automated. It leads to the separation of the management of the hardware and software components. This migration of control allows the underlying infrastructure to be abstracted for applications and network services can treat the network as a virtual entity.

The SDN controller is a centralized entity, which means, it can be composed of various physical or virtual instances but it behaves as a single component. It maintains the global or partial status of the network, allowing enterprises and operators gain control of the entire network from a single logical point. This leads to an enormous design and operation simplification.

Moreover, SDN also simplifies all network devices since it is not necessary to understand and process thousands of protocols, but only accept the instructions of the controller defined by the network administrator.

2.1 Limitations of traditional networking

Meeting current market requirements is virtually impossible with traditional network architectures. Faced with flat or reduced budgets, enterprise IT departments are trying to get the most from their networks using device-level management tools and manual processes. Existing network architectures were not designed to meet the requirements of today’s users, enterprises and carriers. This leads to network designers being constrained by the limitations of current networks. [1]

Formerly, a traditional network architecture consists of a set of transmission media (air, fiber optic, copper) and switching (switches, routers). The management of these networks id distributed, each switching element incorporates a firewall that makes its own decisions based on certain fields in the frames and received packages. While it is true that they have helped to mitigate the effects of requirements of the new network services, it should take into consideration that no longer they are efficient and have limitations. [2]

- The current protocols are based in RFC's, and any modification that some entity wants to introduce will have to pass through a long process of study and approval by the competent organizations.

- Research and development are not promoted due to manufactures and administrators are reluctant to experiment and introduce new models in networks that already work, according to them, in a satisfactory way.

- These traditional network architectures were not designed to support the bandwidth and services currently required, as is the case with streaming or online games.

- Traditional networking has little flexibility and is difficult to manage and configure, since they behave based on the protocols that manufacturers include in their devices. The substitution or migration of a network element would probably require rules of the router, the firewall and other elements. For this reason, the operators tend to maintain an aesthetic structure that causes the loss of dynamism in the network, which leads to a bad adaptation to the changes in traffic.

- Enterprises seek to deploy new capabilities and services in rapid response to changing business needs or user demands. However, their ability to respond is limited by vendors' equipment product cycles, which can range to three years

or more. Lack of standard, open interfaces limits the ability of network operators to tailor rhe network to their individual environments.

2.2 SDN architecture

Software Defined Networking (SDN) is an emerging network architecture where network control is decoupled from forwarding and is directly programmable. This migration of control, formerly tightly bound in individual network devices, into accessible computing devices enables the underlying infrastructure to be abstracted for applications and network services, which can treat the network as a logical or virtual entity.

In a SDN, the key resides in the previously mentioned separation of the control and data management. This leads to a reformulated network architecture, which would consist of three layers that are accessible from various application programming interface (API).

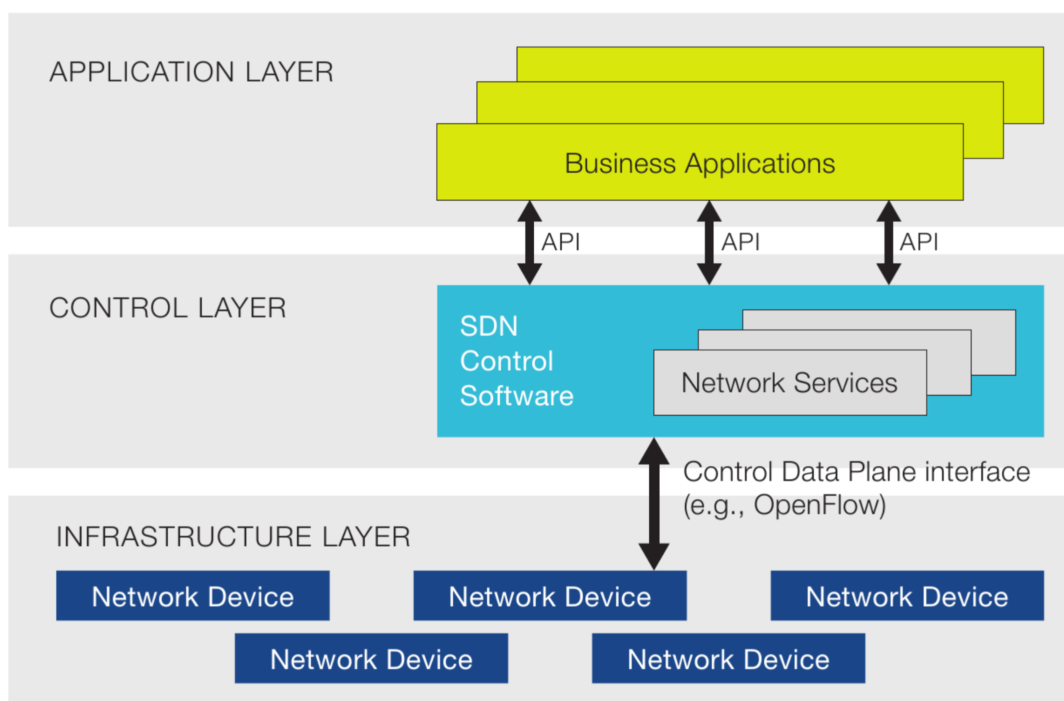


Figure 1: Software Defined Network Architecture

2.2.1 Application Layer

This layer consists of applications intended for end-users who will be the consumers of SDN communications services. The end-users use SDN communication services through the northbound API such as REST, JSON, XML among others. This northbound API is used to connect the SDN controller to the services and applications above, allows services and applications to simplify and automate the tasks of configuring, provisioning and managing new services in the network, offering operators new ways of income, differentiation and innovation, as well as meeting the needs of different applications through the SDN network's programmability.

The northbound API crosses the limit between this layer and the Control Layer. These are the most critical interfaces, as mentioned above; they support a large number of applications and services critical for the end-user.

2.2.2 Control Layer

This layer provides centralized control functionality that monitors the behavior of the data network through an open interface; allows application developers to use network capabilities but abstracted from their topology or functions. In this layer it has to be mentioned the SDN controller, since it is the logical control entity responsible for translating SDN service requests to lower data paths, giving the application layer an abstract view of the network through statistics and possible events.

It could be said that the controllers are the brain of this network architecture, since they have exclusive control over the way to manage and configure network nodes to correctly direct traffic flows. In addition, the architecture allows it to generate a wide range of data plane resources, which offers the potential to unify and simplify its configuration.

It provides a set of common APIs to the application layer (northbound APIs), while implements one or more network protocols for controlling network devices (southbound interface). SDN does not only support SDN-oriented networking

protocols, in fact, it also supports traditional networking protocols like Open Shortest Path First (OSPF), MultiProtocol Label Switching (MLPS) or Border Gateway Protocol (BGP).

The controller may also include a set of modules that allow him to carry out a set of basic networking tasks: inventory of the connected devices, statistics management and other functions. Providers can add new functionalities in the controller's core according to their needs, being this one of the key points of the SDN architecture.

2.2.3 Infrastructure layer

The network nodes that carry out packet switching and routing constitute this layer. It provides a programmable open access through the southbound API, such as OpenFlow. The southbound APIs facilitate control over the network, allowing the controller to make dynamic changes according to the demands in real time and needs.

Southbound APIs facilitate efficient control over the network and enable the SDN Controller to make changes dynamically according to real-time demands and needs. OpenFlow, which was developed by the Open Networking Foundation (ONF), is the first and probably most well-known southbound interface. It is an industry standard that defines the way the SDN Controller should interact with the forwarding plane to make adjustments to the network, so it can better adapt to changing business requirements. With OpenFlow, entries can be added and removed to the internal flow-table of switches and potentially routers to make the network more responsive to real-time traffic demands. Besides OpenFlow, Cisco OpFlex (the company's response to OpenFlow) is also a well-known southbound API.

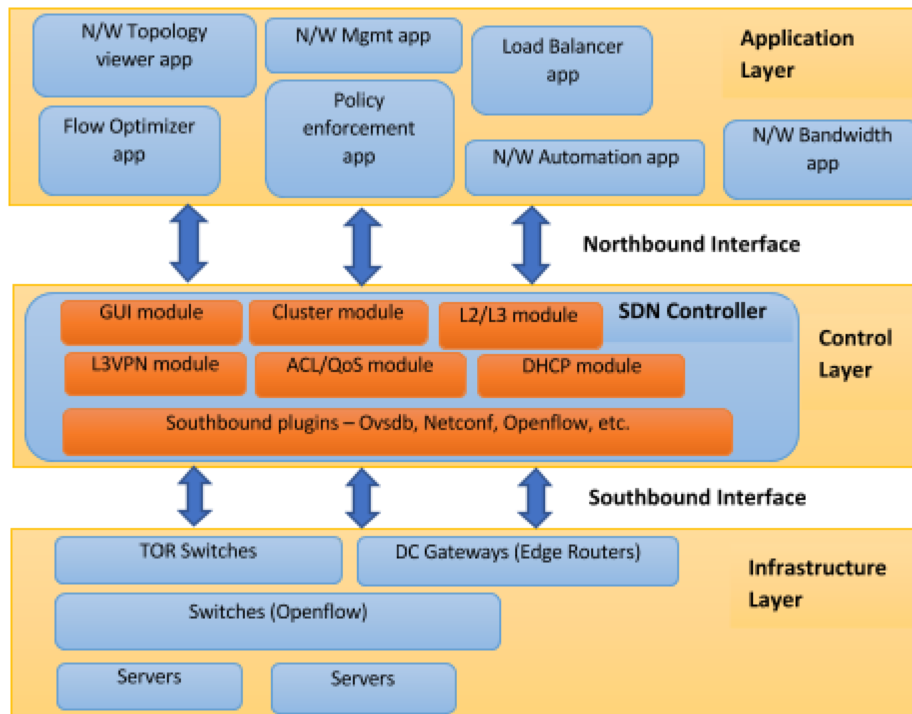


Figure 2: Components of each layer of the SDN architecture

2.3 OpenFlow Protocol

OpenFlow is one of the first communication standards defined between an OpenFlow switch and the controller in an SDN architecture. It facilitates the programmability of the network through the configuration, management and control of data flows from a centralized software. It allows partitioning the traffic; decide the best routes for the packets and how the packages are processed. It is focused on the control of traffic, security and the creation of new forms of routing, among other features.[3]

The first OpenFlow specification was created in 2008 by the Stanford University. They released the version 1.0 by the end of 2009, but with a clear goal of making it open and owned by the community [10]. This is why, since 2011, the Open Network Foundation has been the organization responsible for its promotion and adoption. Currently, it is supported by many switch and router vendors such as Cisco, IBM, Juniper Networks or Hewlett-Packard.

The first version developed already managed by the Open Networking Foundation, the 1.1, was released on 28 February 2011. One year later it was released the version 1.2, and the most recent one is the version 1.4.

OpenFlow allows the actual moving of network control from the switches to the control layer, actually separating the control plane from the data plane. However, as a southbound interface, it needs to be implemented on both sides it, this is, in the SDN controller and the infrastructure devices. These last ones are called OpenFlow Switches.

In a conventional network, each network device has proprietary software that implements a methodology of packet routing. However, with OpenFlow, packet routing decisions are centralized. This allows the network to be programmed independently of the individual switches and data center hardware.

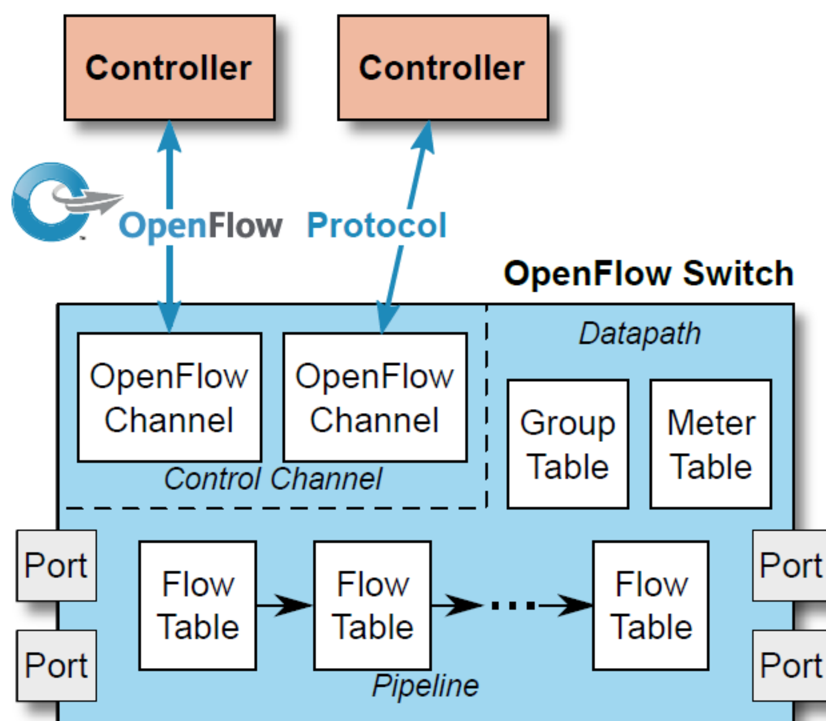


Figure 3: OpenFlow Switch Architecture

A switch that implements the OpenFlow protocol is composed of three components that allow the correct operation of this protocol.

2.3.1 OpenFlow Switch

It is the representation of the actual underlying switch that the SDN controller is going to manage. It consists of one or more OpenFlow tables and a group table, which perform packet analysis and forwarding within the switch. These switches that support OpenFlow can be both virtual or physical.

The physical ones are not only those built with OpenFlow in mind, but also those legacy switches that can be updated to support at least the first version of OpenFlow.

