Performance Evaluation using Multiple Controllers with Different Flow Setup Modes in the Software Defined Network Architecture.

by

Suad El-Geder

A thesis submitted for the degree of

Doctor of Philosophy

Department of Electronic and Computer Engineering

College of Engineering, Design and Physical Sciences

Brunel University London

January 2017

Abstract

In this thesis, a scheme of using multiple controllers which handle multiple network devices has been proposed, while using OpenFlow controllers in the proactive operations paradigm, and this in order to face the problem of using a single controller in the SDN model, including the lack of reliability and scalability on such a model. The main characteristic of this new approach are focused on the ability to design a dynamic and highly programmable network, moving the intelligence from the underlying systems to the network itself through a controller. To evaluate the proper effects of this new approach, different dynamic and programmable networks that could simulate real scenarios and measure their performance contrasting the obtained results with the pragmatic theory has been implemented. The SDN (Software-Defined Network) controller (Open Daylight), has been utilized, and thoroughly examined.

Different sort of nets has been worked out through diverse Open Daylight functionalities, either implementing the intelligence of the controller (bundle), or going through it by an outside intelligent application (External Orchestrator), and eventually sending it through Open Daylight (by making Open Daylight work as an interpreter/translator from its language to OpenFlow or another protocol language).

Summing up, the scheme that has been proposed in this research which is the multiple-proactive mode approach and the single proactive controller has scored no packet loss at all, in which implies the strength of reliability of this scheme, while the multiple reactive mode approach has a range of 1-8% packet loss ratio and the single reactive mode approach has a range of 1-25% packet loss ratio. Also, in case of delay the improvement which was obtained from our approach scored an average reduction of 13% comparing with other tested schemes. Thus, these new and interesting technologies show an astonishing capability to add more efficiency in different types of Networks.

Table of Contents

Abstract ii

Table of Contents iii

Acknowledgements vi

Declaration vii

List of Figures viii

List of Tables xi

List of Abbreviations xii

Chapter 1 Introduction 1

1.1 Background: Networks infrastructure developments 1
1.2 Motivations 3
1.3 Aim and Objectives 4
1.4 Contributions 4
1.5 Thesis Structure 5

Chapter 2 Literature Review 7

2.1 Virtualization Concept 7
2.1.1 Types of virtualization 8
2.1.2 Virtual infrastructure 9
2.1.3 Relationship of Network Virtualization to SDN 10
2.2 Software-Defined Networking 12
2.2.1 SDN Architecture and Components 12
2.2.2 *SDN Switches* 14
2.2.3 *SDN Controllers* 16
2.2.4 Centralization of control in SDN 18
2.2.5 SDN Challenges 20
2.2.6 SDN in the Industry 21
2.2.7 SDN Programming Interfaces 23
2.3 Openflow overview 24
2.3.1 *OpenFlow Protocol* 25
2.3.2 *OpenFlow Protocol Messages* 27
2.3.3 The OpenFlow Flow Table. 29
2.4 Summary 34

Chapter 3 OpenDayLight Controller 36
3.1 Introduction to the controller 36
3.1.1 OpenDaylight primary phases 36
3.1.2 Technology Overview 37
3.1.3 Model View Controller (MVC) platform 37
3.1.4 Fundamental Software Tools 38
3.2 Advantages in front of other controllers 39
3.3 The Structure 40
3.3.1 Two different architectures (AD-SAL and MD-SAL) 42
3.3.2 Packet path in OpenDaylight managing Open- Flow devices 44
3.4 OpenDaylight Communication Technique. 49
3.5 Summary 51
Chapter 4 Multiple-Controller with different operation paradigm 52
4.1 SDN Overview 52
4.2 OpenFlow Protocol 53
4.3 State of the Art 54
4.4 Plan Choices for OpenFlow-Based SDN 55
4.4.1 Physically vs. Logically Centralized in the SDN Control Plane 55
4.4.2 In-Band vs. Out-of-Band Signaling in Control Plane 57
4.4.3 Multiple Controllers in Control Plane 59
4.4.4 Proactive vs. Reactive in Management of Flow Entries 61
4.5 Methodology and Network Design 64
4.6 Proposed scheme 65
4.6.1 Topology 65
4.6.2 Operation steps 66
4.6.3 Network Scenarios 67
4.6.4 Network configuration 69
4.6.5 Evaluation suite for Mininet 70
4.7 Experimental Results and Discussion 71
4.8 Summary 76
Chapter 5 Virtual Tenant Network with Multiple SDN Controller Coordination 77
5.1 Virtual Tenant Network Overview 77
5.2 OpenDaylight Virtual Tenant Network (VTN) 78
5.2.1 VTN architecture 79
5.2.2 Virtual Network Construction 80
5.2.3 vBridge Functions 80
5.2.4 vRouter Functions 81
5.2.5 Mapping of Physical Network Resources 82

5.3 Methodology and Network Design 82
5.4 Proposed scheme 83
5.4.1 *Architecture* 83
5.4.2 Operation steps 83
5.4.3 Network Scenarios 84
5.4.4 Network Configuration 85
5.5 Evaluation results and Discussion 87
5.6 Summary 90
Chapter 6 Conclusions and Future work 91
6.1 Conclusions 91
6.2 Future Work 92
References 94
Appendix A: The python code for the multiple controllers with linear network configuration 101

Acknowledgements

First, I would like to express my deep and sincere gratitude to my supervisor Prof. Hamed Al-Raweshidy, whom I am greatly indebted to for the continuous advices and guidance during this research. I appreciate his patience, support, and encouragement, in which this research would not be completed without. I am very grateful for him for giving me the opportunity to do and achieve this PhD. Also, thanks to my second supervisor Dr Thomas Owen for his great advice and encouragement which gave me the key to encounter the obstacles in which I faced during the journey of my PhD.

I would also like to thank my family, my parents and my sisters, for their support and love, Also, I would like to thank my friends Wasan and Amal Saad for their effective help, and of course a very special thank you for my beloved husband who stood by my side day by day throughout this PhD.

Declaration

I certify that the effort in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree. I also certify that the work in this thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been duly acknowledged and referenced.

Signature of Student

Suad El-Geder

January 2017, London

List of Figures

Figure 2.1 The conventional OS infrastructure Vs the Virtual infrastructure…….9

Figure 2.2 The SDN Architecture……………………………………………..13

Figure 2.3 Traditional Switch versus SDN Switch………………………………15

Figure 2.4 Design of an OpenFlow switch and communication with the controller……………………………………………………………………24

Figure 2.5 OpenFlow-enabled SDN devices…………………..……..……….26

Figure 3.1 OpenDaylight Founding Members…………………………..……..39

Figure 3.2 OpenDaylight modular structure platform…………………….…..40

Figure 3.3 ADSAL architectures……………………………………………...42

Figure 3.4 MD-SAL architectures……………………………………………43

Figure 3.5 Packet path in ADSAL managing OpenFlow devices……………….44

Figure 3.6 Packet path in MD-SAL managing OpenFlow devices……………...46

Figure 3.7 Topology Discovery in SDN……………………………………...49

Figure 4.1 Traditional network Device VS SDN Architecture. ………..………52

Figure 4.2 Main components of an OpenFlow switch……………..………...53

Figure 4.3 Logically centralized control plane…………………………..…55

Figure 4.4 In-band signalling ……………………………………………………57

Figure 4.5 Out-of-band signalling……………………………………………..58

Figure 4.6. Reactive flow management………………………………………61

Figure 4.7 Proactive flow management……………………………….........63

Figure 4.8 Multiple controllers managing multiple network devices…………...65

Figure 4.9 Single controller managing multiple network devices………………65

Figure 4.10 (a) Linear Topology, and (b) Star Topology ………………………67

Figure 4.10 (c) Tree Topology………………………. ……………………….…67

Figure 4.11 Architecture of the D-ITG traffic generator……………………..….70

Figure 4.12 Delay for linear topology and different flow mode in case of multiple-controllers scenarios………………………………………………………...71

Figure 4.13 Delay for linear topology and different flow mode in case of single-controllers scenarios……………………………………………………………..71

Figure 4.14 Packet Loss Ratio for linear topology and different flow mode in case of multiple-controllers scenarios……………………………………………………72

Figure 4.15 Packet Loss Ratio for linear topology and different flow mode in case of single-controller scenarios…………………………………………………………72

Figure 4.16 Delay for different topology in case of proactive-multiple-controller scenarios…………………………………………………………………………….73

Figure 4.17 Throughput for different topology in case of proactive-multiple-controller scenarios…………………………………………………………………73

Figure 4.18 Throughput for proactive-multiple/different-controller scenarios…………………………………………………………………………74

Figure 4.19 Delay for proactive-multiple/different-controller scenarios………..74

Figure 5.1 VTN Architecture…………………………………………………….78

Figure 5.2 GUI of the Opendyalight controller representing the proposed 15 switch in tree topology structure…………………………………………………………84

Figure 5.3 Controllers setup……………………………………………………...84

Figure 5.4 VLAN hosts distribution……………………….…………………….85

Figure 5.5 VLAN host configuration……………………..………………..….85

Figure 5.6 Iperf default settings…………………………………………………86

Figure 5.7 Ping command result for the IP connectivity between hosts in the two VLANs………………………………………………………………………...87

Figure 5.8 Network Connection………………………………………………….88

Figure 5.9 The Throughput for Multi and Single Controller with VTN………...88

Figure 5.10 The Delay for Multi and Single Controller with VTN……………...89

List of Tables

Table 2.1 Virtualization process effects before and after………………………..10

Table 2.2 OpenFlow Messages…………………………………………………..29

Table 4.1 Software used in implementation and experiments……………………64

List of Abbreviations

API	Application Programming Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
CLI	Command Line Interface
FIB	Forwarding Information Base
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
JVM	Java Virtual Machine
NAT	Network Address Translation
NETCONF	Network Configuration
NFV	Network Functions Virtualization
ODL	OpenDaylight
OS	Operating System
OSGi	Open Services Gateway initiative
OVSDB	Open vSwitch Database
QoS	Quality of Service
REST	Representation State Transfer

RPC	Remote Procedure Call
RTT	Round Trip Time
SAL	Service Abstraction Layer
SDN	Software-Defined Networking
SSL	Secure Sockets Layer
TCP	Transmission Control
TLS	Transport Layer Security
VM	Virtual Machine
VTN	Virtual Tenant Network

Chapter 1

Introduction

Today's Internet applications need the underlying networks to be quicker, carry large amounts of traffic, and to deploy a number of distinct, dynamic applications and services. Embracing of the concepts of "inter-connected data centres" and "server virtualization" has amplified network request enormously. In addition to various proprietary network hardware, distributed protocols, and software components, legacy networks are flooded with switching devices that decide on the route taken by each packet discretely; moreover, the data paths and the decision-making processes for switching or routing are collocated on the same device. The decision-making capability or network intelligence is distributed across the various network hardware components. This makes the introduction of any new network device or service a tedious job because it requires reconfiguration of each of the numerous network nodes. Legacy networks have become difficult to automate Networks today depend on IP addresses to identify and locate servers and applications. This approach works fine for static networks where each physical device is recognizable by an IP address, but is extremely laborious for large virtual networks. Managing such complex environments using traditional networks is time-consuming and expensive, especially in the case of virtual machine (VM) migration and network configuration. Thus, the SDN architecture is very valuable in order to simplify the task of managing large networks [1].

1.1 Background: Networks infrastructure developments

In the traditional network approach, the most important part of the network functionality is implemented in the devices like switches or routers, and in

dedicated hardware. This structure involves some difficulties in network functionality, making it to change slowly. Another difficulty is the user usability which is means the hardware networks functionality is under the control of the provider of the appliance making it quite static [2].

Furthermore, networking organizations are under increasing pressure to be as effective and agile as possible with the traditional approach. One source of this pressure arises from the widespread adoption of Cloud Computing. As part of this server virtualization, virtual machines (VMs) are dynamically moved between servers in a matter of seconds or minutes. However, due to its hardware implementation, "if the movement of a VM crosses a Layer 3 boundary, it can take days or weeks to reconfigure the network to support the VM in its new location, with its consequently error prone"[3]. Accordingly, the appearance of Cloud Computing is steadily transforming up to now traditional hardware-centric data network to a software-based network functionality. This means that functions such as encryption or decryption, and the processing of TCP flows, which were previously performed in hardware designed specifically for those functions, are now driven largely, by the need for agility increase, to software running on a general purpose server or on a VM [4].

In a nutshell, the traditional network architectures are unsuitable to meet the requirements of the today's users and enterprises. Correspondingly, a wide range of new opportunities bound to the networking virtualization are opened, such as Software Defined Networking (SDN) and Network Virtualization (NV). With this new approach, the network can develop in scale functionality and easily deployment, performing in traffic engineering with an end-to-end view and better utilization of the resources. Furthermore, the expense of maintenance will be reduced since most of the control will be devolved by software platforms. Finally, network functionality progress more rapidly when it is based on software development lifecycle and it enables applications to dynamically request services

from the network which add more effective security functionality and reduce complexity [5].

1.2 Motivations

The most common debate regarding the SDN networks is doubting the ability to handle a real network with a central controller which leads to the single point of failure problem, and this will lead to trusting issues around the capability of the network to scale and to be reliable. Also there are some concerns regarding the delay and fault tolerance which mean to enable the system to continue operating properly in the event of failure of the OpenFlow (SDN) deployment.

Motivated by the aforementioned concerns, a number of critical execution concerns brought up for the situation of a physically decentralized control plane, which is the way that controllers are put inside the network, as the network execution can be significantly influenced by the number and the physical area of controllers, and in addition by the calculations utilized for their coordination. With a specific end goal to address this, different arrangements have been proposed, from survey the position of controllers as a optimization issue [8] to building up associations of this issue to the fields of local calculations and distributed computing for developing efficient controller coordination protocols [6].

Another concern brought up for the situation of physically appropriated SDN controllers is identified with the consistency of the network state kept up at every controller when performing policy updates, because of simultaneous issues that may happen by the mistake, distributed nature of the logical controller. The arrangements of such an issue can be like those of transnational databases, with the controller being stretched out with a transnational interface characterizing semantics for either committing a policy update or prematurely ending [7].

1.3 Aim and Objectives

In this work, the main aim is to improve the controller’s reliability and scalability, by using multiple controllers operating in the proactive flow mode approach. The research aim is addressed through the following objectives:

1. Literature review of previous work related to the SDN architecture, and all the main component of its network with the relation of the OpenFlow protocol as an innovation of new era of networks.
2. Design and building a model networks of considerable size, which concentrated on the ability to design a dynamic and exceptionally programmable network, moving the intelligence from the basic frameworks to the network itself through a controller.
3. Examine the bottleneck status of the controller to address the limitation of the network, as the control plane is managed by a controller that increase incredibly the user usability of the network: if a client/customer need to build up his own particular network with his own conditions and topology, will have the capacity to do it through the controller.
4. Implement a new scheme design which helps the controller to deal with the reliability issue.
5. Verify the design using Mininet testbed which support OpenFlow switches.

1.4 Contributions

There are three contributions of this thesis which are summarized in the following:

1. Implemented multiple controllers instead of the single, logical controller to manage the network devices by more than one controller connected to each network device in which at least each network device will be connected to 3 or more controllers in the same time, such that they can

share its management. Besides, the proactive flow mode has been used to manage the flow entries.

2. Implemented a different controller topology which have the same configuration to our proposed scheme and a comprehensive performance evaluation has been made between these two approaches.
3. Proposing the configuration of VTN application which is available for the OpenDaylight project with the proposed multiple controller and proactive flow mode paradigm. As its services and features are good examples of NVF.

1.5 Thesis Structure

The thesis is divided in 6 chapters which are as follows:

- ★ Introduction: This chapter talks about the research briefly, the objectives of the research, the place which the research is designed for, the addition to knowledge and the research outline.
- ★ Literature Review: This chapter illustrates some previous studies in the research field including innovations and tools for a new network model, starts the hypothetical piece where to clarify the advances and instruments utilized amid this work.
- ★ OpenDayLight Controller: In this chapter, a concentrated overview about the OpenDaylight controller had been made. The essential device to implement networks using the new paradigms, is the controller. OpenDaylight is one of the most popular controllers used in SDN networks meanwhile, and it was the controller that had been used in this work. An intensive investigation of its features had been made in this chapter, in order to be capable to manage networks with it later.

- ★ Multiple-Controller with different operation paradigm: A new scheme of using multiple controllers which handle multiple network devices, while using

OpenFlow controllers in the proactive operations paradigm has been described in this chapter.

- SDN Applications: The OpenDaylight controller, which was installed during this work, can be easily used using a virtualized network. Hence, in this chapter the VTN Coordinator application had been used, which was installed on a separate VM, to be able to bring NFV features into our network. An implementation for the Multiple Controllers approach with the VTN features has been proposed in this chapter.

- Conclusion and Future Work: This is the final chapter which illustrates the stages of the research in brief and presents some ideas that may improve the system in the future.

Chapter 2

Literature Review

Computer networks are a complex technology that enables end-devices to intercommunicate with each other. A typical network infrastructure contains network devices such as: routers, switches, servers, web-servers, firewalls, load balancers, interruption prevention systems and further devices. The requirements for processing and managing the great amount of data sent over the network, are efficiency, reliability, flexibility and robustness. That has made the manufacturing enterprises of networking devices to implement complex and resource demanding protocols that enable routers and switches to interconnect with each other by packet switching and producing a networking topology for routing purposes[8].

2.1 Virtualization Concept

The concept of virtualization provided individual, dedicated resources from a larger common pool of resources and provided users with the desired customization and control. The field of virtualization expanded in part because of the limitations of shared resources. Although, virtualization has been used since distributed computing started, the combination of virtualization and networking, the core of SDN, has been motivated and enabled by reductions in hardware cost, advances in software, and limitations in current network configurations [9].

Furthermore, the term *virtualization* generally describes the separation of a resource or request for a service from the original physical delivery of that service. For example, with virtual memory, computer software gains access to more memory than is actually installed, via the background swapping of data to disk storage. Similarly, virtualization techniques can be applied to other IT

infrastructure layers as well as networks, storage, laptop or server hardware, operating systems and applications[1].

2.1.1 Types of virtualization

There are four different ways to virtualize a server. Each of these approaches uses a different configuration of the three virtualization components: applications, Operating Systems, and hypervisors.

- ★ **Full virtualization**: The hypervisor is responsible for fully simulating the underlying vendor hardware, when full virtualization has been utilized. This allows unchanged copies of Operating Systems (e.g., Windows, Linux, etc.) to perform on the virtualized server inside their own virtual machines.
- ★ **Hardware-assisted full virtualization**: CPU manufacturers added instructions to their products that support virtualization, as virtualization has become both more popular and more critical to the effective operation of a data centre. Accordingly, the hypervisor can leverage their features to permit guest Operating Systems to work in complete isolation, when these virtualization-enabled CPUs are utilized to power a server. One feature of these CPUs is the introduction of the "ring" concept, which refers to levels of security privileges that are permitted in the code that is currently executing. Applications work at a ring 3 level, rings 1 and 2 are used to perform device drivers, and ring 0 is used to Execute the hypervisor. AMD and Intel have also produced a ring 1 that allows the hypervisor to run computations directly instead of going through the Operating System. This results in an increase in the efficiency of the processing [10].
- ★ **Para-virtualization**: In a virtual server that is using para-virtualization, the guest Operating Systems have each been amended in order to notify them that they are working in a virtual environment. Para-virtualization allows the relocating implementation of critical tasks from the virtual domain to the host domain. As a result of this, guest Operating Systems will spend less time performing operations

that are more difficult in a virtual environment compared to a non-virtualized environment.

★ **Operating System virtualization**: A hypervisor is not in used, when Operating System virtualization has been used on a server. In its place, the virtualization ability is built into the host Operating System. All the functions of a fully virtualized hypervisor is performed by the host Operating System. The major limitation of this approach is that all the virtual machines must run the same Operating System. Each virtual machine remains independent from all the others, but it is not possible to mix and match Operating Systems among them [9].

2.1.2 Virtual infrastructure

A layer of abstraction between computing, storage and networking hardware, and the applications running on it, is offered using balance of virtualization technologies, as demonstrated in Figure 2.1. The user experiences are mostly unchanged, when the deployment of virtual infrastructure has been accomplished. However, virtual infrastructure offers administrators the advantage of managing shared resources across the enterprise, allowing IT managers to be more responsive to dynamic organizational needs and to better leverage infrastructure investments[1].

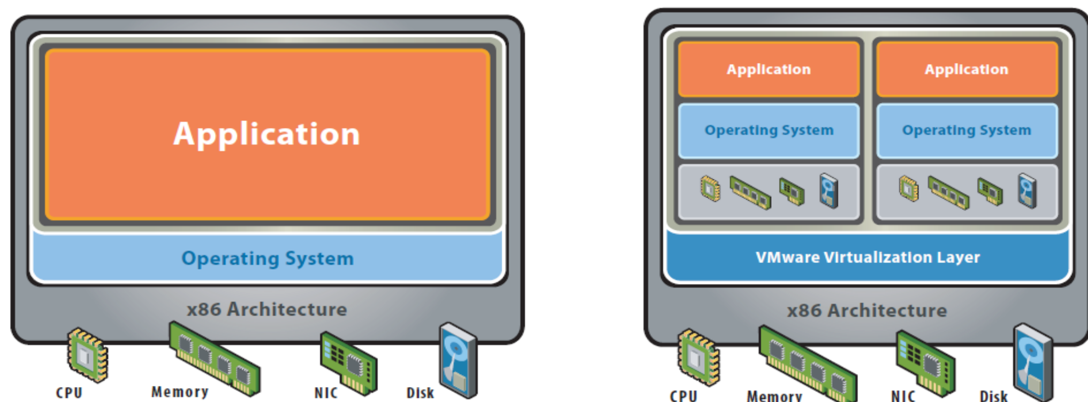


Figure 2.1 The conventional Operating System infrastructure Vs the Virtual infrastructure [1].

Table 2.1 illustrates the main variations occur after visualization:

<table>
<tr><th>Before Virtualization</th><th>After Virtualization</th></tr>
<tr><td>Single Operating System image for each machine.</td><td>Hardware-independence of operating system and applications.</td></tr>
<tr><td>Software and hardware strongly combined.</td><td>Virtual machines can be provisioned to any system.</td></tr>
<tr><td>Running multiple applications on same machine often produces conflict.</td><td rowspan="3">Can manage Operating System and application as a single unit by compressing them into virtual machines.</td></tr>
<tr><td>Underutilized resources.</td></tr>
<tr><td>Inflexible and costly infrastructure.</td></tr>
</table>

Table 2.1 Virtualization process effects before and after [1].

2.1.3 Relationship of Network Virtualization to SDN

The abstraction of the physical network in terms of a logical network is identified as network virtualization, clearly does not need SDN. In the same way, SDN is the separation of a logically centralized control plane from the fundamental data plane, does not imply network virtualization. Interestingly, however, a cooperation between network virtualization and SDN has risen, which has started to stimulate a few new research areas. SDN and network virtualization relate in three fundamental ways [11]:

- ***SDN as a supporting technology for network virtualization***. Cloud providers want a way to permit multiple customers (or "tenants") to share the same network infrastructure, from this demand, network virtualization has gained its importance among cloud computing environment. Nicira's Network

Virtualization Platform (NVP) proposes this abstraction without demanding any support from the underlying networking hardware. In order to provide each tenant with the abstraction of a single switch connecting all of its virtual machines, a solution has been implemented by using overlay networking. Until now, in contrast to previous work on overlay networks, each overlay node is in fact an extension of the physical network a software switch (like Open vSwitch [12]) that encapsulates traffic destined to virtual machines running on other servers. Therefore, to control how packets are encapsulated, and updates these rules when virtual machines move to new locations, a logically centralized controller has installed the rules in these virtual switches.

- ***Network virtualization for assessing and testing SDNs***. It becomes possible to test and assess SDN control applications in a virtual environment before the application is deployed on an operational network, using the ability to separate an SDN control application from the underlying data plane. Mininet [13], [14] uses process-based virtualization to run multiple virtual OpenFlow switches, end hosts, and SDN controllers each as a single process on the same physical (or virtual) machine. To emulate a network with hundreds of hosts and switches on a single machine, Mininet permits the use of process-based virtualization. In such an environment, a researcher or network operator can improve control logic and easily test it on a full-scale emulation of the production data plane; once the control plane has been evaluated, tested, and debugged, it can then be deployed on the real production network.
- ***Virtualizing ("slicing") an SDN***. In traditional networks, each virtual component needs to run own instance of control-plane software, which makes virtualizing a router or switch very difficult. In contrast, virtualizing a "dumb" SDN switch is considerably simpler. The FlowVisor [15] system allows a campus to support a testbed for networking research on top of the same physical equipment that carries the production traffic. The main idea is that each slice has a share of network resources and is managed by a different SDN controller, and this implemented by dividing traffic flow space into "slices" (a

concept introduced in earlier work on PlanetLab [16]). FlowVisor runs as a hypervisor, speaking OpenFlow to each of the SDN controllers and to the underlying switches. To permit different third-party service providers (e.g., smart grid operators) to deploy services on the network without having to install their own infrastructure, a recent work has offered slicing control of home networks [17]. More recent work proposes ways to present each “slice” of a software-defined network with its own logical topology and address space [18].

