

**Sudan University of Science and Technology**  
**Collage of Engineering**  
**Electronics Engineering**



## **A Comparison between Three Legged and Three Part Topologies Using Software Defined Network**

A Research Submitted in Partial Fulfillment for the Requirements of the Degree of B.Sc. (Honors) in Electronics Engineering

### **Prepared By:**

- Ethar Tag-Aldeen Babiker Ahmed.
- Fatma IbnOmer Yousif Salih.
- Salih Kamal-Aldeen Salih Abdaullah.

### **Supervised by:**

Dr. Yassir Obeid Mohammed

**October 2017**

قال تعالى:

(أَفَلَمْ يَسِيرُوا فِي الْأَرْضِ فَتَكُونَ لَهُمْ قُلُوبٌ يَعْقِلُونَ بِهَا أَوْ آذَانٌ يَسْمَعُونَ بِهَا ۗ

فَإِنَّهَا لَا تَعْمَى الْأَبْصَارُ وَلَكِن تَعْمَى الْقُلُوبُ الَّتِي فِي الصُّدُورِ) الحج الآية

(46)

## **Dedication**

To all those who have been supporting us since day one,,

Our beloved parents, our gracious families, our great friends and our kind colleagues.

Without all those people we could never complete this work, much appreciation.

## **Acknowledgement**

First of all we thank Allah for enlighten us through the this journey and to embrace us with his mercy.

Second of all we would love to thank our dear supervisor Dr. Yassir Obeid for supporting us and guiding us all the way and providing us with all the advices we needed to complete this thesis.

And much gratitude for him and his very helpful spirit and elegance attitude, we truly appreciate it.

We also would like to thank Mohmmmed Widaa and Hussam Saif for helping us and giving us their time and support.

## **Abstract**

Unlike traditional networking methods, software-defined networking technology provides a way for many networks to become more agile and adaptive as its design helps in addressing the dynamic needs for modern networks. This project presents the implementation of two different topologies (three-legged and three-part) using SDN, It also indicates the key different between traditional and SDN networks and demonstrates the benefits of SDN technology considering the presented topologies in terms of factors such as: management, security, and performance.

Mininet emulation environment and its components are used. POX SDN controllers is used to provide fair comparison and illustrates the key differences according to various scenarios.

## المستخلص

خلافاً لأساليب الشبكات التقليدية، توفر تقنية الشبكات المعرفة برمجياً وسيلة للعديد من الشبكات لتصبح أكثر مرونة وقابلية للتكيف حيث يساعد تصميمها في معالجة الاحتياجات الديناميكية للشبكات الحديثة. يعرض هذا المشروع تنفيذ بنيتين مختلفتين (ثلاثة أرجل وثلاثة أجزاء) باستخدام الشبكات المعرفة برمجياً، ويشير أيضاً إلى الاختلاف الرئيسي بين الشبكات التقليدية و الشبكات المعرفة برمجياً ويوضح فوائد تكنولوجيا الشبكات المعرفة برمجياً إذا اخذنا بعين الاعتبار الطبقات المقدمة من حيث عدة عوامل مثل: الإدارة، الأمن، والأداء. يتم استخدام بيئة محاكاة Mininet ومكوناتها. و تستخدم وحدات تحكم ال POX SDN لتوفير مقارنة عادلة وتوضح الاختلافات الرئيسية وفقاً لسيناريوهات عديدة.

## Table of Contents

Dedication .....	II
Acknowledgement .....	III
Abstract .....	IV
Abstract in Arabic .....	V
List of Figures .....	VIII
List of Tables .....	IX
List of Abbreviations .....	X
1. Introduction .....	2
1.1 Preview .....	3
1.2 Problem statement .....	5
1.3 Proposed Solution .....	5
1.4 Approach .....	5
1.5 Thesis Outlines .....	5
2.1 Background .....	9
2.2.1 History .....	9
2.1.2 Definition of SDN .....	11
2.1.3 Software-defined networking vs. traditional networking .....	12
2.1.4 SDN basic concepts and principles .....	17
2.1.4.1 Principle .....	17
2.1.4.2 Components .....	19
2.1.5 Network Operating System .....	24
2.1.6 SDN Switches .....	26
2.1.7 SDN Controllers .....	26
2.1.8 OpenFlow API .....	29
2.2 Related Work1 .....	29
2.2.1 Topologies and firewall .....	29

2.2.2 SDN implementation .....	36
3. Approach .....	42
3.1 Introduction .....	43
3.2 Suggested topologies .....	43
3.3 Rules .....	52
4. Results .....	55
4.1 Results .....	56
4.2 Comparison .....	67
5. Conclusion .....	70
5.1 Conclusion .....	71
5.2 Recommendations .....	71
References .....	72
Appendix A .....	1
Appendix B .....	4
Appendix C .....	8



## List of Figures

Figure 2-1: Selected developments in programmable networking over the past 20 years, and their chronological relationship to advances in network virtualization (one of the first successful SDN use cases).....	11
Figure 2-2: SDN vs. traditional networking.....	16
Figure 2-3: SDN vs. traditional networking .....	19
Figure 2-4: SDN Functional Architecture illustrating the data, control and application layers and interfaces.....	24
Figure 2-5: three-part and three-legged network topologies in traditional network.....	31
Figure 2-6: packets are matched against multiple tables.....	39
Figure 2-7: pre-table packet processing .....	39
Figure 2-8: flow table fields .....	40
Figure 2-9: Flowchart detailing packet flow through an OpenFlow switch .....	40
Figure 3-1: Xming .....	46
Figure 3-2: PuTTY page .....	46
Figure 3-3: VirtualBox .....	47
Figure 3-4: mininet command line terminal and Putty terminal...	48
Figure 3-5: MiniEdit opening command .....	49
Figure 3-6: opening MiniEdit .....	49
Figure 3-7: three-legged in MiniEdit.....	50
Figure 3-8: three-part in MiniEdit .....	51
Figure 3-9: Command to activate POX .....	52
Figure 4-1: activation of topology and controller .....	56
Figure 4-2: three-legged rules tests .....	61
Figure 4-3: three-part rules tests.....	66

## List of Tables:

Table 2-1: Most popular open source SDN controllers .....	28
Table 2-2: A comparison of proprietary and non-proprietary SDN strategies .....	38
Table 3-1: Controller rules .....	53

### **List of abbreviations:**

ACL	Access Control Lists.
ACL	Anterior Cruciate Ligament.
ANP	Application Network Profiles.
API	Applection Programming Interface.
ASIC	Application Specific Integrated Circuit.
ASIC	Applection Specific Integrated Circuit.
BGP	Border Gateway Protocol.
BGP	Border Gateway Protocol
CPU	Central Processing Unit.
D-CPI	Data-Controller Plane Interface.
DMZ	DeMilitarized Zone.
DPI	Deep Packet Inspection.
DPID	Data Path Identifier.
ECMP	Equal Cost Multi Path Routing.
FPGA	Field Programmable Gate Array.
FSFW	FlowSpace Firewall.
GPL	General Public License.
GUI	Graphical User Interface.
HA	High-Availability.
HTTP	Hyper Text Transfer Protocol.

ICMP	Internet Control Message Protocol.
IETF	Internet Engineering Task Force.
IO	Input Output.
IT	Information Technology.
JVM	Java Virtual Machine.
MAC	Media Control Access.
NAT	Network Address Translate
NE	Network Element.
NETCONF	Network Configuration.
NFV	Network Functions Virtualization.
NIC	Network Interface Card.
NOS	Network Operating System
NVF	Network Vitalization Function.
OF-Config	OpenFlow Configuration.
ONF	Open Networking Foundation.
ONF	Open Networking Foundation.
OS	Operation System.
OSPF	Open Shortest Path First.
OVS	Open vSwitch.
OVSDB	Open Virtual Switch Data Base.

QoS	Quality of Service.
RIP	Routing Information Protocol.
RPC	Remote Procedure Call.
SDN	Software Defined Networking.
SSH	Secure Shell.
SLA	Service Level Agreement.
TAP	Text Access Point.
TCP	Transmission Control Protocol.
TE	Traffic Engineering.
VLAN	Virtual LAN.
VRF	Virtual Routing and Forwarding.
XNC	Extensible Network Controller.

# **Chapter One**

## **Introduction**

# **Chapter One**

## **Introduction**

- 1.1 Preview
- 1.2 Problem statement
- 1.3 Proposed Solution
- 1.4 Approach
- 1.5 Thesis Outlines

## 1.1 Preview

Networking has experienced limited innovation over the past 20 years. This stagnation has led to overly complex and inflexible networks that no longer meet business requirements. Software Defined Networking (SDN) technology allows engineers to respond quickly to changes in business environments. SDN separates the physical infrastructure from network control and forwarding functionality, so that the network itself becomes more directly programmable.

A few key points of SDN integration within existing networks are:

- Network controls are directly programmable so that one can completely manage all forwarding functionality, plus other network activities.
- Network intelligence is centralized within SDN controllers, giving a single platform to maintain global look at the network.
- Policy enforcement aims to work on local and broad levels.
- There is a reliance on open standards, and SDN technologies also perform best when developed to operate in vendor-neutral settings so it can operate and adjust to specific and existing hardware.
- SDN simplifies the network design by providing instructions from controllers instead of individual devices and protocols, which some vendors can set a limit on older networks.
- Its heart is agile. The systems are designed to grow, move controls and adjust when network demands change. This can be due to growth or increased need, both at local level or network-wide levels.
- SDN technology solutions are designed to integrate with a variety of different software and hardware options. This is a core tenet of SDN technology.



This design focus helps software defined networking deployment address the dynamic needs of modern networks. Those networks still considered “conventional” are falling down in environments like data centers and large enterprise campuses because they struggle to adjust with processing, storage and other networked element needs.

SDN and NVF (network vitalization function) are probably the hottest topics in the field right now,SDN discusses the centralization of the controller, while NVF discusses the centralization of Services.Both work in parallel to make an open source environment that support innovation.

SDN is a technology that enables innovation in how we design and manage networks. The effort is to make computer networks more programmable and manageable as it imposes a centralized architecture using a single controller that makes all the decisions.

As in traditional networks, topology of the network plays a critical role in the implementation of the SDN so it should also follow certain restraints to avoid attacks and malicious actions.SDN uses topology discovery methods which underpins higher level applications and services such as routing and forwarding.

An essential component of each and every network that helps in keeping the network secure is the firewalls. There are several ways a firewall can be established, each depends on the user needstarting from a very simple topology to a highly complicated one each providing a certain level of protection and security. Keeping in mind we are not talking about a firewall which is only a piece of software which runs on the same computer you use to connect to the internet and do your work, but we are talking about a physical computer which is a dedicated firewall.

Here in this thesis we discuss the SDN implementation of two network topologies, one is three-legged network topology, and the other is three-part network topology. In both topologies the firewall plays a critical role. Detailed comparison between them will be established showing the importance and benefits of SDN.

## **1.2 Problem Statement**

In conventional networks once the forwarding policy has been defined, the only way to make an adjustment to the policy is via changes to the configuration of the devices. This has proven restrictive for network operators who are keen to scale their networks in response to rapid changing network demands.

## **1.3 Proposed Solution**

The Use of SDN lead to overcome the introduced problem, as SDN has a centralized control of the network using a device called the controller that has a global view of the network, which makes it easier to adapt to changes in the network.

## **1.4 Approach**

After formulating the problem and proposing a solution for it, the tools to be used in the process has been specified and then we have to learn how it can be used to achieve the goal.

Finally the intended network topologies is implemented using the tools and the comparison is carried out.

## **1.5 Thesis Outlines**

Chapter one: overviews the research and states the problem and proposed solution.

Chapter two: covers a theoretical background of the proposed work and gives an overview of the related work. It also presents the basic concept of the sdn.

Chapter three:the process of implementing and comparing three-legged and three-part topology is carried out in details. Also a sufficient description of tools and technologies used is presented.