Whichever is the case, an OpenFlow Switch will consist of the following components:

- **Ports:** packets will enter the switch and exit it through them. They do not have to be physical as they may be logical ports defined by the switch.
- **OpenFlow tables:** they perform packet analyzing and forwarding. They contain a series of OpenFlow entries, which are used to match and process packets according to their packet headers.
- **Channel:** it is the interface used to communicate the switch with the controller, therefore the switch receives the configuration from it.

2.3.1.1 Flow tables

These tables include information about an action associated with each entry in the table, indicating to the switch how it should process that flow (flow entries).

Each entry consists of the following fields:

Match fields	Priority	Counters	Instructions	Timeout	Cookie

Table 1: Flow entry in a Flow Table

- **Match fields**: it includes information about the header and port.
- **Priority**: assigns the flow entry priority.
- **Counters**: this value is updated when a coincidence is found.

- **Instructions**: this value is used for modifying a set of actions.
- **Timeout**: maximum wait time until a flow expires.
- **Cookie**: this value is used by the controller for flow modifications, filter statistics, among other uses. It is not used when processing packages.

2.3.1.2 Group tables

These tables consist of a group of entries. The groups provide an efficient way to indicate that the same set of actions must be done by multiple flows. For this reason, it is useful to implement both multicast and unicast.

Group identifier	Group Type	Counters	Action Buckets

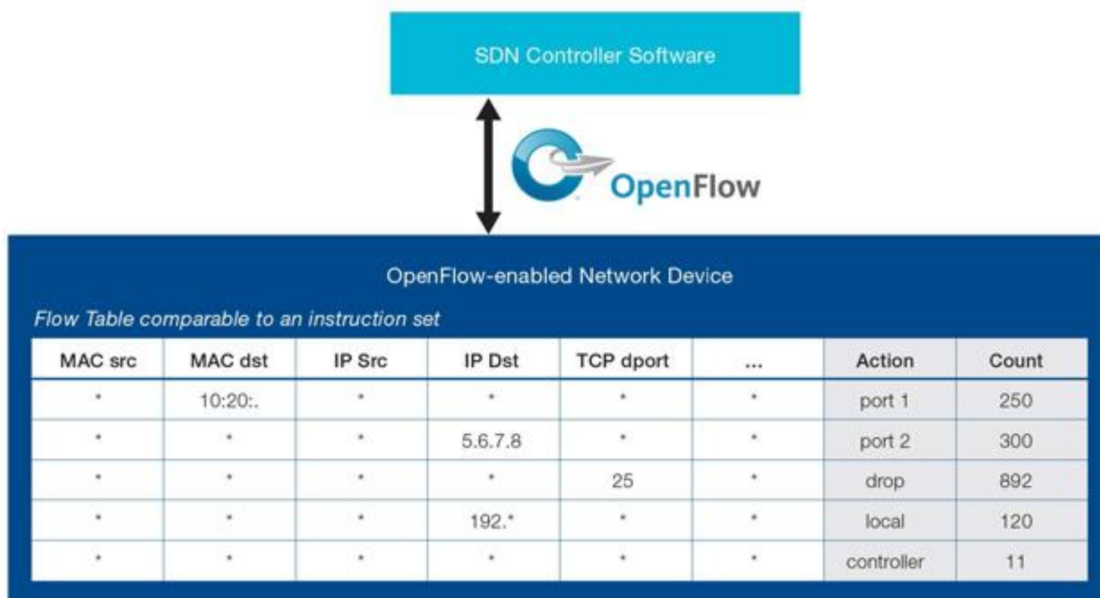
Table 2: Flow Entry in a Group Table

- **Group Identifier**: unique unsigned integer that identifies a group.
- **Group Type**: determines group semantics. Some of them are, i.e.: all (execute all actions of a group) and select (execute one action of a group).
- **Counters**: this value is updated once a group processes the packets.
- **Action buckets**: ordered list of actions. Each action contains a set of actions to execute and associated parameters.

2.3.1.3 OpenFlow Channel

The OpenFlow channel is the interface that connects each OpenFlow switch with a controller. Through this interface, the controller configures and manages the switch, receives events from the switch and sends packets out of the switch. Between the datapath and the OpenFlow channel, the interface implements a specific implementation. However, all OpenFlow channel messages must be formatted

according to the OpenFlow protocol. It is usually encrypted using TLS, but it can be directly used over TCP.



MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Figure 4: Example of OpenFlow Instruction Set

In Figure 3, an OpenFlow Instruction set is compared with a Flow table. This instruction sets updates the flow tables of the network device so that it stays up to date with the network status. Any changes that the controller needs to notify to all network elements are done through this sets. It is also important to note that this instruction sets are personalized for each network device, so when new instructions need to be updated, not all network elements would receive exactly the same instruction set, it would be fitted for each element conditions in the network.

2.4 Open vSwitch

Open vSwitch is a software implementation of a virtual multilayer network switch, designed to enable effective network automation through programmatic extensions, while supporting standard management interfaces and protocols such as NetFlow, OpenFlow, sFlow, SPAN, RSPAN, CLI, LACP and 802.1ag. [4]

In order to define what Open vSwitch (OVS) is, it is extremely important to first understand virtual switching and the new network access layer with in the data center. In the past, servers would physically connect to a hardware-based switch located in the data center. When VMware created server virtualization, the access layer changed from having to be connected to a physical switch to being able to connect to a virtual switch. This virtual switch is a software layer that resides in a server that is hosting virtual machines (VMs). VMs, have logical or virtual Ethernet ports. These logical ports connect to a virtual switch.

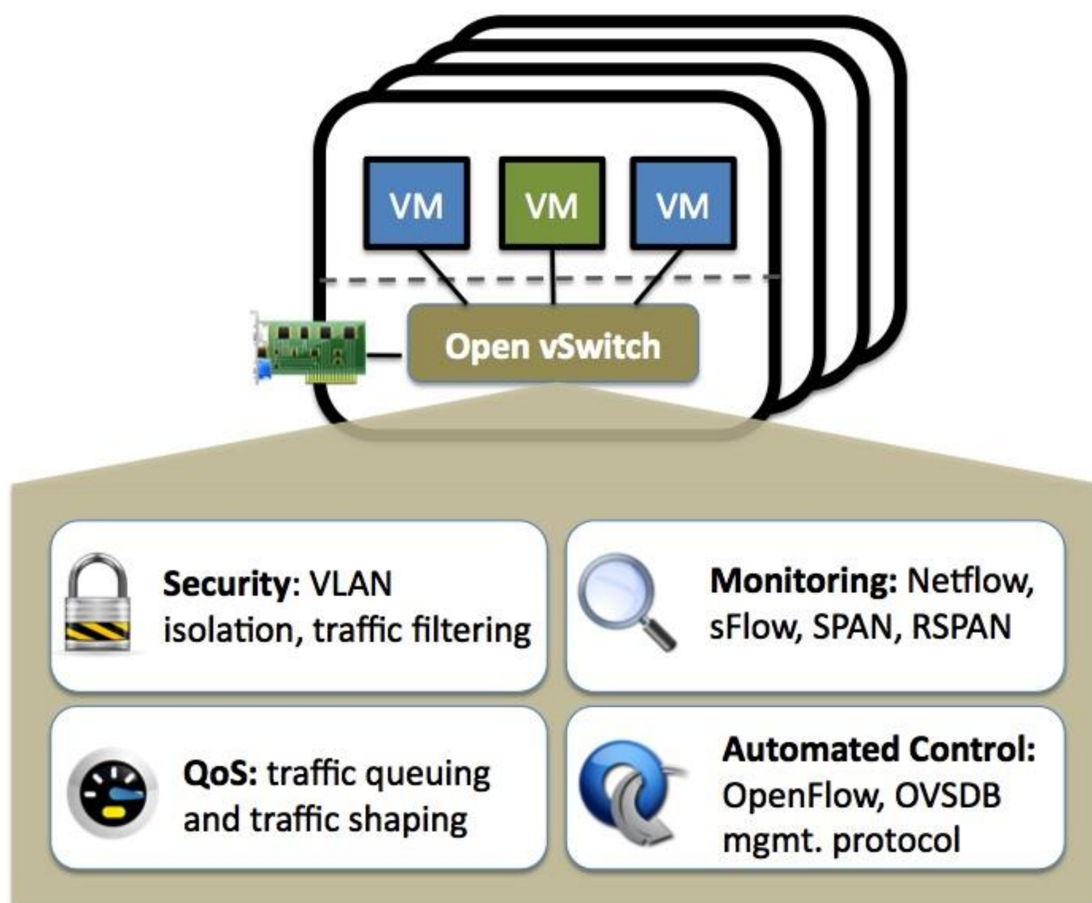


Figure 5: Open vSwitch features and capabilities

From a control and management perspective, Open vSwitch leverages OpenFlow and the Open vSwitch Database (OVSDB) management protocol, which means it can operate both as a soft switch running within the hypervisor, and as the control stack

for switching silicon. OVS is important in Software Defined Networks due its new capabilities that are critical for these networks.

Using OVS for virtual networking is considered the core element of many datacenter SDN deployments and the main use case is multi-tenant network virtualization. OVS can also be used to direct traffic between network functions in service. In some cases, it could be considered critical to many SDN deployments in data centers because it ties together all the virtual machines (VMs) within a hypervisor instance on a server. It is the first entry point for all the VMs sending traffic to the network and is the ingress point into overlay networks running on top of physical networks in the data center.

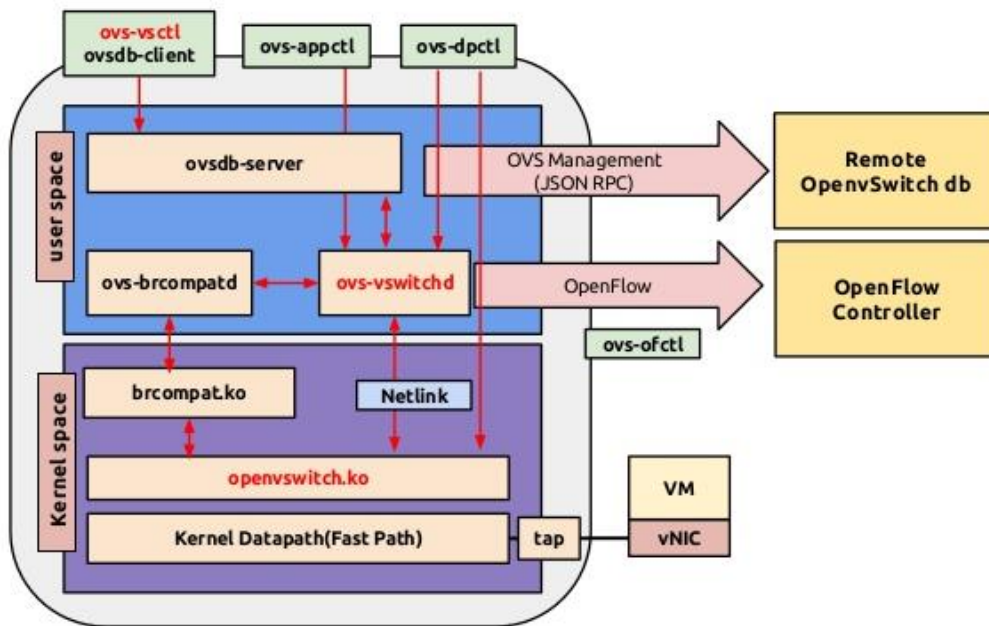


Figure 6: Open vSwitch architecture

Note the OpenFlow protocol is included in the Open vSwitch architecture. This means that this software has full support with this critical protocol and, therefore, there is no need to install any extensions or plugins for its integration.

2.5 The controller

In the SDN architecture, the controller is the main element. The device implements the rules of the network, executes the instructions that are dictated by the applications and distributes them among the different physical layer devices of the network. It is responsible for managing the packets that do not fit in the entries of the flow tables. The controllers differ from each other, in terms of the programming language and platform, but eventually perform the same functions: communicate, using the OpenFlow protocol, with the devices in the network.

An SDN Controller platform typically contains a collection of "pluggable" modules that can perform different network tasks. Some of the basic tasks including inventorying what devices are within the network and the capabilities of each, gathering network statistics, etc. Extensions can be inserted that enhance the functionality and support more advanced capabilities, such as running algorithms to perform analytics and orchestrating new rules throughout the network.

Two of the most well-known protocols used by SDN Controllers to communicate with the switches/routers is OpenFlow and OVSDB. Others protocols that could be used by an SDN Controller is YANG or NetConf. Other SDN Controller protocols are being developed, while more established networking protocols are finding ways to run in an SDN environment. For example, the Internet Engineering Task Force (IETF) working group – the Interface to the Routing System (i2rs) – is developing an SDN standard that enables an SDN Controller to leverage proven, traditional protocols, such as OSPF, MPLS, BGP, and IS-IS.

The controller contributes to the network by managing and monitoring the network status by orchestrating all its services installed. It implements the service throughout the network and designs thedevice topology and discovery mechanism to make all elements interconnected. It also is capable of calculate the routes, and it is intelligent enough to detect a malfunctioning link by rerouting connections, guaranteeing high availability of the network. All devices are connected to the controller through TCP

sessions. It also offers a set APIs that expose the services of the controller to the managing applications.