2.2 Software-Defined Networking

The SDN architecture and its components has been described in this section. Moreover, for a better understanding of the data and control planes, it will be further explained separately. Also, a description of how communication between each layer works, and an explanation for northbound and southbound interfaces will be clarified in this chapter. Finally, an overview of the existing SDN controllers and will be provided with an introduction to the ODL controller.

2.2.1 SDN Architecture and Components

Software-Defined Networking is a developing paradigm that empowers network innovation in view of four basic standards: (*i*) network control and forwarding planes are clearly separated,, (*ii*) instead of destination-based, the forwarding decisions are flow-based, (*iii*) the network forwarding logic is abstracted from hardware to a programmable software layer, and (*iv*) an element, called a controller, is presented to coordinate network-wide forwarding decisions. [19]

The SDN architecture figure 2.2 is established on a principle of decoupling of the control plane or the network plane from the forwarding hardware, and a logical

centralization of a control program or a controller that makes forwarding decisions and installs instructions on switches or routers. [20]

These instructions make the forwarding hardware to switch packets between ports. According to this architecture, the SDN can logically be represented as a three-layered architecture:

* **Infrastructure layer**: This layer is frequently mentioned to as a data plane. It contains forwarding hardware, for example, switches and routers, including forwarding components, and Application Programming Interfaces (API).
* **Control layer**: The other name is a control plane. Network intelligence in a form of logically centralized and software-based SDN controller installed on any UNIX based Operating System (OS) function on any hardware. The control layer manages forwarding hardware and installs forwarding instructions through APIs.

* **Application layer**: Applications and services take control over control and infrastructure layer through Representational state transfer (REST) APIs. The SDN concept allows developers to easily develop applications that execute networking function responsibilities. Applications are usually deployed to separate computers or clouds.[20], [21]

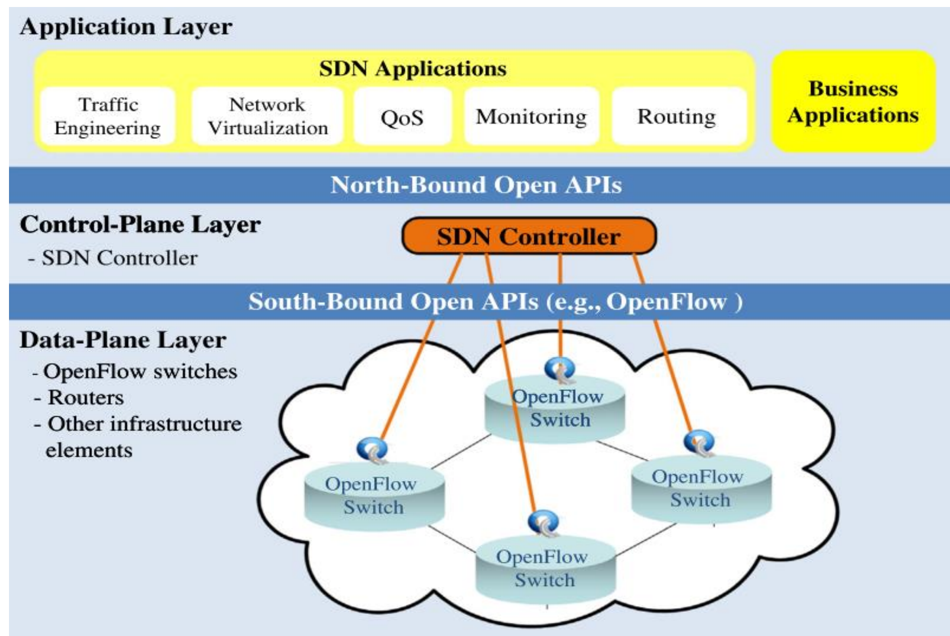


Figure 2.2 SDN Architecture [22].

The APIs in the SDN architecture is frequently called northbound and southbound interfaces. Those are used for the communication between hardware, controllers and applications. The connection between networking devices and controllers is known as the southbound interface, whereas the northbound interface is the connection between applications and the controllers [5].

2.2.2 *SDN Switches*

Figure 2.3 shows the immigration from the traditional switch to the SDN switch, in the traditional networking paradigm the network infrastructure is considered the most essential part of the network. Each network device encapsulates all the functionality that would be required for the operation of the network. For example, a router require to provide the proper hardware like a Ternary Content Addressable Memory (TCAM) for quickly forwarding packets, as well as complicated software for executing distributed routing protocols like BGP. The three-layered SDN architecture presented above in section 2.2.1 changes this, by separating the control from the forwarding operations, which makes the management of network devices much simpler. As previously mentioned, all

forwarding devices contain the hardware that is responsible for storing the forwarding tables (e.g., Application-specific integrated circuits - ASICs - with a TCAM), but are removed of their logic. The controller commands to the switches how packets should be forwarded by installing new forwarding rules via an abstract interface. Each time a packet arrives to a switch its forwarding table is consulted and the packet is forwarded in view of that consultation from the controller. [22]

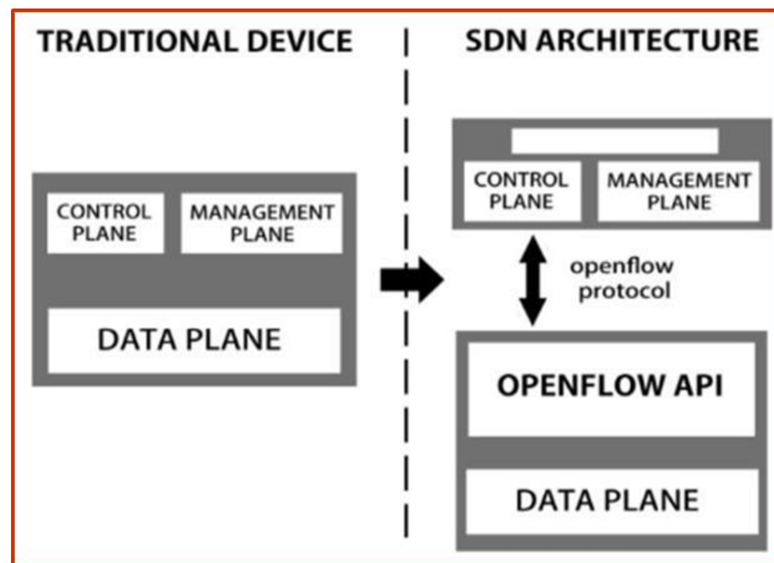


Figure 2.3 Traditional Switch versus SDN Switch.

In contrast, while moving all control operations to a logically centralized controller has the advantage of easier network management, it can also increase scalability issues if physical implementation of the controller is also centralized.

Therefore, it might be beneficial to add some of the logic in the switches. For example in the case of DevoFlow [23], which is an amendment of the OpenFlow model, the packet flows are distinguished into two categories: small ("mice") flows controlled directly by the switches and large ("elephant") flows requiring the involvement of the controller. Likewise, in the DIFANE [24] controller middle switches are used for storing the necessary instructions and the controller is relegated to the simple task of dividing the instructions over the switches.

2.2.3 *SDN Controllers*

As aforementioned, one of the fundamental ideas of the SDN philosophy is the presence of a network operating system placed between the network infrastructure and the application layer. This network operating system is in charge for coordinating and managing the resources of the whole network and for revealing an abstract unified view of all components to the applications executed on top of it. This idea is equivalent to the one followed in a typical computer system, where the operating system lies between the hardware and the user space and is responsible for managing the hardware resources and providing common services for user programs.

In the same way, network administrators and developers are now presented with a homogeneous environment easier to program and configure much like a typical computer program developer would. The SDN model applicable to a wider range of applications and heterogeneous network technologies compared to the traditional networking paradigm, and this can be accomplished by using logically centralized control and the generalized network abstraction. For example, consider a heterogeneous environment composed of a fixed and a wireless network consist of a large number of related network devices (routers, switches, wireless access points, middle-boxes etc.). In the traditional networking paradigm, to make each network device work properly, there is a significant need of individual low level configuration by the network administrator. In addition, since each device aims a different networking technology, it would have its own specific management and configuration requirements, meaning that further effort would be required by the administrator to make the whole network operate as planned. On the other hand, the administrator would not have to worry about low level fine points, using the logically centralized control of SDN. Instead, the network management would be performed by defining a proper high level policy, leaving the network operating

system responsible for communicating with and configuring the operation of network devices.

The controller can work in two different flow setup mode: 1) Proactive mode, and 2) reactive mode. As aforementioned in section 2.2.2 the routers use TCAM for quickly forwarding packets, but the problem with limited TCAM and proactive flow management is a particular concern in carrier networks. They for the most part utilize the Border Gateway Protocol (BGP), which has extensively high state necessities [25]. Accordingly, with regard to different flow management approaches, analyses of the feasibility and practicability of SDN in that context and state-heavy protocols are relevant, at that point viable solutions are needed.

The authors of [26] use a reactive approach to SDN and BGP. In a large network, it is often important to be able to detect high-volume traffic in near real-time. Existing work on the detection and identification of such high volume traffic (called heavy hitters) [27]. Entries for flows with low packet frequency are not carried in the flow table when space is not usable. Those flow entries are saved in the controller and packets belonging to low-frequency flows are transmitted from the switch to the controller, which takes in the complete forwarding knowledge of all flows. The proposed SDN software router is presented in [28]. Other approaches attempt to cut the flow table size. For example, source routing techniques can be leveraged to significantly decrease the number of flow table entries, as shown by Soliman et al. [29]. In their source routing method, the ingress router encodes the path in the form of interface numbers in the packet header. The routers on the path forward the packet according to the interface number of the path in the header. Since packets of source-routed flows contain the path in their headers, the switches on the path do not require a flow table entry for them. However, their method have need of some minor changes to the OpenFlow protocol to support the source routing. Source routing methods for SDN are also discussed in the IRTF [30].

The forwarding state for BGP is also a concern in traditional networks without the challenging limitations of forwarding tables. Various approaches to solve this problem exist today. The authors of [31] attempt to develop the scaling of IP routers using tunnelling and virtual prefixes for global routing. Virtual prefixes are selected in such a way that prefixes are accumulated efficiently. Their solution requires a mapping system to employ the virtual prefixes. Ballani et al. [32] have a similar solution, where they remove parts of the global routing table with virtual prefixes. They illustrate that this technique can reduce the load on BGP routers. The idea of virtual prefixes is currently standardized in the IETF [33]. Distributed hash tables can also be used to effectively reduce the size of BGP tables, as shown in [34].

Other techniques are based on the idea of efficient compression of the forwarding information state, as shown in [35]. The authors use compression algorithms for IP prefix trees in such a way that lookups and updates of the FIB can be done in a timely manner. However, match fields in OpenFlow may contain wildcards and, therefore, have more freedom with regard to aggregation than prefixes in IP routing tables. Appropriate techniques must be developed and tested to reduce the state in OpenFlow switches.

Having discussed the general concepts behind the SDN controller, the following subsections take a closer look at specific design decisions and implementation choices made at this main component that can prove to be serious for the overall performance and scalability of the network.

2.2.4 Centralization of control in SDN

There have been different proposals for physically centralized controllers, like for instance NOX [36] and Maestro [37]. In order to simplify the controller implementation, a physically centralized control has been designed. With all the applications seeing the same network state (which comes from the same

controller), all switches are controlled by the same physical unit, meaning that the network is not subject to consistency related issues. Despite its advantages, the controller acts as a single point of failure for the entire network, which makes this approach suffers from the same weakness that all centralized systems do. An approach to beat this was introduced by associating multiple controllers to a switch, permitting a backup controller to assume control in case of a failure. In this case, all controllers require to have a consistent view of the network, if not applications might fail to work properly. Moreover, since all network devices require to be managed by the same unit, the centralized approach can raise scalability concerns.

One approach that more generalizes the idea of using multiple controllers over the network is to maintain a logically centralized but physically decentralized control plane. In this case, all controllers communicate and maintain a common network view, while each controller is responsible for managing only one part of the network.

Thus, applications view the controller as a single unit, while actually control operations are performed by a distributed system. The advantage of this approach, apart from not having a single point of failure anymore, is that only a part of the network requires to be managed by each individual controller component, which has affect in increasing the performance and scalability. Some distinguished controllers that belong to this category are Onix [38] and HyperFlow [39]. One possible downside of decentralized control is, once more related to the consistency of the network state among controller components. It is possible that applications served by different controllers might have a diverse view of the network, which might make them work improperly, since the state of the network is distributed.

A mix solution that tries to embrace both scalability and consistency is to use two layers of controllers like the Kandoo [40] controller does. The bottom layer is

composed by a group of controllers which do not have knowledge of the entire network state. These controllers only run control operations which need knowing the state of a single switch (local information only). On the other hand, the top layer is a logically centralized controller responsible for performing network-wide operations that need knowledge of the entire network state. The idea is that local operations can be performed faster this way and do not experience any additional load to the high-level central controller, effectively increasing the scalability of the network.

2.2.5 SDN Challenges

There are some concerns about the SDN controllers which can be raised about their performance and applicability over large networking environments. One of the most common concerns raised by SDN doubters is the ability of SDN networks to scale and be quick to respond in cases of high network load. This concern comes essentially from the fact that in the new paradigm control moves out of network devices and goes in a single unit responsible for managing the entire network traffic. Motivated by this concern, performance studies of SDN controller implementations [41] have revealed that even physically centralized controllers can perform really well, having very low response times. For example, it has been presented that even basic single-threaded controllers similar to NOX can handle an average workload of up to 200 thousand new flows per second with a maximum latency of 600ms for networks consist of up to 256 switches. Newer multi-threaded controller implementations have been shown to perform expressively better. For example, NOXMT [42], with an average response time of 2ms in a commodity eight-core machine of 2GHz CPUs, can handle 1.6 million new flows per second in a 256-switch network. In order to increase the

performance even further, newer controller promises to design large industrial servers. For example, the McNettle [43] controller using a single controller of 46 cores with a throughput of over 14 million flows per second, and latency under 10ms, has claimed to be able to serve networks of up to 5000 switches.

Also, the way that controllers are placed within the network, as the network performance can be significantly affected by the number and the physical location of controllers, is consider to be another important performance concern has been raised in the case of a physically decentralized control plane, as well as by the algorithms used for their organization. In order to address this, various solutions have been proposed, from viewing the placement of controllers as an optimization problem [44] to establishing connections of this problem to the fields of local algorithms and distributed computing for developing efficient controller coordination protocols [6].

A final concern that is related to the consistency of the network state maintained at each controller when performing policy updates, due to concurrency issues that might occur by the error prone, distributed nature of the logical controller, is raised in the case of physically distributed SDN controller. The solutions of such a problem can be similar to those of transactional databases, with the controller being extended with a transactional interface defining semantics for either completely committing a policy update or aborting [45].

2.2.6 SDN in the Industry

The advantages that SDN offers compared to traditional networking have also made the industry focus on SDN either for using it as a means to simplify management and improve services in their own private networks or for developing and providing commercial SDN solutions.

Perhaps Google, is one of the most characteristic examples for the adoption of SDN in production networks, which entered in the world of SDN with its B4 network [46] developed for connecting its data centres worldwide. As explained by Google engineers, according to the very fast growth of Google's back-end network, that's was the reason of moving to the SDN paradigm. While computational power and storage become cheaper as scale increases, the same cannot be said for the network. The company was able to choose the networking hardware according to the features it required, by applying SDN principles, while it managed to develop innovative software solutions. Moreover, while at the same time using the centralized network controlled to a reduction of operational expenses, it made the network more efficient and fault tolerant providing a more flexible and innovative environment. More recently, Google revealed Andromeda [47], a software defined network underlying its cloud, which is aimed at enabling Google's services to scale better, cheaper and faster. Also, Facebook and Amazon are planning on building their next generation network infrastructure based on the SDN principles, in order to develop itself in the field of networking and cloud services.

Networking companies have also started showing interest in developing commercial SDN solutions. There is a trend for creating complete SDN ecosystems targeting different types of customers, rather to be this interest only limited in developing specific products like OpenFlow switches and network operating systems. While telecommunication companies like Huawei are designing solutions for the next generation of telecom networks, with a specific interest in LTE and LTE-Advanced networks, another companies like Cisco, HP and Alcatel have entered the SDN market, presenting their own complete solutions intended for enterprises and cloud service providers. In 2012, VMware acquired an SDN startup called Nicira in order to integrate its Network Virtualization Platform (NVP) to NSX, VMware's own network virtualization and security platform for software-defined data centres. With many major companies

like Broadcom, Oracle, NTT, Juniper and Big Switch Networks recognizing the benefits of SDN and proposing their own solutions, the list of providing SDN solutions constantly being grown.

2.2.7 SDN Programming Interfaces

As above-mentioned, the communication of the controller with the other layers is accomplished via a southbound API for the controller-switch interactions and through a northbound API for the controller-application interactions. In this section, a brief explanation about the essential concepts and concerns related to SDN programming had been made by separately examining each point of communication.

2.2.7.1 Northbound API

As of now talked about, one of the essential thoughts pushed in the SDN paradigm is the presence of a network operating system, lying between the network foundation and the high level services and applications, likewise to how a computer operating system lies between the hardware and the user space. Assuming such a centralized coordination unit and based on the simple operating system principles, a clearly defined interface should also be present in the SDN architecture for the interaction of the controller with applications. This interface should permit their communication with other applications, manage the system resources and permit the applications to have the right to use the underlying hardware without having any knowledge of low level network information.

In contrast to the southbound communication, there is currently no accepted standard for the interaction of the controller with applications, whereas the

interactions between the switches and the controller are well-defined through a standardized open interface (i.e. OpenFlow) in the case of the southbound communication [48]. Therefore, to perform controller-application communication, each controller model needs to provide its own methods. Moreover, it is difficult to implement applications with different and many times conflicting objectives that are based in more high-level concepts, and this because the interfaces current controllers implement provide very low-level abstractions (i.e. flow manipulation).

2.2.7.2 Southbound communication

The southbound communication is very important for the handling of the behaviour of SDN switches by the controller. It is the way that SDN going to "program" the network. The most recognizable example of a standardized southbound API is OpenFlow [49]. Most projects related to SDN assume that the communication of the controller with the switches is OpenFlow-based, and therefore it is important to make a detailed demonstration of the OpenFlow approach. Yet, it should be made clear that OpenFlow is just one (rather popular) out of many probable implementations of controller-switch communications. Other alternatives like for example DevoFlow [23] also exist, trying to solve performance issues that OpenFlow faces. In this project, OpenFlow protocol I used as the communication protocol in which make our switches and controllers communicate which each other, in the next section a description of the main features of OpenFlow protocol will be clarified.

2.3 Openflow overview

SDN principle of separating the control and forwarding planes, makes the OpenFlow protocol provide standardized way of managing traffic in switches and of swapping information between the switches and the controller. From Figure 2.4, it can be illustrated that the OpenFlow switch is composed of two logical

components. In order to forward packets, the first component comprises one or more flow tables which is responsible for preserving the information needed by the switch. A simple API permitting the communication of the switch with the controller, consider to be the second component is an OpenFlow client.

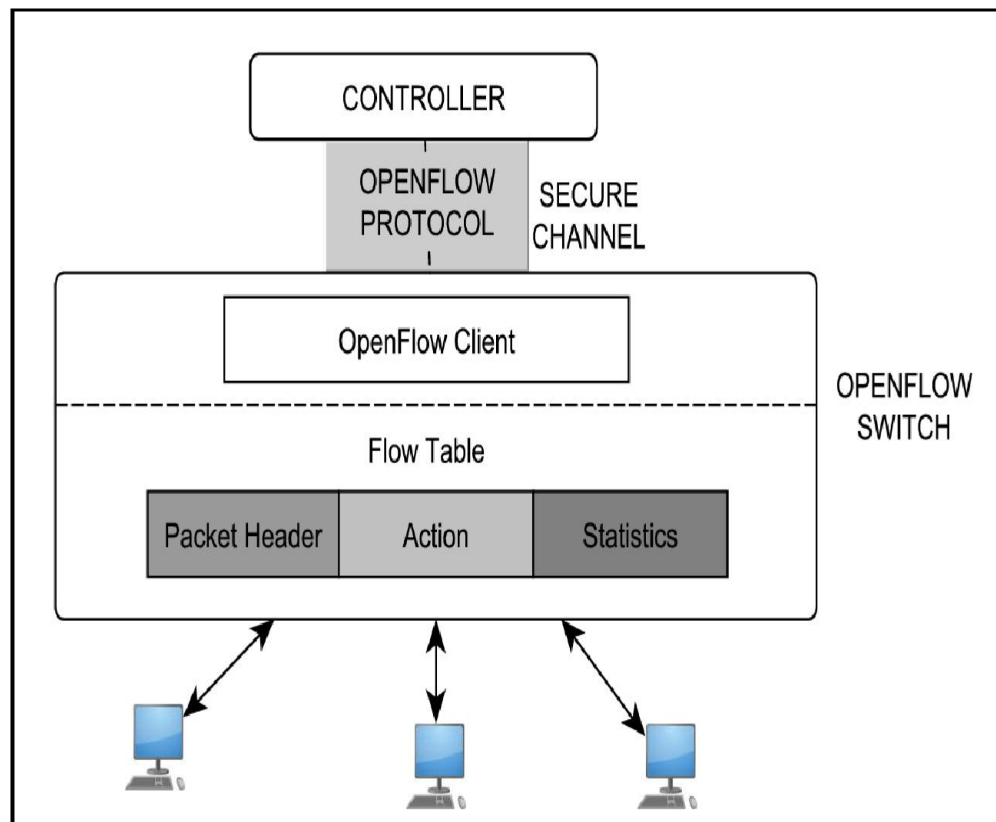


Figure 2.4 Design of an OpenFlow switch and communication with the controller.

2.3.1 *OpenFlow Protocol*

As aforesaid, flow tables comprise of flow entries, each of which defines a set of instructions decisive how the packets belonging to that particular flow will be managed by the switch, that's means how they will be processed and forwarded. Flow table has three fields for each entry as follows:

i) A packet header which defining the flow.

ii) An Action responsible how the packet should be handled.

iii) Statistics, which save route of information same the number of packets and bytes of each flow and the time since a packet of the flow was last forwarded [50].

Inside an OpenFlow switch or router, a way through a grouping of flow tables characterizes how packets should be managed. When a new packet arrives, the lookup procedure starts in the first table and ends either with a match in one of the tables of the pipeline or with a miss, which is when no instruction is found related to that packet. A flow rule can be defined by combining different matching fields, as explained in Figure. 2.5. The packet will be discarded, if there is no default rule. On the other hand, the common event is to install a default rule which tells the switch to send the packet to the controller (or to the normal non-OpenFlow pipeline of the switch). The priority of the rules follows the natural sequence number of the tables and the row order in a flow table. Probable actions include: 1) the packet forwarding to outgoing port(s); 2) encapsulate the packet and forward it to the controller; 3) drop the packet; 4) send the packet to the normal processing pipeline; and 5) send it to the next flow table or to special tables, such as group tables or metering tables presented in the latest OpenFlow protocol [2].

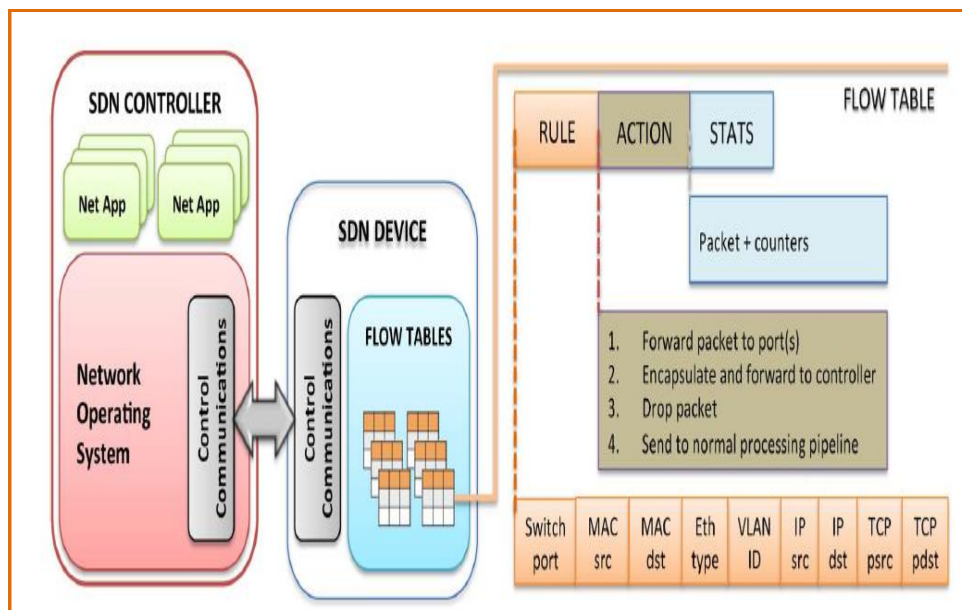


Figure 2.5 OpenFlow-enabled SDN devices[2]

As the OpenFlow protocol defined, the substitution of information between the switch and the controller occurs by sending messages over a secure channel in a standardized way. As explained in the basic controller principles, the controller can handle flows set up in the flow table of the switch, by adding, updating or deleting a flow entry, either proactively or reactively. Consequently, there is no longer a need for network operators to interact directly with the switch, since the controller is able to communicate with the switch using the OpenFlow protocol.