Chapter four: presents result and analyzes them to show what the final output of this thesis is.

Chapter five: provides research conclusion and recommendation.

**Chapter Two**  
**Literature Review**

# **Chapter Two**

## **Literature Review**

- 2.1 Background
- 2.2 Related Work

## **2.1 Background**

Designing and managing networks has become more innovative over the past few years with the aid of SDN. This technology seems to have appeared suddenly but it is actually part of long history of trying to make computer networks more programmable.

### **2.1.1 History**

The term software-defined networking (SDN) has been coined in recent years. However, the concept behind SDN has been evolving since 1996, driven by the desire to provide user-controlled management of forwarding in network nodes. Implementations by research and industry groups include Ipsilon (proposed General Switch Management protocol, 1996), The Tempest (a framework for safe, resource-assured, programmable networks, 1998) and Internet Engineering Task Force (IETF) Forwarding and Control Element Separation, 2000, and Path Computation Element, 2004. Most recently, Ethane (2007) and OpenFlow (2008) have brought the implementation of SDN closer to reality. Ethane is a security management architecture combining simple flow-based switches with a central controller managing admittance and routing of flows. OpenFlow enables entries in the Flow Table to be defined by a server external to the switch. SDN is not, however, limited to any one of these implementations, but is a general term for the platform. [2]

SDN was originally created in response to demand from large data centers which have found problems coping with totally unpredictable traffic patterns. Those traffic patterns would cause very high demand on particular resources that couldn't meet with the existing infrastructure. So they had two choices, the first is to scale the network infrastructure to

meet the peaks; this solution is not only very expensive but also the majority of the network will be underutilized most of the time.

The second choice is to build the network in such a way it can reconfigure itself automatically to cope with these peaks and channel the resources to meet with the appropriate demand, and this is what SDN does.

SDN uses a programmatic method where the customer can alter the network according to their own business rules to meet those peak and demand at a very short notice.

SDN is ideally suited for:

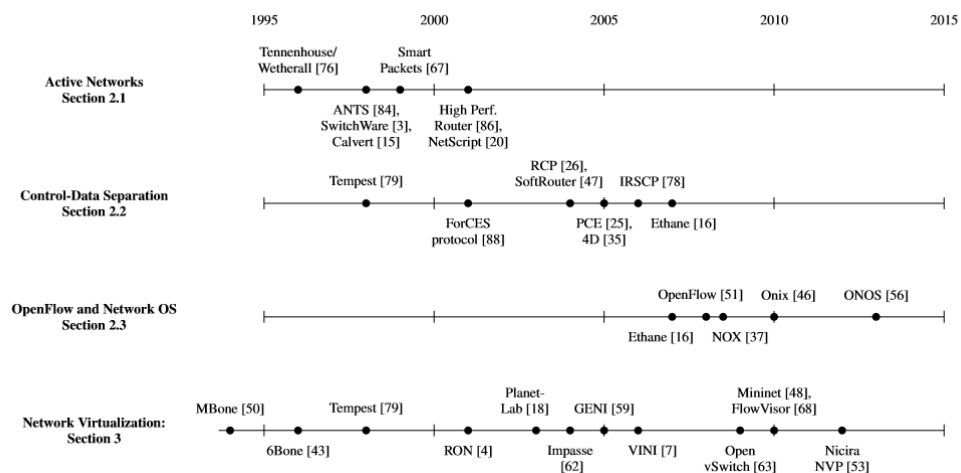
- Rapidly changing traffic patterns, in which customers have rapid changes in their day-to-day networkload, Such as social networkingsites.
- Large data centers that may have geographically disbursedresources and they have to reconfigure their network to meet the demand in the specific resource.

Designing and managing networks has become more innovative over the past few years with the aid of SDN. This technology seems to have appeared suddenly but it is actually part of long history of trying to make computer networks more programmable.

The history of SDN can be divided into three stages, as shown in Figure 2-1. Each stage has its own contributions to the history: (1) active networks (from the mid-1990s to the early 2000s), which introduced programmable functions in the network to enable greater to innovation; (2) control and data plane separation (from around 2001 to 2007), which developed open interfaces between the control and data planes; and (3) the OpenFlow API and network operating systems (from 2007 to around

2010), which represented the first instance of widespread adoption of an open interface and developed ways to make control-data plane separation scalable and practical.[1]

Figure 2-1: Selected developments in programmable networking over the past 20 years, and their chronological relationship to advances in network virtualization (one of the first successful SDN use cases).



**Figure 2-1** Selected developments in programmable networking.

### 2.1.2 Definition of SDN

Open Networking Foundation (ONF) definition of SDN: “In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications.” [2]

Also Nick Mckeown - professor in electrical engineering and computer science department in Stanford university defines Software Defined Networking as: [3] a network in which the control plane is physically separate from the forwarding plane. And a single control plane controls several forwarding devices.



### **2.1.3 Software-defined networking vs. traditional networking**

Today, business and technical network requirements include enhancing performance and realizing broader connectivity. Companies have to meet more and more industry-specific security regulations and there is a growing demand for mobility. In order to comply with all of these criteria, networking protocols have evolved significantly over the last few decades. However, the way traditional networks are set up, deploying one protocol to realize these needs organization-wide is quite the challenge.

The traditional approach to networking is characterized by two main factors:

- i. Network functionality is mainly implemented in a dedicated appliance. In this case, ‘dedicated appliance’ refers to one or multiple switches, routers and/or application delivery controllers.
- ii. Most functionality within this appliance is implemented in dedicated hardware. An Application Specific Integrated Circuit (ASIC) is often used for this purpose.

Organizations are increasingly confronted with the limitations that accompany this hardware-centric approach, such as:

- Traditional configuration is time-consuming and error-prone:

Many steps are needed when a network administrator needs to add or remove a single device in a traditional network. First, he will have to manually configure multiple devices (switches, routers, firewalls) on a device-by-device basis. The next step is using device-level management tools to update numerous configuration settings, such as ACLs, VLANs and QoS. This configuration approach makes it that much more complex for an administrator to deploy a consistent set of policies. As a result,

organizations are more likely to encounter security breaches, non-compliance with implications. So that the highly administrative hassle that is traditional configuration interferes with meeting business networking standards.

- Multi-vendor environments require a high level of expertise:

The average organization owns a variety of equipment of different vendors. To successfully complete a configuration, an administrator will therefore need extensive knowledge of all present device types.

- Traditional architectures complicate network segmentation:

A development further complicating networking matters, is the connectivity evolution that is currently taking place. In addition to tablets, PCs and smartphones, other devices such as alarm systems and security cameras will soon be linked to the internet. The predicted explosion of smart devices is accompanied by a new challenge for organizations: how to incorporate all these devices of different vendors within their network in a safe and structured manner. Many traditional networks place all types of devices in the same zone. In case of a compromised device, this design risks giving external parties access to the entire network. This can be hackers exploiting the internet connection of smart devices or vendors who can remotely log onto their devices. In both cases, there is no apparent reason for giving them access to all network components. However, the administrative hassle described earlier makes network segmentation a complex process and quickly leads to network clutter.

In conclusion, to overcome these and other traditional networking limitations, the time has come to introduce a new perspective on network management.

- TRADITIONAL NETWORKING-basics in light of SDN

The control plane is responsible for configuration of the node and programming the paths to be used for data flows. Once these paths have been determined, they are pushed down to the data plane. Data forwarding at the hardware level is based on this control information.

- i. This network configuration we consider traditional is still used by many networks today. There are two main factors that determine the traditional layout:
- ii. Functions in the network are typically implemented in a dedicated device for each purpose. These devices include switches, routers, and application delivery controllers.
- iii. Functionality within each device is executed in a specific piece of dedicated hardware. This will often make the use of application specific integrated circuits.

Traditional network devices have a control plane that gives them desired information which they use to create a forwarding table, and the data plane references this forwarding tables to make decisions. Packets delivered to the device are then sent based on the control's plane rules, and both of these planes exist within the network device itself.

Development of traditional network requires specific, planned steps that tend to require an IT administrator to be present even during maintenance. Traditional networking often require manual configuration of the devices on a device-by-device basis before network management tools can be applied for device-level management.

If a traditional network uses hardware from multiple vendors, each individual element will need to be configured to work with other devices and deployments, but much of this work will have to occur on a case-by-case basis with direct configuration.

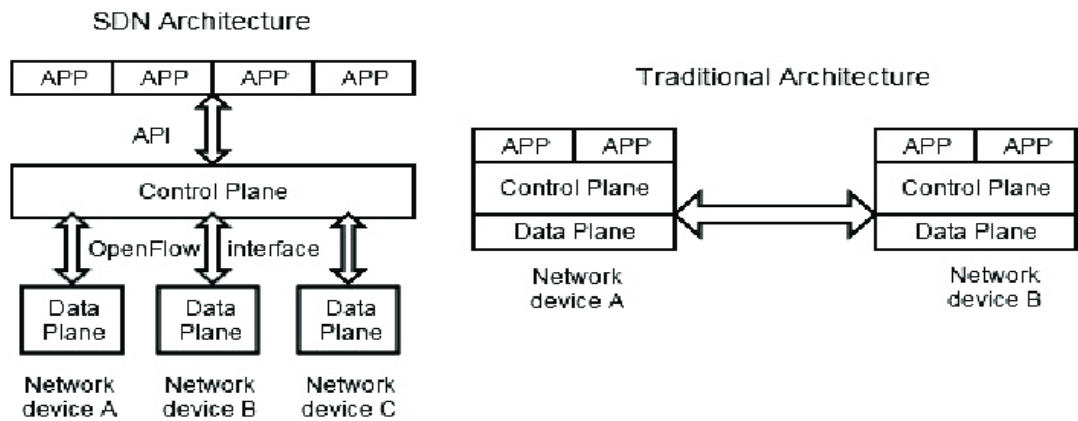
- Software-defined networking-in light of SDN traditional networking

Software-defined networking (SDN) is the virtualization of the network by decoupling hardware from software, as shown in Figure 2.2 making adjustments on the two devices planes.

The control plane is removed from the device and is placed on what is essentially a layer of networking software. The data stays on the networked hardware so that it can still execute commands and direction from the control plane.

What this means for the network is that the hardware no longer controls data paths, allowing centralized software that makes decisions and creates a virtual control mesh in the network to be build out.

SDN allows a network to be flexible to change data handling and flow when the business needs change. If network automation controls, such as APIs are also integrated then the SDN controller can deliver instructions throughout the network quickly and without the need to understand command line and code for each different vendor whose equipment is used.



**Figure 2-2** SDN vs. traditional networking

- Benefits of SDN over traditional networking:

There are two big business benefits to consider when making the traditional networking vs. software-defined networking choice:

Data flow optimization:

Data flow plays a significant role in networking, SDNs perform better for business needs because the SDN controller is able to identify and use multiple path per data flow. Traditional networks are typically stuck with a single path and don't allow you to split traffic across multiple nodes. Using multiple nodes and adjusting flow based on traffic size can help maintain a quick network and prevent a slowdown.

Both need to have a network configuration that supports business goals:

Traditional networks are configured through largely manual processes on a per-device scheme. The core of SDN support is that devices can be automatically configured and adjusted, improving network responsiveness and preventing from taking up all of the time of the system administrator whenever any network element needs to change. Thus, flexibility and speed are two big differences between SDN and traditional networking.

In addition, SDN is usually able to deliver significant cost savings by simply reducing the amount of spend you have to put toward infrastructure-when optimizing devices to sometimes run multiple different use cases dynamically as the traffic changes leading to the most out of existing devices without needing to purchase and install new equipment for each possible use case.

With these cost savings also comes increased visibility, because the system administrators are working from a view point of overall network functionality. They can then adjust resources where needed, changes and flow changes so that the network automatically scales and adjusts. Pre-stage commands for quick responses and even automate control

#### **2.1.4 SDN basic concepts and principles**

This sub-section introduces the principles of SDN, and the functional entities and relationships that form the SDN architecture.