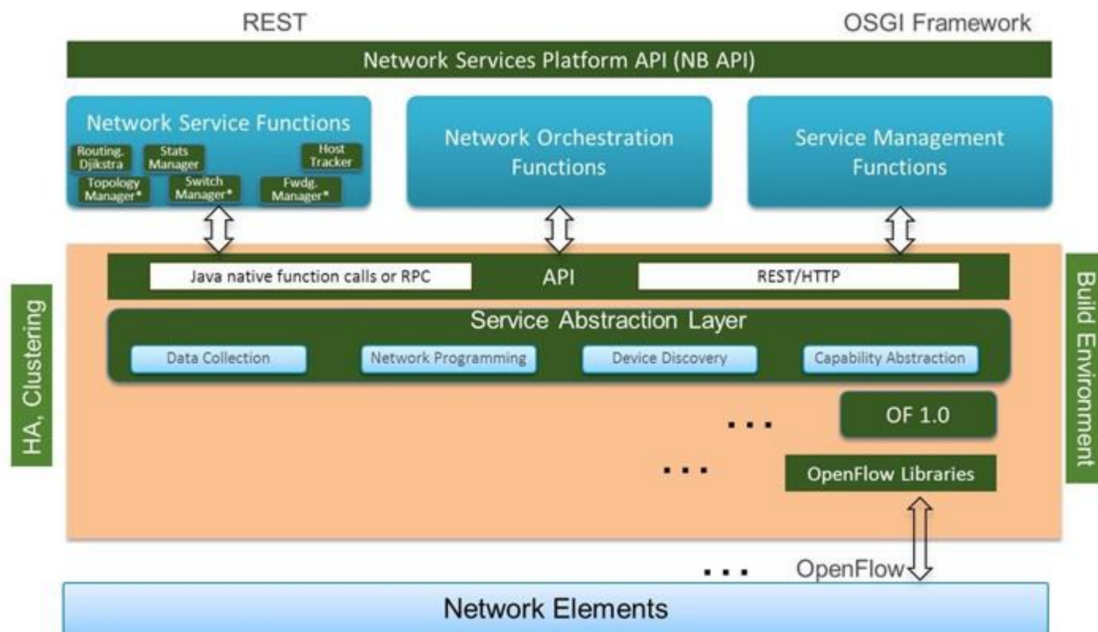


Figure 7: Software Defined Network Controller Architecture

SDN controllers can be useful in many environments, including cloud and data center networks, where they can offer better utilization of resources and faster turnaround times for multitenant segregation, and in enterprise campus networks, in which the benefits of network access control and network monitoring can be leveraged. They also show great promise for service provider networks, where optimization and control of application flows traffic is essential to business growth and success.

There are multiple open source SDN controllers focusing mainly in enterprise environments [5]:

- **NOX** was the first OpenFlow controller, developed at Nicira Networks (acquired by VMware in 2012) in parallel with OpenFlow. It was written in C++ as a program to manage switches. Nicira donated NOX to the research community in 2008.

- **POX** is the successor to NOX, written in Python. POX is still under active development with the goal to develop the archetypal, modern SDN controller.
- **Beacon** is written in Java and works with the Eclipse integrated development environment. Although limited to star topologies, it was the first SDN controller that allowed programmers without extensive experience to enable SDN environments.
- **Floodlight** is a Java-based OpenFlow controller that is also enterprise-class and Apache-licensed. It is part of a collection of open source projects done by Big Switch. The controller supports a range of virtual and physical OpenFlow switches and it can handle mixed OpenFlow and non-OpenFlow networks. The Controller includes support for the OpenStack cloud orchestration platform as well. Floodlight has already been used in a number of applications, including the OpenStack Quantum Plug-in and the Floodlight Virtual Switch.

2.5.1 OpenDayLight (ODL)

OpenDayLight [6] is an Open Source project led by the Linux Foundation that aims to accelerate the diffusion of innovation in design and implementation of an open and transparent standard of SDN. It aims to become an open platform used by all companies, preventing private applications from restricting market growth, while reducing development costs.

The main advantage of this proposal is that eliminates barriers, since some organizations do not want to commit to specific manufacturers that may block their development in the future. Being a common platform, companies may opt for technologies from different manufacturers.

The platform that provides this project (ODL) can be deployed directly without the need for any other component. This is due to the architecture of the platform, which provides a set of basic functions for the applications and the wide variety of collaborators that contribute to the project.

One of the main features that introduces this controller in a SDN is the microservice architecture. In this case, a microservice is a particular network protocol or service that a user wants to allow within their installation of the OpenDayLight driver, for instance; BGP protocol, an AAA service (Authentication, Authorization and Accounting). By implementing this architecture, the controller only implements the protocols the user needs and thus reduces costs and implementations. Moreover, this type of architecture allows the enterprise to easily scale according to the business needs.

OpenDayLight works perfectly with the OpenFlow protocol due to the same SDN implementation. Nevertheless, it also provides support for a wide range of protocols, not only OpenFlow, but also includes well-known protocols such as SNMP, NETCONF, OVSDB, BGP, PCEP, LIS, among others.

Due to its Software-Defined focus, it supports custom development of new functionalities composed of protocols and additional network services. It provides high-level abstraction of its capabilities so that network engineers and developers can create new applications to customize the configuration and administration of networks.

The controller, thus, has the main action of centralized control of physical and virtual network devices, controlling them with standards and open source protocols.

ODL controller is strictly implemented in software and is contained within its own Java Virtual Machine (JVM). As such, it can be deployed on any hardware and operating system platform that supports Java [8].OpenDaylight controller relies on the following technologies:

- **Maven:** is a project management tool that simplifies and automates dependencies between a project or different projects. This tooling will help developers to manage all the required plugins and dependencies of its applications, as well as to provide a project start-up by using its defined archetypes.
- **Java**: it is the programming language that is used to develop applications and features in the OpenDaylight's controller. Developing in Java provides a

valuable compile-time safety, as well as an easy way to implement defined services.

- **Open Service Gateway Interface (OSGi)**: it is the backend of OpenDayLight as it allows to dynamically load bundles and JAR packages (they compose the applications), and bind modules together for exchanging information.
- **Karaf**: it is an application container built on top of OSGi, which simplifies aspects of packaging and installing applications.
- **YANG**: it is the key-point of the model-driven behavior in the controller. Developers will use YANG to model application functionality, and to generate APIs from the defined models, which will be later used to provide its implementations. YANG supports modelling operational and configuration data, as well as RPC and notifications.

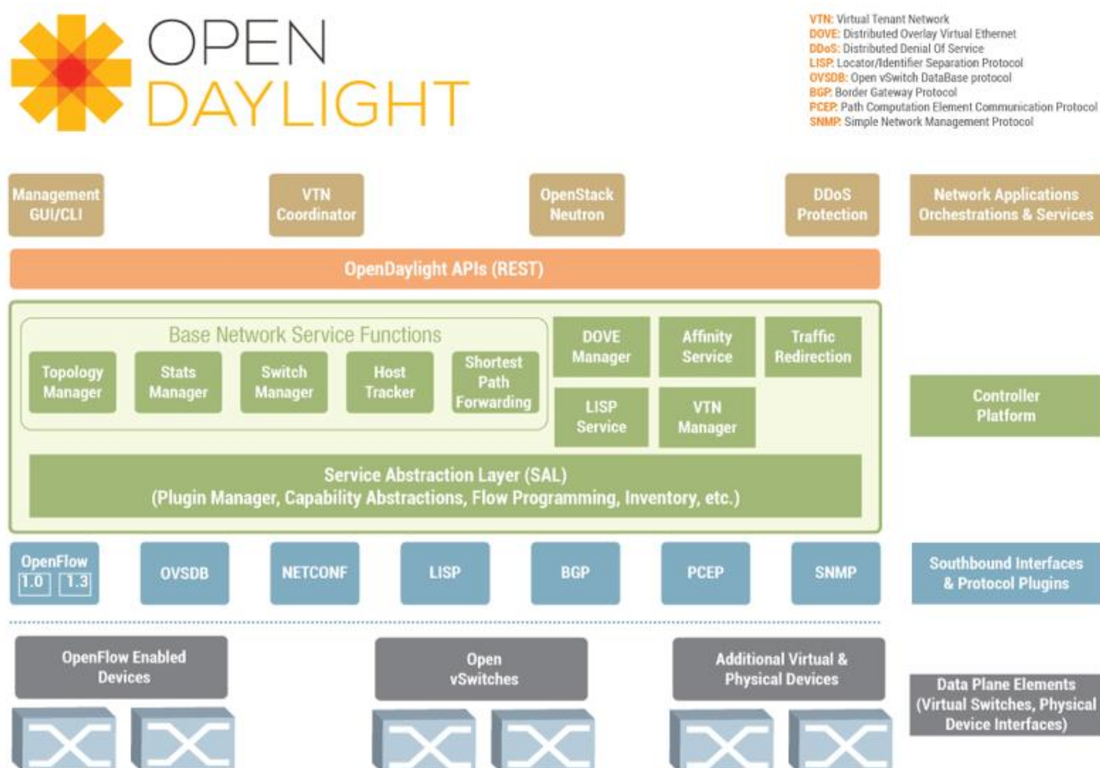


Figure 8: OpenDayLight Architecture

2.6 Benefits of Software Defined Networks

For enterprises, this new network architecture makes it possible for the network to be a competitive differentiator. By using OpenFlow-based SDN enable IT departments to address the high bandwidth, dynamic nature of today's applications, adapt the network to ever-changing business needs, and significantly reduce operations and management complexity. The benefits may include:

- **Centralized management of multi-vendor environments:** SDN control software can control any OpenFlow-enabled network device from any vendor, including switches, routers and virtual switches. This eliminates the management of group devices from individual vendors, so that the IT departments can use SDN-based orchestration and management tools to quickly deploy, configure and update devices across the entire network.

- **Reduced complexity through automation:** OpenFlow-based SDN offers a flexible network automation and management framework, which makes it possible to develop tools that automate many management tasks that are done manually today. These automation tools will reduce operational overhead and decrease network instability introduced by operator error. In addition, with SDN, cloud-based applications can be managed through intelligent orchestration and provisioning systems, further reducing operational overhead while increasing business agility.

- **Increased network reliability and security:** SDN makes it possible for enterprises to define high-level configuration and policy statements, which are then translated down to the infrastructure via OpenFlow. An OpenFlow-based SDN architecture eliminates the need to individually configure network devices each time an end point, service, or application is added or moved, or a policy changes, which reduces the likelihood of network failures due to configuration or policy inconsistencies.

- **Granular network control:** OpenFlow‘s flow-based control model allows IT to apply policies at a very granular level, including the session, user, device, and application levels, in a highly abstracted, automated fashion. This control enables cloud operators to support multi-tenancy while maintaining traffic isolation, security, and elastic resource management when customers share the same infrastructure.

2.7 SDN implementations

There are functional implementations that are serving large organizations for their business activity. The main benefits of SDN are mainly targeted at large computational infrastructure and datacenters. One successful implementation is the University of Pittsburgh Medical Center. [7]

The University Of Pittsburgh Medical Center (UPMC) is a $10 billion integrated global nonprofit health enterprise that has more than 62,000 employees, 21 hospitals, and 400 clinical locations including outpatient sites and doctors’ offices serving a 2.2 million-member health insurance division, as well as commercial and international ventures.

As a large healthcare provider, UPMC has a world class IT infrastructure environment that is 80 percent virtualized serving over 4 Petabytes of storage within its data centers supported by a private MPLS network.

With such a large and dynamic compute environment, UPMC found that its traditional IP network was suffering from network configuration delays due to the complexity of the workflow between the IT and IP teams within their organization and the exchange of detailed network information (IP addresses, VLAN tags, QoS requirements, and security profiles) that needed configuration setup for each application instance.

As such, UPMC looked at the SDN technology market for ways to streamline the provisioning aspects of the network, and to provide an increase in the visibility and

control of the network for the IP team. UPMC tried Nuage Networks SDN solution [8] over a six-month period from May 2013 through October 2013 and has moved forward with the deployment of SDN starting in February 2014.

With traditional (bare metal) server deployments where hosts were deployed for long production lifecycles (three to four years) and network configuration was configured once and then left alone did not apply. With the virtualized compute environment, demands changed to require instant deployment for peak periods and a significant increase in moves and changes to the network. The increased workload suffered from the traditional IT to network team workflow processes and increased the likelihood of human based configuration errors.

During the trial period, UPMC tested the functionality of SDN to provide network virtualization overlays. It also validated the assumptions that SDN's automation and abstraction principles would significantly improve the organization's ability to react to changes driven by its business and improve the service delivery from its IT department.

The implementation of this SDN environment begun in February 2014, with a long-term strategy to expand the SDN environment and to transition the production network onto the SDN based network during the latter half of 2014 and into 2015.

The migration to SDN provides a number of benefits to UPMC. There is a notable gain from overall network efficiency and they have decreased the network configuration time for both applications changes and new deployments.

In Figure 8, an overview of the solution deployed for the UPMC infrastructure is described. The top layer includes the cloud management platform, which controls the datacenter infrastructure in the view of an infrastructure operator.

The underlying plane is the control plane of the datacenter. It communicates directly with the cloud management platform so that all management operations of all infrastructure dictated in this layer are passed through this control plane of the datacenter. In this layer reside the multiple SDN controllers that orchestrate all the network devices that each controller is responsible of managing. Note that there is more than one controller in this plane. All controllers work in a federated mode, which

they maintain all network status and changes in the network synchronized throughout all controllers and, eventually, network elements.

Lastly, the data plane is where all network devices, hosts and the remaining datacenter elements that transfer critical business data between them. This plane is completely orchestrated from the control plane above it, where all controllers reside and communicate with them. Operators do not have access directly to these elements because the federated controllers monitors and manage them on behalf of the operators. This represents one of the main advantages of the SDN implementation, the centralized control of the entire network from one single point, and erases the micromanagement of each network element.