Implying the matching process to the header of the packet does not need to be precise, as OpenFlow packet header field can be a wildcard. Different network devices like switches, routers and middle-boxes have a similar forwarding behaviour, and this was the idea behind producing this approach, also they only varying regarding which header fields they use for matching, and the actions they execute. In order to indicate how thoughtfully binds together a wide range of sorts of network devices, OpenFlow permits the utilization of any subset of these header fields for applying rules on traffic flows. For example a firewall would be emulated through a packet header field containing extra data like the source and destination IP addresses and port numbers, and also the transport protocol utilized, while a switch could be emulated by a flow entry utilizing a packet header performing a match just on the IP address [50].

2.3.2 *OpenFlow Protocol Messages*

The message exchange that occur between an OpenFlow controller and an OpenFlow switch is identify as OpenFlow protocol. Commonly, in order to give a secure OpenFlow channel, the protocol is being executed on top of SSL or Transport Layer Security (TLS).

So as to perform add, update, and delete actions to the flow entries in the flow tables, the OpenFlow protocol has enabled the controller to do the job. OpenFlow protocol supports three types of messages, as shown in Table 2.2.

- ★ *Controller-to-Switch:* These messages at times, require a reaction from the switch, and they are started by the controller. This class of messages empowers the controller to deal with the logical condition of the switch, including its configuration and details of flow and group-table entries. Also, the `Packet-out` message is included in this class. When a switch sends a packet to the controller and the controller chooses not to drop the packet but rather to direct it to a switch output port, then this message has to be utilized.

- ★ *Asynchronous:* These kinds of messages are sent without consultation from the controller. This class includes different status messages to the controller. Additionally included is, the `Packet-in` message, which is used to send a packet to the controller when there is no flow table match, and it is being utilized by the switch.
- ★ *Symmetric:* These messages are sent without demand from either the controller or the switch. They are simple however helpful. When the connection is first established `Hello` messages are commonly sent back and forth between controller and switch. Both the controller and the switch are using `Echo` request and reply messages, to measure the latency or bandwidth of a controller-switch connection or just verify that the device is operating. To stage features to be built into future versions of OpenFlow, the Experimenter message is being applied.

Description	Message
Controller-to-Switch	
Features	Ask for the abilities of a switch. Switch reacts with features answer that indicates its abilities.
Configuration	Set and inquiry configuration parameters. Switch reacts with parameter settings.

Table 2.2 OpenFlow Messages.

Modify-State	Add, remove, and amend flow/group entries and set switch port properties.
Read-State	Gather information from switch, such as existing configuration, statistics, and capabilities.
Packet-out	Direct packet to a specified port on the switch.
Barrier	Barrier request/reply messages are utilized by the controller to certify message dependencies have been met or to receive notices for accomplished operations.
Role-Request	Set or query role of the OpenFlow channel. Beneficial when switch attaches to multiple controllers.
Asynchronous-Configuration	Set filter on asynchronous messages or query that filter. Convenient when switch attaches to multiple controllers.
Asynchronous	
Packet-in	Transfer packet to controller.
Flow-Removed	Notify the controller about the removal of a flow entry from a flow table.
Port-Status	Notify the controller of a modification on a port.
Error	Notify controller of error or problem situation.
Symmetric	
Hello	Swapped between the switch and controller upon connection startup.
Echo	Echo request/reply messages can be sent from either the controller or the switch, and they must return an echo reply.
Experimenter	For further functions.

2.3.3 The OpenFlow Flow Table.

There are three types of tables in logical switch architecture, had been defined by OpenFlow protocol specification. A *Flow Table* identifies the

functions that are to be performed on the packets and matches received packets to a specific flow. As described successively, there may be multiple flow tables that work in a pipeline manner. A flow table may direct a flow to a *Group Table,* in the case of generating a variety of actions that affect one or more flows. Furthermore, a variety of performance-related actions can be generated on a flow as *a Meter Table*. It is useful to define what the term *flow* means, before continuing. Inquiringly, this term is not defined in the OpenFlow specification, nor is there a try to define it in virtually all of the literature on OpenFlow. Generally, a flow is a sequence of packets crossing a network that share a set of header field values. For instance, a flow might consist of all packets with similar source and destination IP addresses, or all packets with identical VLAN identifier. More particular definition are subsequently provided [51].

2.3.3.1 Flow-Table Components

The basic structure block of physical switch architecture is the flow table. Each packet that arrives a switch passes through one or more flow tables. There are six components, in which each flow table contains entries is consisting of:

1. *Match Fields:* Used to choose packets that have the same values in the fields.
2. *Priority:* Relative priority of table entries.
3. *Counters:* Updated for matching packets. The OpenFlow specification defines a selection of timers. For instance, the number of received bytes and packets per port, per flow table, and per flow-table entry; number of dropped packets; and duration of a flow.
4. *Instructions:* Actions to be taken if a match take place.
5. *Timeouts:* The maximum amount of idle time before a flow is expired by the switch.
6. *Cookie:* Uncertain data value chosen by the controller. Thus, this value may be used by the controller to filter flow statistics, flow modification, and flow deletion; but not used when processing packets.

A flow table may include a *table-miss* flow entry, in every field is a match regardless of value, and has the lowest priority (priority 0). The Match Fields component of a table entry consists of the following required fields:

* *Ingress Port:* Presents the port on the switch where the packet arrived. Also, it may be a switch-defined virtual port or a physical port.
* *Ethernet Source and Destination Addresses:* Each entry can be a wildcard value (match any value), or an exact address, a bit-masked value for which only some of the address bits are checked.
* *IPv4 or IPv6 Protocol Number:* A protocol number value, so that indicate the next header in the packet.
* *IPv4 or IPv6 Source Address and Destination Address:* Each entry can be a wildcard value, or an exact address, a bit-masked value, a subnet mask value.
* *TCP Source and Destination Ports:* Wildcard value or Exact match.
* *User Datagram Protocol (UDP) Source and Destination Ports:* Wildcard value or Exact match.

Any OpenFlow-compliant switch must be supported by the previous match fields. The following fields may be optionally supported:

* *Physical Port:* Used in order to designate underlying physical port when packet is received on a logical port.
* *Metadata:* During the processing of a packet, this field carrying additional information that can be passed from one table to another. Its use is discussed afterwards.
* *Ethernet Type:* Ethernet Type field.

* *VLAN ID and VLAN User Priority:* Fields in the IEEE 802.1Q Virtual LAN header.

* *IPv4 or IPv6 DS and ECN:* Explicit Congestion Notification fields and Differentiated Services.

* *Stream Control Transmission Protocol (SCTP) Source and Destination Ports:* Wildcard value or Exact match.

* *Internet Control Message Protocol (ICMP) Type and Code Fields:* Wildcard value or Exact match.

* *Address Resolution Protocol (ARP) Opcode:* Exact match in Ether-net Type field.

* *Source and Target IPv4 Addresses in Address Resolution Protocol (ARP) Payload:* Can be a wildcard value, or an exact address, a bit-masked value, a subnet mask value.

* *IPv6 Flow Label:* Wildcard value or Exact match.

* *ICMPv6 Type and Code fields:* Wildcard value or Exact match.

* *IPv6 Neighbour Discovery Target Address:* In an IPv6 Neighbour Discovery message.

* *IPv6 Neighbour Discovery Source and Target Addresses:* Link-layer address options in an IPv6 Neighbour Discovery message.

* *Multiprotocol Label Switching (MPLS) Label Value, Traffic Class, and Bottom of Stack (BoS):* Fields in the top label of an MPLS label stack[50].

Thus, OpenFlow can be used with network traffic including a variety of protocols and network services. Note that at the MAC/link layer, only Ethernet is supported.

Accordingly, OpenFlow as currently defined cannot control Layer 2 traffic over wireless networks.

Now an accurate definition of the term *flow* can be offered. A flow is a sequence of packets that matches a specific entry in a flow table, and this from the point of view of an individual switch. Considering, the definition is packet-oriented, in the sense that it is a function of the values of header fields of the packets that constitute the flow, and not a function of the path they follow through the network. A flow that is bound to a specific path is defined as a combination of flow entries on multiple switches.

2.3.3.2 The Instructions Component

If the packet matches, then the entry of a table entry consists of a set of instructions that will be executed. Before describing the types of instructions, it's useful to explain the terms "Action" and "Action Set". Packet forwarding, packet modification, and group table processing operations, can be described as an actions [51]. The OpenFlow specification contains the following actions:

* *Output:* Forward packet to identified port.
* *Set-Queue:* Sets the queue ID for a packet. When the output action occurred, after the packet is forwarded to a port, the queue id determines which queue dedicated to this port, in order to schedule and forward the packet. Forwarding behaviour is dictated by the configuration of the queue and is utilized to offer basic QoS support.
* *Group:* Process packet via specified group.
* *Push-Tag/Pop-Tag:* Push or pop a tag field for a VLAN or MPLS packet.

* *Set-Field:* By using field type, the various Set-Field actions can be identified; they modify the values of respective header fields in the packet.

* *Change-TTL:* In order to modify the values of the IPv4 Time To Live (TTL), IPv6 Hop Limit, or MPLS TTL in the packet, the various Change-TTL actions is being used.

An ***Action Set*** is a list of actions associated with a packet that are accumulated while the packet is processed by each table and executed when the packet exits the processing pipeline. Instructions are of four types:

* *Direct packet through pipeline:* The Goto-Table instruction directs the packet to a table farther along in the pipeline. The Meter instruction directs the packet to a specified meter.

* *Perform action on packet:* Actions may be performed on the packet when it is matched to a table entry.

* *Update action set:* Merge specified actions into the current action set for this packet on this flow, or clear all the actions in the action set.

* *Update metadata:* A metadata value can be associated with a packet. It is used to carry information from one table to the next [50].

2.4 Summary

OpenFlow and its associated standards organization, the ONF is recognized with starting the discussion of SDN and providing the first proposal of modern SDN control, which comprises from: a centralized point of control, a northbound API that discoveries topology, path computation, and provisioning services to an application above the controller, in addition to a standardized southbound protocol for instantiating forwarding state on a multivendor infrastructure.

Unfortunately, the OpenFlow architecture does not offer a standardized northbound API, nor does it offer a standardized east-west state distribution protocol that permits both application portability and controller vendor interoperability. Standardization may progress through the newly reproduced Architecture Working Group, or even the new open source organization OpenDaylight Project. OpenFlow provides a great deal of flow/traffic control for those platforms that can exploit the full set of OpenFlow primitives. The ONF has produced a working group to address the description/discovery of the capabilities of vendor hardware implementations as they apply to the use of the primitive set to implement well-known network application models.

Chapter 3

OpenDayLight Controller

As above-mentioned, OpenDaylight is the controller that has been used in this thesis, Thus, in order to further use of the OpenDayLight Controller, this chapter introduce an accurate and theoretical overview of this controller, to program the control of different network scenarios [52].

3.1 Introduction to the controller

The OpenDaylight project is a collaborative open source project, that the Linux Foundation has been hosted. According to its partners, the purpose of the project is to speed up the implementation of Software Defined Networking and create a solid foundation for Network Functions Virtualization.

3.1.1 OpenDaylight primary phases

The Linux Foundation announced the founding of the OpenDaylight Project, on April 8, 2013. This was after some months of the news of an industry merger forming around SDN, had been broken by the Software Defined Networking site "SDN Central”. Therefore, in order to raise new innovation, accelerate adoption, and create a more open and transparent approach to Software Defined Networking and Network Functions Virtualization, this platform has been exposed, by a community-led and industry-supported open source framework.

From that point on, three major versions have been released: Hydrogen (February 2014), Helium (September 2014), Lithium (June 2015), Beryllium (February 2016), and the current Boron-SR2 (October 2016). During this period, an increasing number of companies have given support to the project: since the

foundational and very powerful ones such as Cisco, Intel, Microsoft... to other newer names like Huawei or Lenovo, until reach the remarkable number of almost 50 influential brands [53].

3.1.2 Technology Overview

OpenDaylight is a modular platform written in Java with most modules (bundles) reusing some common services and interfaces (Service Abstraction Layer, SAL).

Each module is a service offered by the controller, and it is developed under multi-vendor sub-project following the idea of SDN, each user can deploy these already implemented bundles and also can develop his own one, in order to control his particular network. This gives many possibilities to personalize the network to the user.

To leverage functionality in other platform bundles, the idea of building applications on the OpenDaylight platform with this bundle structure has been created. Each of which export important services through Java interfaces. The major portion of these services are built on a provider-consumer model over an adaptation layer called (Service Abstraction Layer, SAL). Therefore, SAL is consider to be a layer which used to establish the connection between everybody, not only between bundles, but also between inside the controller (bundles) and network devices (nodes, switches...), and even with external applications.

3.1.3 Model View Controller (MVC) Platform

OpenDaylight is a Model-View-Control platform. In order to separate internal representations of information from the ways that information is submitted to or taken from the user, the application has been split in three interconnected parts, and these parts are:

* **Model→** *YANG*: It's a model for data, Remote Procedure Call (RPC) and notifications
* **View →** *REST API*: View self-generated and reachable through Northbound (ADSAL, or RESTconf. It is our user interface to see the information (e.g. a flow).
* **Control →** *Java Implemented Code*: To handle data changes, notifications and RPC call backs.

3.1.4 Fundamental Software Tools

OpenDayLight uses the following software tools/paradigms. It is essential to the main usage of them, in order to use them in our design and configurations. These tools are listed as follows:

★ **Java interfaces**: These interfaces are used for event listening, specifications and forming patterns. This is the main way in which specific bundles implement call-back functions for events and also to indicate awareness of specific state.

★ **Maven**: It is a software tool used for the management and construction of Java projects. OpenDayLight uses Maven for easier build automation. Maven uses pom.xml (Project Object Model for this bundle) to script the dependencies between bundles and also to describe what bundles to load on start.

★ **OSGi**: This framework in the backend of OpenDayLight allows dynamically loading bundles and packaged JAR files, and binding bundles together for information exchange.

★ **Karaf**: is a small OSGi based runtime which provides a lightweight container for loading different modules. It will be available after the first release, so for Helium and Lithium releases [52].

3.2 Advantages in front of other controllers

Different kind of controllers can be used in order to deploy networks in the Software Defined Networking paradigm: like POX, RYU, Floodlight or ONOS. Among the available ones, there are some reasons to choose OpenDaylight:

- ★ Open Source: As an open source platform, OpenDaylight platform provide a universal access via a free license to the platform, and universal redistribution of that platform including subsequent improvements to it by anyone.

According to this:

- The Linux Foundation manages it.
- Anyone interested is free to collaborate.
- Free access to the platform.
- Its services and behaviour can be modified by changing its prebuilt modules of implementing new ones.

- ★ **Industry support**: The industrial support of this platform encompasses the major part of the most powerful companies in the IT field. Some very important firms like Cisco, Intel, Microsoft... among many others, are part of the OpenDaylight members.

- ★ **Novel functionalities**: As a new platform, OpenDaylight implements some innovative functionalities respect other controllers. For instance the possibility of contacting with external applications that are not strictly connected with the network. This functionality is very useful and it is widely explained during the thesis, reaching the point that Chapters 5 and 6 are focalized on this topic.

★ **Running in a personal Java Virtual Machine (JVM):** The main advantage is the portability that it offers. Running the platform in its own JVM proportionate the possibility to use and develop OpenDaylight on any hardware that support Java.

★ **OpenDaylight is a cloud friendly environment**: it will include an Open Stack Neutron, OpenStack's virtual networking plug-in, and the Open vSwitch Database project will allow management from within OpenStack [10].

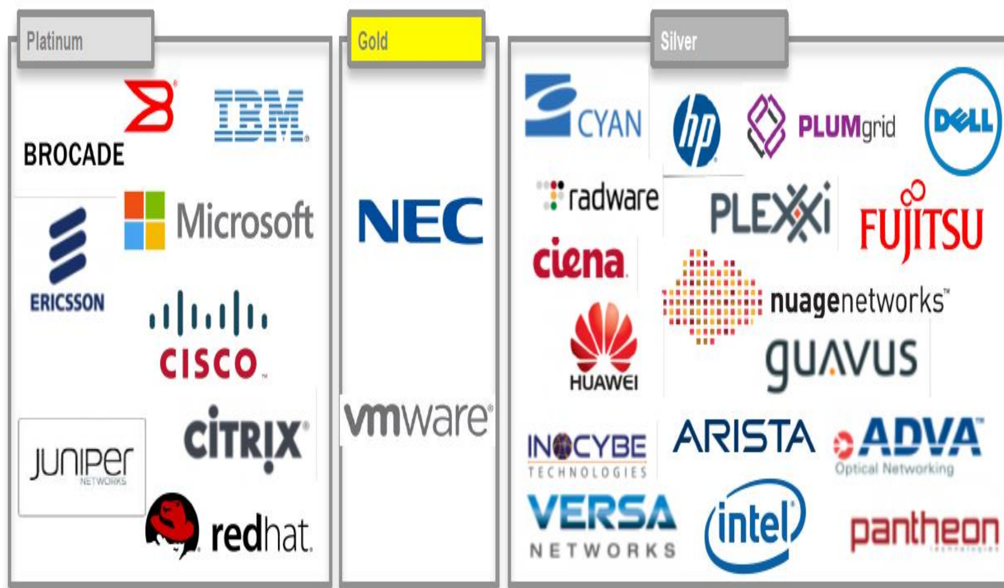


Figure 3.1 OpenDaylight Founding Members [54]**.**

3.3 The Structure

The following picture is offered by the OpenDaylight official site and shows the structure of the platform and the tools or devices, which it can contact with.

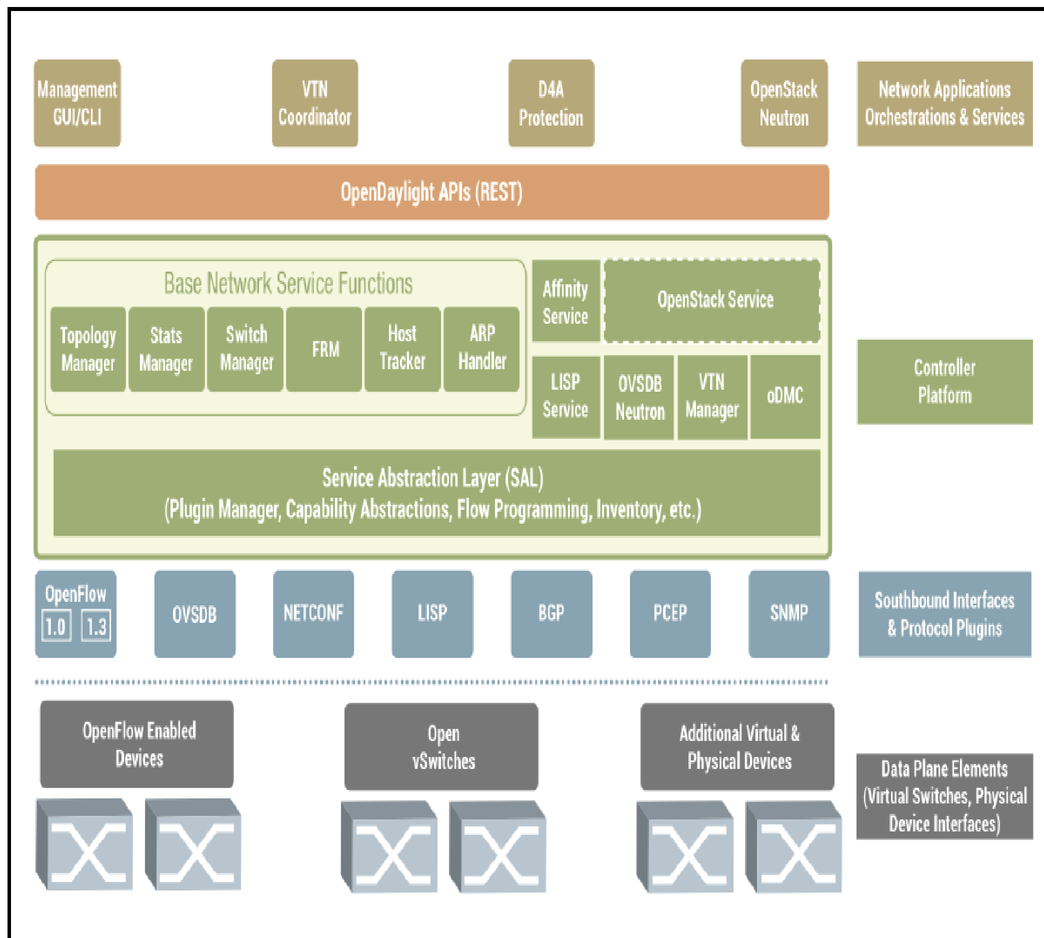


Figure 3.2: OpenDaylight modular structure platform[55].

To know the full platform overview, it is important to understand the above diagram (figure 3.2):

* **OpenFlow Enabled Devices**: This is the network infrastructure managed via OpenDaylight. In this case OpenFlow plug-in.
* **Protocol Plugins**: OpenDaylight supports these protocols for managing your network devices. One of the plugins is OpenFlow.
* **Service Abstraction Layer (SAL)**: This layer does all the plumbing between the applications and the underlying plugins.
* **Controller Platform**: These are the applications that come pre-bundled with OpenDaylight to manage the network. It is possible to write our own bundle.
* **Bundles:** They are each of the green small boxes inside the Controller Platform.

* **Network Applications**: These are applications with leverage REST NBI of OpenDaylight to build intelligence.

3.3.1 Two different architectures (AD-SAL and MD-SAL)

The Service Abstraction Layer, or SAL, is nothing more than a pipe that connects the Protocol Plugins with the bundles and the REST APIs as shown in Figure 3.2. Thus, the SAL API are the contract that the Protocol Plugins and the NFS (Application on the top of SAL layer: bundles, external applications...) sign, in order to be able to communicate to each other. It is also used to talk between bundles themselves or bundles and REST API.

There are two different approaches to the SAL that can be taken into account when programming applications for OpenDaylight: the API-Driven SAL (AD-SAL) and the Model-Driven SAL (MD-SAL).

3.3.1.1 AD-SAL

AD-SAL approach is more useful in case of requesting to control or program a network where it is allowed to access directly to the controller. It is also important to mention that the controller should not need to interchange much information between an external programs through the Northbound Interface, to OpenFlow through the Southbound Interface, since this approach does not provide a common REST API.

In these favourable cases, the advantages of this architecture fall on the usability of the service adaptation which is well developed and with a wide range of Java classes and methods to process a packet-in.

Figure 3.3 shows the AD-SAL approach, which has the following main characteristics:

* It can be used with both southbound and northbound plugins.
* It is stateless.
* It is limited to flow-capable devices and services only.

* The applications are programmed into the controller as OSGi bundles.
* The flow programming is reactive, by receiving events from the network.

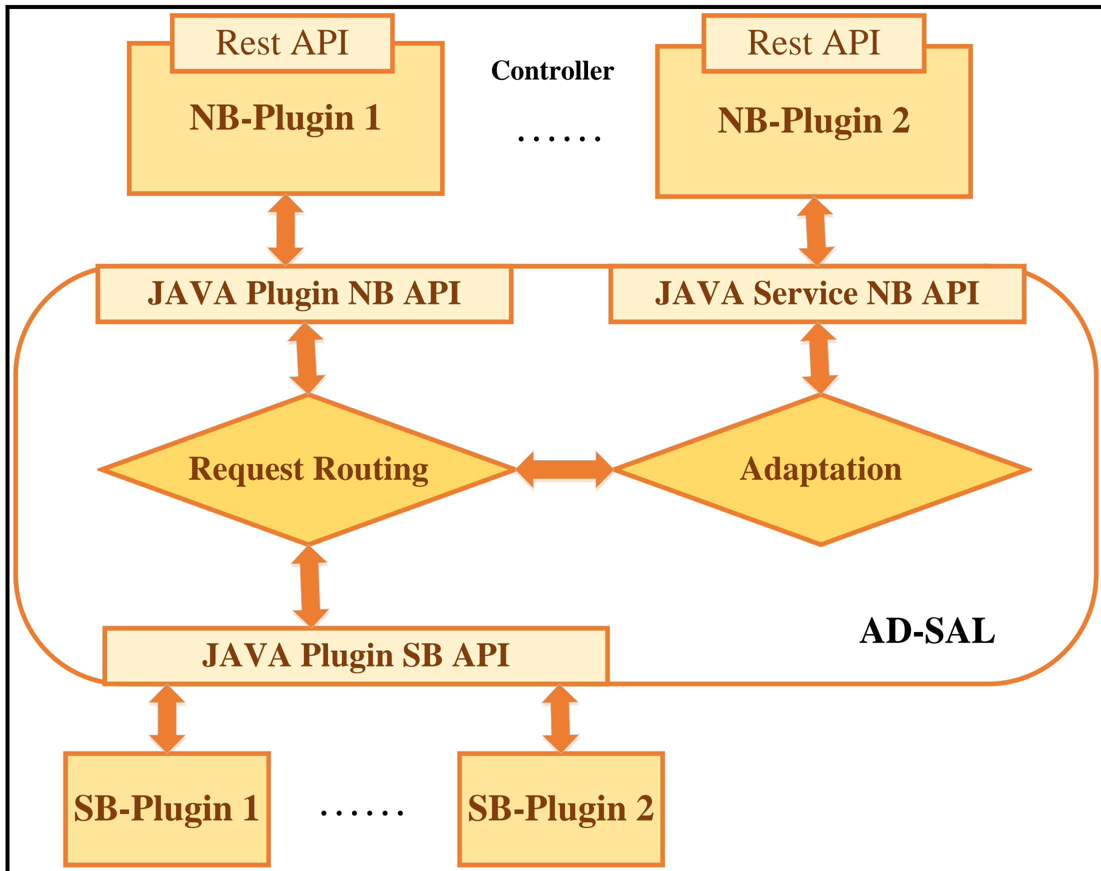


Figure 3.3 ADSAL architectures.[56]

3.3.1.2 MD-SAL

MD-SAL approach is preferable if the programmer cannot access physically to program the controller or he need to access the controller by an external application that needs to often contact OpenFlow. The reason is that MD-SAL provides a common infrastructure where data and functions defined in models can be accessed by means of a common REST API. So, installing an OpenFlow flow can be directly through the Northbound is quite simpler.