SDN basic components are shown in Figure 2-3

##### **2.1.4.1 Principles**

From this and other sources, several basic principles of SDN may be adduced.SDN focuses on four key features:

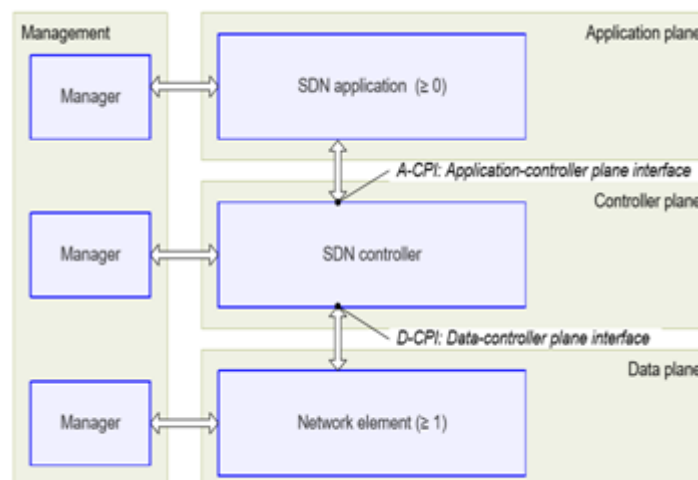
Separation:

A fundamental characteristic of the SDN architecture is the physical separation of the control plane from the forwarding (data) plane.This principle calls for separable controller and data planes. However, it is understood that control must necessarily be exercised within data plane systems. The D-CPI between SDN controller and network element is defined in such a way that the SDN controller can delegate significant functionality to the NE, while remaining aware of NE state.

Conventional routers and switches embody a tight integration between the control and data planes. This coupling made various network-management tasks, such as debugging configuration problems and predicting or controlling routing behavior, exceedingly challenging. To address these challenges, various efforts to separate the data and control planes began to emerge. [3]

Centralization:

- In comparison to local control, a centralized controller has a broader perspective of the resources under its control, and can potentially make better decisions about how to deploy them. Scalability is improved both by decoupling and centralizing control, allowing for increasingly global but less detailed views of network resources. SDN controllers may be recursively stacked for scaling or trust boundary reasons
- Openness: open interfaces between the devices in the control plane and those in the data plane.
- Programmability.



**Figure 2-3** SDN overview

### **2.1.4.2 Components**

Data plane:

The data plane comprises a set of one or more network elements, each of which contains a set of traffic forwarding or traffic processing resources. Resources are always abstractions of underlying physical capabilities or entities.

Data plane functions:

- Forwards traffic to the next hop along the path to the selected destination network according to control plane logic.
- Data plane packets go through the router.
- The routers/switches use what the control plane built to dispose of incoming and outgoing frames and packets.

Control plane:

The control plane comprises a set of SDN controllers, each of which has exclusive control over a set of resources exposed by one or more network elements in the data plane (its span of control).

The minimum functionality of the SDN controller is to faithfully execute the requests of the applications it supports, while isolating each application from all others. To perform this function, an SDN controller may communicate with peer SDN controllers, subordinate SDN controllers, or non-SDN environments, as necessary. A common but non-essential function of an SDN controller is to act as the control element in a feedback loop, responding to network events to recover from failure, re-optimize resource allocations, or otherwise.



Data plane functions:

- Makes decisions about where traffic is sent.
- Control plane packets are destined to (like telnet) or locally originated by the router itself .
- The control plane functions include the system configuration, management, and exchange of routing table information.
- The route controller exchanges the topology information with other routers and constructs a routing table based on a routing protocol, for example, RIP, OSPF or BGP.
- Control plane packets are processed by the router to update the routing table information.
- It is the Signaling of the network.
- Since the control functions are not performed on each arriving individual packet, they do not have a strict speed constraint and are less time-critical.

Application plane:

The application plane comprises one or more applications, each of which has exclusive control of a set of resources exposed by one or more SDN controllers. Additional interfaces to applications are not precluded. An application may invoke or collaborate with other applications. An application may act as an SDN controller in its own right.

Manager:

Each application, SDN controller and network element has a functional interface to a manager. The minimum functionality of the manager is to allocate resources from a resource pool in the lower plane to a particular client entity in the higher plane, and to establish reachability information that permits the lower and higher plane entities to mutually

communicate. Additional management functionality is not precluded, subject to the constraint that the application, SDN controller, or NE have exclusive control over any given resource.

#### Administration:

Each entity in a north-south progression through the planes may belong to a different administrative domain. The manager is understood to reside in the same administrative domain as the entity it manages.

#### Open Programmable Interfaces:

A standardized programmable interface, OpenFlow [4], was adopted by the industry in order to program multiple flavors of forwarding devices (i.e. ASIC, FPGA-based, Network Processors, virtual switches) thereby abstracting the complexity of the underlying hardware. [6]

Several interfaces are identified in Figure 2-4: the Control-Data Interface (also known as the Southbound API such as OpenFlow, OF-Config, OVSDB, NETCONF), the Application-Control Interface (also known as the Northbound API such as REST API) and the East-West Interface between Controllers. The East-West interface refers to the bidirectional and lateral communication between SDN controllers. It is noted that in [6], Jarschel et al. propose a definition referring to the east interface for communication between SDN controllers and the west interface for communication between an SDN controller and other, non-SDN control planes. The interoperability between SDN and legacy control planes is, however, out of scope of this work.

#### ONF protocols:

A companion interface to the programmable interface described above is the switch management protocol (e.g. OF-Config, OVSDB). Such a protocol is required to standardize the configuration and management

functions of the programmable hardware. For instance, the OF-Config protocol is used to configure and manage an OpenFlow capable switch as well as multiple logical switches that can be instantiated on top of the device. Internally, the protocol uses NETCONF as the transport protocol that defines the set of operations over a messaging layer (RPC), which exchanges the switch configuration information between the configuration point and the packet forwarding entity. [7]

#### Third-party Network Services:

SDN allows the integration of third-party network services in the architecture. In a monolithic SDN controller implementation (e.g. RYU, POX, NOX), these applications are compiled and run as part of the controller module while controllers like OpenDayLight allow the instantiation of applications at run-time, without restarting the controller module. This is analogous to operating systems, where in software modules and libraries can be downloaded and integrated within a running environment. From a deployment standpoint, this drives innovation, allows customization of services, introduces flexibility in the overall architecture to adapt to new features, and reduces the cost of proprietary services. Depending on the controller implementation, third-party services can communicate to a controller module via internal APIs or open northbound APIs (e.g. REST APIs) supported by the controller.

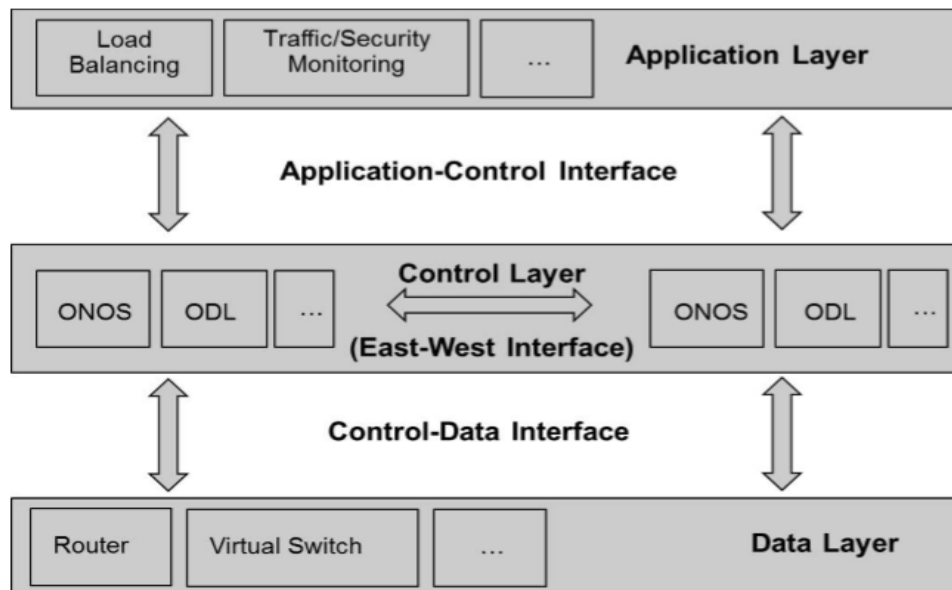
#### Virtualized Logical Networks:

Virtualizing the SDN components supports multi-tenancy in the infrastructure. In a typical SDN network, multiple logical switches can be instantiated in a shared physical substrate such that each entity can represent individual customers. The goal here is to containerize the SDN components thereby guaranteeing customized performance, security, and

Quality of Service (QoS) based on customer requirements. While SDN is developing in the IT community, Network Functions Virtualization (NFV) is being developed by the Telecommunications industry. NFV uses IT virtualization technologies to virtualize network functions/services previously implemented in proprietary hardware appliances. This supports dynamic and agile network service provision. NFV and SDN are closely connected offering a software-based networking paradigm.

#### Centralized Monitoring Units:

Although not unique to the SDN architecture, a centralized monitoring unit unifies the analytical capabilities of the infrastructure and creates a feedback control loop with the controller to automate updates to the networking function. For example, a TAP monitoring unit can feed data traffic to Deep Packet Inspection (DPI) engines that can assess the data traffic, identify attack patterns and then programmatically update the forwarding table to block attack traffic. While the SDN entities can internally include several monitoring capabilities, a typical network deployment would consider deploying dedicated monitoring solutions in the infrastructure. For example, the OpenFlow protocol provides statistical and status information about the switch and its internal state.



**Figure 2-4** SDN Functional Architecture

Figure 2-4 represent theSDN Functional Architecture illustrating the data, control and application layers and interfaces.

### 2.1.5 Network Operating System

- Network elements has two components:
  - OpenFlow client.
  - Forwarding hardware with flow tables.
- The SDN controller must implement the network OS functionality
  - Provide abstraction to the upper layer.
  - Provide control to the underlying hardware.
  - Managing the resources.

OS vs. NOS:

OS:

- Resources managed
  - CPU, memory, disk, IO devices, etc.
- Applications:
  - User programs that use the resources
- OS functionality (abstraction):
  - CPU virtualization
  - Memory virtualization
  - IO virtualization
  - File systems

NOS:

- Resources managed
  - Connected switches/routers/NICs
- Applications
  - Firewall, migration, network virtualization, NAT, TE, etc.
- NOS functionality
  - Network abstraction – this is a new concept that is not well understood.

SDN switches are controlled by a network operating system (NOS) that collects information using the API and manipulates their forwarding plane, providing an abstract model of the network topology to the SDN controller hosting the applications. The controller can therefore exploit complete knowledge of the network to optimize flow management and support service-user requirements of scalability and flexibility.

### **2.1.6 SDN Switches**

OpenFlow switch is a software program or hardware device that forwards packets in SDN environment. OpenFlow switches are either based on the OpenFlow protocol or compatible with it.

Open vSwitch the 'v' stands for virtual. This is a virtual OpenFlow switch. Apart from OpenFlow, it also support other switch management protocols

OpenFlow switch identification:

Each OpenFlow instance on a switch is identified by a Datapath Identifier. This is a 64-bit number determined as follows according to the OpenFlow specifications:

“The datapath\_id field uniquely identifies a datapath. The lower 48 bits are intended for the switch MAC address, while the top 16 bits are up to the implementer. An example use of the top 16 bits would be a VLAN ID to distinguish multiple virtual switch instances on a single physical switch.” [10]

Every OF switch should have a datapath id (DPID), which should be available at the controller when the switch registers with it. The DPID should be unique for every switch that is to be handled by a single controller. Each time a new OpenFlow switch connects to controller, it have to send its own information including DPID port statistics to the controller.