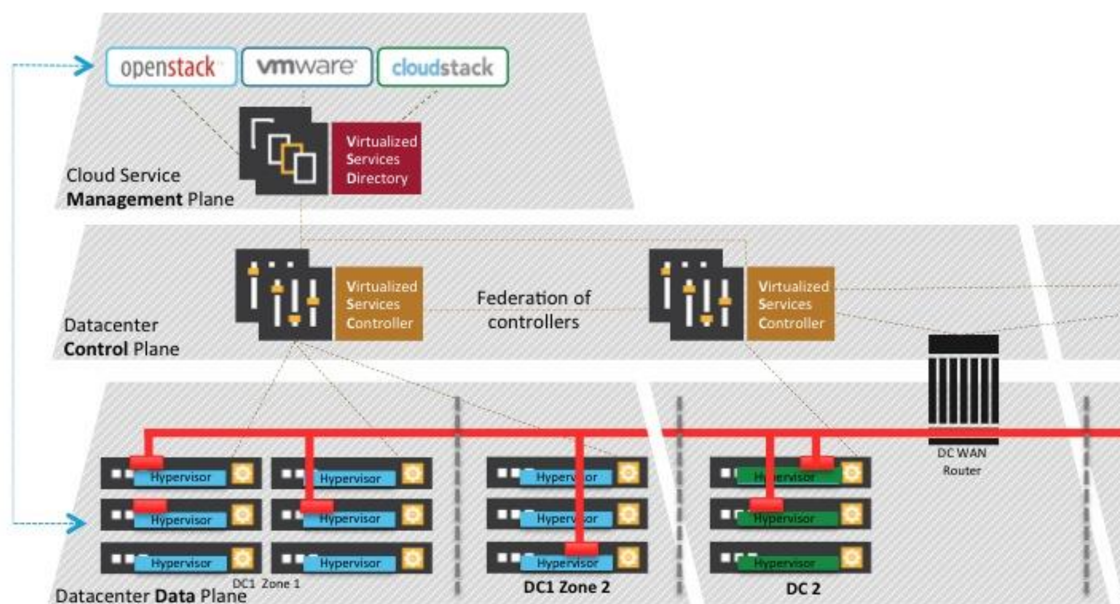


Figure 9: UPMC Software Defined Networking Architecture Solution

3 Implementation

Starting from the point where all the tools needed to perform a SDN simulation are completely configured and ready to be used, it is intended to start with the implementation of a common network architecture to understand the operation of them, as well as try to give a global vision of each tool.

The tool used for the creation of SDN simulations is called Mininet; by introducing console commands, using the Mininet API or by executing scripts in Python language containing the desired topology, as this tool is based on Python.

3.1 Mininet

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking. [9]

Mininet supports research, development, learning, prototyping, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC.

The main features that introduce Mininet are the provisioning of a simple and inexpensive network testing environment for developing OpenFlow applications, it enables developers to work independently on the same topology. It provides a complex topology testing without the need to wire up a physical network and supports arbitrary custom topologies and parametrize them depending on the characteristics on the topology it is intended to test.

It also includes a command-line interface that is topology and OpenFlow aware, for debugging and testing network-wide tests. A Python API is provided for network creation and experimentation.

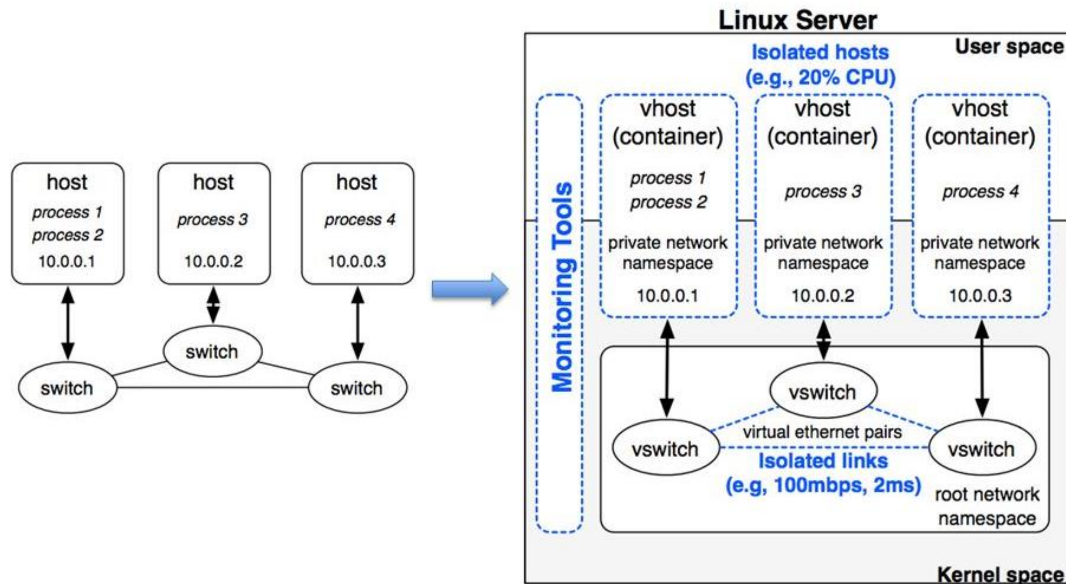


Figure 10: Mininet architecture

Being an open source project and the availability of a Python API included in Mininet, it is easy to develop and extend Mininet with plugins or complements that allows the additions of features for this software. One that will be used in this thesis is the Miniedit extension for Mininet. This extension allows us to design graphically any custom network topology that is going to be created and simulated in a graphical manner that is simple and intuitive to use. After the design, Miniedit is able to translate the graphical design into a python script that Mininet can interpret and run this script for network simulation.

In the following section, the installation and use of both Mininet and Miniedit are described and used for emulating the proposed architectures.

3.2 Environment Setup

It is proposed the creation of a common scenario in Mininet, which will have an external controller (OpenDayLight) to see the topology created and the information that circulates through the network devices. This will be done using the Python language and the Miniedit API.

Mininet will be executed in a virtual machine that is already provided by their creators. It is an Ubuntu Server distribution with all the packages and dependencies needed for the correct operation of the Mininet simulator. It is also needed another Linux virtual machine with a graphical interface for the controller machine, which comes with a web interface and a REST API. The host machine is a Windows 10 with the VMware Workstation Player 12 installed for the execution of Virtual Machines.

Due to the need of interconnectivity between the two guest machines, the VMware Workstation Player needs to be configured with an external interface accessible by the host and the other guest machines. This is configured in the Player with the following configuration:

VM	IP	Interface	Use
Mininet VM	192.16.169.130	eth1	Mininet Simulator
Ubuntu 17.10	192.168.169.129	eth1	OpenDayLight Controller

Table 3: VMware Player virtual machines network configuration

3.3 OpenDayLight controller setup

The controller requires a virtual machine with a graphical user interface as it includes a web manager for displaying the network architecture and manage it. The Linux distribution that will be used is an Ubuntu 17.10 and the latest version of the OpenDayLight controller will be installed on it, which is called Nitrogen (v0.7).

Once the Ubuntu distribution is installed as a virtual machine, it has to be checked that the Internet connection due to the need of obtaining the packages that include all the installation of the controller software. Because the previous network configuration done in the VMware Workstation Player was configured with this need in mind, the virtual machine has internet connection after the installation.

The controller is programmed based on the Java language. It includes the Apache Karaf [10] software architecture, which is a polymorphic container that can host any kind of applications, such as OSGi, Spring, WAR and much more.

Therefore, it is necessary to install the Java run-time by introducing the following commands in the Ubuntu terminal:

$$sudo\ apt-get\ update\ \&\&\ sudo\ apt-get\ install\ default-jre-headless$$

After the installation of Java run-time, it is possible to download the OpenDayLight package with the following command:

$$wget\ https://nexus.opendaylight.org/content/groups/public/org\\ /opendaylight/integration/karaf/0.7.1/karaf-0.7.1.tar.gz$$

After the download, it is necessary to extract the contents of the compressed package:

$$tar-xvzf\ karaf-0.7.1.tar$$

That will create a new folder called ‘karaf-0.7.1’ that contains the OpenDayLight software. As this is all needed to start the controller, it is proceeded to execute the service by typing in the console:

$$./karaf-0.7.1\ /bin/karaf$$

It should appear a message that Karaf is booting up. After the startup sequence has completed, another command-line interface should appear with the OpenDayLight logo. It is necessary to install some dependencies that are needed in order to provide a graphical interface, a RESTful API and the interaction between the controller and the Open vSwitch software. The installation of these dependencies is done with the following command:

$$feature: install\ odl - restconf\ odl - l2switch - switch\ odl - mdsal - apidocs\ odl - dlux - core\ odl - dluxapps - nodes\ odl - dluxapps - topology\ odl - dluxapps - yangui\ odl - dluxapps - yangvisualizer\ odl - dluxapps - yangman$$

The ‘dluxapps’ dependencies are necessary for deploying the web application for the controller. The ‘restconf’ package allows the access to the RESTCONF API, which is the API that allows the configuration of the network from the controller to the network devices. The ‘mdsal-apidocs’ is used for the Yang API. Finally, the ‘l2switch-switch’ packet provides similar functionality to an Ethernet switch to the network.

3.3.1 Controller access and management

After OpenDayLight is completely installed in the virtual machine, it is possible to access to the web application. To perform that, a web browser is enough and write the following URL:

$$http://192.168.169.129:8181/index.html$$

The application exposes the application to the interfaces of the virtual machine at the application default port, which is 8181. When the page is loaded, a login interface will appear.

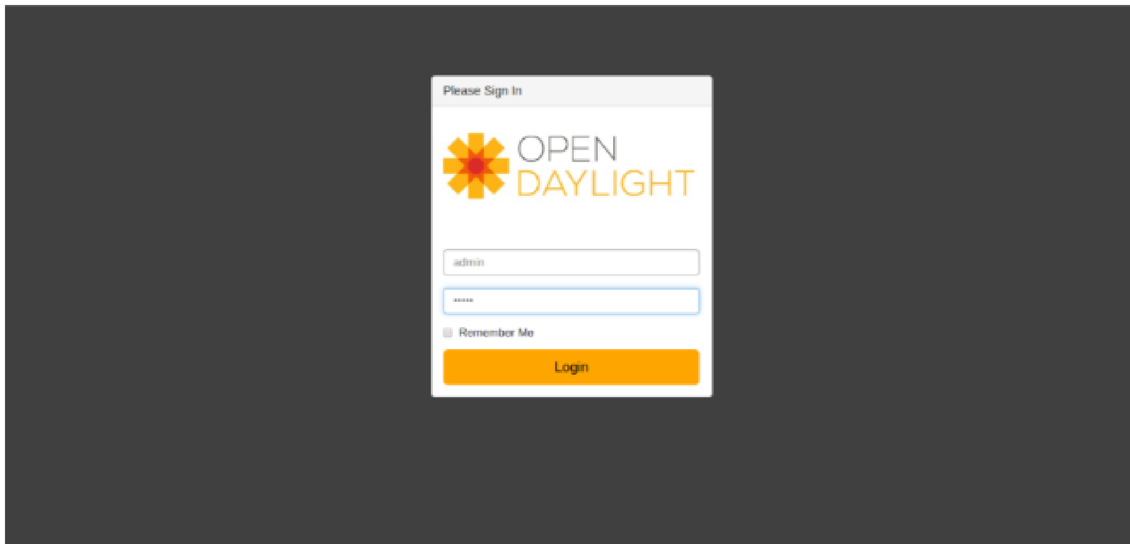


Figure 11: OpenDayLight login interface

The default login credentials are 'admin' for both user and password. Once login has been successful, the web interface is not difficult to use. It is composed of a main work area and left side menu of four sections: Topology, Nodes, Yang UI and Yang Visualizer.

The network topology can be mapped in the Topology section. It is useful if to overview all network connections if the network has been designed without any graphical user interface. It will track any change the network suffers while it is operational.

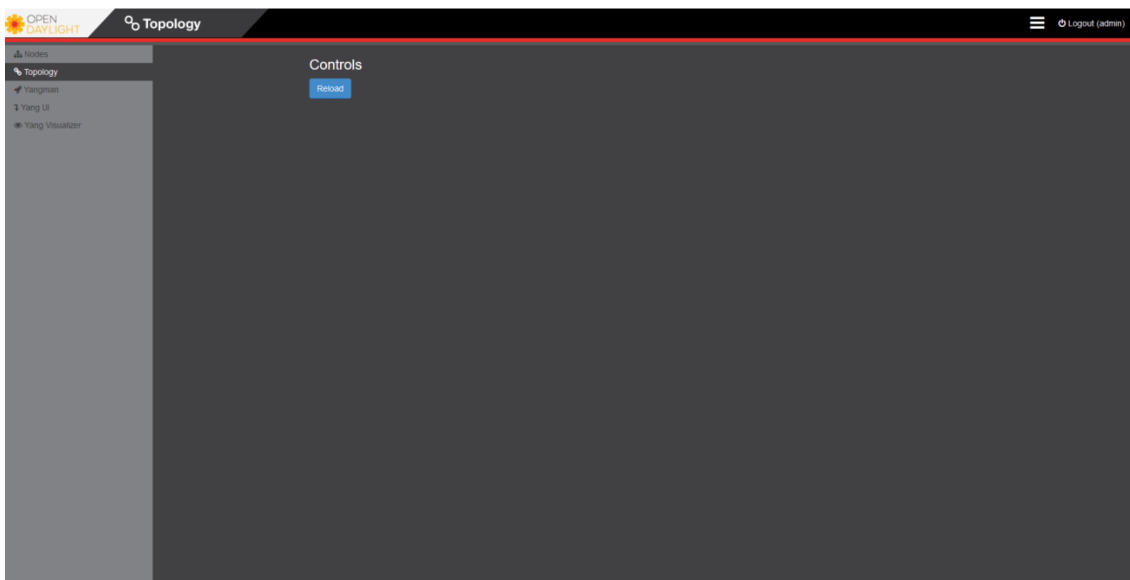


Figure 12: OpenDayLight Topology section

The ‘Nodes’ section allows to track information of every network device connected to the controller, such as network statistics, node connectors and configured flows.