Figure 3.4 illustrated MD-SAL approach, which has the following features:

* It has a common REST API for all the modules.
* It can store data for models in permanent or volatile APIs.

* It is model agnostic. It supports any device or service models.
* The applications are programmed outside the controller.
* The flow programming is proactive, without the possibility to receive events from the network.

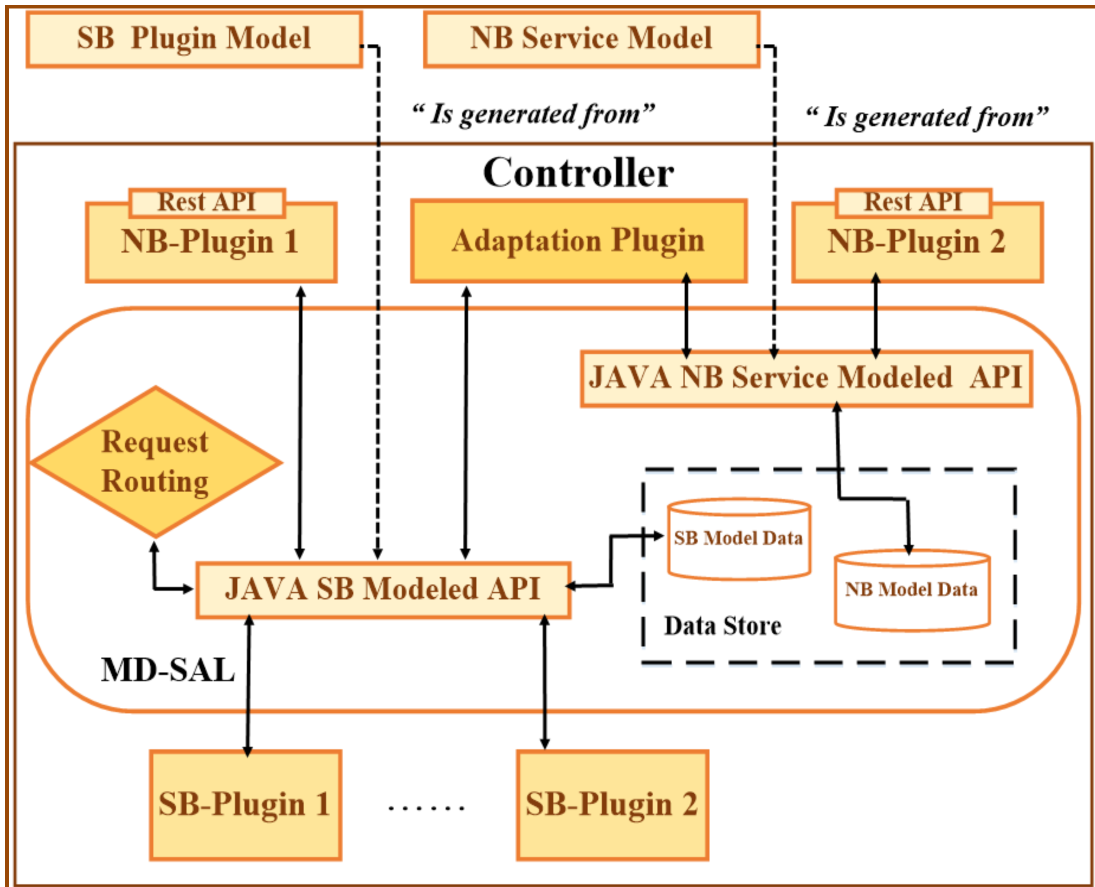


Figure 3.4 MD-SAL architectures.[56]

3.3.2 Packet path in OpenDaylight managing Open- Flow devices

After the knowledge of the details about OpenDaylight architecture, let's see which elements are involved in the installation of a flow in the both possible SAL approaches:

3.3.2.1 In AD-SAL Architecture:

In AD-SAL the packet path begins from an OpenFlow enabled network device and goes to the controller. Then, the controller processes it and it comes back to the network device.

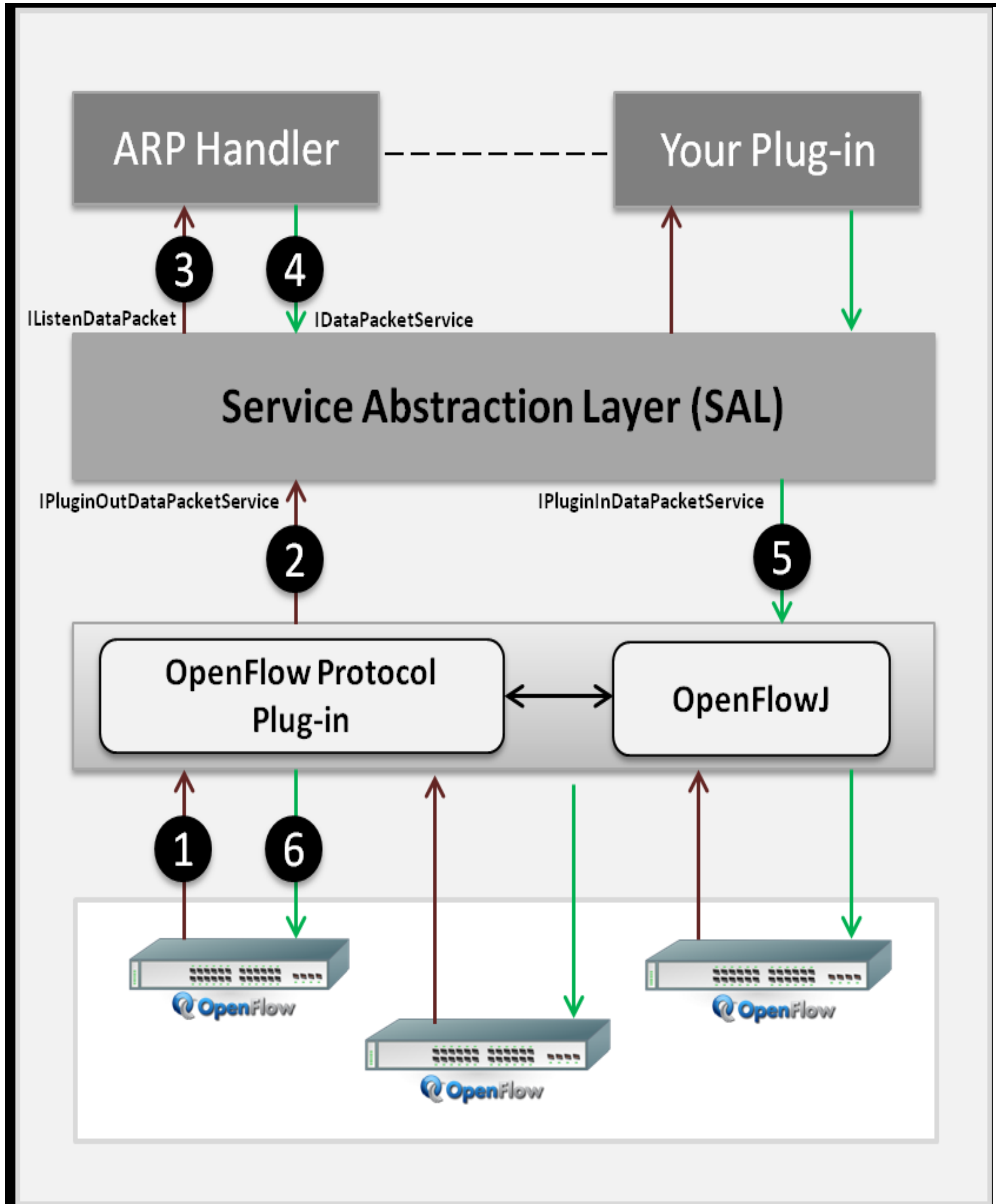


Figure 3.5 Packet path in ADSAL managing OpenFlow devices [31].

Figure 3.5 described how a packet flows between an OpenFlow enabled network device to the controller and which interfaces are being used. It is also explained which the return back path of the packet is:

1. A packet arriving at a network device, like an OpenFlow switch, is sent to the appropriate protocol plugin of OpenDaylight which is managing the switch.

2. IPluginOutDataPacketService: The plugin (in our case OpenFlow Plug-in) will parse the packet, generating an event for Service Abstraction Layer (SAL) for further processing.

3. IListenDataPacket: SAL will dispatch the packet to all the modules listening for DataPacket. If the first packet is just an ARP Request, SAL will send it to the ARP Handler. However, if it is another kind of packet, it will be processed by its appropriate bundle.

This configuration and also all these bundles can be changed. Furthermore, other bundles can be created following the user necessities.

4. IDataPacketService: The Application/Module that has been commented in (3) is in charge of the packet processing in accordance with its own needs, so it is the user who can actually program it. Finally, a PACKET OUT is sent using IdataPacketService interface. For example, after processing the packet, the PACKET OUT can be a rule of where the OpenFlow switch has to send the packet and also install in this device a new flow.

5. IPluginInDataPacketService: SAL receives the DataPacket and dispatches it to the modules listening for plug-in DataPackets. In this case OpenFlow plugin.

6. OpenFlow plugin then sends the packet back to the device from where the packet was originated. Following the previous example, once the switch receives the DataPacket, it will know where to send the packet previously sent to the

controller. In addition, it will install a new flow in order to flow directly the upcoming packets with the same characteristics.

3.5.2 *In MD-SAL Architecture:*

AD-SAL is not the only way to insert flows. Figure 3.6 shows a scenario where an external application adds a flow by means of RESTconf API of the controller. Whereas in the previous example were about dealing with a reactive approach, in this case it is a proactive scenario.

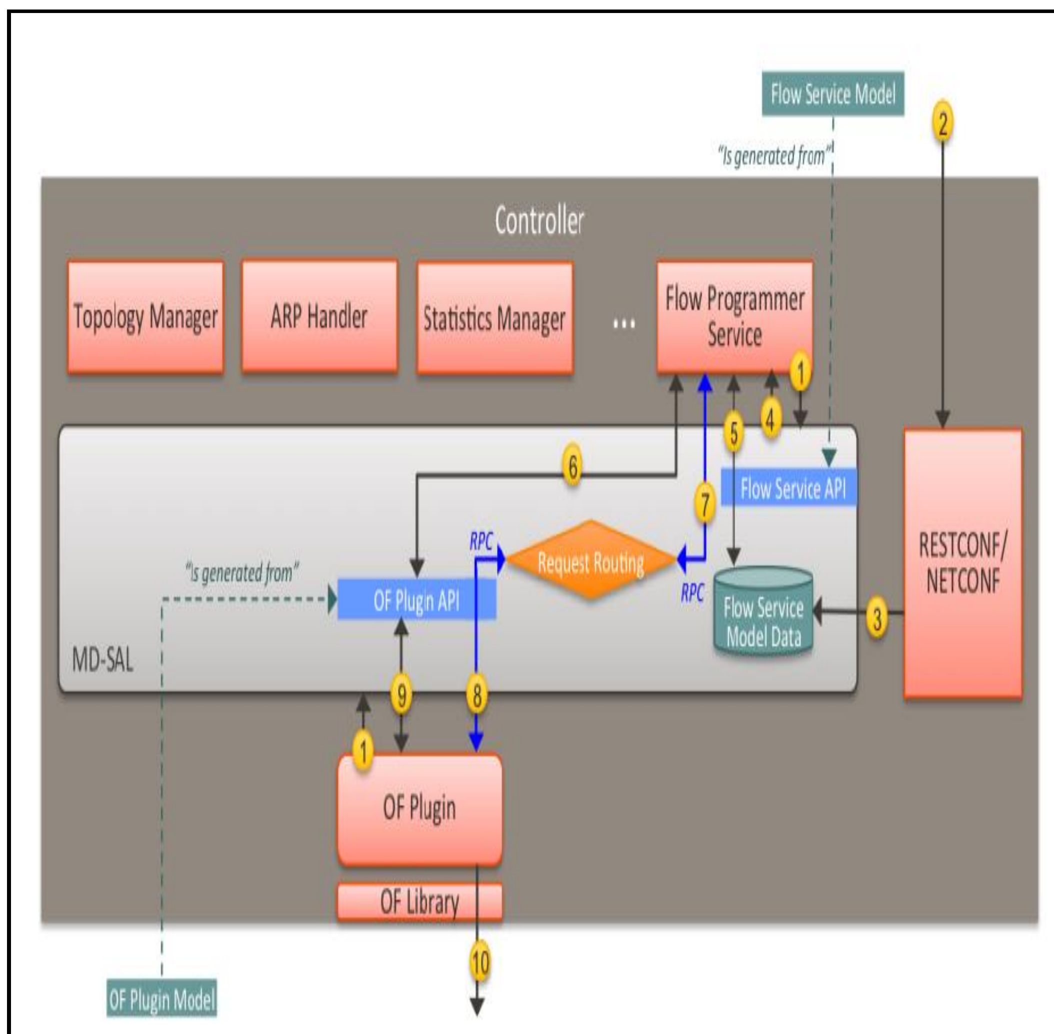


Figure 3.6 Packet path in MD-SAL managing OpenFlow devices [21].

1. When the controller is started with its corresponding plugins, the followings registrations are performed:

* The Flow Programmer Service registers with the MD SAL for Flow configuration data notifications.
* The registration between the OF Plugin and the MDSAL between a Remote Procedure Call (RPC) to establish a connection between both platforms. Note that the RPC is defined in the OF Plugin model, and that the API is generated during build time.

2. A client application that would be running anywhere establish a REST connection with the REST API of the OpenDaylight controller sending a flow add request. As has already been said, in AD-SAL there is a dedicated Northbound REST API on top of the Flow Programming Service. The MD-SAL provides a common infrastructure where data and functions defined in models can be accessed by means of a common REST API. The client application provides all parameters for the flow in the REST call.

3. Data from the 'Add Flow' request is de-serialized, and a new flow is created in the Flow Service configuration data tree. (Note that in this example, the configuration and operational data trees are separated; this may be different for other services). Note also that the REST call returns success to the caller as soon as the flow data is written to the configuration data tree.

4. The MD-SAL generates a 'data changed' notification to the Flow Programmer Service since this one is registered to receive notifications for data changes in the Flow Service data tree.

5. The Flow Programmer Service reads the newly added flow, and performs a flow add operation.

6. At this point the Flow Programmer Service tells the OF Plugin to add the flow in the appropriate switch. The Flow Programmer Service uses the OF Plugin generated API to create the RPC input parameter, called DTO, for the 'AddFlow'

RPC of the OF Plugin. So basically, the Flow Programmer Service communicates the OF Plugin API that it wants to add a new flow.

7. The Flow Programmer Service gets the service instance (actually, a proxy), and invokes the 'AddFlow' RPC on the service. The MD-SAL will the request to the appropriate OF Plugin (which implements the requested RPC).

8. The 'AddFlow' RPC request is routed to the OF Plugin, and the implementation method of the 'AddFlow' RPC is invoked.

9. The 'AddFlow' RPC implementation uses the OF Plugin API to read values from the DTO of the RPC input parameter. (Note that the implementation will use the getter methods of the DTO generated from the yang model of the RPC to read the values from the received DTO.)

10. The 'AddFlow' RPC is further processed (pretty much the same as in the AD-SAL) and at some point, the corresponding flow-mod is being sent to.

3.4 OpenDaylight Communication Technique.

Finally, in order to have a full idea of the all necessary functions that OpenDayLight process, it’s useful to understand the following steps which describes the communication procedure between the OpenDayLight controller and the OpenFlow switch.

1. A switch is discovered by the controller whenever a transport channel between them is first established.

2. The controller then discovers the physical ports of a connected switch along with its other features using **Feature Request** and **Feature reply** [57] OpenFlow messages.

3. Next, the controller determines the switch-to-switch adjacency (neighbouring switches) by generating periodic LLDP messages.

4. At first, the controller creates an LLDP message, puts it in the payload of an OpenFlow **PacketOut** message, and then sends it to one of its connected switches (switch A in Figure 3.7).

5. The action field in this **PacketOut** message tells the receiving switch to forward the LLDP packet to a particular port. The switch (switch B in Figure 3.7) at the other end of this link (if there is one) receives this packet, encapsulates it in an OpenFlow PacketIn message, and sends the encapsulated packet to its controller.

6. The controller then de-capsulate the packet and determines the originating switch (i.e. switch A) as well as discovering the switch-to switch adjacency (here, link A-B). Note that the LLDP messages traverse only one hop in this technique.

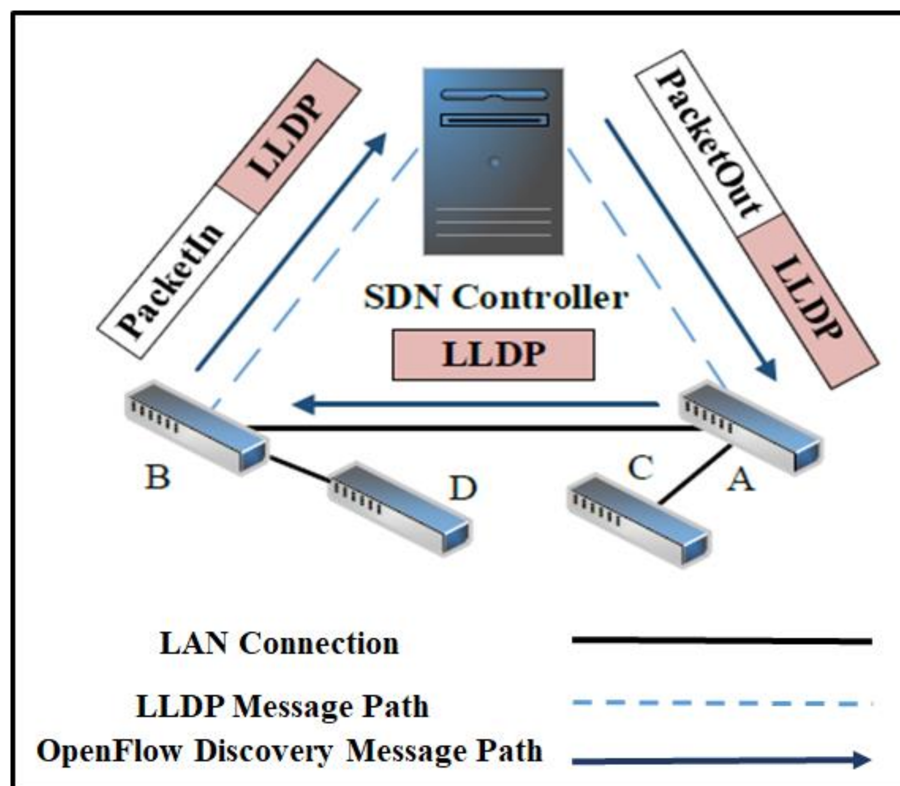


Figure 3.7 Topology Discovery in SDN.

3.5 Summary

The main improvement that OpenDaylight offers is the possibility of controlling a network by three different ways:

Scenarios can be controlled by implementing a bundle inside the controller, this module will probably be developed in the AD-SAL architecture due to the facilities this layer give when a programming inside the platform is being done.

In some occasions the control of the network has to be implemented outside the controller for different causes, in these situations, MD-SAL is the most intelligent choice as it provides a common REST API from outside the controller to the network devices.

Finally, a hybrid solution can be developed by programming the intelligence of the network externally but also by taking advantage of some bundles already implemented in OpenDaylight.

Chapter 4

Multiple-Controller with Different Operation Paradigm

Large scale complex networks face a lot of challenges, in particular, they require re-policing or reconfigurations from time to time. To this end, SDN, a model where a central program, known as a controller, orders the whole network behaviour, is increasingly gaining importance. The underlying concept of SDN is that of separating the traditional merging of the control plane and data plane, into different two entities, where the network routers/switches simply forward packets and the control/management plan is implemented by a centralised controller. At the present time, OpenFlow is the most common SDN protocol/standard, is concerned with the southbound interface between the controller and the routers/switches. However, a single controller being used in such a model has major difficulties, including lack of reliability and scalability. In this research, a scheme using multiple controllers which handle multiple network devices, while using OpenFlow controllers in the proactive operations paradigm had been proposed.

4.1 SDN Overview

SDN has gained a lot of attention in recent years, because;

- It addresses the lack of programmability in existing networking architectures; and
- Enables easier and faster network innovation.

SDN [22] is a paradigm that clearly distinguishes the data plane from the control plane and this promising architecture enables software implementations of

complex networking applications, with the expectation of less specific and cheaper hardware as well as the prospect of greater flexibility. With SDN, network Ethernet devices become simple packet forwarding devices [58], while the "brain" or control logic is implemented in the controller [59], [60], as shown in figure 4.1. SDN controllers and forwarding devices communicate with each other by the southbound SDN interfaces and OpenFlow [61] is one of the most common of this type. Many marketers, including HP, NEC, NetGear, and IBM, produce OpenFlow-capable network switches that are available in the marketplace [61]. The Open Networking Foundation (ONF) is responsible for standardising the OpenFlow protocol. There are a variety of OpenFlow controllers, for example, NOX, Floodlight, and OpenDayLight.

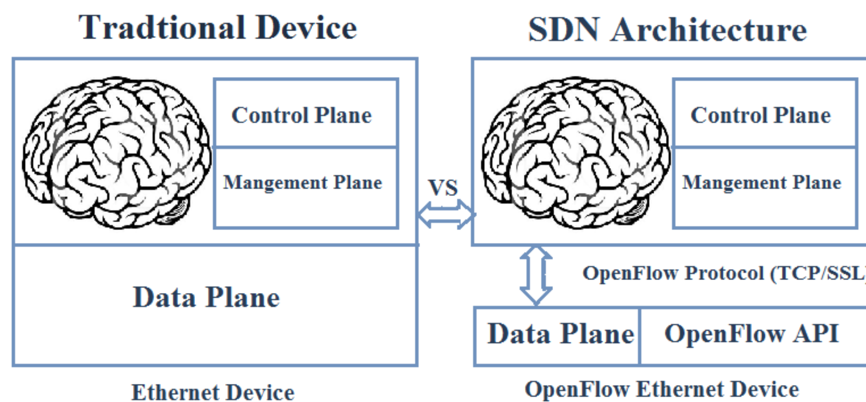


Figure 4.1 Traditional network Device VS SDN Architecture.

4.2 OpenFlow Protocol

The components of the OpenFlow architecture are: 1) the OpenFlow controller, 2) the OpenFlow device (switch), and 3) the OpenFlow protocol. Figure 4.2 shows the various components of this architecture. The OpenFlow protocol is the communication language through which the switch and the controller can understand each other, such that the controller is able to manage the switch by adding, updating, and deleting flow entries in flow tables, no matter which flow setup mode (reactive or proactive) is used.

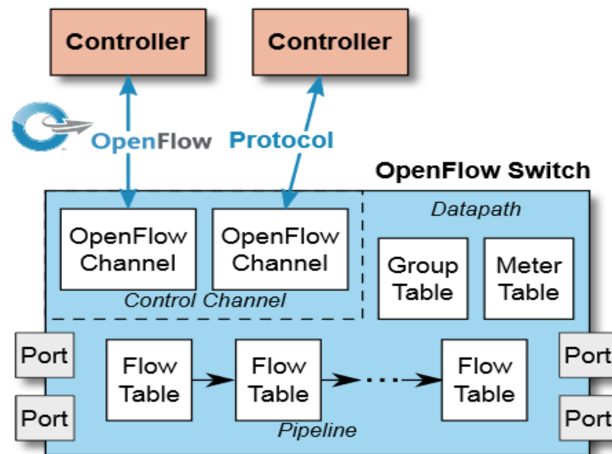


Fig 4.2 Main components of an OpenFlow switch [50]**.**

4.3 State of the Art

In distributed control hierarchy the load can be reduced significantly and this approach has been applied in many applications, such as FlowVisor, Hyperflow, and Kandoo. HyperFlow tries to provide scalability, using as many controllers as necessary to reach it, but keeping the network control logically centralised. Specifically, the Hyperflow mechanism is aimed at synchronizing the state of a network with all available controllers[61]. Kandoo sorts applications by its scope: local and network wide, with the former being deployed in the locality of datapath for processing requests and messages there, thereby reducing controller load. Network-wide applications are handled by a controller, and in a distributed hierarchical arrangement, a root controller takes care of them and updates them to all other controllers in SDN [24]. Flowvisor slices the network, with each slice being handled by a controller or a group of controllers, thus reducing the load and making an efficient decision handling mechanism [15].

Another proposal, the DevoFlow [23] and The "Doing It Fast And Easy" (DIFANE) [24]. In the DevoFlow proposal the aim is to deal with the scalability problem by evolving network control to switches with an aggressive use of a flow wildcard and introducing a mechanism to improve visibility [39]. While the

DIFANE approach examines the scalability issues that arise with OpenFlow in large networks and with many fine-grained flow entries. It proactively pushes all states in the data path (addressing the lack of data path memory in a scalable manner), by installing all forwarding information in the fast path.