### **2.1.7 SDN Controllers**

The control plane is an essential part of the SDN architecture, so it is very important to give proper attention to any proposal or design of an SDN controller. During the past few years, several controllers have been

developed and several studies have been done to evaluate, compare and test the performance of these controllers.

The followings are Examples of the most common SDN controllers are those and additionalcontroller are shown in Table 2-1.

- NOX: is a first-generation OpenFlow controller, it's one of the most widely used controllers because it is stable, open source and it supports OpenFlow. It is available in two versions either NOX-Classis which is implemented in C++ or Python but is no longer supported. Or either, a newer supported version of NOX which is implemented in C++ only but with a cleaner code base and better performance.
- POX: is essentially the same as NOX but it's implemented in Python only and it's relatively easier to deal with it in terms of writing and reading codes. It is a good choice if the performance is not an issue.
- Ryu: is an open-source Python controller that supports OpenFlow and OpenStack. It had relatively poor performance with respect to other commercial grade controllers.
- Floodlight is an open-source Java controller that supports OpenFlow and is maintained by Big Switch Networks. It's considered the optimum choice for multi-tenant's cloud environment because is supports OpenStack and this type of implementation, it is also suitable when production level performance is needed and it is well documented, however, it has a steep learning curve.
- OpenDayLight: The OpenDayLight Project is a recent opensource project founded by some of the big vendors such as: Big Switch Networks, Brocade, Cisco, Citrix, Ericsson, HP,



IBM, Juniper Networks, Microsoft, NEC, Red Hat and VMware. OpenDayLight is developed as a modular, pluggable, and flexible controller platform. This controller is completely programmed and it is integrated within its own Java Virtual Machine (JVM). Hereby, it can be deployed on any hardware and operating system platform that has Java environment installed. Chapter 3 Background 22 Table 2-1 shows a summary of some properties of some controllers.

Table 2-1: most popular open source SDN controllers

	Program- Ming Language	GUI	Docum- Entation	Modularity	Distributed/ Centralized	Platform Support	Productivity	Southbound Apis	Northbound Apis	Partener	Multithreadin g Support	Openstack Support
<b>ONOS</b>	Java	Web Based	Good	High	D	Linux,MAC OS, And Windows	Fair	OF1.0, 1.3, NETCO NF	REST API	ON.LAB, At&T, Ciena,Cisco, Ericsson,Fujitsu, Huawei,Intel, Nec,Nsf.Ntt Communication, Sk Telecom	Y	N
<b>Open- Day- Light</b>	Java	Web Based	Very Good	High	D	Linux,MAC OS, And Windows	Fair	OF1.0, 1.3, 1.4, NET-CONF/YANG, OVSDDB, PCEP, BGP/Ls, LISP, SNMP	REST API	Linux Foundation With Memberships Covering Over 40 Companies, Such As Cisco, IBM, NEC	Y	Y
<b>NOX</b>	C++	Python + QT4	Poor	Low	C	Most Supported On	Fair	OF 1.0	REST API	Nicira	NOX_MT	N
<b>POX</b>	Python	Python + QT4	Poor	Low	C	Linux,MAC OS, And Windows	High	OF 1.0	REST API	Nicira	N	N
<b>RYU</b>	Python	Yes	Fair	Fair	C	Most Supported On Linux	High	OF 1.0, 1.2, 1.3, 1.4, NETCO NF, OF-CONFIG	REST For South bound	Nippo Telegraph And Telephone Corporation	Y	Y
<b>Beacon</b>	Java	Web Based	Fair	Fair	C	Linux,MAC OS, And Windows	Fair	OF 1.0	REST API	Stanford University	Y	N
<b>Maestro</b>	Java	-	Poor	Fair	C	Linux,MAC OS, And Windows	Fair	OF 1.0	REST API	RICE, NSF	Y	N
<b>Flood- Light</b>	Java	Web/ Java Based	Good	Fair	C	Linux,MAC OS, And Windows	Fair	OF 1.0 , 1.3	REST API	Big Switch Networks	Y	N
<b>Iris</b>	Java	Web Based	Fair	Fair	C	Linux,MAC OS, And Windows	Fair	OF 1.0, 1.3, OVSDDB	REST API	ETRI	Y	N
<b>MUL</b>	C	Web Based	Fair	Fair	C	Most Supported On Linux	Fair	OF 1.4, 1.3, 1.0, OVSDDB, OF-	REST API	Kulcloud	Y	Y
<b>Runos</b>	C++	Web Based	Fair	Fair	D	Most Supported On Linux	Fair	OF 1.3	REST API	ARCCN	Y	N
<b>Lib- Fluid</b>	C++	-	Fair	Fair	-	Most Supported On Linux	Fair	OF 1.0, 1.3	-	ONF	Y	N

### **2.1.8 OpenFlow API**

Before the emergence of OpenFlow, the ideas underlying SDN faced a tension between the vision of fully programmable networks and pragmatism that would enable real-world deployment. OpenFlow struck a balance between these two goals by enabling more functions than earlier route controllers and building on existing switch hardware through the increasing use of merchant-silicon chipsets in commodity switches. Although relying on existing switch hardware did somewhat limit flexibility, OpenFlow was almost immediately deployable, allowing the SDN movement to be both pragmatic and bold. The creation of the OpenFlow API<sup>51</sup> was followed quickly by the design of controller platforms such as NOX<sup>37</sup> that enabled the creation of many new control applications.

## **2.2 Related Work**

### **2.2.1 Topologies and firewall**

Firewall topologies include Simple Dual-Homed Firewall (the simplest and possibly most common way to use a firewall. The Internet comes into the firewall directly), Two-Legged Network (exposes DeMilitarized Zone (DMZ), which is a computer host or small network imposed as a "neutral zone" between a private network and the outside public network. It prevents outside users from getting direct access to a server which contains data), Three-Legged Network, Three-part Network, and so on.

This thesis discusses two firewall topologies, three-legged topology and three-part topology. Shown in figure 2-5. Three-legged has a single firewall while three-part has two firewall devices.

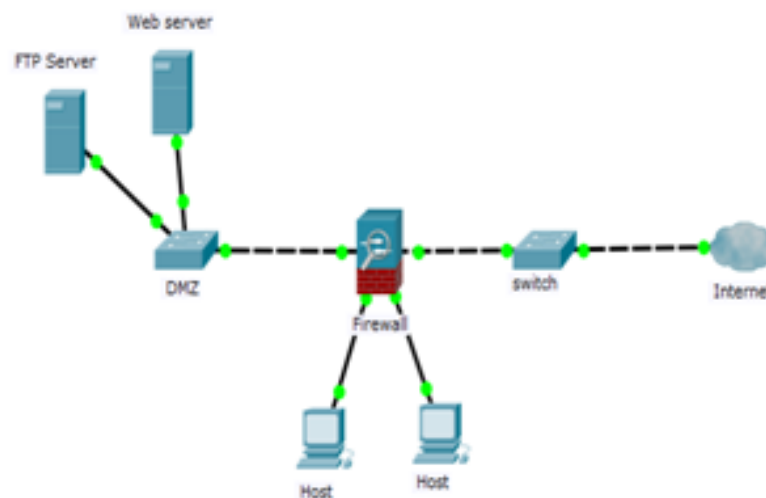
In the former, the firewall is configured to route packets between the outside world and the DMZ differently than between the outside world and the private network.

One of the most important features of this topology is that it can be modified to work as the Simple Dual-Homed Network. The primary disadvantage of the three-legged firewall is the additional complexity. Access to and from the DMZ and to and from the internal network is controlled by one large set of rules.

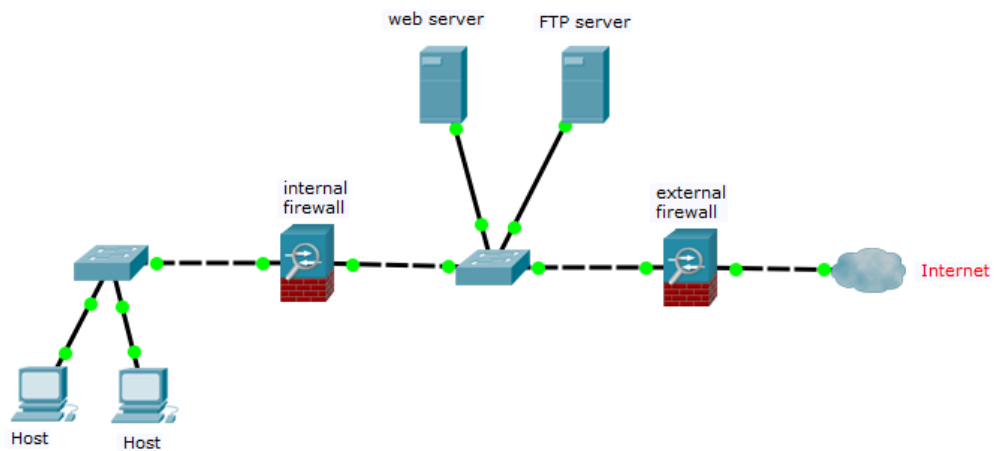
The latter has the following three specialized layers:

- DMZ.
- A router that acts as an inside packet filter between the corporate internetwork and DMZ.
- Another router that acts as an outside packet filter between the DMZ and the outside internetwork.

These two topologies will be implemented and discussed in an attempt to meet security and performance requirements.



Three-legged network topology



### Three-part network topology

**Figure 2-5** the two network topologies under investigation  
SDN Switch a New Form of a Firewall

SDN switches can behave like a firewall. Many people anticipated that enterprise organizations would adopt Software Defined Network (SDN) technologies later than service providers or multi-tenant data centers and cloud service providers. We are now seeing more use of Network Functions Virtualization (NFV) within enterprises and some enterprises are starting SDN pilot projects. As enterprises consider how to utilize SDN technologies in their data center environments, they start to consider what new security capabilities SDN can provide. SDN switches can drop packets for flows that are not permitted by the controller.

Software Defined Networking evolved from the concept of decoupling the lower-layer packet/frame forwarding from the control function that intelligently determines how application traffic should be transported. The separation of the control plane from the forwarding plane allows networks to facilitate packet processing in new and innovative ways and

created a new paradigm for network virtualization. SDN has opened up a whole new world of network design and enabled creative approaches to networking. SDN has also caused us to reconsider how security policies are enforced within the network.

In the OpenFlow SDN model, the flows within a network switch are placed there by an OpenFlow controller. If a flow is not present (table-miss), then the switch punts to the controller to ask for help determining how the packet should be forwarded. The OpenFlow technical specifications state that if the table-miss flow entry is not present in the switch and there is no rule to send the packet to the controller, then the packet is dropped by the switch. If the switch punts the packet to the controller, then the controller processes the Packet-in message and determines the fate of that packet. The controller then determines if the packet should be forwarded or dropped. This behavior sounds like the SDN switch is behaving like a firewall and enforcing the “that which is not included in the flow table, is dropped” standard security policy. This can be thought of as similar to the default “Fail-Safe Stance ” that was also mentioned in the book *Building Internet Firewalls* , by Elizabeth D. Zwicky, Simon Cooper and D. Brent Chapman. On first blush, this sounds like a great new form of security and makes it seem like every port of an SDN switch can behave like a firewall.

Many SDN switches behave much like a standard Ethernet switch and flood traffic out all ports for Ethernet frames destined to broadcast, multicast or unknown MAC addresses. Most SDN switches flood normal ARP traffic like a typical hardware-based Ethernet switch. In most situations, the default behavior for an SDN switch is to act like an Ethernet bridge, or learning switch. However, it is possible to put an

SDN switch into an explicit forwarding mode whereby only flows allowed or configured/pushed by the controller are allowed.