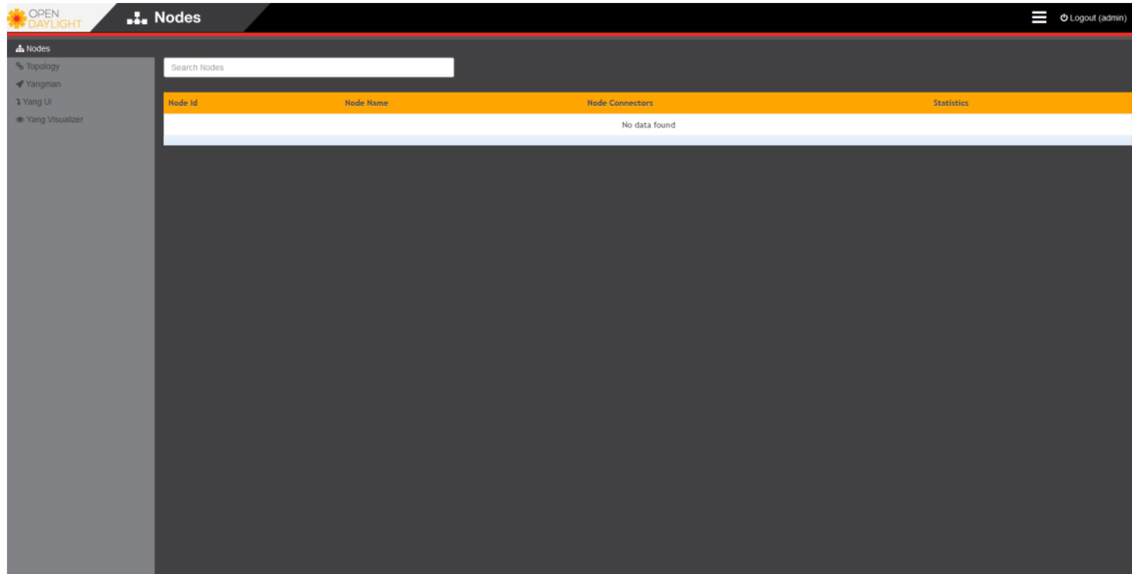


Figure 13: OpenDayLight Nodes section

The last three sections use the Yang API. Yang is a data modeling structure similar to others based on SNMP, SMI and MIB. It provides a functionality to the SDN switches in a similar way as SMI does for the switches that are not SDN. Yang UI is a REST graphic client to configure and sent REST requests to the OpenDayLight local database. It can be used to get information or modify it in the database.

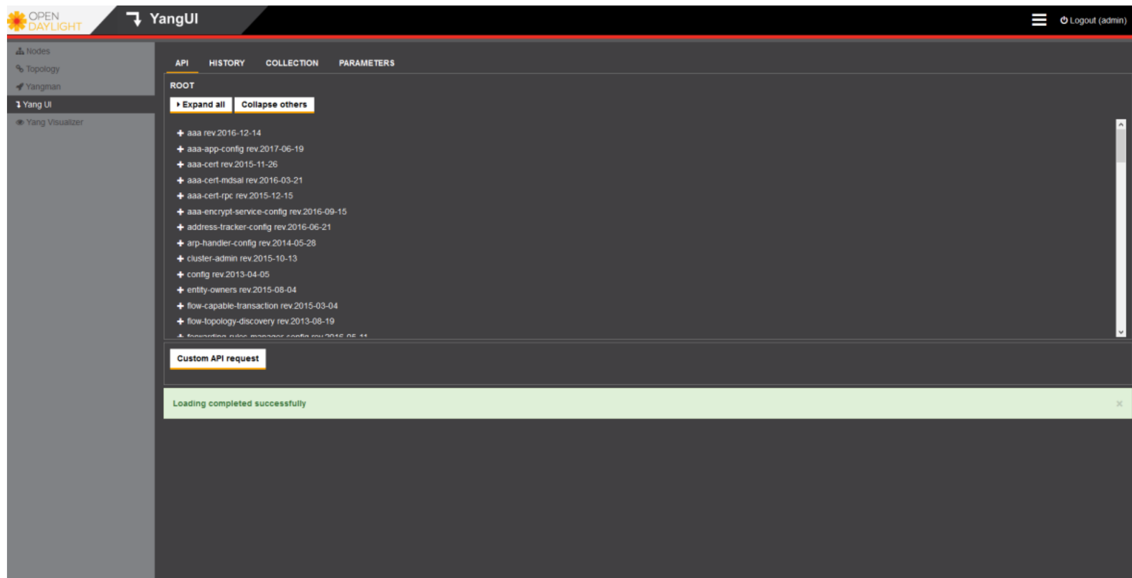


Figure 14: OpenDayLight Yang UI section

This section shows all available APIs within this client from the ‘Expand All’ button. Not all these APIs will work because all the features are not installed. One of those that works is the so-called 'Inventory API', which will be launched by expanding the menu with the name 'inventory' by going to the 'nodes' tab and sending the GET API request to the controller from the 'Send' button. To the bottom, all the information about the network will be displayed: nodes, ports, statistics and much more. If any switch or interface listed is clicked, it will display details of each one of them.

3.4 Network Design

Once we have decided and installed all the tools that are going to be used, it is now proceeded with the design of two network architectures, traditional and Software Defined Networking. The architectures should have a similar pattern in order to compare similar networks in terms of number of network devices yet each architecture uses its main network technology as its core. The routing and devices are also identical or very similar.

As the Mininet VM is set up and running, there are several tools that are useful for achieving the objectives in this project. One of the most useful is the Miniedit tool, which is a graphical user interface developed in python that allows the creation of complex network topologies defined by the user. It allows configuring the network devices and, once the design is completed, exporting the designed topology into a python script for future simulations.

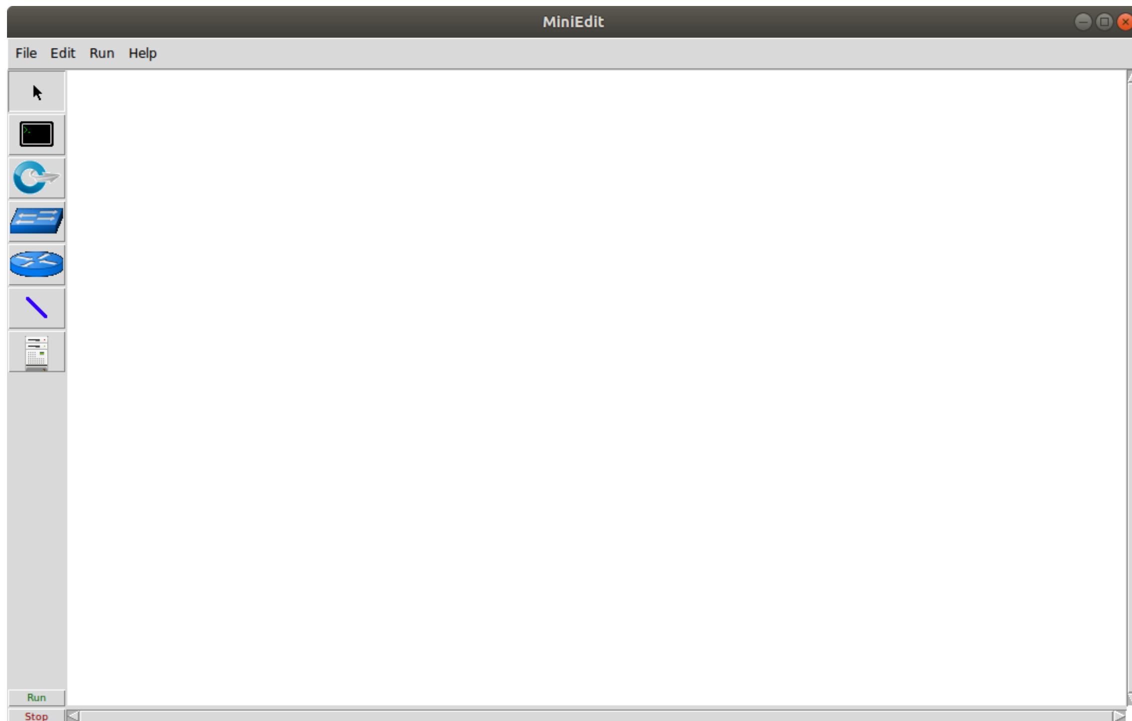


Figure 15: Miniedit graphical interface

It presents a simple interface with a toolbar in the left side of the window and a menu bar in the upper section of the window. The toolbar is filled with multiple tools that allow the design of the network. Most of them are common network elements that are needed for a network implementation, such as routers, switches, links, hosts and other devices. By clicking on an element, it is possible to drag the element to the design area and drop it wherever is wanted.

One of the main objectives of this thesis is to compare the performance between traditional network and SDN architecture; therefore, two network designs are proposed to be evaluated using Miniedit.

3.4.1 SDN network architecture design

The first design consists in a SDN architecture by implementing a controller connected to a set of switches forming a tree topology. The first level is exclusive for the controller, as it is the network orchestrator and monitors all switches connected to it. The controller should be considered as the core of the network. The second level is composed of three switches connected between them. These switches act as the distribution layer of the network, guarantying the redundancy and the availability of the network. The last level consists of four switches, connecting three host machines each one. This level is the access level.

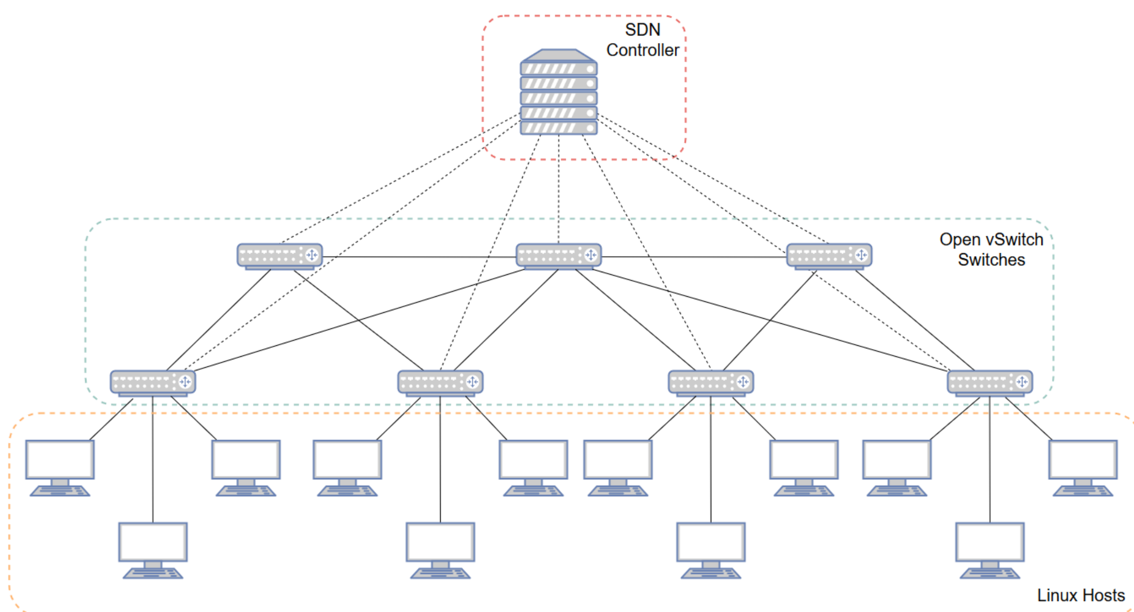


Figure 16: SDN proposed architecture

The SDN controller is orchestrated with OpenDayLight controller. It connects directly to all switches of the network. The switches are implemented with the Open vSwitch specification. This software is designed for having full support for the OpenFlow protocol that all the Software Defined Network is going to use for routing. The hosts are all composed of a Linux operating system.

This design has to be translated to the Miniedit tool in order to perform the simulation, so the architecture designed can be tested in this tool.

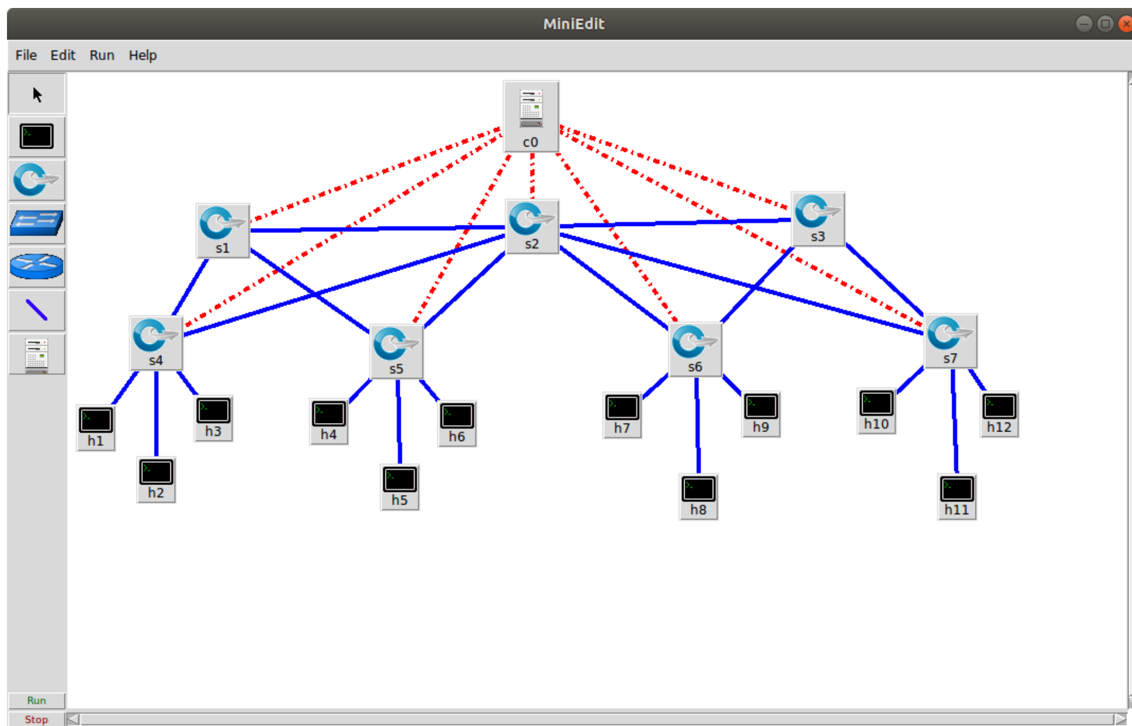


Figure 17: SDN proposed architecture in Miniedit

Once the design is completed, Miniedit implements a feature that can translate the proposed design into a Python script that can be later executed in Mininet. This design is coded in Python using this feature, generating a Python script file. This script, however, should be modified at the controller declaration to specify the IP address and port where the controller service is located.

```
info( '*** Adding controller\n' )
c0=net.addController(name='c0',
                     controller=RemoteController,
                     ip='192.168.169.129',
                     protocol='tcp',
                     port=6633)
```

Figure 18: Controller configuration in python script

Once the change is made in the script, this tests scenario is ready to be executed and simulate it in Mininet.