4.4 Plan Choices for OpenFlow-Based SDN

Today, SDN is mostly applied for flexible and programmable data centres. There is a need for network virtualization, energy efficiency and dynamic formation and enforcement of network policies. An important feature is the dynamic creation of virtual networks, commonly referred to as network-as-a-service (NaaS). Even more complex requirements arise in multi-tenant data centre environments. SDN can provide these features easily, due to its flexibility and programmability.

However, SDN is also discussed in a network or Internet service provider (ISP) context. Depending on the use case, the design of SDN architectures varies a lot. In this section, a clarification of the architectural design choices for SDN had been demonstrated. A discussion of their implications regarding to performance, reliability and scalability of the control and data plane will be made.

4.4.1 Physically vs. Logically Centralized in the SDN Control Plane

Originally, a centralized control plane is considered for SDN. It provides a global view and knowledge of the network and allows for optimization and intelligent control. It can be implemented in a single server, which is a physically centralized approach. Obviously, a single controller is a single point of failure, as well as a potential bottleneck. Thus, a single control server is most likely not an appropriate solution for networks, due to a lack of reliability and scalability.

As an alternative, a logically centralized control plane may be used to provide more reliability and scalability [62]. It consists of physically distributed control

elements that interface with each other through the so-called east- and west-bound interface which is illustrated in Figure 4.3. Since that distributed control plane interfaces with other layers, like a centralized entity, the data plane and network applications see only a single control element. A challenge for logically centralized control is the consistent and correct network-wide behaviour. Another common term for the SDN logically centralized control plane is the "network operating system" (network OS).

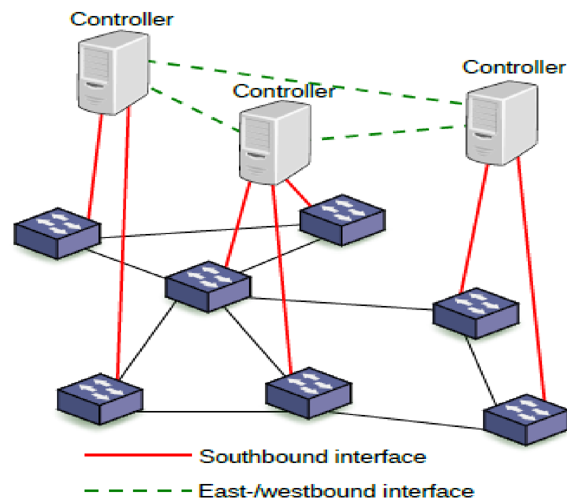


Figure 4.3. Logically centralized control plane [62]**.**

Several studies investigated the feasibility, scalability and reliability of logically centralized control planes. Some of other researches, investigate the issue of the placement of the distributed control elements inside the network [44][38].

The importance of limited latency for control communication in OpenFlow-based networks has been highlighted by McKeown in [44]. Thus, to meet these latency constraints, they propose a number of required controllers with their position in the network. Another consideration has been done by Hock et al. [63] to optimize the placement of controllers with regard to latency, in addition to controller load, reliability and resilience. Their proposed scheme can be used to implement a scalable and reliable SDN control plane. In the HyperFlow proposal, the authors of [39] intend a control plane that is based on the NOX OpenFlow controller.

They discuss the management of network applications and the consistent view of the framework in detail. For example, when a link breaks, one controller notices the failure, but other controllers may not be aware of the connection failure. The HyperFlow architecture ensures in such cases that network applications operate in a consistent state of the network, even though control elements may not share identical knowledge about the network. A hierarchical control platform called Kandoo is proposed in [40], which arranges controllers in a layers structure as there are some lower and higher layers are constructed. Controllers in lower layers process local network events and program the local portions of the network under their control. Controllers on higher layers sort network-wide decisions. In particular, they instruct and query the local controllers at lower layers.

Another study [64] compares the performance of network applications that run on a distributed control platform. Network applications that are aware of the physical decentralization showed better performance than applications that assume a single network-wide controller.

4.4.2 In-Band vs. Out-of-Band Signalling in Control Plane

In the following, a discussion about the communication between the control components and the forwarding devices on the data plane will be clarified. This control channel has to be strong and dependable. In SDN-based data centre networks, this control channel is often constructed as a separate physical network in parallel with the data plane network. Carrier and ISP networks have different demands. They often cross over a rural area or a continent. Hence, a separate physical control network might not be price-effective or feasible at all. Therefore, two main design options for the control channel exist: in-band control plane and out-of-band control plane.

With in-band control, the control traffic is sent like data traffic over the same infrastructure. This is depicted in Figure 4.4. This variant does not involve an

additional physical control network, but has other major disadvantages. Firstly, the control traffic is not distinguished from the data traffic, which raises security worries. Failures of the data plane will likewise touch on the control plane. Therefore, it is possible that a failure disconnects the switch from its control element, which makes the restoration of the network more complicated.

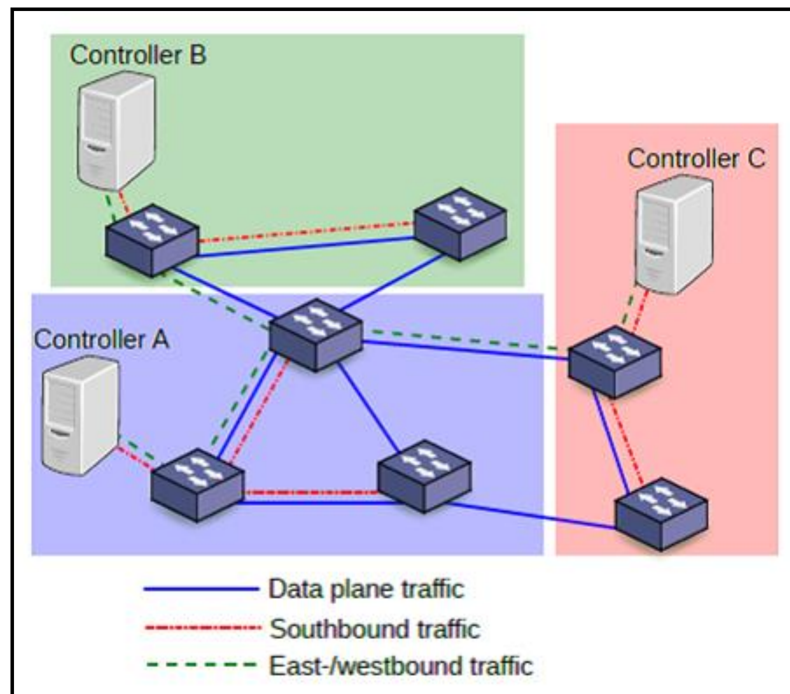


Figure 4.4 In-band signalling [62].

Out-of-band control requires a separate control network in addition to the data network, as illustrated in Figure 4.5. This is a common approach in data centres that are fixed in geographical size. In the data centre context, the maintenance and cost of an additional control network are commonly satisfactory. This may be different for wide-ranging networks, such as carrier networks, where a separate control network can be costly with regard to CAPEX and OPEX. The advantages of an out-of-band control plane are that the separation of data and control traffic improves security. Data plane network failures do not affect control traffic, which eases network restoration. Moreover, a separate control plane can be implemented more securely and dependably than the data plane. This ensures the high

availability of the control plane and can be crucial for disruption-free network operation [62].

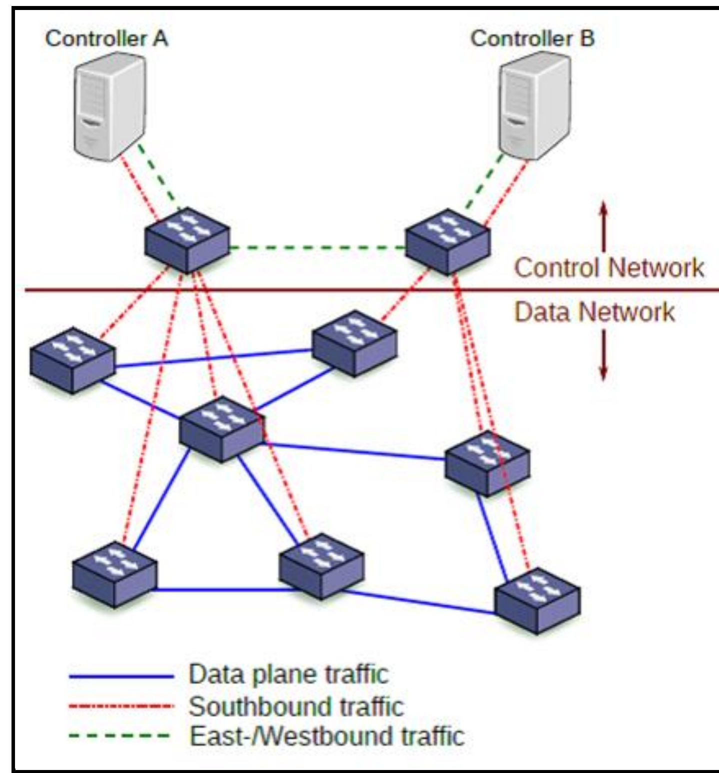


Figure 4.5 Out-of-band signalling [62].

A potential approach that combines in-band and out-of-band control planes for carrier networks is based along the control planes in optical networks, such as synchronous optical network (SONET) or synchronous digital hierarchy (SDH) and optical transport networks (OTN). In such networks, an optical supervisory channel (OSC) may be shown on a separate wavelength, only along the same fibre over which data traffic is transported. In a similar way, a dedicated optical channel could be allocated for SDN control traffic when an optical layer is available. Moreover, other lower layer separation techniques may be utilized to implement separated control and data networks over the same physical base.

4.4.3 Multiple Controllers in Control Plane

The switch can establish communication with a single controller or with multiple ones [50][39]. Having multiple controllers improves reliability, as a switch can continue to operate in OpenFlow mode, if one controller or controller connection fails. The handover between controllers is initiated by the controllers themselves, which enables quick recovery from failure and also controller load balancing. The controllers coordinate the management of the switch between themselves, and the aim of the multiple controller functionality is only to help synchronise controller handoffs performed by the controllers. When OpenFlow operation is started, the switch must connect to all controllers it is configured with, and attempt to maintain connectivity with all of them concurrently. Although many controllers send controller-to-switch commands to the switch, the reply or error messages related to those commands must only be sent on the controller connection accompanying with that command. Asynchronous messages might need to be sent to multiple controllers, the message being duplicated for each eligible OpenFlow channel and each of them are sent when the respective controller connection allows it[50].

There are three operational modes for controllers that have been defined in OpenFlow 1.3: (1) master, (2) slave and (3) equal [50]. The default role of a controller is `OFPCR_ROLE_EQUAL`. In this role, the controller is equal to other controllers with the same role and has full access to the switch. A controller can demand its role to be changed to `OFPCR_ROLE_SLAVE`, whereby it has read-only access to the switch. Otherwise, the controller can request its role to be changed to `OFPCR_ROLE_MASTER`, which is similar to `OFPCR_ROLE_EQUAL` such that it has full access to the switch, with the difference being that the switch make sure it is the only controller in this role. When a controller changes its role to `OFPCR_ROLE_MASTER`, the switch modifies all other controllers with that role to `OFPCR_ROLE_SLAVE`.

A switch may be at the same time connected to multiple controllers in equal state, multiple controllers in slave state, and at most one controller in the master state. Each controller can communicate its role to the switch via an `OFPT_ROLE_REQUEST` message and the latter must remember the role of each controller connection, which can change at any time.

4.4.4 Proactive vs. Reactive in Management of Flow Entries

Starting with explaining why flow tables in OpenFlow switches are limited in size, and then, discussing two approaches for flow table entry management.

In the SDN architecture, the control plane is the element which is responsible for the configuration of the forwarding devices. By using the OpenFlow protocol, the controller installs flow table entries in the forwarding tables of the switches. As discussed in Chapter 2 Section 2.3.3.1, an entry consists of match fields, counters and forwarding actions. The OpenFlow match fields are wildcards that match to precise header fields in the packets. Wildcards are typically installed in ternary content-addressable memory (TCAM) to certify fast packet matching and forwarding. Though, TCAM is very expensive, so that it needs to be small; as a consequence, only a modest number of flow entries can be accommodated in the flow table. In the following, a description of two flow management approaches will be explained, which are the proactive and the reactive flow management. Both options are not equally exclusive: it is common in OpenFlow networks to install some flows proactively and the remaining flows reactively.

The flow entries can be installed temporary and in particular before they are actually needed by the controller. This approach is referred to as proactive flow management [58]. However, this approach has a disadvantage: flow tables must hold many entries that might not fit into the expensive TCAM.

To manage with small flow tables, flow entries can also be installed reactively, i.e., installed on demand. This is clarified in Figure 4.6. (1) Packets can arrive at a switch where no corresponding rule is installed in the flow table, and therefore, the switch cannot forward the packet on its own; (2) The switch notifies the controller about the packet; (3) The controller identifies the path for the packet; (4) The controller installs appropriate rules in all switches along the path; (5) Then, the packets of that flow can be forwarded to their destination.

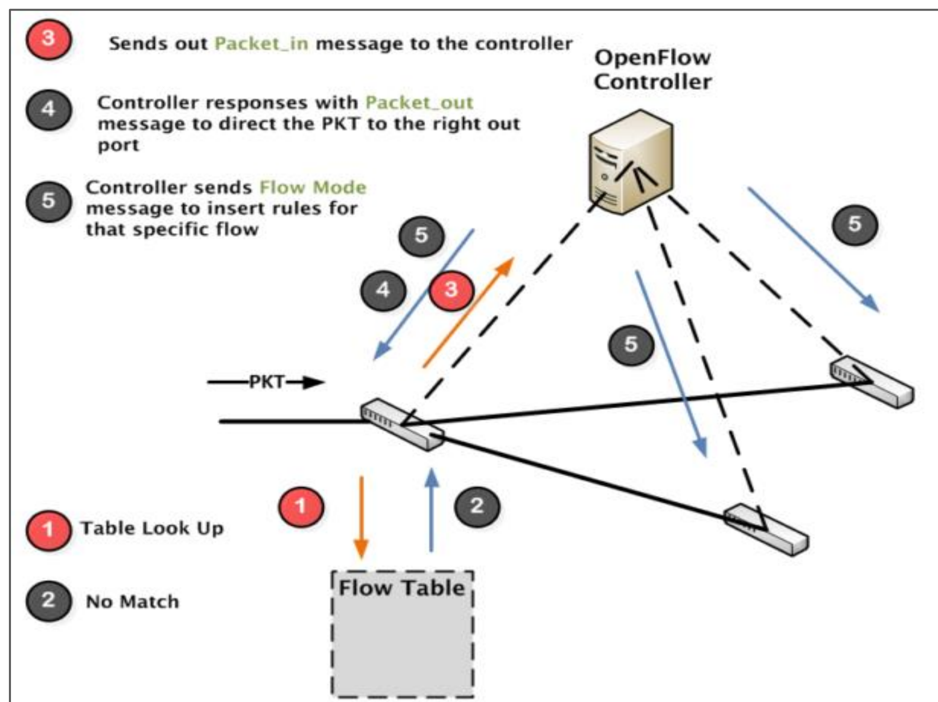


Figure 4.6 Reactive flow management.

The mechanisms to enable reactive flow management in OpenFlow are based on timeouts. The controller sets an expiry timer that defaults to one second. The switch tracks the duration to the last match for all entries. Idle entries are removed from the switch. When more packets of an expired flow arrive, the controller must be asked for path installation again.

Both flow management approaches have different advantages and disadvantages. The ***reactive flow management*** approach holds only the lately used flow entries in the table. On the one hand, this allows one to handle with small forwarding tables.

On the other hand, controller interface is required if packets of a flow arrive in a switch that has no appropriate entry in the flow table. After some time, the correct entry will be ultimately installed in the switch, so that packets can be forwarded. The resulting delay depends on the control channel and the current load of the controller. Thus, reactive flow management reduces the state in the switches and relaxes the need for large flow tables, but it increases the delay and reliability requirements of the control channel and control plane software. Particularly, failures of the control channel or the controller will have substantial impact on the network performance if flow entries cannot be installed in a timely manner [62].

With ***a proactive flow management*** approach, all required flow entries are installed in the switches by the controller as described in figure 4.7. Depending on the use case and network, a huge amount of flows must be installed in switches, e.g., BGP routing tables can have hundreds of thousands IP prefixes. This may require switch memory hierarchy optimizations that are not needed with reactive flow management. While proactive flow management increases state requirements in switches, it reduces the requirements on the control plane and software controller performance and is stronger against network failures in the control plane. That means that if the controller is overloaded or the communication channel fails, the data plane is still fully functional.

Based on the above-mentioned explanation, the main advantage of the proactive mode is the ability to make the system function properly in case of any fail of the communication channels comparing with the reactive mode, which makes the network reliable and strong, also it reduces the time delay and that’s why this was my best choice to use for my proposed scheme.

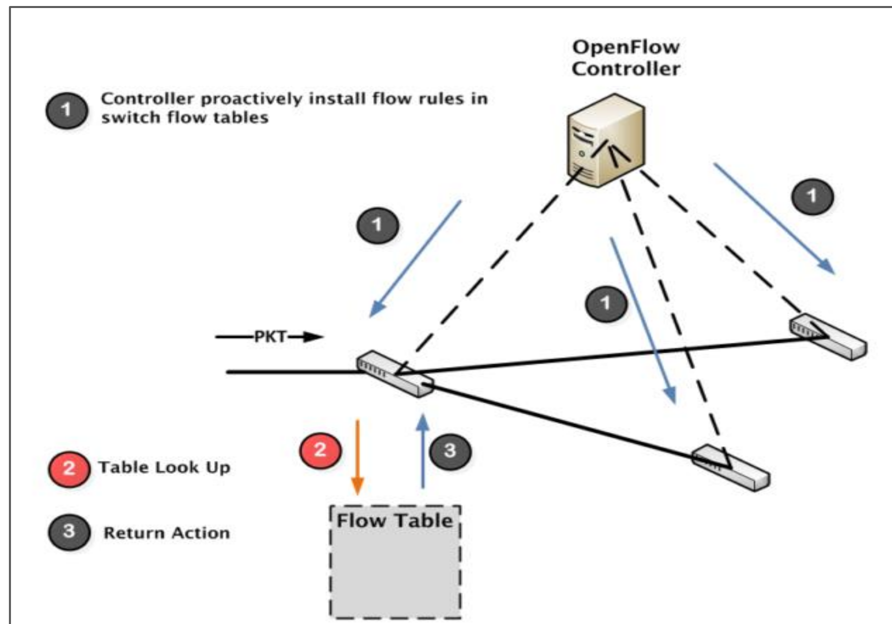


Figure 4.7 Proactive flow management.

4.5 Methodology and Network Design

In order to test and evaluate our network, Mininet is used which is a network emulator that simulates a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization which is the capability of an operating system to be installed directly on the server hardware and provides the functionality to create Virtual Servers, and this to make a single system look like a complete network. Mininet is important tool to the open source SDN community as it is commonly used as a simulation, verification, testing tool, and resource. Furthermore, it can be used to create an accurate virtual network with real working components but running on a single machine for ease of testing. It's an open source project hosted on GitHub which allows us to create custom topologies, and provides the ability to create hosts, switches and controllers via: 1) Command Line, 2) Interactive User Interface, and 3) Python application [65].

The reason of using Mininet in our experiments is how widely it used for experimentation that it allows us to create custom topologies, many of which have been demonstrated as being quite complex and realistic, such as larger, Internet-

like topologies that can be used for BGP research. Another important feature of Mininet is that it allows for the full customization of packet forwarding.

In table 4.1, an illustration of the software which had been used in the

i

Software	Function	Version
Mininet	Network Emulator	2.2.1
Open vSwitch	Virtual SDN Switch	2.0.2
OpenDayLight	SDN Controller Platform	Lithium-SR4
VMWare Workstation	Virtualization Software	11
Linux (Ubuntu)	Host Operating System	14.4
Python	Programming Language	2.7.6

p
l
e
m
entation and experiments of this research.

Table 4.1 Software used in implementation and experiments.

Furthermore, as aforementioned the test environment chosen for this research was the OpenFlow emulation virtual machine using Mininet. The experiment was built using VMware Workstation 11.1.0. The ***hp*** EliteBook workstation PC had been used in this research and it has the following specifications; processor Intel ® Core ™ i7 – 3630QM CPU 2.4 GHZ,16GB of RAM running the operating system Windows 8.1 Pro 64bits. As such, the test environment implements and performs the actual protocol stacks that communicate with each other virtually. That is, the Mininet environment allows for the execution of real protocols in a virtual network.

4.6 Proposed scheme

In an attempt to start describing our design, an explanation of our architecture, operation steps and network scenarios will be clarified as follows:

4.6.1 Topology

In this research, multiple controllers had been used as shown in Figure 4.8 instead of the single, logical controller which shown in Figure 4.9. That is, it was recognised that a network device can be managed by more than one controller such that they can share its management, and used the proactive flow mode to manage the flow entries.

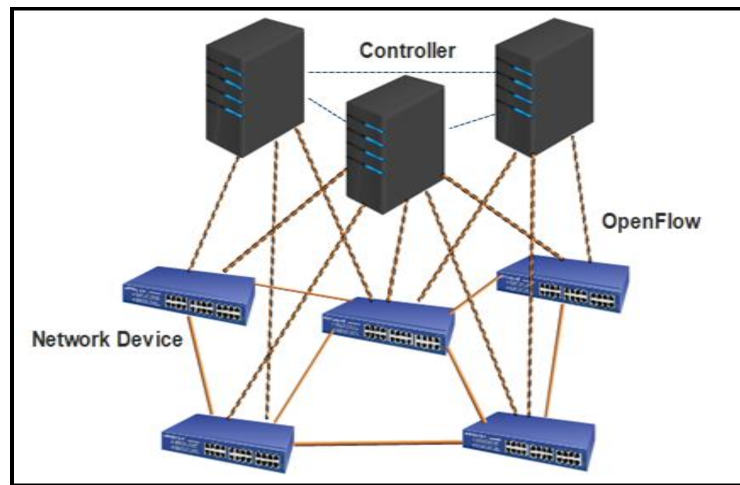


Figure 4.8 Multiple controllers managing multiple network devices.

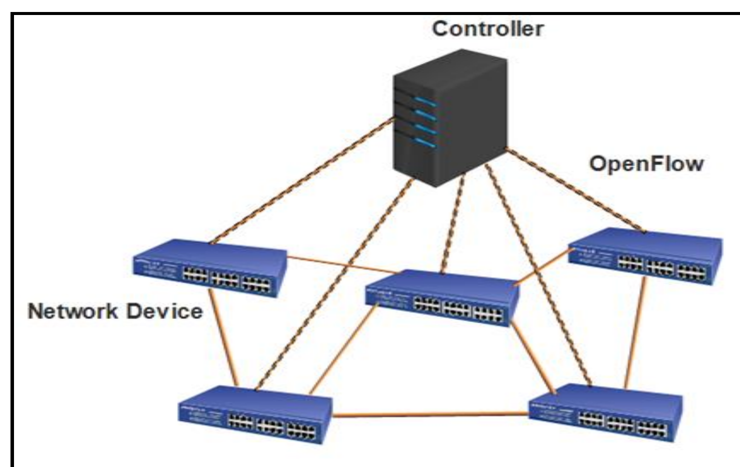


Figure 4.9. Single controller managing multiple network devices.

4.6.2 Operation steps

1. Initiate a cluster of multiple controllers.

2. Configure the OpenFlow switches to connect to the multiple controllers of the cluster whenever the transport channel between them has been established and consequently, each switch has an active connection to more than a single controller.

3. Each controller chooses to accept the connection initiated by OpenFlow switches.

4. After a connection is set up, the controller sends a message to get the OpenFlow switch's information.

5. The controller uses some form of broadcast request and ACK mechanism, like ARP, to determine the reachability of the target host. The information collated by the Link Discovery module is used to build the neighbour database in the controller by capturing all the OpenFlow neighbours of a given OpenFlow Switch in the network.

6. By using the information that has been gathered in step5 the controllers populate the flows proactively in the flow tables of the switches.

7. When a connection is broken, after a random time interval, the network device will re-initiate a connection to the controller. Then go to step3.

4.6.3 Network Scenarios

A network with the proactive mode had been created to make a comparison with the reactive mode using a centralised controller in one experiment, whilst in another a different controllers had been used, and in a third a multiple peer to peer controllers with equal roles had been used, subsequently comparing all the three scenarios. The controller that was used in all of our experiments is the ODL controller, which is a collaborative, open source project aimed at advancing SDN. In this experiment, an installation of the ODL controller in four Ubuntu servers

had been made, and they were connected remotely to the network that had been created in this research. Layer 2 connectivity is discovered by an SDN controller. Three network topologies were used to test this scheme. So, linear topology, star topology, and tree topology had been used as shown in Figure 4.10 (a), (b), and (c) respectively captured from the GUI of the Opendaylight controller that had been used in this work.