If every Ethernet switch in the environment could perform like a traditional firewall, it would change the way security policy is implemented in a networked environment. Imagine if every Ethernet switch were a multi-port firewall, then firewall policies could be implemented throughout the network at every ingress switch port and on every link between switches. There would be firewalls for every server, desktop, every link, and the firewall policy would be implemented by a controller that maintained a global view of the current application traffic and what traffic should be permitted. Having security policy enforced throughout the environment would mean the complete erosion of the security perimeter. Having that many security policies implemented and maintained manually would be an administrative nightmare. However, with a controller architecture, the policy would be created once and then pushed down to every network device for enforcement.

Network slicing is one of the popular use cases of SDN. A network can be logically carved up into logically separated networks overlaid upon the same physical network hardware. Network slicing is a popular use case within universities because they would like to separate different departments (admissions, finance, residence halls, computer science departments, etc.) into their own logical network enclaves. The SDN can separate the networks similar to Virtual Routing and Forwarding (VRF) instances may be used to separate layer-3 forwarding. This can also be done by adding a slicing layer between the control plane and the data plane, thus making the security policies slice-specific. Enforcement of strong isolation between slices in “Flowspace” means that actions in one slice do not affect another slice. For more information look at Flowvisor

and the FSFW: FlowSpaceFirewall. An example of this would be the Cisco Extensible Network Controller (XNC) with the Networking Slicing application. In these ways, SDNs can provide the “Diversity of Defense” concept which was also mentioned in Building Internet Firewalls.

The key concept to the feasibility of using an SDN-enabled switch as a firewall is the state that it would maintain the application traffic flows. Access Control Lists (ACLs) are not stateful and do not have awareness of when the connection started or ended. Even with the good-old Cisco ACL CLI parameter “established”, the ACLs became only slightly “statefulish”. ACLs typically do not pay any attention to the three-way TCP handshake (SYN, SYN-ACK, and ACK) or the FIN/ACK session teardown. Stateful firewalls, on the other hand, observe the session establishment and close process and apply their policies directionally using Stateful Inspection.

So, how do modern SDN products implement security and could they behave like a traditional firewall? When it comes to Cisco’s Application Centric Infrastructure (ACI), the Nexus 9000 switches operate in a stateless manner. The Application Network Profiles (ANPs) configured in the Application Policy Infrastructure Controller (APIC) are deployed to the switches in the ACI fabric in a stateless fashion. Therefore, an ACI system would not be able to operate with the same level of security as a standard stateful firewall. This is why ACI allows for Layer-4-to-7 Service Graphs to be configured and integrated into the ACI fabric.

When it comes to OpenvSwitch (OVS), it has supported only stateless matches on policies. It is possible to configure OVS policies that match TCP flags or configure rules to use a “learn” method to establish the

return flow of traffic. However, neither of these methods are stateful like a traditional stateful inspection firewall. There is work being done by the Open vSwitch community to have connection tracking ( Conntrack ) to allow the OVS to notify a Netfilter (think ip\_tables) connection tracker and maintain a state table of existing sessions.

Project Floodlight can configure ACLs, however, these also operate like a stateless firewall. Floodlight has a Firewall application module that enforces ACL rules by inspecting Packet-In behavior. This works in a reactive way, with the first packet helping to instantiate the flow and traffic is allowed or denied based on the priority-sorted policy rule-set. Rules are allowed to have overlapping flowspace but the priority creates the first-match rule action top-down policy.

VMware NSX has the ability to configure security policies within the SDN environment. NSX for vSphere supports logical switching/routing, firewall, load balancing, and VPN functionality. The firewall rules are enforced at the vNIC, but the firewall policy is associated with the VM and when the host moves, so does the policy. The NSX Distributed Firewall is a kernel loadable module and provides stateful L2/L3/L4 dual-protocol firewalling and can do anti-spoofing. The VMware NSX firewall policies operate like a Cisco router with a reflexive ACL. When it comes to Equal Cost Multi-Path (ECMP) designs or High-Availability (HA), the NSX Edge Services Gateway firewall functions in a stateless manner. In other words, stateful firewalling and load balancing or NAT are not supported by the Edge Services Gateway with an HA or ECMP topology.

There are groups who are working to try to create SDN systems that provide robust security policy enforcement. Research projects like



FlowGuard and a paper titled “An OpenFlow-Based Prototype of SDN-Oriented Stateful Hardware Firewalls” written by Jacob Collings at the University of North Dakota show that there may be potential to establish statefulness within the SDN network devices.

From this analysis, we can conclude that SDN switches that obtain their forwarding policies from a controller are not necessarily stateful. These SDN-enabled switches are, therefore, unable to provide the same level of protection as a stateful firewall. It is important to ask the vendor about the details of the statefulness of their firewall capabilities in their SDN solutions and understand how they operate. Because many of these SDN systems may operate in a stateless way, if the organization requires stateful firewall protections, then SDN policies must be used to steer the traffic with service-chaining toward a stateful packet inspection Network Functions Virtualization (NFV) firewall.

### **2.2.2 SDN implementation**

Strategies:

i. Proprietary SDN Strategy :

One of the two distinctive SDN trends in the networking world with respect to SDN is to have products with proprietary software components. This approach gives importance to programmability but puts restrictions on the openness by having proprietary components in a programmable infrastructure. Customers are interested in the programmability of computer networks while the industry is deciding about the placement of OpenFlow and SDN in the current network system. Therefore, while OpenFlow and SDN are important developments, the thing that will get customers excited is exposing the intelligence that's already there in the network and being able to program networks as per their needs. Having such an approach means that instead of dealing with protocol

configuration such as border gateway protocol (BGP) and virtual local area networks (VLAN) setups, a user just passes on a network the requirement of a connection between two points under certain service level agreement (SLA). The underlying network will make the arrangement to complete this networking task. All industrial players in the networking world agree on the requirement of programmability, but they differ in how this programmability should practically be implemented. Some hardware vendors want to use proprietary components in implementing programmability but the other type of vendors may consider it a compromise on being open source. [8]

Many vendors, such as VMware and Cisco, sell proprietary SDN controllers along with higher-level software applications as a part of their programmable networking system. Cisco offers, in particular, a range of products to suite to various levels of networking

ii. Open Source and Non-proprietary :

The flowers of the open source and non-proprietary SDN believe that the main purpose of SDN survives by being available to all as an open source provision with no hidden proprietary components so that the independent use and development of SDN could grow further. Some hardware vendors might like the notion of being open source or not, but the competition in the market is forcing them to consider open source options as their competitor shave already found a fit of the open standard in their product lines. Based on this aspect, NSX product of VMware is capable of creating a virtual network overlay that is loosely coupled to the physical network underneath. Similarly, OpenContrail by Juniper is an SDN controller freely available through an open-source license. A re-

branded version comes with services and support on per socket cost basis.

The open programmable SDN product suite by Big Switch enables easy adaptation of new network applications in an easier way as compared to the adaptation of new applications with traditional and non-programmable networks. This hardware platform-independent suite supports open standards and APIs including OpenFlow. HP also backs up open standard and offers an OpenFlow-enabled SDN controller and switches.

A comparison between proprietary and non-proprietary SDN strategies is shown in tabular format in table 2-2.

Table 2-2: proprietary vs. non-proprietary SDN strategies.

SDN Strategy	Control)	Feedback	Support	Standardization
Proprietary	Restricted	Yes	Yes	Under control of one vendor
Non-proprietary	Open to all	No	Limited without payment	Wait and see

### Firewall implementation:

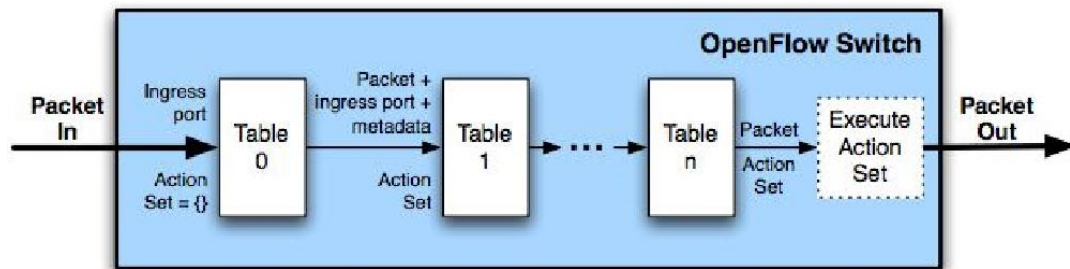
In a recent work simple firewall is implemented in SDN using OpenFlow.

One can think of the control plane as being the network’s “brain” as it is responsible for making all decisions, while the data plane is what actually moves the data.

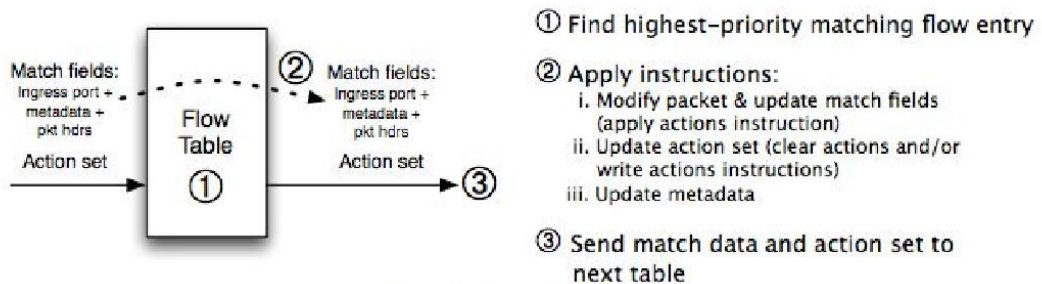
The SDN control plane is implemented by the “controller” and the data plane by “switches”. The controller acts as the brain of the network, and sends rules to the switches on how to handle traffic. OpenFlow has emerged as the de facto SDN standard and specifies how the controller

and the switches communicate as well as the rules controllers install on switches.

Mininet, pox and OpenFlow 1.3 -the version the OpenFlow protocol supported within the Mininet environment- used to establish the firewall. Figure 2-6 and Figure 2-7 explain the operation of OpenFlow switches.



**Figure 2-6** Packets are matched against multiple tables



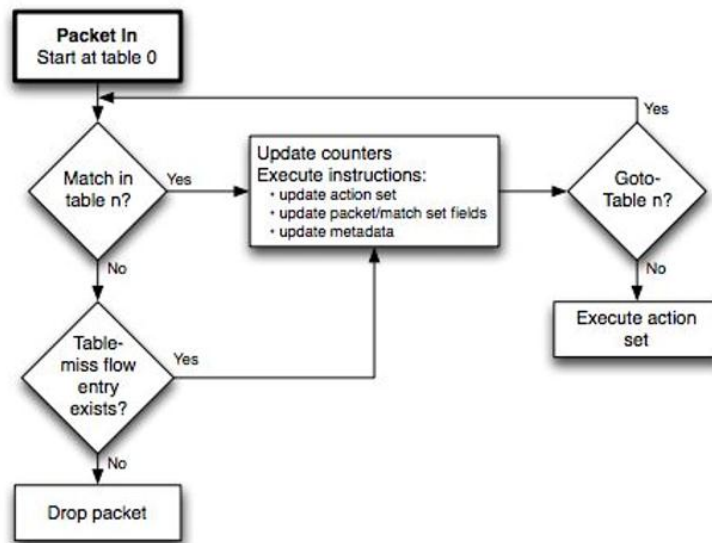
**Figure 2-7** Pre-table packet processing

When the packet comes into an OpenFlow switch, the switch will reference a table containing “rules” and “actions”. This flow table looks like the one shown in Figure 2-8 and it contains the fields showed in Figure 2-7.

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie
--------------	----------	----------	--------------	----------	--------

**Figure 2-8** Flow table fields

Figure 2-9 below shows the flow of execution that follows.



**Figure 2-9** Packet flow through an OpenFlow switch Flowchart.

If an `ofp_packet_in` does not match any of the flow entries and the flow table does not have a “table-miss” flow entry, the packet will be dropped. If the packet matches the “table-miss” flow entry, it will be forwarded to the controller. If there is a match-entry for the packet, the switch will execute the action stored in the instruction field of the corresponding flow table.