3.4.2 Standard network architecture design

Due to the need of comparing a SDN and a standard network architecture, an almost similar to the previous proposed SDN architecture is designed for the tests that are going to be performed.

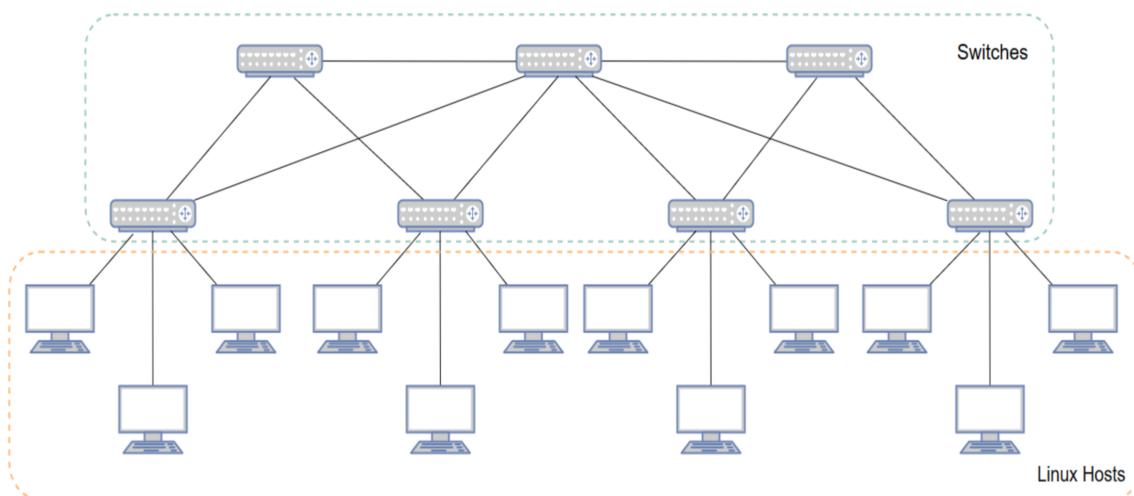


Figure 19: Traditional network proposed architecture

This proposed design has only two levels. The first level is composed of three switches connected between them. These switches act as the distribution layer of the network, guarantying the redundancy and the availability of the network. The last level consists of four switches, connecting three host machines each one. Each switch of this design is a 'legacy' switch in Miniedit. That is, a switch that does not implement any of the SDN oriented protocols. In this case, these switches implement all of the protocols except OpenFlow.

This proposed architecture is translated in the Miniedit tool.

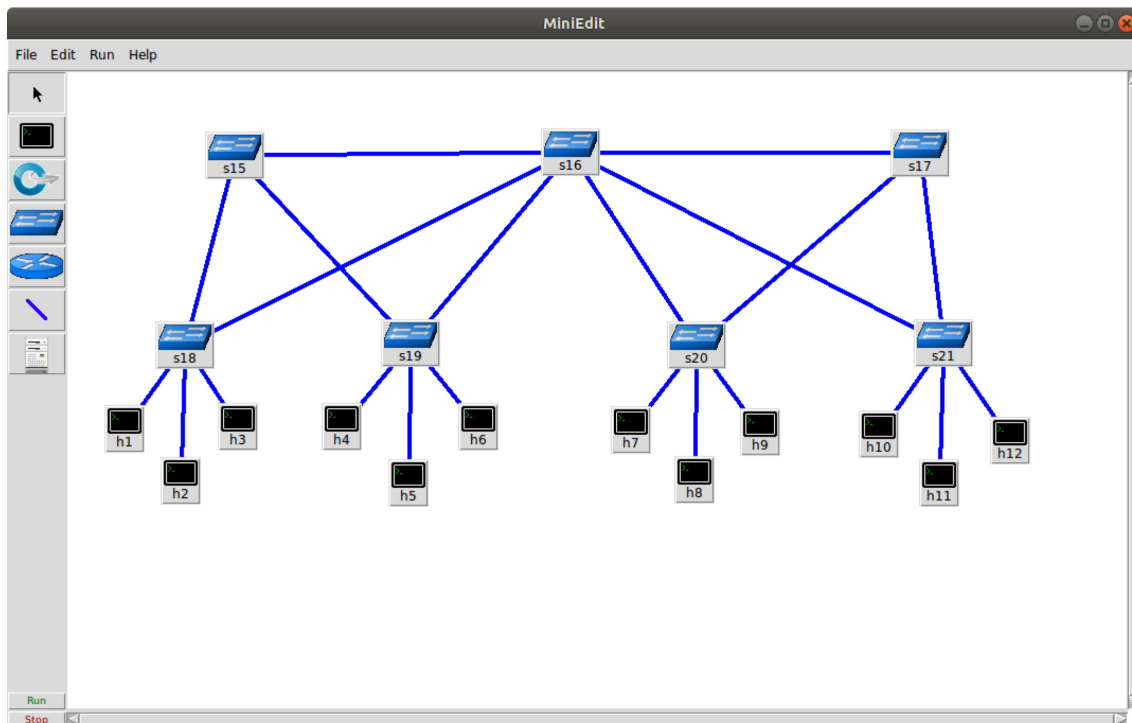


Figure 20: Standard network proposed architecture

Note the absence of the controller. In traditional networks, there is not a centralized control of the whole network and each switch has to be configured manually. The software that runs the switches is most likely to be proprietary, due to the lack of open source viable alternatives, such as Cisco Nexus switches or Juniper. In this case, a simple virtualization in the same Ubuntu system satisfies our needs. Finally, the hosts are the same as the Software Defined Networking architecture.

3.5 Connecting Mininet with OpenDayLight controller

Once both network designs are completed, it is possible to start up both the controller and Mininet services and link them to test the SDN network. Note that this step is only needed in the SDN architecture.

The first step is to start up the OpenDayLight controller in the virtual machine. This has been explained in section 3.2. After that, the Mininet virtual machine is started to simulate the SDN proposed network for testing purposes.

The following command is typed in the Mininet virtual machine terminal to begin the simulation:

$$sudo\ python\ tfmtopoSDN.py$$

This produce the following output, which shows that the simulation creation went well:

```
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h9 h3 h4 h11 h7 h2 h6 h1 h10 h12 h8 h5
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
```

Figure 21: Mininet simulation environment creation output

After the creation of the network devices, the controller should have got the basic information from the switches that are directly connected to it:

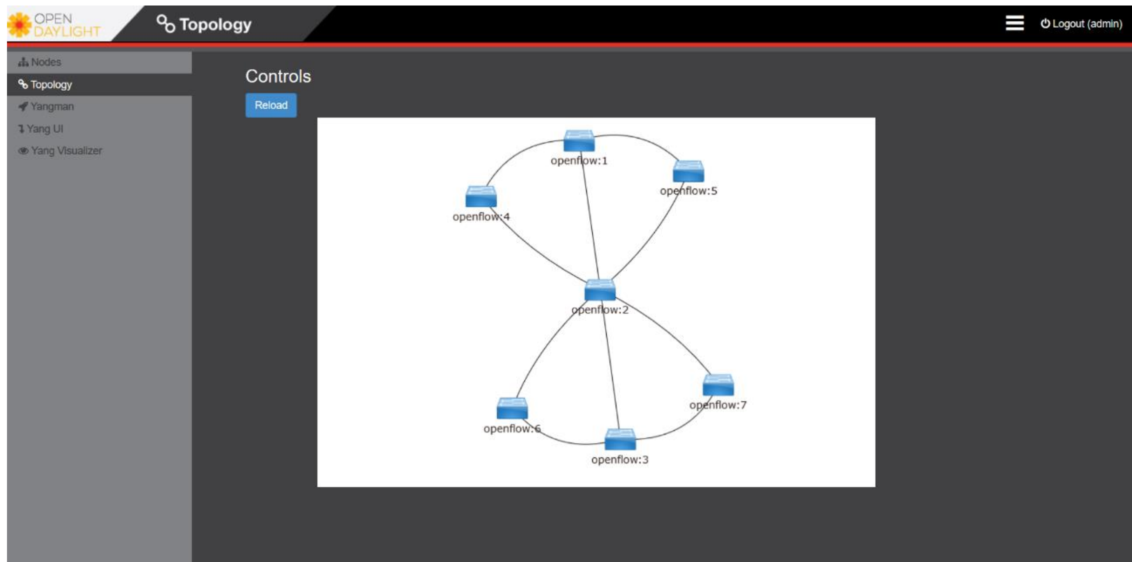


Figure 22: OpenDayLight controller first network devices detection

The controller knows the first status of the network but does not know what elements are connected to the switches. This is due to there is still no activity in the network yet. The elements will show in the controller once a minimum traffic flows through the network and the controller will then know the status of these elements. This is done by typing the following command in the Mininet virtual machine:

$$mininet > pingall$$

This command executes ping commands from all host machines to the other host machines, so that every host machine connects to every other host machine.

```
mininet> pingall
*** Ping: testing ping reachability
h9 -> h3 h4 h11 h7 h2 h6 h1 h10 h12 h8 h5
h3 -> h9 h4 h11 h7 h2 h6 h1 h10 h12 h8 h5
h4 -> h9 h3 h11 h7 h2 h6 h1 h10 h12 h8 h5
h11 -> h9 h3 h4 h7 h2 h6 h1 h10 h12 h8 h5
h7 -> h9 h3 h4 h11 h2 h6 h1 h10 h12 h8 h5
h2 -> h9 h3 h4 h11 h7 h6 h1 h10 h12 h8 h5
h6 -> h9 h3 h4 h11 h7 h2 h1 h10 h12 h8 h5
h1 -> h9 h3 h4 h11 h7 h2 h6 h10 h12 h8 h5
h10 -> h9 h3 h4 h11 h7 h2 h6 h1 h12 h8 h5
h12 -> h9 h3 h4 h11 h7 h2 h6 h1 h10 h8 h5
h8 -> h9 h3 h4 h11 h7 h2 h6 h1 h10 h12 h5
h5 -> h9 h3 h4 h11 h7 h2 h6 h1 h10 h12 h8
*** Results: 0% dropped (132/132 received)
```

Figure 23: Pingall command output

By doing this, the controller would have a minimal traffic generated by each host and would map the current connections of the network. This change is appreciated in the OpenDayLight controller:

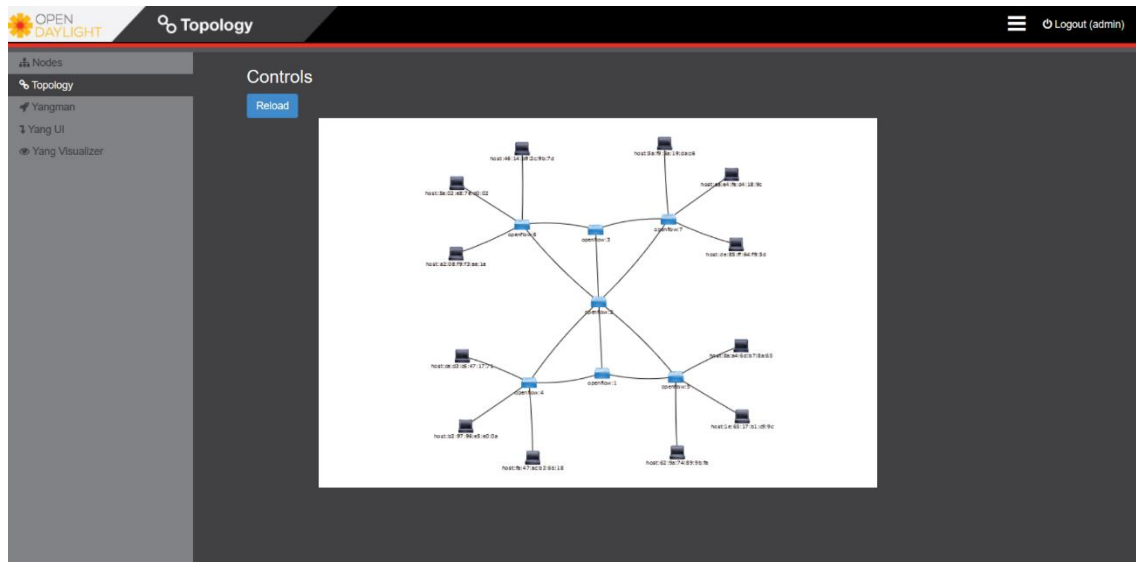
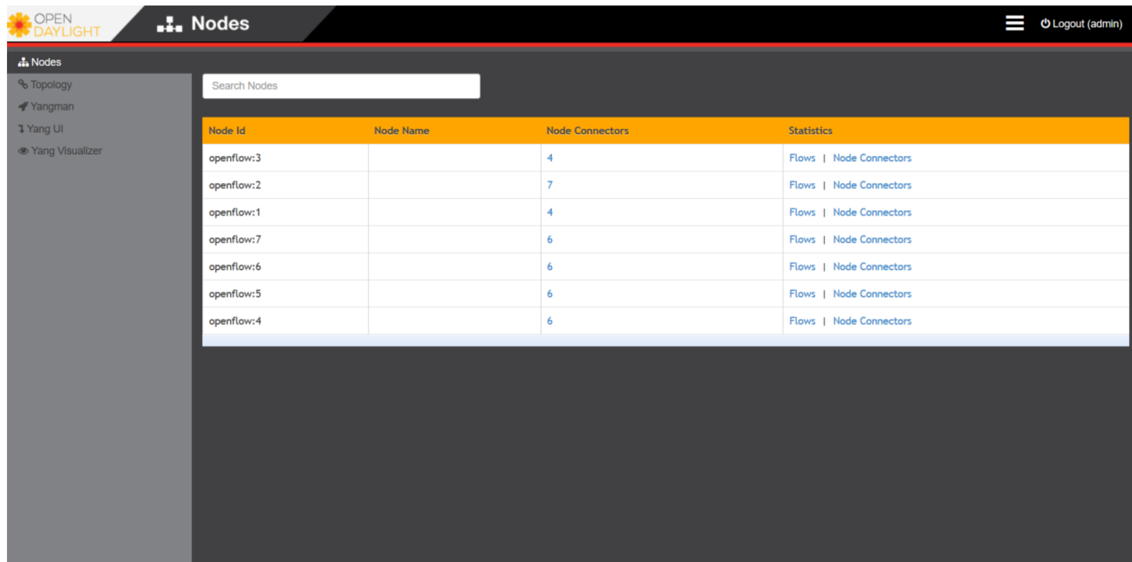