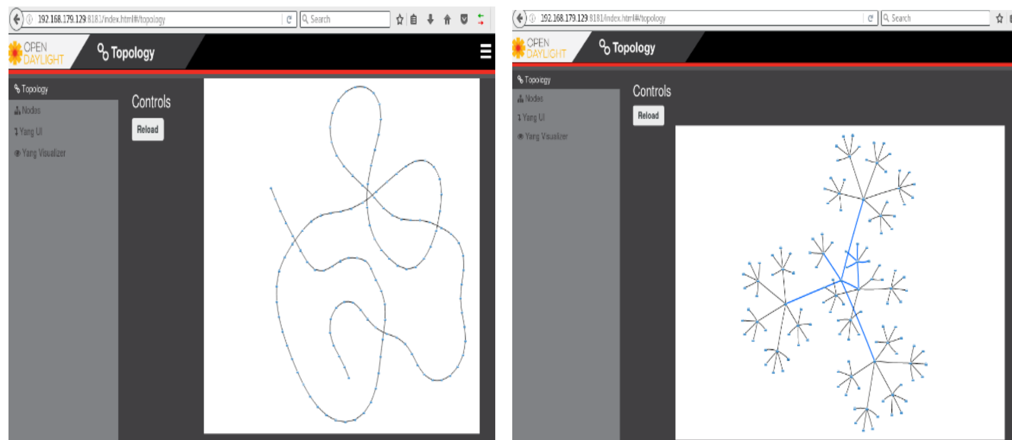
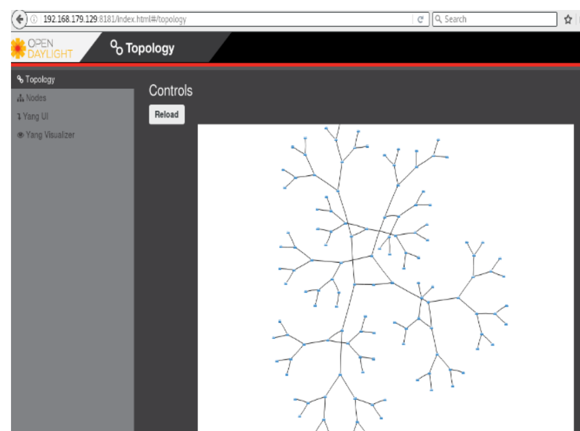


Figure 4.10 (a) Linear Topology, (b) Star Topology,



(c) Tree Topology.

The Mininet environment allows the execution of real protocols in a virtual network. First, four experiments had been created to compare among them in which is the best result, for the *first* comparison the subsequent scenarios had been created:

1. Proactive-multiple controllers with linear network topology scenario.
2. Proactive-single controllers with linear network topology scenario.
3. Reactive-multiple controllers with linear network topology scenario.
4. Reactive-single controllers with linear network topology scenario.

For the *second* comparison the following scenarios had been made:

1. Proactive-multiple controllers with Tree network topology scenario.
2. Proactive-multiple controllers with Star network topology scenario.
3. Proactive-multiple controllers with linear network topology scenario.

In the *third* comparison an evaluation to distinguish the difference between multiple controllers and the use of different controllers had been made, which is how most researchers built their consumptions regarding the distributed controller architecture.

1. Proactive-multiple controllers with linear network topology scenario.
2. Proactive-different controllers with linear network topology scenario.

4.6.4 Network configuration

After execution of Python script in Mininet to launch our networks, a network model is created 128 virtual hosts, each with a separate IP address as defined in Python code, and created 128 OpenFlow software switches in the kernel having required ports, in which one virtual hosts per switch with a virtual Ethernet cable. Furthermore, 128 virtual hosts are distributed linearly between 128 switches and all switches are connected with each other as well via same virtual Ethernet cable. Next the process of setting the MAC address of each host equal to its IP address is established. Finally, the configuration progression of the OpenFlow switches to be connected to our four remote controllers.

4.6.5 Evaluation suite for Mininet

Another concern of Mininet networks is how to evaluate these networks. After being able to create all of our network scenarios and configured all of our network elements, a test of this proposed scheme had been made, the evaluation of several metrics needs to be concerned. And for traffic generation, the D-ITG [66] had been used in this research.

4.6.5.1 Distributed Internet Traffic Generator D-ITG

To evaluate the performance of Mininet D-ITG in version 2.8.1-r1023 was used: Distributed Internet Traffic Generator (DITG) is a platform capable to produce traffic that accurately adheres to patterns defined by the inter departure time between packets (IDT) and the packet size (PS) stochastic processes [67]. Therefore, it offers a rich variety of probability distributions for the traffic generation and uses some models proposed to emulate sources of various protocols. With it, it is possible to generate various packet streams and collect statistics with a logging server. In Figure 4.11 all the important modules of the D-ITG are depicted. The ITGSend module is responsible for the traffic generation, while the ITGRecv module is the sink for the packets, which are delivered over a Data Channel. To collect logging information both, the ITGSend and ITGRecv are communicating via a Log Channel with the ITGLog module. For remote control the ITGManager offers the functionalities to adjust parameters of ITGSend through the Signalling Channel[68]. Consequently, every module is connected through several communication channels to other modules. ITGSend performs the traffic generation and sending processes, while ITGRecv receives it. ITGLog is a storage to collect all log files, while ITGManager is used for the remote control [67].

Finally, the D-ITG decoder (ITGDec) analyses the results collected by the ITGLog module. It calculates the packet loss, throughput, jitter and delay, both

the one-way delay (OWD) and the round-trip time (RTT). Moreover, it can analyse log information in real-time, e.g., if the sender is instructed by a controller entity to adapt the transmission rate based on channel congestion and receiver capacity.

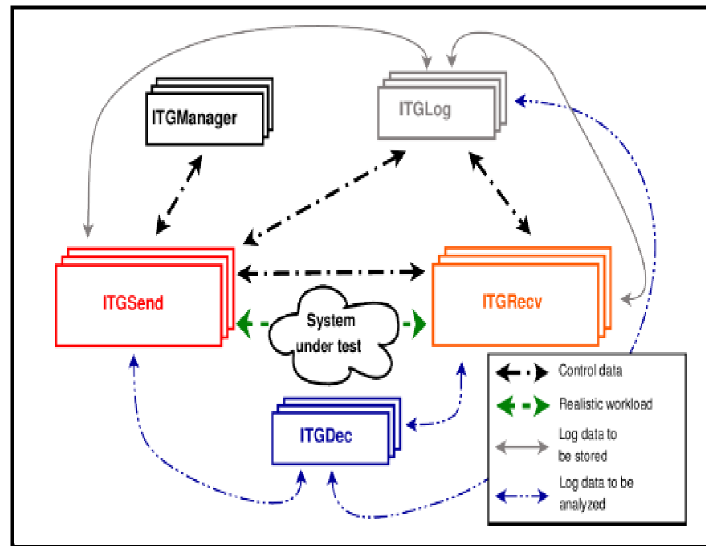


Figure 4.11 The architecture of the D-ITG traffic generation[66]**.**

4.7 Experimental Results and Discussion

First the performance test result is shown in Figure 4.12 and 4.13 presents the round-trip time (RTT) delay occurred in *first* comparison that had been mentioned above in the section 4.6.3. In Figure 4.12 the proposed scheme of multiple-proactive mode approach has the better performance and the delay was almost neglected compare with the multiple-reactive mode approach. Also, Figure 4.12 shows that the proposed scheme of this research reduced the RTT by 13.03% in case of comparing the proposed proactive multiple-controller scheme with the reactive multiple-controller scheme, and in Figure 4.13 the RTT has been reduced by 13.15% in case of comparing the reactive single-controller scheme with the proactive single-controller scheme.

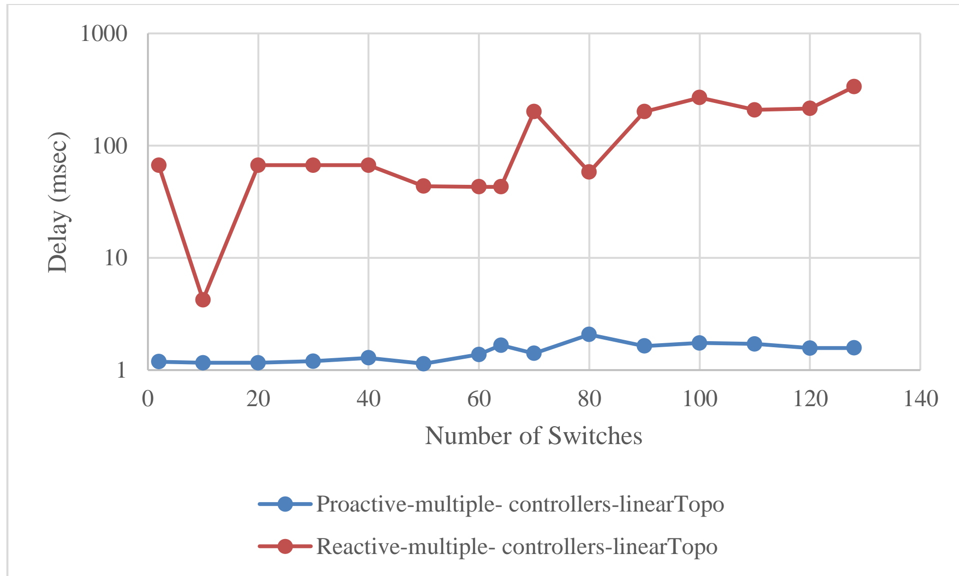


Figure 4.12 Delay for linear topology and different flow mode in case of multiple-controllers scenarios.

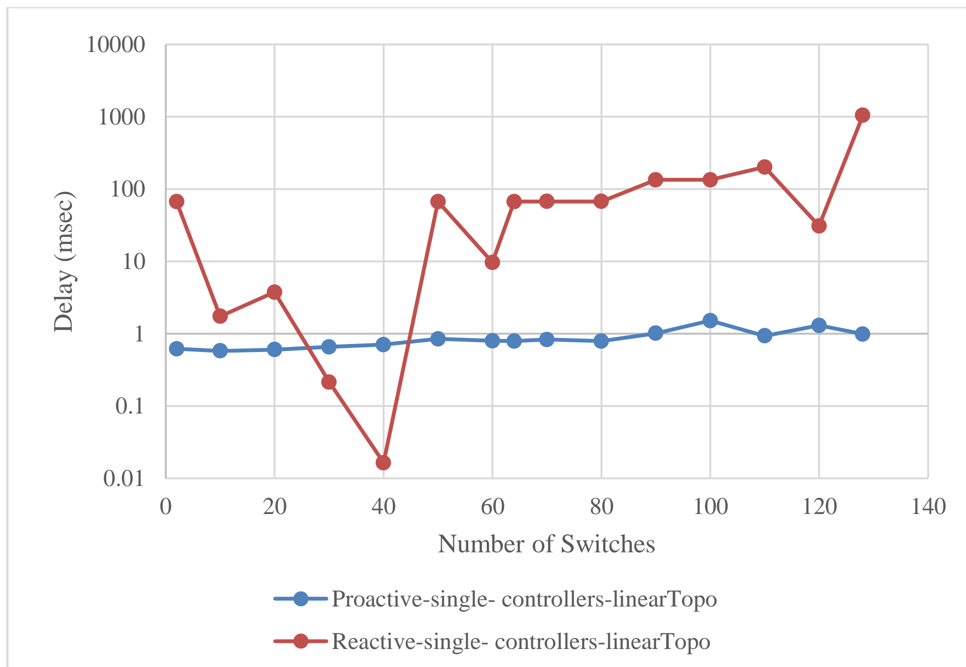


Figure 4.13 Delay for linear topology and different flow mode in case of single-controllers scenarios.

Figures 4.14 and 4.15 presents the packet loss ratio happened also in *first* comparison. It can be seen that in Figures 4.14 and 4.15 the proposed scheme of multiple-proactive mode approach and single proactive controller has no packet

loss at all, while the multiple reactive mode approach has a range of 1-8 % packet loss ratio and the single reactive mode approach has a range of 1-25 % packet loss ratio. These results considerable and, more desirably, therefore the improvement increases when the network gets larger, which indicates an aggregating effect and is meaningful to network reliability and scalability.

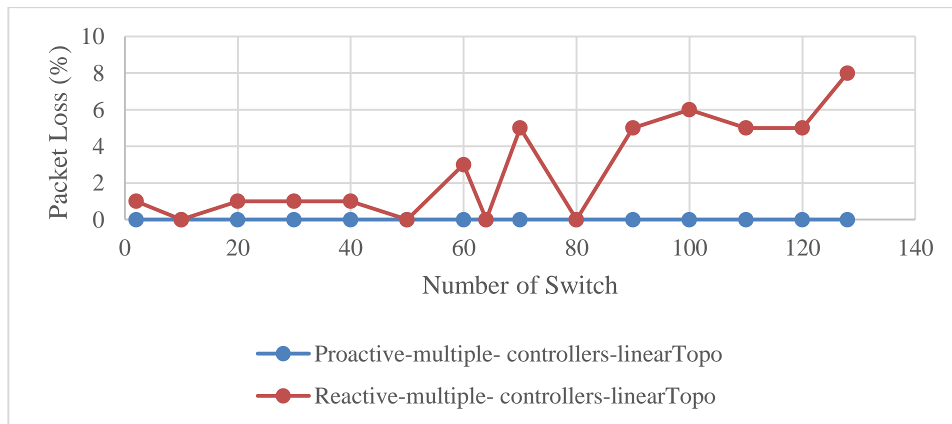


Figure 4.14 Packet Loss Ratio for linear topology and different flow mode in case of multiple-controllers scenarios.

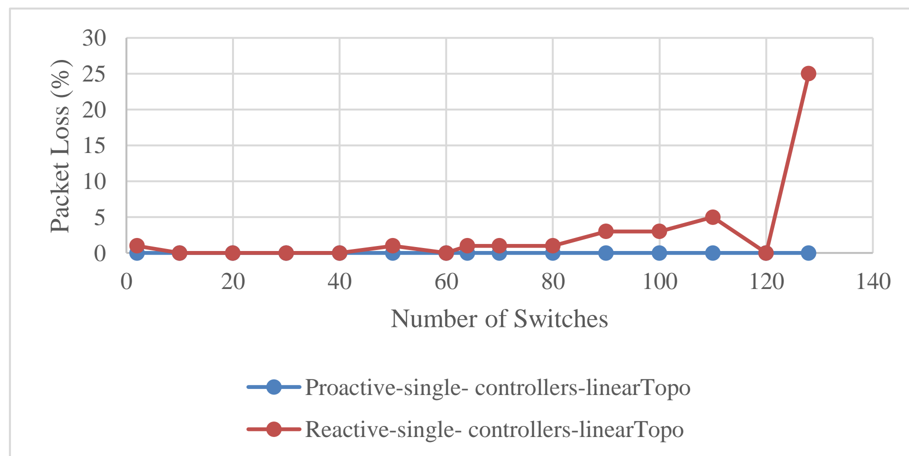


Figure 4.15 Packet Loss Ratio for linear topology and different flow mode in case of single-controller scenarios.

Second, a Proactive-multiple controllers scenarios for three network topologies had been created to test the proposed scheme, so, linear topology, star topology, and tree topology had been used. The performance test result is shown in figure 4.16 shows that the star topology has the lowest delay by 25.16% compared with

the linear topology and by 6.22% compared with the tree topology. In addition, from Figure 4.17 it can be seen that the average throughput was the highest in the case of the tree topology by 18.39% comparing with the linear topology, and by 5.52% comparing with the star topology. This leads us to conclude that the linear topology is the worst case for network topology, and that's why the experiments had been done using the linear topology to see the performance of the proposed network scheme in the worst conditions.

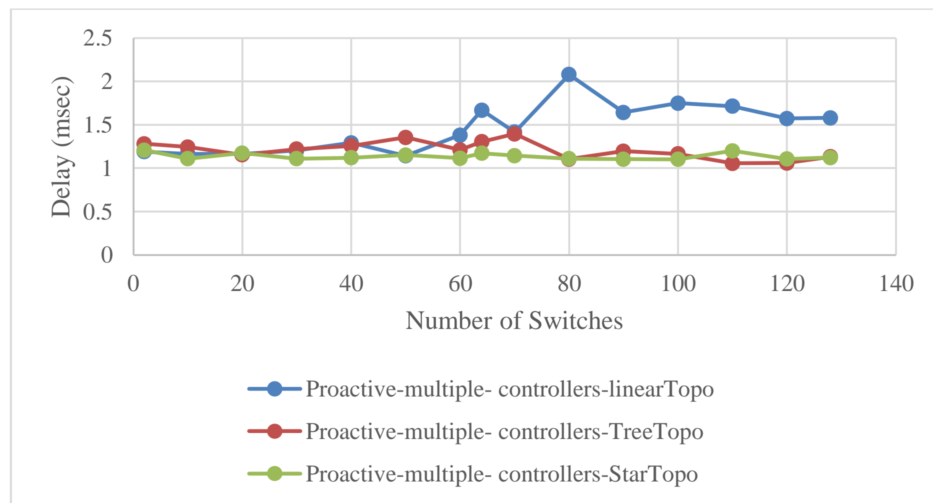


Figure 4.16 Delay for different topology in case of proactive-multiple-controller scenarios.

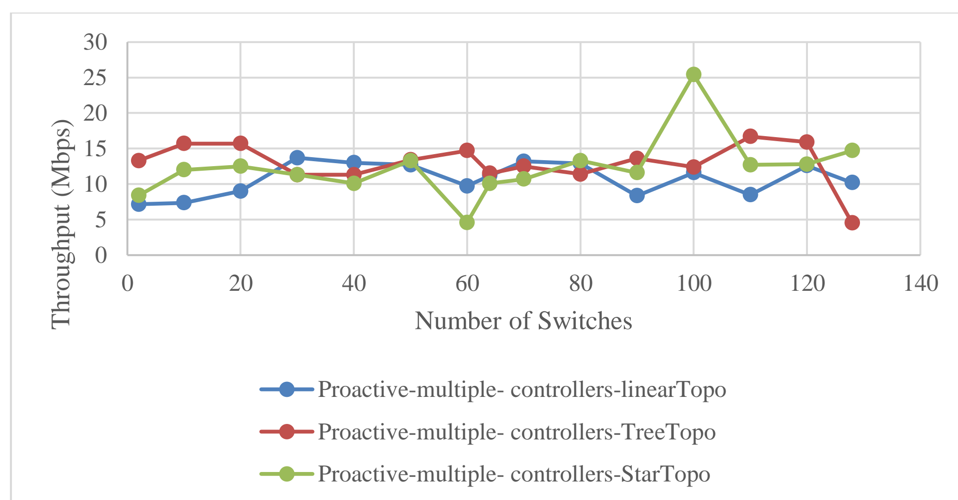


Figure 4.17 Throughput for different topology in case of proactive-multiple-controller scenarios.

Finally a Proactive-Different controllers scenario had been created in which a number of switches had been assigned to just one of each existing controllers, and compare it with the proactive multiple controller network which had been proposed in this research. The performance test result is demonstrated in Figure 4.18 and 4.19 respectively can be seen that the proposed scheme has the highest throughput by 11.89% compared with the proactive different controller scheme, but with more delay by 7.44% and its reasonably for the delay to be less in the case of use different controllers, because each controller has less OpenFlow switches to manage which reduces the control message packets through the network.

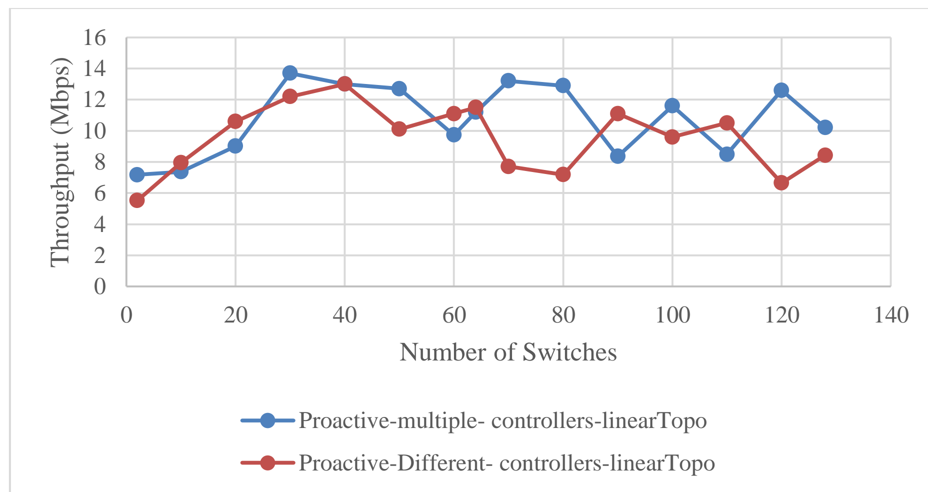


Figure 4.18 Throughput for proactive-multiple/different-controller scenarios.

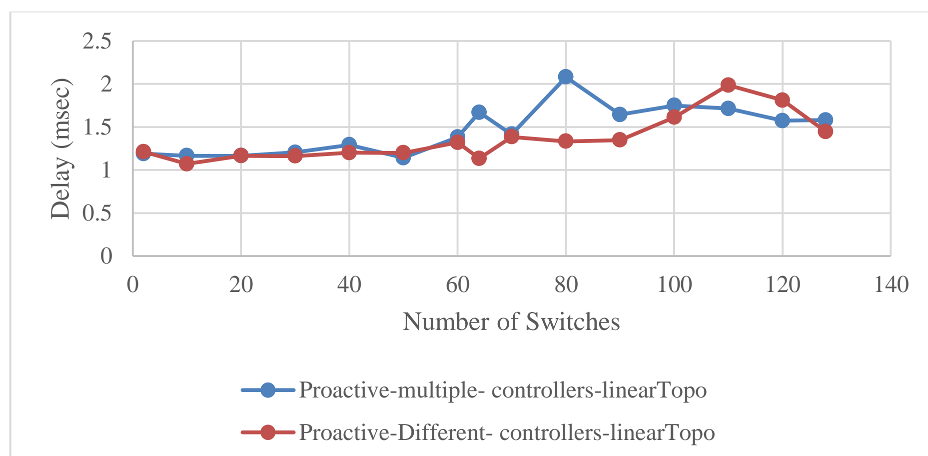


Figure 4.19 Delay for proactive-multiple/different-controller scenarios.

While considering that the OpenFlow architecture has a promising future due to its simplicity and suitability to new technologies, its centralized architecture gives scalability problems. This problem has been studied by researchers from several points of view. This research analyses the performance of different and Multiple Open- Flow controller operating in reactive and proactive approach. All evaluation's results show the increase on controller performance when it used the proactive approach with Multiple-controllers scheme. Our proposal tries to solve this problem by using multiple-controllers with equal role functionality in the proactive flow mode approach which improves the performance of proactive approach but maintaining the facility of reactive approach.

4.8 Summary

In this chapter, various significant studies and experiments had been reviewed which have been conducted to develop and improve SDN reliability and scalability issues. By focusing on the work that had been done to develop and improve the control plane and management plane, which is the most relevant to this research. Thus, in this work the proposed scheme was using multiple controllers which handle multiple network devices, while using OpenFlow controllers in the proactive operations paradigm. Several experiments had been conducted to assess the performance of the multiple-proactive controllers framework proposed in this research. In the experiments, the implementation of the framework was done by using ODL controller. By examining and evaluating different network scenarios, considering the results obtained in this Chapter, the proposed scheme showed promising results and is worth being applied in SDN networks, as it makes the network more reliable and easy to scale without affecting the performance of the network.

Chapter 5

Virtual Tenant Network with Multiple SDN Controller Coordination

This chapter covers installation and configuration of one application available for the OpenDaylight project. Applications use the northbound APIs to interconnect with underlying networking devices. They provide additional services using data collected by a controller. VTN is an application, because its services and features are good examples of NVF. In addition, it can be used with the Mininet network.

5.1 Virtual Tenant Network Overview

Multi-inhabitant networks, are data-centre networks that are separated and consistently partitioned into tinier, segregated networks. Similar to tenants in an apartment compound, multi-tenant networks work all alone network with no ability to see into the other logical networks, yet share the physical networking gear. It's been some time, since the ability to divide networks into logical entities has been available through the use of VLANs. Accordingly, virtualized data-centres and cloud computing concepts have brought multi-tenancy back to the consideration of network administrators. The need to separate and control parts of the network does not convert when moving to the cloud, whether the applications needed by the business processes of the organization or federal regulatory requirements. In order to ensure that local data is secure and may even help develop the IT association into one focused on services, and presents more willingly than simply bits and bytes, because of that, the multi-tenant network approach that public cloud provider's has been utilized this approach. Beside the security aspects of multi-tenancy networks, breaking the data-centre network by

application and service could be the foundation to develop the IT organization from an operations mode into principally an internal vendor, providing applications and services to business units in much the same manner as public cloud service providers. Building a private cloud infrastructure would enable enterprise applications to be delivered to business units as a service, by using service providers, and with clearly defined service-level agreements and bill-back processes. This effort has been supported by multi-tenant networks, which allows specific quality of service and security policies to be set for each "customer" of IT services [69].

5.2 OpenDaylight Virtual Tenant Network

OpenDaylight Virtual Tenant Network (VTN) is an application that offers multi-tenant virtual network on an SDN controller. It provides API for creating a common virtual network regardless of the physical network.