# **Chapter Three**

## **Approach**

# **Chapter Three**

## **Approach**

- 3.1 Introduction
- 3.2 Suggested topologies
- 3.3 Rules

### **3.1 Introduction**

In this chapter research methodology will be demonstrated in a step-by-step basis using Mininet emulator tool, with the use of the POX controller and OpenFlow switches to implement the three-legged and the three-part network topology.

Mininet will be operated in a VirtualBox –Linux environment to enable us to have the privileges that Linux have while other operating system haven't to allow us to implement the topologies-presented earlier using SDN technology to compare them and extract the advantages that SDN provides.

Also all the used tools and components will be explained to give the reader a good overview so that even a common person with no previous knowledge knows what this thesis is about, then it will be taken to a higher level that needs back knowledge of various concepts that only a developer/engineer can understand.

### **3.2 Suggested topologies**

- i. Three-legged topology:

Three-legged topology-as its name proclaims-has three-legged,all connected to one single firewall that apply network policies to protect the internal network in the first place.

One of the legs represents the DMZ area, the two others represents Intranet and Internet.

DMZ contains servers that are frequently accessed by the Internet (untrusted) clients.

Intranet implies the network in need for protection because it contains sensitive data.



ii. Three-part topology:

As the name suggests it, consist of three parts managed by two firewalls (Dual-firewall topology).

Middle part is the DMZ, the two bounding areas are the Internet and the Intranet.

The external firewall connects the Internet with the DMZ, while the internal firewall connects the DMZ to the internal network.

- Operating Environment :

Oracle VM VirtualBox :

VirtualBox is a powerful x86 and AMD64/Intel64 virtualization product for enterprise as well as home use. It is a hypervisor used to run operating systems in a virtual machine, on top of the existing operating system. VirtualBox has an ever growing list of features. It comes with a GUI (graphical user interface), as well as headless and SDL command-line tools for managing and running virtual machines. In order to integrate functions of the host system to the guests, including shared folders and clipboard, video acceleration and a seamless window integration mode, guest additions are provided for some guest operating systems. Not only is VirtualBox an extremely feature rich, high performance product for enterprise customers, it is also the only professional solution that is freely available as Open Source Software under the terms of the GNU General Public License (GPL) version 2.

VirtualBox runs on Windows, Linux, Macintosh, and Solaris hosts and supports a large number of guest operating systems.

Mininet emulator:

Mininet is an open-source network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking. It is designed to support research and education in the field of SDN systems as it creates a simulated network that runs real software on the components of the network so it can be used to interactively test networking software. It helps in creating complex custom scenarios using the Mininet Python API.

Mininet provides a simple GUI editor that helps in the implementation of network topologies and also provide a code for the illustrated topologies. As a conclusion,Mininet combines many of the best features of emulators, hardware testbeds, and simulators.

POX:

POX is an open source development platform for Python-based software-defined networking (SDN) control applications, such as OpenFlow SDN controllers. It enables rapid development and prototyping. POX started life as an OpenFlow controller but can now also function as an OpenFlow switch and can be useful for wiring networking software in general.The ultimate goal for POX is to use it to create “an archetypal, modern SDN controller.”

- Configuration:

Xming:

At a very early stage the X windows system server (Xming).Figure 3-1.Must be activated to obtain basic framework for a GUI environment.

Xming allows us to pipe graphics out of our Mininet VM, receive them via putty (explained below), and display them via Xming.

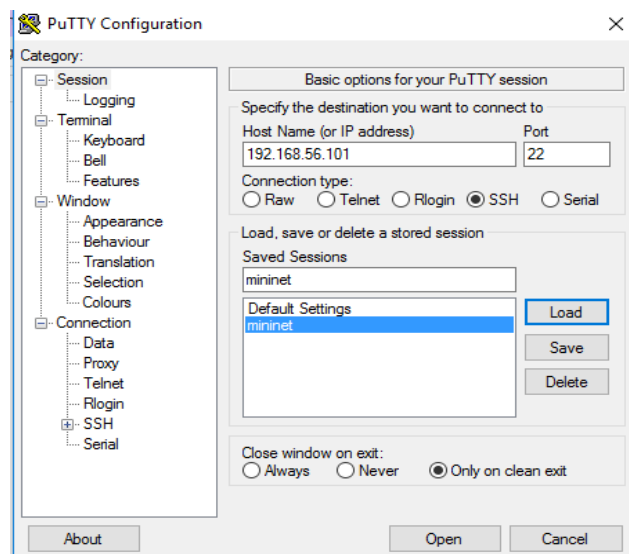


**Figure 3-1** Xming

Putty:

is an SSH(secure shell) /Telnet client that gives a keen terminal to talk to the Mininet VM.it can also do SSH X11 forwarding which is important for getting XTerms running in the VM on the host (windows in our case)desktop. Putty also allows copy/paste features.

Mininet is loaded into the putty with the host IP address 192.168.56.101 in our case, and port number 22, and SSH connection type as shown in Figure 3-2.



**Figure 3-2** Putty page.

Then Mininet is activated on VirtualBox to start the implementation of the two topologies (three-legged and three-part).Figure3-3.

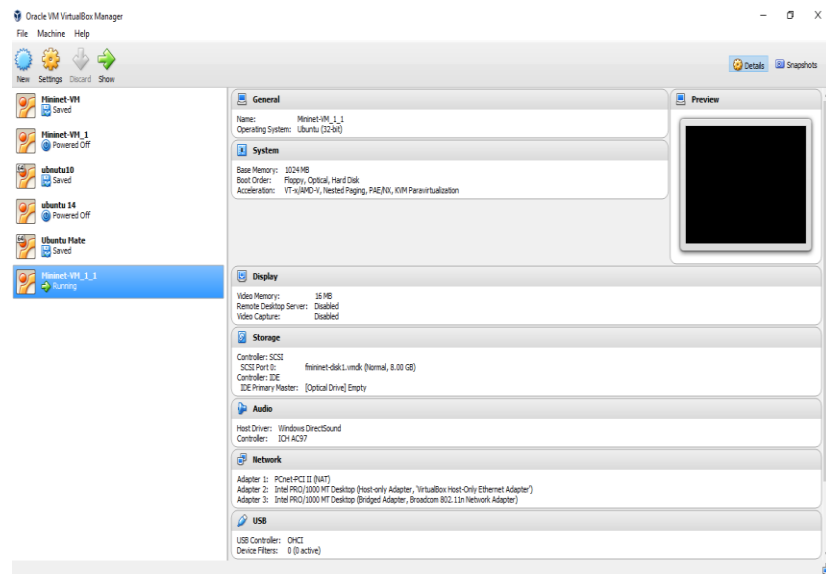


Figure3-3 VirtualBox

Then Mininet terminal opens as shown in Figure 3-4 (A) and all nexts step will be done using this terminal and the Putty terminal of Figure 3-4 (B) with the aid of Linux command line (as we mentioned earlier Mininet is a Linux-based environment).

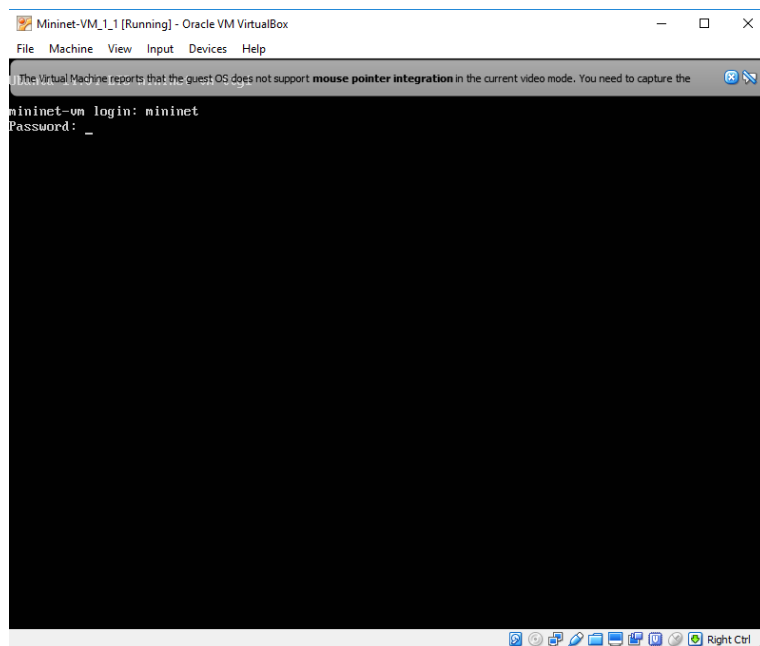
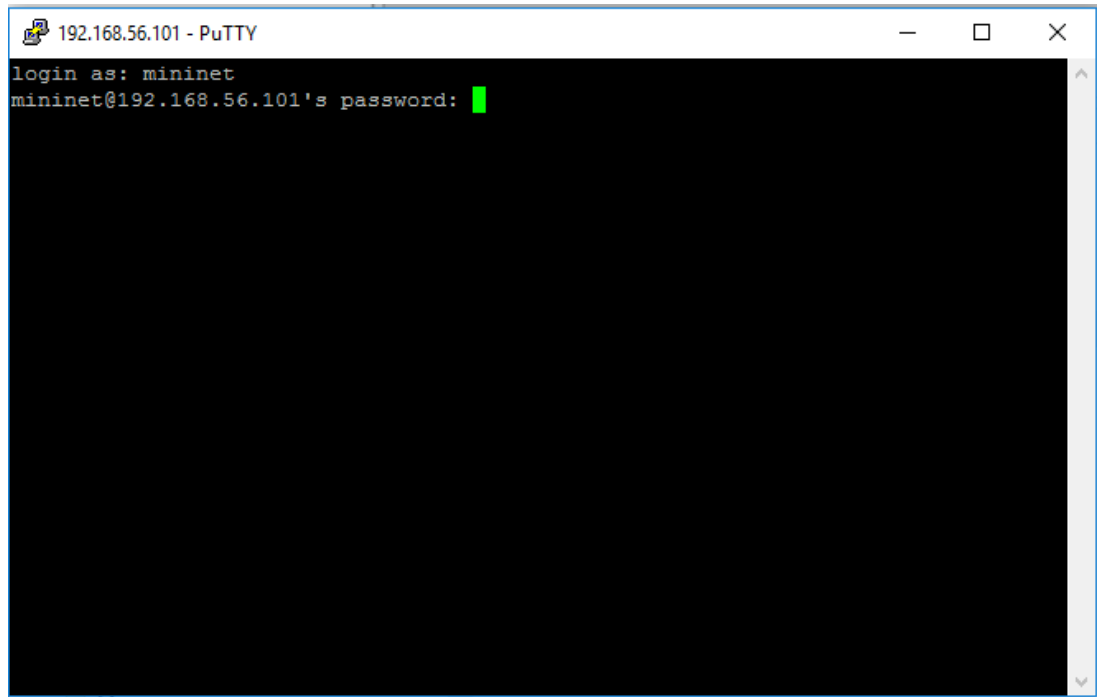