Figure 24: OpenDayLight controller detection of all network elements

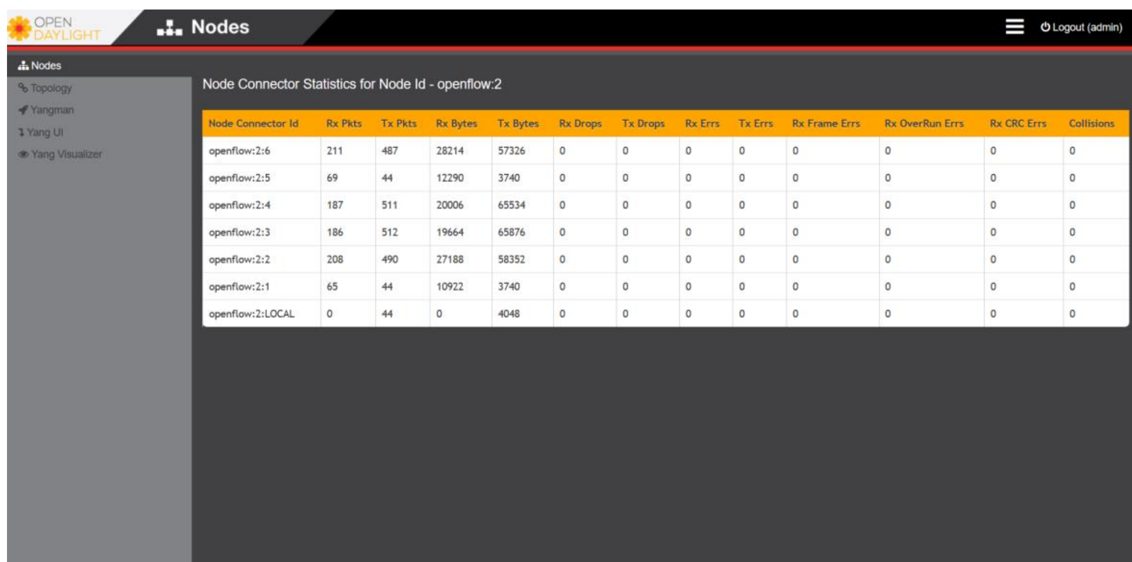
The ‘Nodes’ section shows information about the seven simulated switches, as shown in the next image:



Node Id	Node Name	Node Connectors	Statistics
openflow:3		4	Flows | Node Connectors
openflow:2		7	Flows | Node Connectors
openflow:1		4	Flows | Node Connectors
openflow:7		6	Flows | Node Connectors
openflow:6		6	Flows | Node Connectors
openflow:5		6	Flows | Node Connectors
openflow:4		6	Flows | Node Connectors

Figure 25: Nodes section with switch list

Clicking on any 'Node Connectors' of any listed switch, the controller will show traffic statistics that flows through that node.



Node Connector Id	Rx Pkts	Tx Pkts	Rx Bytes	Tx Bytes	Rx Drops	Tx Drops	Rx Errs	Tx Errs	Rx Frame Errs	Rx OverRun Errs	Rx CRC Errs	Collisions
openflow:2:6	211	487	28214	57326	0	0	0	0	0	0	0	0
openflow:2:5	69	44	12290	3740	0	0	0	0	0	0	0	0
openflow:2:4	187	511	20006	65534	0	0	0	0	0	0	0	0
openflow:2:3	186	512	19664	65876	0	0	0	0	0	0	0	0
openflow:2:2	208	490	27188	58352	0	0	0	0	0	0	0	0
openflow:2:1	65	44	10922	3740	0	0	0	0	0	0	0	0
openflow:2:LOCAL	0	44	0	4048	0	0	0	0	0	0	0	0

Figure 26: Traffic statistic of a selected switch

After all that setup, the SDN environment is ready to perform the tests.

3.6 Standard network simulation

The standard network architecture environment is more simple than the SDN architecture environment. In this case, the Mininet virtual machine is the only machine that will be used, since the network does not have the controller device. Thus, the OpenDayLight virtual machine will not be used.

The following command is typed in the Mininet virtual machine to begin the simulation of the standard network architecture:

$$sudo\ python\ tfmtopoNOSDN.py$$

Mininet will begin the simulation process. A similar output will be displayed when the SDN network was being simulated.

If Mininet does not encounter any problem during the simulation setup, the connectivity of all network devices is tested with the 'pingall' command:

$$mininet > pingall$$

If all packets have been received without any dropped packet, the standard network environment is ready to be tested.

4 Results

There are several metrics in networking architecture that allow comparing different architectures designs in terms of performance. Most of the time, the performance is modeled and simulated instead of measured, which is one of the aims of this thesis.

Some measures are often considered important and comparable through all network designs. In this thesis, the measures that are emphasized are the latency and the throughput.

4.1 Throughput

It is normally understood as throughput as the amount of data transferred from one place to another or processed in a specified amount of time. Data transfer rates for networks are measured in terms of throughput. This measure is typically measured in kbps, Mbps and Gbps. The throughput is also defined as the actual speed of data transport through a network and it will always be less than the bandwidth, which is the theoretical channel capacity. This is due to the limitations of underlying physical medium, available processing power of the system components and end-user behavior.

Mininet provides a command that evaluates the throughput performance between network devices in a simulated network. The most used is the tool called 'iperf', which evaluates the TCP bandwidth between network elements.

The throughput test will consist of different 'iperf' experiments between two hosts, beginning from the host h1, which is one host located at one extreme of the network, to another hosts located in the network until the host located at the other extreme of the network. In this network, the tests are h1->h3, h1->h6 and h1->h12.

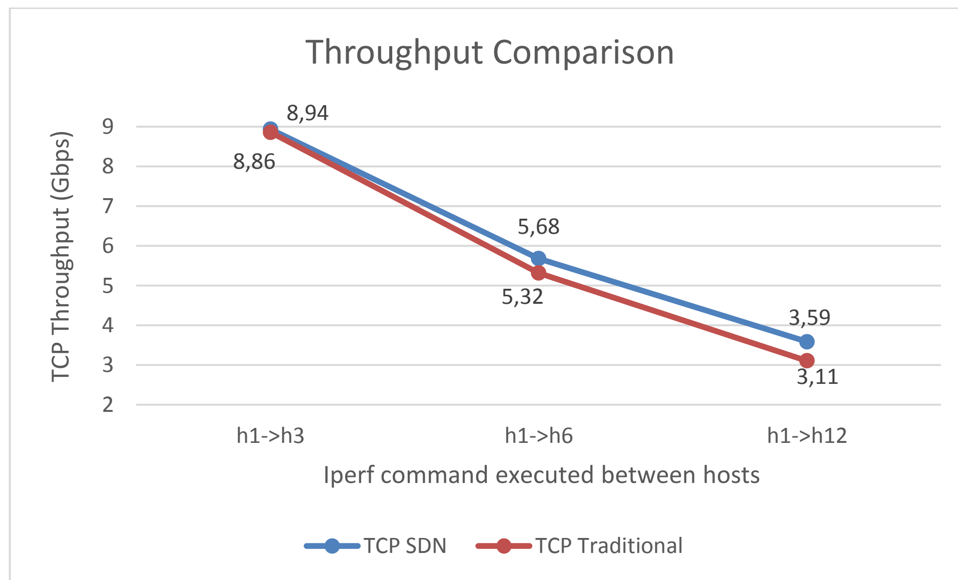


Figure 27: Throughput measure between different hosts

The results that are shown in the previous figure proves that in all cases the SDN network performs better in terms of throughput compared to the traditional network architecture.

4.2 Latency

Latency is a time delay between the cause and the effect of some physical change in the system being observed. In a network environment, the latency is measured either one-way (the time from the source sending a packet to the destination receiving it), or round-trip delay time (the one-way latency from source to destination plus the one-way latency from the destination back to the source).

Many software platforms provide a service called 'ping' that can be used to measure round-trip latency. 'Ping' performs no packet processing; it merely sends a response back when it receives a packet. Due to the use of ICMP protocol, Ping cannot perform accurate measurements and differs from real communication protocols such as TCP.

The latency test will consist of different 'ping' experiments between two hosts, beginning from the host h1, which is one host located at one extreme of the network, to another hosts located in the network until the host located at the other extreme of the network. In this network, the tests are h1->h3, h1->h6 and h1->h12.

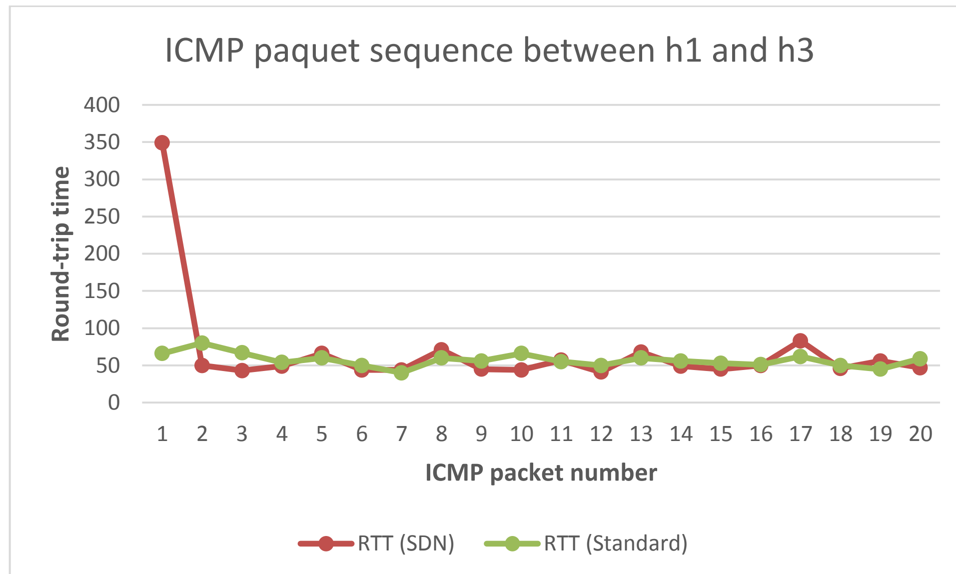


Figure 28: Latency measure between h1 and h3 in both network environments

In this experiment environment it is observed that the delay of the first ICMP packet in the SDN network is always much higher than that of the traditional network. This difference is because in SDN, when the first packet sent by host h1 arrives at the switch, this network element does not know how to route it, encapsulates it in the OpenFlow protocol and forwards all the contents of the incoming packet to the controller, being responsible for managing the installation of the flow tables in each switch. This initial process of establishing flow consumes a time that introduces a latency in the network.

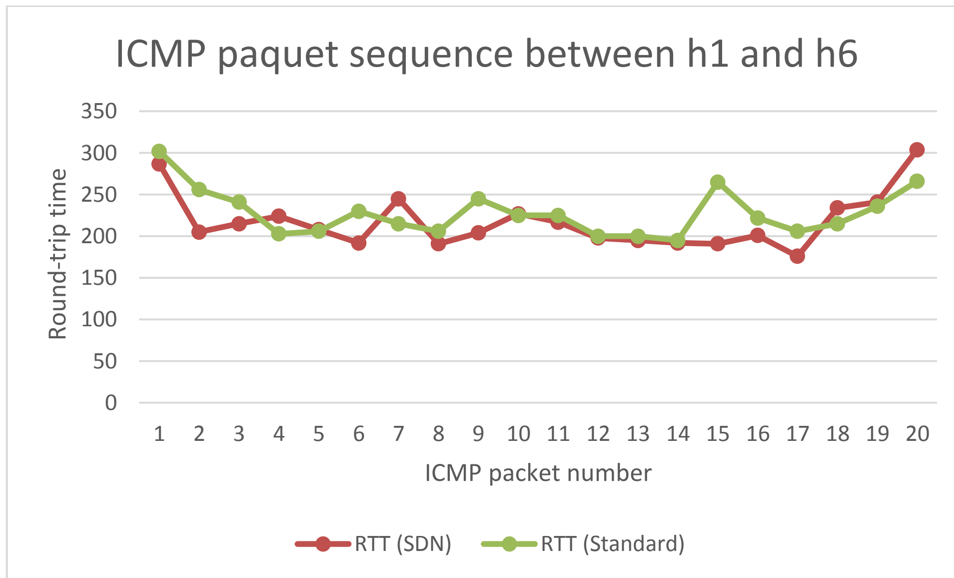


Figure 29: Latency measure between h1 and h6 in both network environments

The effect related in the previous experiment is repeated in this experiment but in this case, the traditional network introduces much higher latency. This is due to the latency that introduces the hops the packets must pass through the switches that are connected, in this case three switches. The SDN architecture also suffers from this hops but lesser than the traditional network.