Conventionally, the network is configured as a silo for each department and system, which implies huge investment in the network systems and operating expenses are needed. Therefore, several network appliances must be installed for each tenant, and those boxes cannot be pooled with others. It is a hard work to design, implement and operate the entire complex network. The logical abstraction plane, considered to be the uniqueness feature of VTN. This allows the complete separation of logical plane from physical plane. Users can design and deploy any desired network without knowing the physical network topology or bandwidth restrictions. The network can be defined corresponding to the users, by using VTN, and this with a look and feel of conventional L2/L3 network. Once the network is composed on VTN, it will consequently be mapped into underlying physical network, and afterward configured on the individual switch utilizing SDN control protocol. The definition of logical plane makes it conceivable, not just to hide the complexity of the underlying network, but additionally better to

deal with network resources. It accomplishes reducing reconfiguration time of network services and minimizing network configuration errors[70].

5.2.1 VTN architecture

Figure 5.1 shows the major components of VTN. VTN includes VTN manager and VTN coordinator. In order to manage the information of the virtual network, including network topology and mapping information, VTN manager has been deployed inside SDN controller in the form of plug-in. VTN coordinator manages multiple SDN controller and provides REST APIs for VTN application [71]. VTN applications are network applications used in the virtual network.

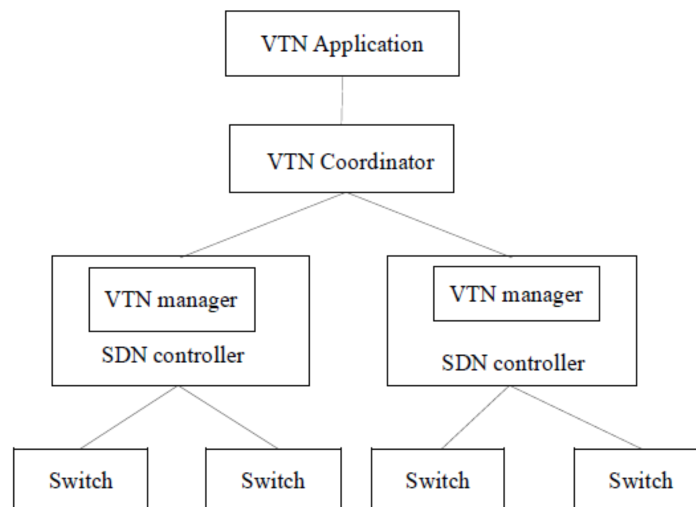


Figure 5.1 VTN Architecture [72].

5.2.1.1 VTN Coordinator

In the OpenDaylight architecture, the VTN Coordinator is part of the network application, orchestration and services layer. Its key function is to generate the virtual network through the OpenDaylight REST APIs. It also provides its own REST APIs to external northbound applications while supporting the use of multiple controllers, and their coordination, at the same time[70].

5.2.1.2 VTN Manager

In the Opendaylight VTN project, the intelligence has been provided by the manager module. It is a plug-in inside the controller which, interacting with other plug-ins to implement the elements of the created VTN model.

5.2.2 Virtual Network Construction

In the VTN, a virtual network is made using virtual nodes (vBridge, vRouter) and virtual interfaces and links. The connection of virtual interfaces which has been made on virtual nodes via virtual links, can probably configure a network which has an L2 and L3 transfer function[70]. Follows brief description for the virtual nodes, which has above-mentioned.

* vBridge: Represents L2 switch functions.
* vRouter: Provides router functions.
* vTep: It is a representation of Tunnel End Point (TEP).
* vTunnel: It is a Tunnel logical representation.
* vBypass: Provides connectivity between controlled networks.
* Virtual interface: Represents virtual node's end points.
* Virtual Link (vLink): Is a connection between virtual interfaces.

5.2.3 vBridge Functions

It's worth to clarify, that according to the destination MAC address, the vBridge provides the bridge function that transfers a packet to the proposed virtual port. When the destination MAC address has been learned, the vBridge looks up the MAC address table and transmits the packet to the corresponding virtual

interface. In the event, when the destination MAC address has not been learned, the process is to transmit the packet to all virtual interfaces other than the receiving port. The learning process of MAC addresses as follows:

* ***MAC address learning***: The vBridge learns the MAC address of the connected host. The source MAC address of each received frame is mapped to the receiving virtual interface, and this MAC address is stored in the MAC address table created on a per-vBridge basis.
* ***MAC address aging***: The MAC address stored in the MAC address table is saved as long as the host gives back the ARP reply. After the host is disconnected, the address is kept until the aging timer times out. MAC addresses can be registered manually, in order to have the vBridge learn MAC addresses constantly.

5.2.4 vRouter Functions

The vRouter transfers IPv4 packets between vBridges. Routing, ARP learning, and ARP aging functions, is supported by the vRouter. The functions are described as follows:

* ***Routing function***: When an IP address is registered with a virtual interface of the vRouter, the default routing information for that interface is registered. It is also possible to statically register routing information for a virtual interface.

* ***ARP learning function***: The vRouter links a destination IP address, MAC address and a virtual interface, based on an ARP request to its host or a reply packet for an ARP request, and maintains this information in an ARP table prepared for each routing domain. The registered ARP entry is saved until the aging timer, has times out. The vRouter transmits an ARP request on an individual aging timer basis and deletes the associated entry from the ARP

table if no reply is returned. For static ARP learning, ARP entry can be registered information manually.

5.2.5 Mapping of Physical Network Resources

Map physical network resources to the constructed virtual network. Mapping detects which virtual network each packet transmitted or received by an OpenFlow switch belongs to, as well as which interface in the OpenFlow switch transmits or receives that packet. There are two mapping methods. Port mapping is first searched for the corresponding mapping definition, when a packet is received from the OpenFlow switch, then VLAN mapping is searched, and the packet is mapped to the appropriate vBridge according to the first matching mapping.

★ ***Port mapping***

Maps physical network resources to an interface of vBridge using Switch ID, Port ID and VLAN ID of the incoming L2 frame. Untagged frame mapping is also supported.

★ ***VLAN mapping***

Maps physical network resources to a vBridge using VLAN ID of the incoming L2 frame. Maps physical resources of a particular switch to a vBridge using switch ID and VLAN ID of the incoming L2 frame.

★ ***MAC mapping***

Maps physical resources to an interface of vBridge using MAC address of the incoming L2 frame (The initial contribution does not include this method).

5.3 Methodology and Network Design

In this experiment four Opendaylight controllers had been used, using VMware Workstation 11.1.0, each controller was installed in Ubuntu server 14.04

with 2GB of RAM 2 cores per processor and size of 32GB of hard disk, a VTN manager was installed also per controller which is the plugin provides REST APIs to internal components. Also, one Fedora20 server with 2GB of RAM, 2 cores per processor and size of 60GB of hard disk had been used, in this server, a VTN coordinator which is an external application that utilizes the interface provided by the manager and provides a user with a VTN Virtualization had been installed.

In order to test and evaluate our network, the ***Mininet*** network emulator had been used to simulate the proposed network.

5.4 Proposed scheme

To describe the design, the system architecture, algorithm and network scenarios are described first:

5.4.1 *Architecture*

By using the network abstractions, VTN enables to configure virtual network across multiple SDN controllers. This provides highly scalable network system. In our network VTN had been created on each SDN controller. Each user can manage those multiple VTNs with one policy, those VTNs can be united to a single VTN. This feature can be deployed to multi data centres environment. Even if those data centres are geographically separated and controlled with different controllers, a single policy virtual network can be realized with VTN.

5.4.2 Operation steps

8. Initiate a cluster of multiple controllers.

9. In the *"karaf"* console of each controller, execute "`feature:install odl-vtn-manager-rest`" to install the VTN manager features.

10. Starting the VTN coordinator by using the following command "`/usr/local/vtn/bin/vtn_start`".

11. After running the Mininet script of the proposed network, the OpenFlow switch transit unicast packet.

12. Through the OpenFlow plugin in the ODL controller a "PACKET_IN" notification sent to the VTN manager to notifies it, with the unicast packet.

13. The VTN manager determines the VBridge to which the packet is mapped.

14. In the VBridge Mac address table the process of searching to find the output destination is occurred.

15. After finding a match for the output destination a packet forwarding is instructed to AD-SAL, if physical network path is present.

16. Flow entry setting are instructed by using the "PACKET_OUT" message.

17. Finally, packet are transmitted and Flow entry is set.

5.4.3 Network Scenarios

A network with multiple controllers managing 15 Openflow switch with tree topology structure had been created as shown in Figure 5.2, the network had been divided into two vlans to test the VTN features in one experiment, and compare it with the same network topology and configuration but by using single controller. Therefore, two experiments had been created to compare among them to examine the effective characteristic between the two approaches, and these scenarios are:

1. Multiple Controllers with VTN feature.
2. Single Controller with VTN feature.

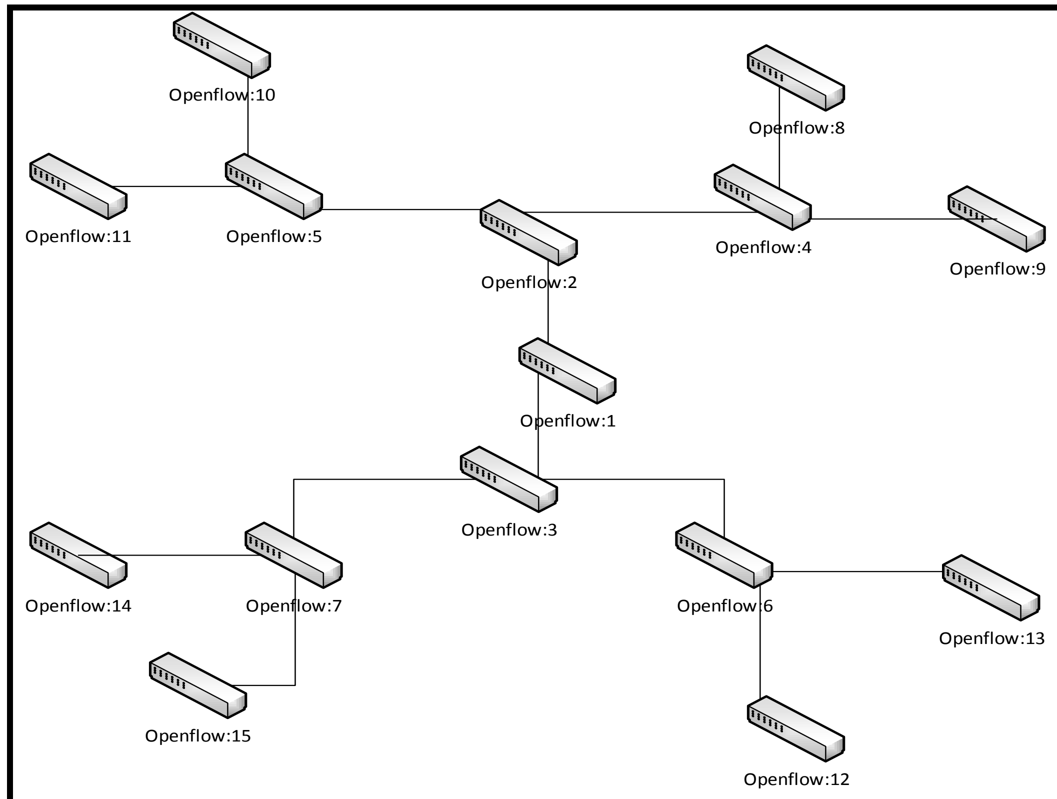


Figure 5.2 the proposed 15 switch in tree topology structure.

5.4.4 Network Configuration

In our python script, the Controllers information had been created as illustrated in Figure 5.3. This will map the controller instance with the VTN application.

```
 GNU nano 2.2.6                File: vlan2_multi4tree16.py

def emptyNet():

   net = Mininet(controller=RemoteController, switch=OVSKernelSwitch)

   c1 = net.addController('c1', controller=RemoteController, ip="192.168.179.129", port=6633)
   c2 = net.addController('c2', controller=RemoteController, ip="192.168.179.131", port=6633)
   c3 = net.addController('c3', controller=RemoteController, ip="192.168.179.132", port=6633)
   c4 = net.addController('c4', controller=RemoteController, ip="192.168.179.136", port=6633)
```

Figure 5.3 Controllers setup.

Figures 5.4 and 5.5 demonstrate our piece of script which used to create a VTN plane, a vBridge1 had been made to be used as a gateway between hosts in VLAN 200, also made a vlan map with vlanid 200 for the vBridge1, Also, a vBridge2 had been made for VLAN 300 and finally a vlan map with vlanid 300 had been made as well for the vBridge2.

```
h1 = net.addHost( 'h1', ip='10.0.0.1', cls=VLANHost, vlan=200 )
h2 = net.addHost( 'h2', ip='10.0.0.2', cls=VLANHost, vlan=200 )
h3 = net.addHost( 'h3', ip='10.0.0.3', cls=VLANHost, vlan=300 )
h4 = net.addHost( 'h4', ip='10.0.0.4', cls=VLANHost, vlan=300 )
h5 = net.addHost( 'h5', ip='10.0.0.5', cls=VLANHost, vlan=200 )
h6 = net.addHost( 'h6', ip='10.0.0.6', cls=VLANHost, vlan=200 )
h7 = net.addHost( 'h7', ip='10.0.0.7', cls=VLANHost, vlan=300 )
h8 = net.addHost( 'h8', ip='10.0.0.8', cls=VLANHost, vlan=300 )
h9 = net.addHost( 'h9', ip='10.0.0.9', cls=VLANHost, vlan=200 )
h10 = net.addHost( 'h10', ip='10.0.0.10', cls=VLANHost, vlan=200 )
h11 = net.addHost( 'h11', ip='10.0.0.11', cls=VLANHost, vlan=300 )
h12 = net.addHost( 'h12', ip='10.0.0.12', cls=VLANHost, vlan=300 )
h13 = net.addHost( 'h13', ip='10.0.0.13', cls=VLANHost, vlan=200 )
h14 = net.addHost( 'h14', ip='10.0.0.14', cls=VLANHost, vlan=200 )
h15 = net.addHost( 'h15', ip='10.0.0.15', cls=VLANHost, vlan=300 )
```

Figure 5.4 VLAN hosts distribution.

```
class VLANHost( Host ):
        def config( self, vlan=100, **params ):
                """Configure VLANHost according to (optional) parameters:
                        vlan: VLAN ID for default interface"""
                r = super( Host, self ).config( **params )
                intf = self.defaultIntf()
# remove IP from default, "physical" interface
                self.cmd( 'ifconfig %s inet 0' % intf )
# create VLAN interface
                self.cmd( 'vconfig add %s %d' % ( intf, vlan ) )
# assign the host's IP to the VLAN interface
                self.cmd( 'ifconfig %s.%d inet %s' % ( intf, vlan, params['ip'] ) )
# update the intf name and host's intf map
                newName = '%s.%d' % ( intf, vlan )
# update the (Mininet) interface to refer to VLAN interface name
                intf.name = newName
# add VLAN interface to host's name to intf map
                self.nameToIntf[ newName ] = intf
                return r
```

Figure 5.5 VLAN host configuration.

5.5 Evaluation results and Discussion

Experiments have been conducted to assess the performance of the multiple-proactive controllers with VTN framework proposed in our research. In the experiments, the traffic flow used in this test is generated by ping and Iperf tests. As per well-known Iperf is a tool to measure the bandwidth and the quality of a network link which can be used for evaluation of parameters such as bandwidth, delay, window size, and packet loss. It is used in evaluation of both TCP and UDP traffic [73]. The bandwidth presented by Iperf is the bandwidth from the client to the server. In the default settings, the Iperf client connects to the Iperf server on the TCP port 5001, and this has been demonstrated as shown in Figure 5.6.

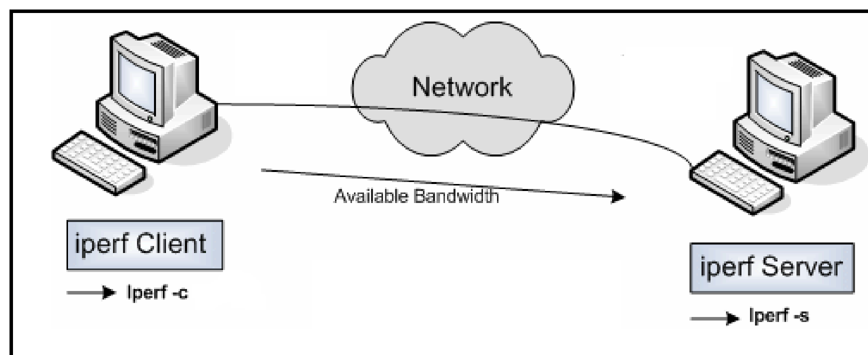


Figure 5.6 Iperf default settings.

The metrics that had been evaluated were (1) the round trip time (RTT) which is the interval between the instance `packet-in` is sent out and the instance `packet-out` or `flow-mod` is received, and this to show the flow setup latency which had been measured on all switches and then take the average, also (2) end-to-end throughput, aggregated over all flows, which were measured using Iperf. Note that RTT is essentially a control-plane metric and throughput a data-plane metric; using these two metrics allows us to see how the two planes quantitatively interact. In all networks, link capacity is 1 GBPS, and Open- Flow channels are provisioned out of band with band- width also 1 GBPs. Flow-table entry timeout

is 60 seconds by default. All the results are averaged over 15 equal-setting runs. The test environment implements and performs the actual protocol stacks that communicate with each other virtually. Mininet environment allows the execution of real protocols in a virtual network.

First, in order to test the IP connectivity between hosts in VLAN 200 as well as between hosts in VLAN 300, while there is no connectivity between the two VLAN groups, by executed the "`pingall`" command, the result should that the connection was successively made as can be seen in Figure 5.7 part of the project. The OpenDaylight controller was tested with the VTN application using NVF possibilities

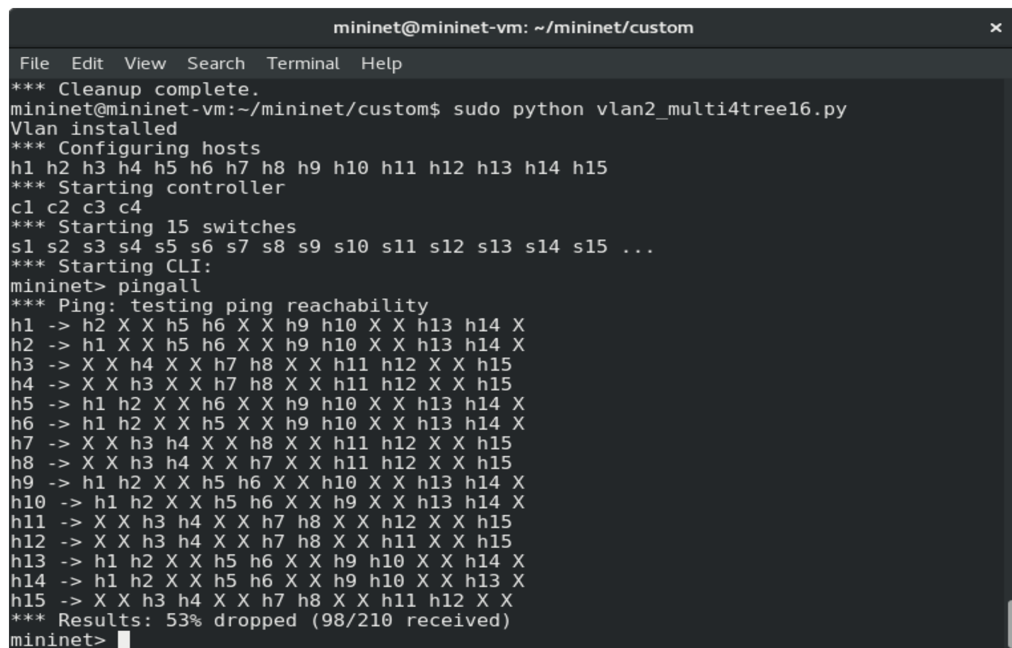


Figure 5.7. Ping command result for the IP connectivity between hosts in the two VLANs.

Figure 5.8 demonstrate the connection of the network after executing the "`net`" command in the CLI of the Mininet.

```
mininet@mininet-vm: ~/mininet/custom
File  Edit  View  Search  Terminal  Help
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15
*** Starting controller
c1 c2 c3 c4
*** Starting 15 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 ...
*** Starting CLI:
mininet> net
h1 h1-eth0.200:s1-eth1
h2 h2-eth0.200:s2-eth1
h3 h3-eth0.300:s3-eth1
h4 h4-eth0.300:s4-eth1
h5 h5-eth0.200:s5-eth1
h6 h6-eth0.200:s6-eth1
h7 h7-eth0.300:s7-eth1
h8 h8-eth0.300:s8-eth1
h9 h9-eth0.200:s9-eth1
h10 h10-eth0.200:s10-eth1
h11 h11-eth0.300:s11-eth1
h12 h12-eth0.300:s12-eth1
h13 h13-eth0.200:s13-eth1
h14 h14-eth0.200:s14-eth1
h15 h15-eth0.300:s15-eth1
s1 lo:  s1-eth1:h1-eth0.200 s1-eth2:s2-eth2 s1-eth3:s3-eth2
s2 lo:  s2-eth1:h2-eth0.200 s2-eth2:s1-eth2 s2-eth3:s4-eth2 s2-eth4:s5-eth2
s3 lo:  s3-eth1:h3-eth0.300 s3-eth2:s1-eth3 s3-eth3:s6-eth2 s3-eth4:s7-eth2
s4 lo:  s4-eth1:h4-eth0.300 s4-eth2:s2-eth3 s4-eth3:s8-eth2 s4-eth4:s9-eth2
s5 lo:  s5-eth1:h5-eth0.200 s5-eth2:s2-eth4 s5-eth3:s10-eth2 s5-eth4:s11-eth2
s6 lo:  s6-eth1:h6-eth0.200 s6-eth2:s3-eth3 s6-eth3:s12-eth2 s6-eth4:s13-eth2
s7 lo:  s7-eth1:h7-eth0.300 s7-eth2:s3-eth4 s7-eth3:s14-eth2 s7-eth4:s15-eth2
s8 lo:  s8-eth1:h8-eth0.300 s8-eth2:s4-eth3
s9 lo:  s9-eth1:h9-eth0.200 s9-eth2:s4-eth4
s10 lo:  s10-eth1:h10-eth0.200 s10-eth2:s5-eth3
s11 lo:  s11-eth1:h11-eth0.300 s11-eth2:s5-eth4
s12 lo:  s12-eth1:h12-eth0.300 s12-eth2:s6-eth3
s13 lo:  s13-eth1:h13-eth0.200 s13-eth2:s6-eth4
s14 lo:  s14-eth1:h14-eth0.200 s14-eth2:s7-eth3
s15 lo:  s15-eth1:h15-eth0.300 s15-eth2:s7-eth4
c1
c2
c3
c4
mininet>
```

Figure 5.8 Network Connection.

Second, the result which are obtained from performing the `Iperf` and the `ping` test were compared to get the throughput and the delay for each of the scenarios that have been accomplished as shown in Figure 5.9 and Figure 5.10 respectively.

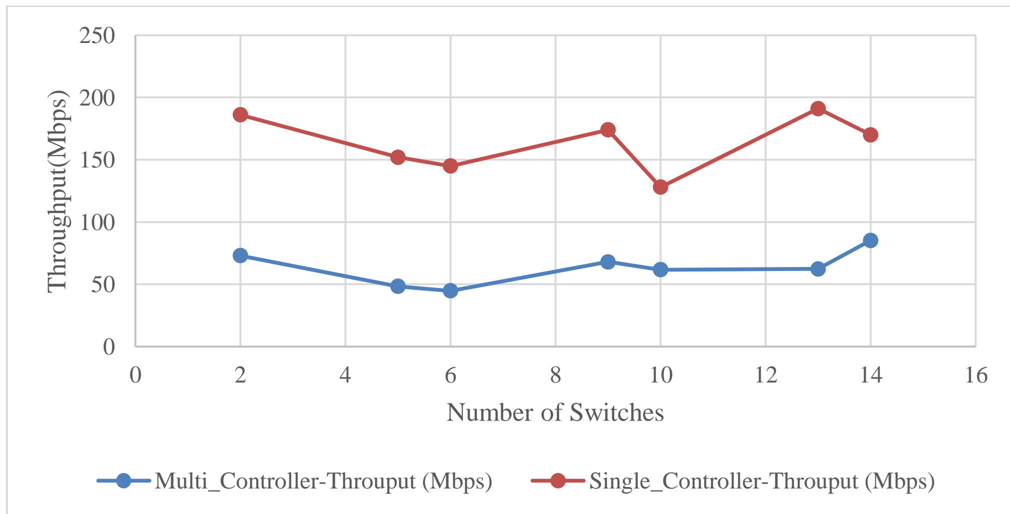


Figure 5.9 The Throughput for Multi and Single Controller with VTN.