Figure 3-4

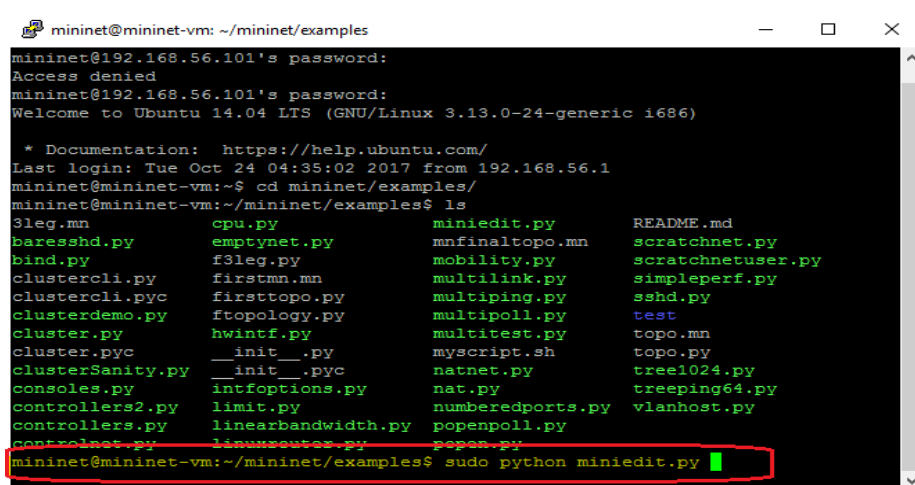
(A):Mininetcommandline terminal



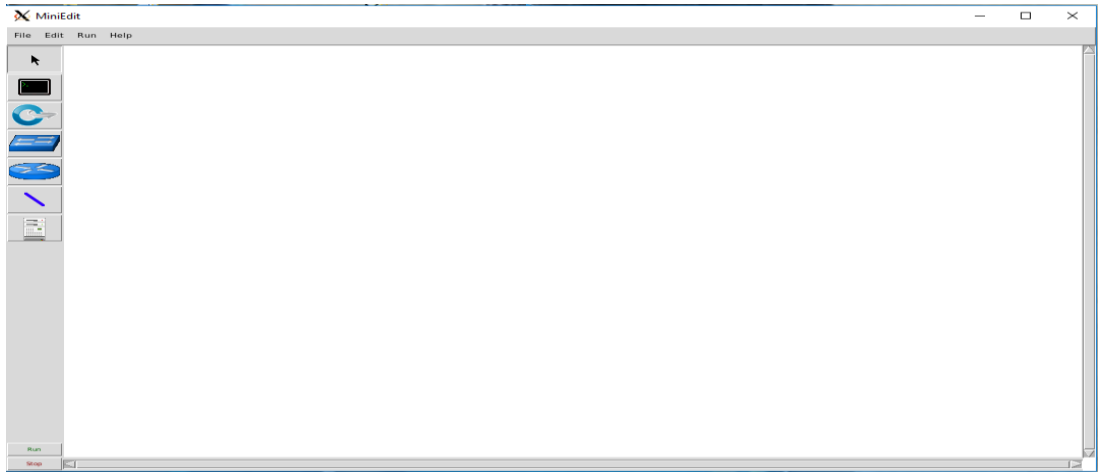
**Figure 3-4** (B):Putty terminal

Logging in with the username “mininet” and password is also “mininet” (it will not appear as you type it).

Then heading to MiniEditGUI (Figure3-5) using the command shown in Figure 3-6 to implement the two topologies that is place in /mininet/examples directory.



**Figure 3-5** MiniEdit opening command

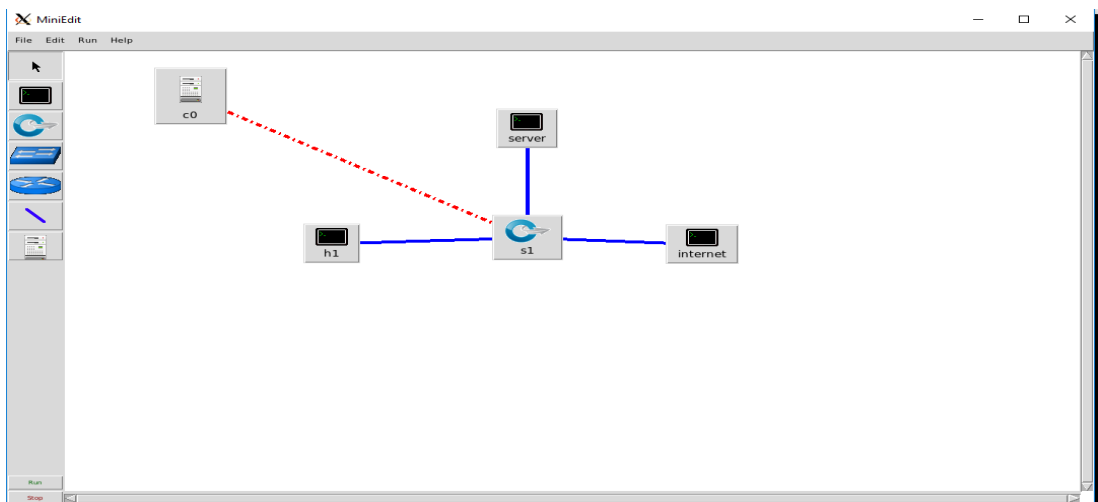


**Figure 3-6** Opening MiniEdit

Then the topologies can be implemented in an environment that looks like the PacketTracer environment used to demonstrate traditional networks. Which contains host, OpenFlow switch, legacy switch, legacy router, netlink, and the SDN controller.

Three-legged topology in SDN:

The firewall is implemented using an OpenFlow switch that can be configure to work as a filter to supervise and monitor packets that come in and packets go out. Figure 3-7 shows three-legged topology implementation on the MiniEdit.

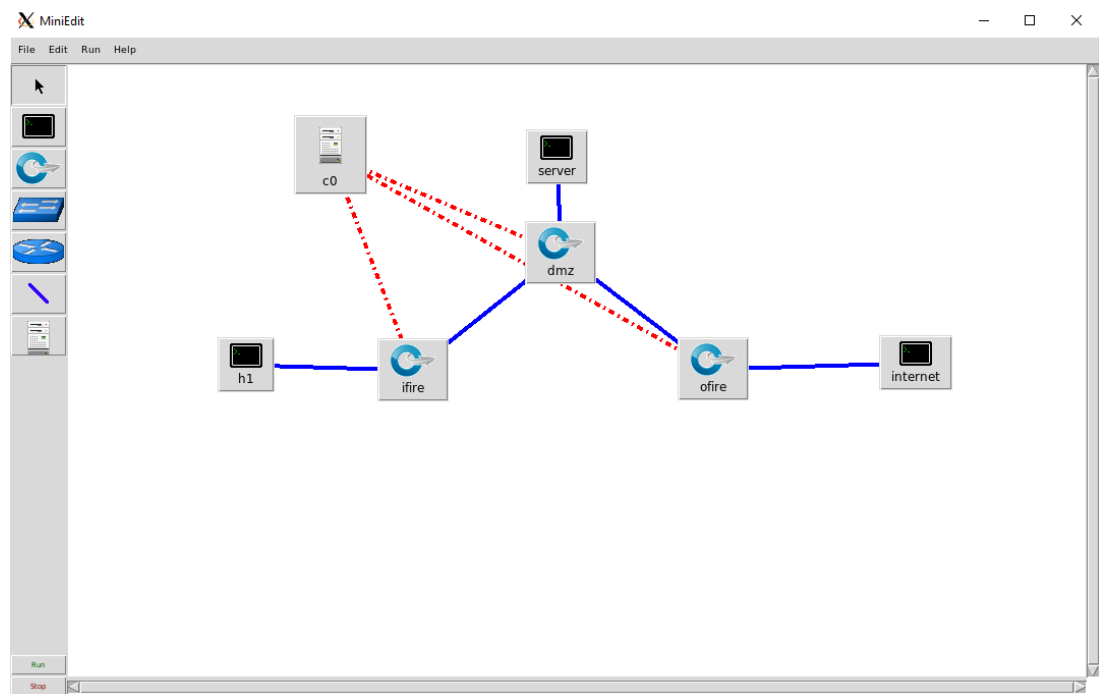


**Figure3-7** Three-legged in MiniEdit

The switch is directly connected to the POX controller, so it can get the rules from the controller to accomplish its forwarding job and to know exactly how to deal with each packet type based on the controller's guide.

Three-part topology in SDN:

The same method followed in implementing the three-legged topology will be followed to implement the three-part topology as shown in Figure 3-8.



**Figure3-8** Three-part in MiniEdit

Mininet has a great feature which is generation of the code of the topologies implemented on the MiniEdit. See appendix A and appendix B.

POX controller configuration:

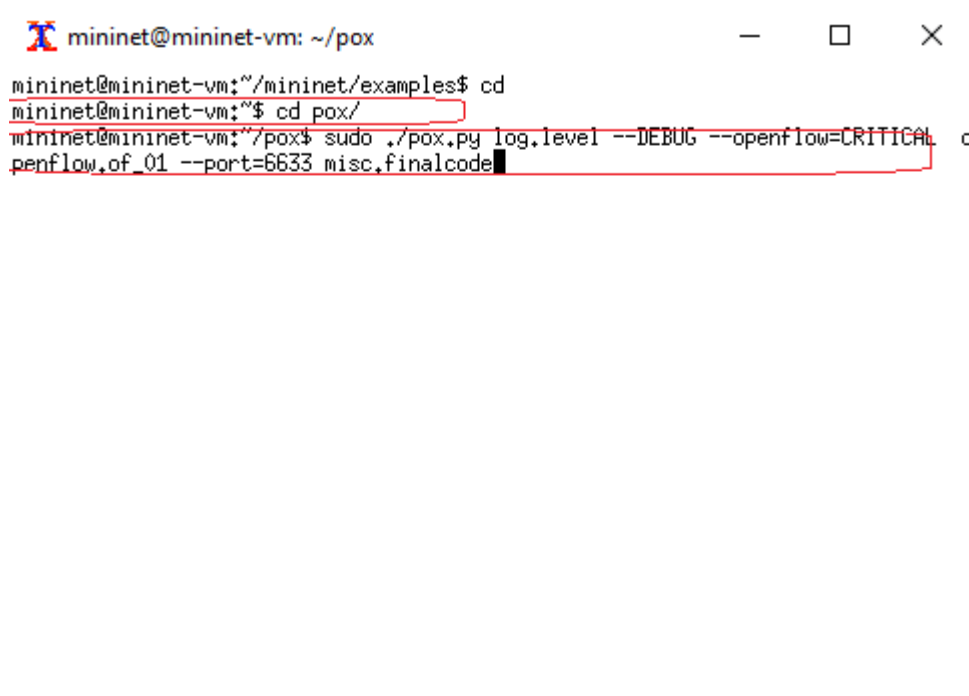
As explained in details in the previous chapter, SDN basic characteristic is the decoupling of the control plane and the data plane.

The controller is the brain of the network. All configuration information are implemented and stored in the controller and then distributed to the rest of the network components (OpenFlow switches).

As we are dealing with a firewall, the rules and policies that govern the flow of the packets in both ways through the OpenFlow switches are stored in the POX controller to ensure the safety of the data in the internal network.

Same rules will be implemented on both controllers of the two implemented topologies.

The POX is activated and open using the command shown in figure 3-9



```
mininet@mininet-vm: ~/pox
mininet@mininet-vm:~/mininet/examples$ cd
mininet@mininet-vm:~$ cd pox/
mininet@mininet-vm:~/pox$ sudo ./pox.py log.level --DEBUG --openflow=CRITICAL
penflow.of_01 --port=6633 misc.finalcode o
```

**Figure3-9** Command to activate POX

### 3.3 RULES



POX is a python-based controller, so in order to apply the rules the code must be written in python programming language.

As shown in Figure 3-6 and Figure 3-7 we have one server in the DMZ, one host (h1) in the internal network, and the internet (of course additional terminals can be added, but we made it as simple as possible to help the reader to understand the configuration), HTTP (hypertext transfer protocol) and ICMP (internet control message protocol) protocols are used to demonstrate the concept.

h1 can have HTTP services and can ping both of the server and the Internet, while the server cannot do both because mainly its job is only to provide services, clients from the Internet can have HTTP service from the server, but cannot ping the server (for security purposes), Internet clients cannot neither ping or have HTTP services from the host in the Intranet. These details are presented in Table 3-1.

We configured these basic rules into the POX controller with aid of a python script. See appendix C.