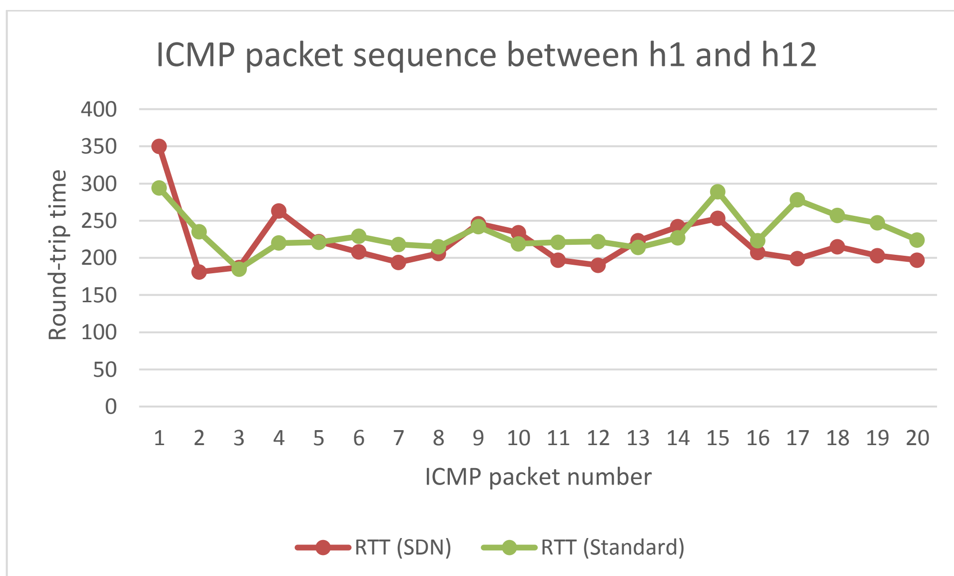


Figure 30: Latency measure between h1 and h12 in both network environments

As the ping increments in distance of hops, the ping is greater for both networks. Nevertheless, the traditional network suffers more than the SDN architecture. This is because in an SDN network, after the flows are installed, the routes to route the frames are stored in the memory to achieve better performance. In a traditional network it is necessary to analyze all the incoming frames in each switch, as part of the process of learning MAC addresses as well as updating an internal database in the switch that has the physical address of the host and the source port.

5 Conclusions

The SDN are a still developing technology that, as we have seen throughout this work, pretend to be the solution to the problems of the current network structures, which are appearing due to an increasing demand for resources by the emergence of new technologies.

The current growth and the estimated demands for the coming years of information technologies and telecommunications require major changes in the infrastructures of current networks. The Software Defined Networks are the alternative that is envisaged to face the current and future challenges in the telecommunications networks.

Trends such as user mobility, server virtualization, cloud-based infrastructure, IT-as-a-Service, and the need rapidly to respond to changing business conditions place significant demands on the network demands that today's conventional network architectures can't handle. Software-Defined Networking provides a new, dynamic network architecture that transforms traditional network backbones into rich service-delivery platforms.

By decoupling the network control and data planes, OpenFlow-based SDN architecture abstracts the underlying infrastructure from the applications that use it, allowing the network to become as programmable and manageable at scale as the computer infrastructure that it increasingly resembles. An SDN approach fosters network virtualization, enabling IT staff to manage their servers, applications, storage, and networks with a common approach and tool set. Whether in a carrier environment or enterprise data center and campus, SDN adoption can improve network manageability, scalability, and agility.

The future of networking will rely more and more on software, which will accelerate the pace of innovation for networks as it has in the computing and storage domains. SDN promises to transform today's static networks into flexible, programmable platforms with the intelligence to allocate resources dynamically, the scale to support

enormous data centers and the virtualization needed to support dynamic, highly automated, and secure cloud environments. With its many advantages and astonishing industry momentum, SDN is on the way to becoming the new norm for network.

This thesis has focused on comparing the current network architectures with the emerging SDN topologies, integrating the new network devices this new architecture introduces, such as the OpenDayLight controller, and test and compare their performance in terms of latency and throughput with simulation software tools as Mininet.

The use of latest technologies is one of the strengths of this thesis, as the technologies applied in this thesis are the ones to be implemented in the main technological companies in the near future. On the other hand, one of the possible weaknesses of this thesis could be the non-deepening of the OpenFlow protocol and not knowing how to take advantage of the capabilities of this protocol to obtain better performance or new features in SDN networks.

5.1 Future lines of work

As this work focused solely on comparing both architectures, there has been a series of tasks that would have been interesting to perform, which were beyond the scope of this thesis. Deepen the knowledge of the OpenFlow protocol could have led to interesting configurations and features to the SDN architecture, which improves the overall capacities compared to traditional network architectures. Investigating more about the capacities of a SDN controller and the unique capabilities it offers is also an interesting future line of work.

6 Budget

All work done on this thesis has consisted in implementing a simulation environment for performance testing. The economic impact found in this thesis has been an estimation of hours dedicated to the realization of this project, evaluated at a price of junior engineer (10€ / hour) and the licenses needed for the software used. In this case, the software used is open source and free of use, so there is no license costs in this thesis.

Task	Hours (estimated)	Cost
Documentation and Information Research	50	500.00 €
Environment design and Implementation	140	1,400.00 €
Simulation and tests	90	900.00 €
Total	**250**	**2,500.00 €**

Table 4: Budget estimation

7 References

[1] Open Network Foundation. *Software-Defined Networking: The New Norm for Networks.* ONF White Paper. April 13, 2012. Available at: [http://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf]. Accessed 13 November 2017

[2] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller and Navneet Rao. Queen's University Belfast, Cisco Systems, Tabula and Netronome. *Are We Ready for SDN? Implementation Challenges for Software-Defined Networks*. Future Carrier Networks, IEEE Communications Magazine, July 2013, Pages 36-43. Available at: [https://ieeexplore.ieee.org/abstract/document/6553676/]

[3] Open Network Foundation. *OpenFlow Switch Specification.* ONF White Paper. December 31, 2009. Available at: [https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf]. Accessed 15 November 2017

[4] Usenix, The Advanced Computing Systems Association, Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, and Pravin Shelar. *The Design and Implementation of Open vSwitch*. May 4 – May 6, 2015. Available at: [https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf]. Accessed 25 November 2017.

[5] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado and Rob Sherwood. *On Controller Performance in Software-Defined Networks*. Usenix, The Advanced Computing Systems Association. 2013. Available at: [https://www.usenix.org/system/files/conference/hot-ice12/hotice12-final33_0.pdf]

[6] The Linux Foundation Projects. OpenDayLight. Platform Overview. Available at: [https://www.opendaylight.org/what-we-do/odl-platform-overview]. Accessed 15 November 2017

[7] Gigaom, Ben Kepes. *Analyst Report: SDN meets the real world: implementation benefits and challenges*. March 25, 2014. Available at: [https://gigaom.com/report/sdn-meets-the-real-world-implementation-benefits-and-challenges]. Accessed 14 December 2017.

[8] Nuage Networks, Nokia. *Nuage Networks Provides Comprehensive Software-Defined Networking (SDN) Solution to Leading Health Provider UPMC*. January 16, 2014. Available at: [http://www.nuagenetworks.net/news/nuage-networks-provides-comprehensive-software-defined-networking-sdn-solution-leading-health-provider-upmc/]. Accessed 14 December 2017.

[9] Mininet. Mininet, An Instant Virtual Network on your Laptop (or other PC). 2017. Available at: [http://mininet.org/]. Accessed 5 October 2017.

[10] Apache Karaf, The Apache Software Foundation. *The Apache Karaf Open Source Project.* 2017. Available at: [https://karaf.apache.org/]. Accessed 23 November 2017.

8 Annexes

In this section, the code that has been used in order to perform the simulation is described. There are two code versions, the first one is the code used for simulating the SDN network architecture and the second code is the legacy network architecture.

8.1 SDN network architecture code

In this code all elements of the network are declared and configured using the Python language functions and API explicitly designed for Mininet.

First, an instantiation of a Mininet network is declared and the principal characteristics of the network are also declared. In this case, the topology is custom so the *topo* parameter is set to *None*. Then an *ipBase* is set in order to define the routing of the entire network. This is the network routes that would be assigned to the network elements.

Then, a declaration of the Controller is followed. Note the IP where is located. This is the IP used in the Ubuntu machine described in the implementation section where the controller is running. The type of controller is set to *RemoteController* in order to indicate the Mininet not to virtualize the controller and connect to an external controller.

Finally, the network elements such as switches and hosts are declared, connected and configured. After this declaration, the switches and the controller are started and the whole network is starting to run and ready to be tested.´

```
#!usr/bin/python

from subprocess import call
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
```

```
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf

def myNetwork():

    net = Mininet(topo=None, build=False, ipBase='10.0.0.0/8')

    info('*** Adding OpenDayLight controller\n')

    c0 = net.addController(name='c0', controller=RemoteController,
ip='192.168.169.129', protocol='tcp', port=6633)

    info('*** Adding switches\n')

    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
    s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
    s8 = net.addSwitch('s8', cls=OVSKernelSwitch)
    s9 = net.addSwitch('s9', cls=OVSKernelSwitch)

    info('*** Adding hosts\n')

    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
    h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
    h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
    h7 = net.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
    h8 = net.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)

    info('*** Adding links\n')

    net.addLink(h1, s1)
    net.addLink(s1, h2)
    net.addLink(h3, s2)
    net.addLink(s2, h4)
    net.addLink(h5, s3)
    net.addLink(s3, h6)
    net.addLink(s1, s5)
    net.addLink(s5, s2)
    net.addLink(s2, s6)
    net.addLink(s6, s3)
    net.addLink(s3, s7)
```

```
    net.addLink(s7, s4)
    net.addLink(s4, h7)
    net.addLink(s4, h8)
    net.addLink(s9, s6)
    net.addLink(s9, s7)
    net.addLink(s5, s8)
    net.addLink(s8, s6)

    info('*** Starting network\n')

    net.build()

    info('*** Starting Controllers\n')
    for controller in net.controllers:
        controller.start()

    info('*** Starting switches\n')

    net.get('s1').start([c0])
    net.get('s2').start([c0])
    net.get('s3').start([c0])
    net.get('s4').start([c0])
    net.get('s5').start([c0])
    net.get('s6').start([c0])
    net.get('s7').start([c0])
    net.get('s8').start([c0])
    net.get('s9').start([c0])

    info('*** Post configure switches and hosts\n')
```

8.2 Legacy Network architecture code

In this code, an almost identical logic is developed as in the SDN simulation code. The most outstanding change is the absence of the controller and, therefore, the instantiation of this element is no longer required. Apart from this change, the simulation code respects the same logic from the previous simulation code.

```
#!usr/bin/python

from subprocess import call
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
```

```
from mininet.link import TCLink, Intf

def myNetwork():

    net = Mininet(topo=None, build=False, ipBase='10.0.0.0/8')

    info('*** Adding switches\n')

    s1 = net.addSwitch('s1', cls=UserSwitch)
    s2 = net.addSwitch('s2', cls=UserSwitch)
    s3 = net.addSwitch('s3', cls=UserSwitch)
    s4 = net.addSwitch('s4', cls=UserSwitch)
    s5 = net.addSwitch('s5', cls=UserSwitch)
    s6 = net.addSwitch('s6', cls=UserSwitch)
    s7 = net.addSwitch('s7', cls=UserSwitch)
    s8 = net.addSwitch('s8', cls=UserSwitch)
    s9 = net.addSwitch('s9', cls=UserSwitch)

    info('*** Adding hosts\n')

    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
    h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
    h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)
    h7 = net.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
    h8 = net.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)

    info('*** Adding links\n')

    net.addLink(h1, s1)
    net.addLink(s1, h2)
    net.addLink(h3, s2)
    net.addLink(s2, h4)
    net.addLink(h5, s3)
    net.addLink(s3, h6)
    net.addLink(s1, s5)
    net.addLink(s5, s2)
    net.addLink(s2, s6)
    net.addLink(s6, s3)
    net.addLink(s3, s7)
    net.addLink(s7, s4)
    net.addLink(s4, h7)
    net.addLink(s4, h8)
    net.addLink(s9, s6)
    net.addLink(s9, s7)
    net.addLink(s5, s8)
    net.addLink(s8, s6)
```

```
info('*** Starting network\n')

net.build()

info('*** Starting switches\n')

net.get('s1').start([c0])
net.get('s2').start([c0])
net.get('s3').start([c0])
net.get('s4').start([c0])
net.get('s5').start([c0])
net.get('s6').start([c0])
net.get('s7').start([c0])
net.get('s8').start([c0])
net.get('s9').start([c0])

info('*** Post configure switches and hosts\n')
```