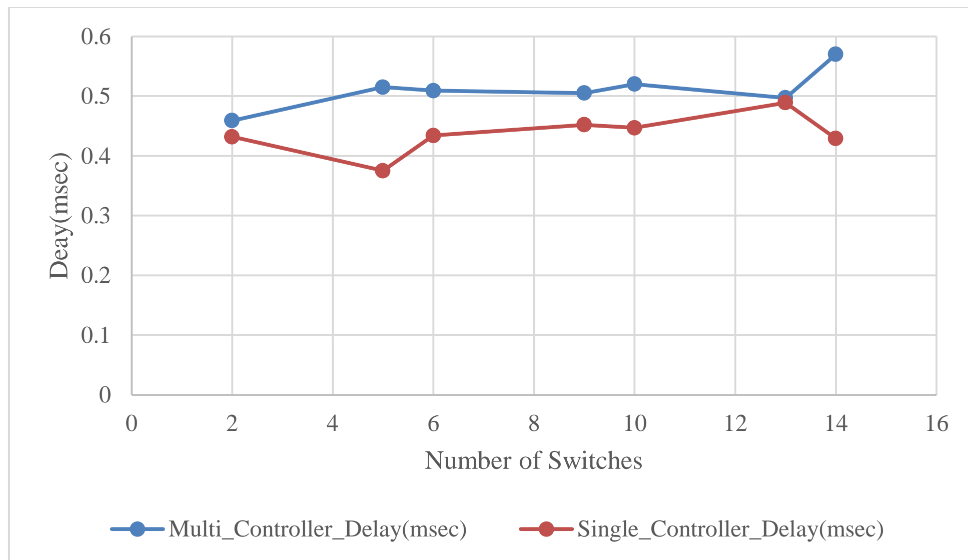


Figure 5.10 The Delay for Multi and Single Controller with VTN.

From both of the results graphs, It can be found that the Single Controller VTN approach has higher throughput and less delay than the Multi Controller VTN approach, and this very reasonable because of the setup time of each controller and the time consumed in the control messages between the controllers and the switches which affect the overall running time but the Multi Controller VTN has no packet loss and improves the reliability, the scalability, and provide efficient end-to-end services comparing with the single point of failure that may happen in case of the Single Controller VTN which risk all the network to fall down.

5.6 Summary

The SDN system was built in a virtualized environment with some advanced features installed. The OpenDaylight controller, which was installed during the project, can be easily used using a virtualized network. Moreover, the VTN Coordinator application, which was installed on a separate VM, was able to bring NFV features into the system. The OpenDaylight controller (Lithium version at the time of writing) was able to test the Multiple Controller approach which was a problem in earlier Opendaylight versions.

Chapter 6

Conclusions and Future work

In this chapter, the stages of the work of this thesis are illustrated in brief, and a number of methods that may improve the system in the future are presented.

6.1 Conclusions

While considering that the OpenFlow architecture has a promising future due to its simplicity and suitability to new technologies, its centralized architecture gives reliability and scalability problems. This problem has been studied by researchers from several points of view.

This research analyses the performance of different and Multiple Open-Flow controller operating in reactive and proactive approach. All evaluation's results show the increase on controller performance when it used the proactive approach with Multiple-controllers scheme. Our proposal tries to solve this problem by using multiple-controllers with equal role functionality in the proactive flow mode approach which improves the performance of proactive approach but maintaining the facility of reactive approach.

Our proposed scheme of multiple-proactive mode approach and single proactive controller has scored almost no packet loss in which implies the strength of reliability our scheme had, while the multiple reactive mode approach has a range of 1-8 % packet loss ratio and the single reactive mode approach has a range of 1-25 % packet loss ratio.

Also in case of delay the improvement which was obtained from our approach scored an average reduction of 13% comparing with other tested schemes.

Consequently, these results indicate an aggregating effect and is meaningful to network reliability and scalability.

In addition, as the VTN project is one of mature projects in OpenDaylight, which provides network virtualization support for OpenFlow. And its most key features are Virtual network, provisioning, flexible traffic control and automatic detouring on link failures.

The ODL VTN module, with or without OpenStack, provides an easy, flexible and efficient method of configuration for multi-tenancy in a virtualized environment. This configuration capability is error tolerant and done over reduced time spans.

The VTN feature had been used accompanied with multiple-proactive controller and was successfully worked in a way to provide NFV in our network. From all our experiment it can be declared that the multiple controller is a great solution for reliability and scalability issue in the SDN-OpenFlow networks.

6.2 Future Work

There are several possible methods and techniques that can be applied in the future to develop and improve that has been done in this thesis. In regards of Network Sharing for different types of networks that are implementing solutions based on SDN and NVF, It can be improved by sharing architecture that is based on VTN provided to VNFs over SDN. By virtualizing the tenant SDN control functions of a VTN, and moving them into the cloud by using one of the cloud computing platform that supports all types of cloud environments, such as OpenStack project.

The control of a VTN is a key requirement related with network virtualization, since it allows the dynamic programming, in which direct control and configuration of the virtual resources allocated to the VTN. For future work an

evaluation of the first SDN/NFV orchestration architecture can be done, in a multi-partner testbed to dynamically deploy independent SDN controller instances for each VTN and to provide the required connectivity within minutes. Moreover, it is suggested to investigate further applications of VTN to extend functionalities offered to network providers.

References

[1] VMware, "Virtualization overview," *White Pap. http//www. vmware. com/pdf/virtualization. pdf*, pp. 1–11, 2006.

[2] D. Kreutz, F. M. V Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, S. Member, and S. Uhlig, "Software-Defined Networking : A Comprehensive Survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[3] J. Metzler, "The 2016 Guide to SDN and," 2016.

[4] S. E. E. Inside, "Spring 2012," *Crosslink®, Aerosp. Corp. Mag. Adv. Aerosp. Technol.*, 2012.

[5] O. Fundation, "Software-defined networking: The new norm for networks," *ONF White Pap.*, pp. 1–12, 2012.

[6] S. Schmid and J. Suomela, "Exploiting locality in distributed SDN control," *Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. Netw. - HotSDN '13*, pp. 1–6, 2013.

[7] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid, "Software transactional networking: Concurrent and consistent policy composition," *Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. Netw.*, pp. 1–6, 2013.

[8] T. D. Nadeau and K. Gray, *SDN Software Defined Networks*, First Edit. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. O'Reilly, 2013.

[9] P. Hui and T. Koponen, *Software Defined Networking*, vol. 2, no. 9. 2012.

[10] N. McKeown, *Software Defined Networking*, vol. 17, no. 2. 2009.

[11] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," *ACM Sigcomm Comput. Commun.*, vol. 44, no. 2, pp. 87–98, 2014.

[12] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending Networking into the Virtualization Layer," *8th ACM Work. Hot Top. inNetworks*, vol. VIII, pp. 1–6, 2009.

[13] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," *Work. Hot Top. Networks, ACM*, pp. 1–6, 2010.

[14] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol. - Conex. '12*, pp. 253–264, 2012.

[15] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. Mckeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," *Network*, p. 15, 2009.

[16] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the Internet," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 59–64, 2003.

[17] Y. Yiakoumis, "Slicing Home Networks," *ACM SIGCOMM Workshop on Home Networking (Homenets), Toronto, Ontario, Canada.*, 2011. [Online]. Available: http://conferences.sigcomm.org/sigcomm/2011/papers/homenets/p1.pdf.

[18] D. Drutskoy, E. Keller, and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," *IEEE Internet Comput.*, pp. 2–9, 2012.

[19] P. H. Isolani, J. A. Wickboldt, and C. B. Both, "Interactive Monitoring , Visualization , and Configuration of OpenFlow-Based SDN," *2015 IFIP/IEEE Int. Symp. Integr. Netw. Manag.*, pp. 207–215, 2015.

[20] G. Computing, "Software Defined Network - Architectures," *Parallel, Distrib. Grid Comput. (PDGC), 2014 Int. Conf.*, pp. 451–456, 2014.

[21] E. Haleplidis, J. H. Salim, and D. Meyer, "Software-Defined Networking (SDN): Layers and Architecture Terminology," *Internet Res. Task Force*, pp. 1–35, 2015.

[22] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in SDN-OpenFlow networks," *Comput. Networks*, vol. 71, pp. 1–30, 2014.

[23] A. A. R. Curtis, J. C. J. J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks," *Proc. {ACM} {SIGCOMM} 2011 Conf. Appl. Technol. Archit. Protoc. Comput. Commun. Toronto, {ON}, Canada, August 15-19, 2011*, pp.

254–265, 2011.

[24] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, “Scalable flow-based networking with DIFANE,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 351–362, 2010.

[25] “BGP Reports.” Available online: http://www.cidr-report.org/as2.0/ 2013.

[26] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, “Leveraging Zipf S Law For Traffic Offloading,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 1, pp. 16–22, 2012.

[27] L. Yang, B. Ng, and W. K. G. Seah, “Heavy hitter detection and identification in software defined networking,” in *2016 25th International Conference on Computer Communications and Networks, ICCCN 2016*, 2016.

[28] N. Sarrar, A. Feldmann, S. Uhlig, R. Sherwood, and X. Huang, “Towards hardware accelerated software routers,” *Proc. ACM Conex. Student Work. - Conex. ’10 Student Work.*, no. 9, pp. 1–2, 2010.

[29] M. Soliman, B. Nandy, I. Lambadaris, and P. Ashwood-Smith, “Source routed forwarding with software defined control, considerations and implications,” *Proc. 2012 ACM Conf. Conex. student Work.*, pp. 43–44, 2012.

[30] “draft-ashwood-sdnrg-state-reduction-00 - SDN State Reduction.” https://tools.ietf.org/html/draft-ashwood-sdnrg-state-reduction-00.

[31] X. Zhang, P. Francis, J. Wang, and K. Yoshida, “Scaling IP routing with the core router-integrated overlay,” *Proc. - Int. Conf. Netw. Protoc. ICNP*, pp. 147–156, 2006.

[32] H. Ballani, P. Francis, and J. Wang, “Viaggre: Making Routers Last Longer,” *Proc. ACM Work. Hot Top. Networks (HotNets), Calgary, AB, Canada*, pp. 109–114, 2008.

[33] “draft-ietf-grow-va-06 - FIB Suppression with Virtual Aggregation.” https://tools.ietf.org/html/draft-ietf-grow-va-06.

[34] C. Zhang, B. Gu, S. Xu, K. Yamoriy, and Y. Tanaka, “Time-dependent pricing for revenue maximization of network service providers considering users preference,” *Netw. Oper. Manag. Symp. (APNOMS), 2013 15th Asia-Pacific*, pp. 1–6, 2013.

[35] G. Retvari, J. Tapolcai, A. Korosi, A. Majdan, and Z. Heszberger, “Compressing IP Forwarding Tables: Towards Entropy Bounds and beyond,”

IEEE/ACM Trans. Netw., vol. 24, no. 1, pp. 149–162, 2016.

[36] N. Gude, J. Pettit, and S. Shenker, "NOX : Towards an Operating System for Networks," *ACM SIGCOMM Comput. Commun. Rev. 38.3*, pp. 105–110, 2008.

[37] A. L. Cox, Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro : A System for Scalable OpenFlow Control Maestro : A System for Scalable OpenFlow Control," *Tech. Rep. TR10-08, Rice Univ.*, 2010.

[38] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, Others, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," *Proc. 9th USENIX Conf. Oper. Syst. Des. Implement.*, pp. 1--6, 2010.

[39] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," *Proc. 2010 internet Netw.*, pp. 1–6, 2010.

[40] S. H. Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," *Proceeding HotSDN '12 Proc. first Work. Hot Top. Softw. Defin. networks*, pp. 19–24, 2012.

[41] A. Shalimov, D. Zimarina, and V. Pashkov, "Advanced Study of SDN / OpenFlow controllers," *Proceeding CEE-SECR '13 Proc. 9th Cent. East. Eur. Softw. Eng. Conf. Russ.*, pp. 1–6, 2013.

[42] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," *Proceeding Hot-ICE'12 Proc. 2nd USENIX Conf. Hot Top. Manag. Internet, Cloud, Enterp. Networks Serv.*, pp. 1–6, 2012.

[43] A. Voellmy and J. Wang, "Scalable software defined network controllers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, p. 289, 2012.

[44] B. Heller, R. Sherwood, and N. Mckeown, "The Controller Placement Problem," *Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. networking.*, pp. 7–12, 2012.

[45] M. Canini, T. U. B. T-labs, D. Levin, and S. Schmid, "Software Transactional Networking : Concurrent and Consistent Policy Composition," *Proceeding HotSDN '13 Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. Netw.*, pp. 1–6, 2013.

[46] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4 :

Experience with a Globally-Deployed Software Defined WAN," *Proc. ACM SIGCOMM 2013 Conf. SIGCOMM*, pp. 3–14, 2013.

[47] A. Vahdat, "Google Cloud Platform Blog Enter the Andromeda zone - Google Cloud Platform's latest networking stack." .

[48] B. Astuto, a Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *Ieee Commun. Surv. Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[49] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, and S. Louis, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[50] V. W. Protocol, "OpenFlow Switch Specification 1.5," *Open Netw. Found. ONF*, vol. 0, pp. 1–205, 2013.

[51] W. Stallings, "Software-Defined Networks and OpenFlow," *Internet Protoc. Journal, Vol. 16, No. 1*, vol. 16, no. February, pp. 1–14, 2013.

[52] CD-adapco, "OpenDayLight User Guide, Lithium," 2015.

[53] M. Report, "SDN Controllers Report 2015 Custom Edition for Cisco Systems," *sdx Cent. Trust. News Resour. Site SDx, SDN, NFV, Cloud Virtualization Infrastruct.*, pp. 1–17, 2015.

[54] L. Paraschis, "Software Innovations and Control Plane Evolution in the new SDN Transport Architectures," *Cisco Connect summit Toronto*, pp. 1–74, 2015.

[55] Z. K. Khattak, M. Awais, and A. Iqbal, "Performance evaluation of OpenDaylight SDN controller," *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, vol. 2015–April, pp. 671–676, 2015.

[56] "OpenDaylight Controller MD-SAL FAQ - OpenDaylight Project," *https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:FAQ*. .

[57] V. W. Protocol, "OpenFlow Switch Specification THIS SPECIFICATION HAS BEEN APPROVED BY THE BOARD OF DIRECTORS OF TO PUBLICATION AND SUCH CHANGES MAY INCLUDE THE ADDITION OR DELE- TION OF NECESSARY CLAIMS OF PATENT AND OTHER INTELLECTUAL PROPERTY," vol. 0, pp. 1–205, 2013.

[58] M. P. Fernandez, "Comparing OpenFlow controller paradigms scalability: Reactive and proactive," *Proc. - Int. Conf. Adv. Inf. Netw. Appl. AINA*, pp. 1009–1016, 2013.

[59] F. Hu, Q. Hao, and K. Bao, "A Survey on Software Defined Networking (SDN) and OpenFlow: From Concept to Implementation," *IEEE Commun. Surv. Tutorials*, vol. 16, no. c, pp. 2181–2206, 2014.

[60] M. K. M, V. Kaarthek, S. Shetty, and R. Kadam, "Software Defined Networking - A Networking Paradigm to meet the emerging trends," vol. 3, no. 2, pp. 5563–5566, 2014.

[61] Y. Li, L. Dong, J. Qu, and H. Zhang, "Multiple Controller Management in Software Defined Networking," *2014 IEEE Symp. Comput. Appl. Commun.*, no. 2013, pp. 70–75, 2014.

[62] W. Braun and M. Menth, "Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices," *Futur. Internet*, vol. 6, no. 2, pp. 302–336, 2014.

[63] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, "Pareto-Optimal Resilient Controller Placement in SDN-based Core Networks," pp. 2–3, 2014.

[64] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized? State distribution trade-offs in software defined networks," *HotSDN*, pp. 1–6, 2012.

[65] "Documentation · mininet_mininet Wiki · GitHub." https://github.com/mininet/mininet/wiki/Documentation.

[66] D. Manual, A. Botta, W. De Donato, A. Dainotti, S. Avallone, and A. Pescap, "D-ITG 2.8.1 Manual," pp. 1–35, 2013.

[67] S. Guadagno, D. Emma, A. Pescap, and N. Federico, "D-ITG Distributed Internet Traffic Generator," *Proc. First Int. Conf. Quant. Eval. Syst.*, no. ii, pp. 1–2, 2004.

[68] M. Großmann and S. J. A. Schuberth, "Auto-Mininet : Assessing the Internet Topology Zoo in a Software-Defined Network Emulator," *MMBnet 2013*, pp. 1–10, 2013.

[69] Y. Y. Shin, S. H. Kang, J. Y. Kwak, B. Y. Lee, and S. Hyang, "The study on configuration of multi-tenant networks in SDN controller," *Int. Conf. Adv.*

Commun. Technol. ICACT, pp. 1223–1226, 2014.

[70] "OpenDaylight User Guide, Lithium Release," 2015.

[71] L. Li and W. Chou, "Design and describe REST API without violating REST: A Petri Net based approach," *Proc. - 2011 IEEE 9th Int. Conf. Web Serv. ICWS 2011*, pp. 508–515, 2011.

[72] O. Access, Y. Mao, Y. Gui, and S. Shen, "Research and Implementation on Virtual Network Mapping Mechanism based on SDN," *Open Cybern. Syst. J.*, no. Vl, pp. 1449–1452, 2015.

[73] S. S. Kolahi, S. Narayan, D. D. T. Nguyen, and Y. Sunarto, "Performance monitoring of various network traffic generators," *Proc. - 2011 UKSim 13th Int. Conf. Model. Simulation, UKSim 2011*, no. March 2011, pp. 501–506, 2011.

Appendix A: The python code for the multiple controllers with linear network configuration

```
#!/usr/bin/python

"""
This example creates a multi-controller network from semi-scratch by
using the net.add*() API and manually starting the switches and controllers.

This is the "mid-level" API, which is an alternative to the "high-level"
Topo() API which supports parametrized topology classes.

Note that one could also create a custom switch class and pass it into
the Mininet() constructor.
"""

from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch, RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.topo import Topo
from mininet.log import lg
from mininet.util import irange, quietRun
from mininet.link import TCLink
from functools import partial

def multiControllerNet():
    "Create a network from semi-scratch with multiple controllers."

    net = Mininet( controller=Controller, switch=OVSSwitch )

    print "*** Creating (reference) controllers"
    c1 = net.addController( 'c1', controller=RemoteController, ip='192.168.179.129',
port=6633)
    c2 = net.addController( 'c2', controller=RemoteController, ip='192.168.179.131',
port=6633)
    c3 = net.addController( 'c3', controller=RemoteController, ip='192.168.179.132',
port=6633)
    c4 = net.addController( 'c3', controller=RemoteController, ip='192.168.179.133',
port=6633)
```

```
print "*** Creating switches"
s1 = net.addSwitch( 's1', protocols='OpenFlow13' )
s2 = net.addSwitch( 's2', protocols='OpenFlow13' )
s3 = net.addSwitch( 's3', protocols='OpenFlow13' )
 .
 .
 .
 s128 = net.addSwitch( 's128', protocols='OpenFlow13' )

print "*** Creating hosts"
h1 = net.addHost( 'h1' )
h2 = net.addHost( 'h2' )
h3 = net.addHost( 'h3' )
 .
 .
 .

h128 = net.addHost( 'h128' )

print "*** Creating links"

net.addLink( s1, h1 )
net.addLink( s2, h2 )
net.addLink( s3, h3 )
 .
 .
 .

net.addLink( s128, h128 )

net.addLink( s1, s2 )
net.addLink( s2, s3 )
net.addLink( s3, s4 )
net.addLink( s4, s5 )
net.addLink( s5, s6 )
net.addLink( s6, s7 )
net.addLink( s7, s8 )
net.addLink( s8, s9 )
net.addLink( s9, s10 )
net.addLink( s10, s11 )
net.addLink( s11, s12 )
net.addLink( s12, s13 )
net.addLink( s13, s14 )
net.addLink( s14, s15 )
net.addLink( s15, s16 )
net.addLink( s16, s17 )
net.addLink( s17, s18 )
net.addLink( s18, s19 )
net.addLink( s19, s20 )
net.addLink( s20, s21 )
```

```
net.addLink( s21, s22 )
net.addLink( s22, s23 )
net.addLink( s23, s24 )
net.addLink( s24, s25 )
net.addLink( s25, s26 )
net.addLink( s26, s27 )
net.addLink( s27, s28 )
net.addLink( s28, s29 )
net.addLink( s29, s30 )
net.addLink( s30, s31 )
net.addLink( s31, s32 )
net.addLink( s32, s33 )
net.addLink( s33, s34 )
net.addLink( s34, s35 )
net.addLink( s35, s36 )
net.addLink( s36, s37 )
net.addLink( s37, s38 )
net.addLink( s38, s39 )
net.addLink( s39, s40 )
net.addLink( s40, s41 )
net.addLink( s41, s42 )
net.addLink( s42, s43 )
net.addLink( s43, s44 )
net.addLink( s44, s45 )
net.addLink( s45, s46 )
net.addLink( s46, s47 )
net.addLink( s47, s48 )
net.addLink( s48, s49 )
net.addLink( s49, s50 )
net.addLink( s50, s51 )
net.addLink( s51, s52 )
net.addLink( s52, s53 )
net.addLink( s53, s54 )
net.addLink( s54, s55 )
net.addLink( s55, s56 )
net.addLink( s56, s57 )
net.addLink( s57, s58 )
net.addLink( s58, s59 )
net.addLink( s59, s60 )
net.addLink( s60, s61 )
net.addLink( s61, s62 )
net.addLink( s62, s63 )
net.addLink( s63, s64 )
net.addLink( s64, s65 )
net.addLink( s65, s66 )
net.addLink( s66, s67 )
net.addLink( s67, s68 )
net.addLink( s68, s69 )
net.addLink( s69, s70 )
net.addLink( s70, s71 )
```

```
net.addLink( s71, s72 )
net.addLink( s72, s73 )
net.addLink( s73, s74 )
net.addLink( s74, s75 )
net.addLink( s75, s76 )
net.addLink( s76, s77 )
net.addLink( s77, s78 )
net.addLink( s78, s79 )
net.addLink( s79, s80 )
net.addLink( s80, s81 )
net.addLink( s81, s82 )
net.addLink( s82, s83 )
net.addLink( s83, s84 )
net.addLink( s84, s85 )
net.addLink( s85, s86 )
net.addLink( s86, s87 )
net.addLink( s87, s88 )
net.addLink( s88, s89 )
net.addLink( s89, s90 )
net.addLink( s90, s91 )
net.addLink( s91, s92 )
net.addLink( s92, s93 )
net.addLink( s93, s94 )
net.addLink( s94, s95 )
net.addLink( s95, s96 )
net.addLink( s96, s97 )
net.addLink( s97, s98 )
net.addLink( s98, s99 )
net.addLink( s99, s100 )
net.addLink( s100, s101 )
net.addLink( s101, s102 )
net.addLink( s102, s103 )
net.addLink( s103, s104 )
net.addLink( s104, s105 )
net.addLink( s105, s106 )
net.addLink( s106, s107 )
net.addLink( s107, s108 )
net.addLink( s108, s109 )
net.addLink( s109, s110 )
net.addLink( s110, s111 )
net.addLink( s111, s112 )
net.addLink( s112, s113 )
net.addLink( s113, s114 )
net.addLink( s114, s115 )
net.addLink( s115, s116 )
net.addLink( s116, s117 )
net.addLink( s117, s118 )
net.addLink( s118, s119 )
net.addLink( s119, s120 )
net.addLink( s120, s121 )
```

```
net.addLink( s121, s122 )
net.addLink( s122, s123 )
net.addLink( s123, s124 )
net.addLink( s124, s125 )
net.addLink( s125, s126 )
net.addLink( s126, s127 )
net.addLink( s127, s128 )

print "*** Starting network"
net.build()
c1.start()
c2.start()
c3.start()
c4.start()

s1.start( [ c1, c2, c3, c4 ] )
s2.start( [ c1, c2, c3, c4 ] )
s3.start( [ c1, c2, c3, c4 ] )
.
.
.
s128.start( [ c1, c2, c3, c4 ] )

print "*** Testing network"
#net.pingAll()
h1, h2  = net.hosts[0], net.hosts[1]
print h1.cmd('ping -c15 %s' % h2.IP())
h1, h10  = net.hosts[0], net.hosts[9]
print h1.cmd('ping -c15 %s' % h10.IP())
h1, h20  = net.hosts[0], net.hosts[19]
print h1.cmd('ping -c15 %s' % h20.IP())
h1, h40  = net.hosts[0], net.hosts[39]
print h1.cmd('ping -c15 %s' % h40.IP())
h1, h60  = net.hosts[0], net.hosts[59]
print h1.cmd('ping -c15 %s' % h60.IP())
h1, h80  = net.hosts[0], net.hosts[79]
print h1.cmd('ping -c15 %s' % h80.IP())
h1, h100  = net.hosts[0], net.hosts[99]
print h1.cmd('ping -c15 %s' % h100.IP())
h1, h120  = net.hosts[0], net.hosts[119]
print h1.cmd('ping -c15 %s' % h120.IP())

print "testing bandwidth between h1 and hn"
h1, h2 = net.get( 'h1', 'h2' )
net.iperf((h1, h2))
h1, h10 = net.get( 'h1', 'h10' )
net.iperf((h1, h10))
h1, h20 = net.get( 'h1', 'h20' )
net.iperf((h1, h20))
h1, h40 = net.get( 'h1', 'h40' )
```

```
    net.iperf((h1, h40))
    h1, h60 = net.get( 'h1', 'h60' )
    net.iperf((h1, h60))
    h1, h80 = net.get( 'h1', 'h80' )
    net.iperf((h1, h80))
    h1, h100 = net.get( 'h1', 'h100' )
    net.iperf((h1, h100))
    h1, h120 = net.get( 'h1', 'h120' )
    net.iperf((h1, h120))


    print "*** Running CLI"
    CLI( net )

    print "*** Stopping network"
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )  # for CLI output
    multiControllerNet()
```