Table 3-1: controller rules

<b>Source</b>	<b>Destination</b>	<b>Protocol</b>	<b>Action</b>
h1	Server	http/icmp	Accept
h1	Ithesnترنت	http/icmp	Accept
Internet	Server	http	Accept
Internet	Server	Icmp	Deny
Internet	h1	http/icmp	Deny
Server	-	-	Deny

## **Chapter Four**

### **Results**

# **Chapter Four**

## **Results**

4.1 Results

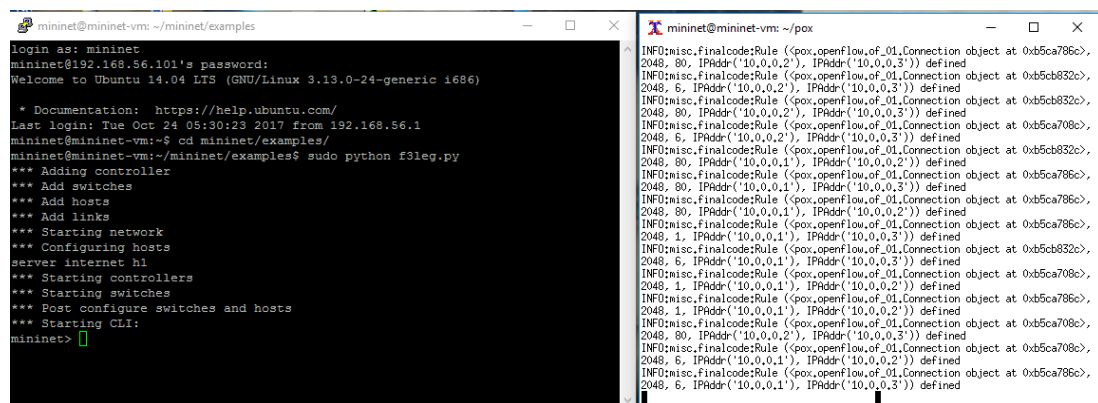
4.2 Comparison

This chapter verifies the convenience of the implementation of the two topologies, and presents the result of applying the rules proposed in the previous chapter.

Depending on the results a comparison between three-legged and three-part network topologies that already implemented using SDN in the previous chapter is provided.

## 4.1 Results

Topology and controller are activated using the commands shown in Figure 4-1 to start making the tests.



```
mininet@mininet-vm: ~/mininet/examples
Login as: mininet
mininet@192.168.56.101's password:
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic 1686)

 * Documentation:  https://help.ubuntu.com/
Last login: Tue Oct 24 05:30:23 2017 from 192.168.56.1
mininet@mininet-vm:~$ cd mininet/examples/
mininet@mininet-vm:~/mininet/examples$ sudo python f3leg.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server internet h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> 
```

```
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca786c>,
2048, 80, IPAddr('10.0.0.2'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5cb832c>,
2048, 6, IPAddr('10.0.0.2'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5cb832c>,
2048, 80, IPAddr('10.0.0.2'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca708c>,
2048, 6, IPAddr('10.0.0.2'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5cb832c>,
2048, 80, IPAddr('10.0.0.1'), IPAddr('10.0.0.2')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca786c>,
2048, 80, IPAddr('10.0.0.1'), IPAddr('10.0.0.2')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca786c>,
2048, 1, IPAddr('10.0.0.1'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5cb832c>,
2048, 6, IPAddr('10.0.0.1'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca708c>,
2048, 1, IPAddr('10.0.0.1'), IPAddr('10.0.0.2')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca786c>,
2048, 1, IPAddr('10.0.0.1'), IPAddr('10.0.0.2')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca708c>,
2048, 80, IPAddr('10.0.0.2'), IPAddr('10.0.0.3')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca708c>,
2048, 6, IPAddr('10.0.0.1'), IPAddr('10.0.0.2')) defined
INFO:misc.Finalcode:Rule (<pox.openFlow.of_01.Connection object at 0xb5ca786c>,
2048, 6, IPAddr('10.0.0.1'), IPAddr('10.0.0.3')) defined
```

**Figure 4-1** Activation of topology and controller

In the figures below different scenarios are executed to verify the feasibility of the employed rules in both topologies.

Three-legged tests:

Tests conducted for the three-legged network are shown in the screens of Figure 4-2 below.

```
mininet@mininet-vm: ~/mininet/examples
cluster.pyc      __init__.py      myscript.sh      topo.py
clusterSanity.py __init__.py      natnet.py        tree1024.py
consoles.py     intfoptions.py  nat.py           treeping64.py
controllers2.py limit.py         numberedports.py vlanhost.py
controllers.py  linearbandwidth.py popenpoll.py
controlnet.py   linuxrouter.py  popen.py
mininet@mininet-vm:~/mininet/examples$ sudo python f3leg.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server internet h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> h1 ping server
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=89.4 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.126 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.060 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 0.060/29.870/89.424/42.111 ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
mininet> h1 ping internet
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=147 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=1.56 ms
^X64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.059 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 3 received, 25% packet loss, time 3006ms
rtt min/avg/max/mdev = 0.059/49.556/147.046/68.938 ms
mininet> h1 ping internet
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=19.2 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.218 ms
^C
--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1003ms
rtt min/avg/max/mdev = 0.218/9.727/19.236/9.509 ms
mininet> internet ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1002ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
mininet@mininet-vm:~/mininet/examples$ sudo python f3leg.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server internet h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> h1 ping server
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=89.4 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.126 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.060 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2005ms
rtt min/avg/max/mdev = 0.060/29.870/89.424/42.111 ms
mininet> internet ping server -c 1
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
mininet@mininet-vm:~/mininet/examples$ sudo python f3leg.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server internet h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> server ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4000ms
mininet> server ping internet
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 999ms
```

```
mininet@mininet-vm: ~/mininet/examples
mininet> server python -m SimpleHTTPServer 80 &
mininet> internet python -m SimpleHTTPServer 80 &
mininet> h1 wget -O - - server
--2017-10-22 12:19:05-- http://10.0.0.3/
Connecting to 10.0.0.3:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2298 (2.2K) [text/html]
Saving to: 'STDOUT'

0% [-----] 0 --.-K/s
!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="3leg.mn">3leg.mn</a>
<li><a href="__init__.py">__init__.py</a>
<li><a href="__init__.pyc">__init__.pyc</a>
<li><a href="baesshd.py">baesshd.py</a>
<li><a href="bind.py">bind.py</a>
<li><a href="cluster.py">cluster.py</a>
<li><a href="cluster.pyc">cluster.pyc</a>
<li><a href="clustercli.py">clustercli.py</a>
<li><a href="clustercli.pyc">clustercli.pyc</a>
<li><a href="clusterdemo.py">clusterdemo.py</a>
<li><a href="clusterSanity.py">clusterSanity.py</a>
```

```
mininet@mininet-vm: ~/mininet/examples
^[A^[[A^?^?^?^?^?^?^?^?
^C
mininet> h1 wget -O - - internet
Serving HTTP on 0.0.0.0 port 80 ...
10.0.0.3 - - [23/Oct/2017 01:38:20] "GET / HTTP/1.1" 200 -
--2017-10-23 01:41:13-- http://10.0.0.2/
Connecting to 10.0.0.2:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2298 (2.2K) [text/html]
Saving to: 'STDOUT'

0% [-----] 0 --.-K/s
!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="3leg.mn">3leg.mn</a>
<li><a href="__init__.py">__init__.py</a>
<li><a href="__init__.pyc">__init__.pyc</a>
<li><a href="baesshd.py">baesshd.py</a>
<li><a href="bind.py">bind.py</a>
<li><a href="cluster.py">cluster.py</a>
```

```
mininet@mininet-vm: ~/mininet/examples
mininet> internet python -m SimpleHTTPServer 80 &
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> server wget -O - h1
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 01:38:20-- http://10.0.0.1/
Connecting to 10.0.0.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2298 (2.2K) [text/html]
Saving to: 'STDOUT'

0% [ 1000 1 0 --.-K/s
!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="3leg.mn">3leg.mn</a>
<li><a href="__init__.py">__init__.py</a>
<li><a href="__init__.pyc">__init__.pyc</a>
<li><a href="baresshd.py">baresshd.py</a>
<li><a href="bind.py">bind.py</a>
<li><a href="cluster.py">cluster.py</a>
<li><a href="cluster.pyc">cluster.pyc</a>
```

```
mininet@mininet-vm: ~/mininet/examples
dmz ifire ofire
*** Stopping 3 hosts
internet h1 server
*** Done
mininet@mininet-vm:~/mininet/examples$ sudo python f3leg.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server internet h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> h1 python -m SimpleHTTPServer 80 &
[1] 25335
mininet> internet python -m SimpleHTTPServer 80 &
mininet> server python -m SimpleHTTPServer 80 &
mininet> server wget -O - internet
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 07:30:34-- http://10.0.0.2/
Connecting to 10.0.0.2:80...
```



```

mininet@mininet-vm: ~/mininet/examples
2017-10-22 12:19:05 (196 MB/s) - written to stdout [2298/2298]
mininet> internet wget -O - server
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-22 12:23:11-- http://10.0.0.3/
Connecting to 10.0.0.3:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2298 (2.2K) [text/html]
Saving to: 'STDOUT'

 0% [          ] 0 --.-K/s
!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="3leg.mn">3leg.mn</a>
<li><a href="__init__.py">__init__.py</a>
<li><a href="__init__.pyc">__init__.pyc</a>
<li><a href="baresshd.py">baresshd.py</a>
<li><a href="bind.py">bind.py</a>
<li><a href="cluster.py">cluster.py</a>
<li><a href="cluster.pyc">cluster.pyc</a>
<li><a href="clustercli.py">clustercli.py</a>
<li><a href="clustercli.pyc">clustercli.pyc</a>
<li><a href="clusterdemo.py">clusterdemo.py</a>
<li><a href="clusterSanity.py">clusterSanity.py</a>

```

```

mininet@mininet-vm: ~/mininet/examples
s1
*** Stopping 3 hosts
server internet h1
*** Done
mininet@mininet-vm:~/mininet/examples$ sudo python f3leg.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
server internet h1
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> internet wget -O - h1
--2017-10-23 07:34:50-- http://10.0.0.1/
Connecting to 10.0.0.1:80...
^C
mininet> server wget -O - internet
--2017-10-23 07:35:14-- http://10.0.0.2/
Connecting to 10.0.0.2:80... ^C
mininet>

```

**Figure 4-2** Three-legged rules tests

Three-part:

Tests conducted for the three-part network are shown in the screens of Figure 4-3 below.

```
mininet@mininet-vm: ~/mininet/examples
*** Stopping 1 switches
s1
*** Stopping 3 hosts
server internet h1
*** Done
mininet@mininet-vm:~/mininet/examples$ sudo python ftopology.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
internet h1 server
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> h1 ping server
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=240 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.684 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.063 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2022ms
rtt min/avg/max/mdev = 0.063/80.372/240.369/113.135 ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
mininet@mininet-vm:~/mininet/examples$ clear
mininet@mininet-vm:~/mininet/examples$ sudo python ftopology.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
internet h1 server
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> h1 ping internet
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=355 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.881 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.071 ms
^C
--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.071/118.821/355.513/167.366 ms
mininet> internet ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
```

```
mininet@mininet-vm: ~/mininet/examples
<li><a href="popen.py">popen.py</a>
<li><a href="popenpoll.py">popenpoll.py</a>
<li><a href="README.md">README.md</a>
<li><a href="scratchnet.py">scratchnet.py</a>
<li><a href="scratchnetuser.py">scratchnetuser.py</a>
<li><a href="simpleperf.py">simpleperf.py</a>
<li><a href="sshd.py">sshd.py</a>
<li><a href="test/">test/</a>
<li><a href="topo.mn">topo.mn</a>
<li><a href="topo.py">topo.py</a>
<li><a href="tree1024.py">tree1024.py</a>
<li><a href="treeping64.py">treeping64.py</a>
<li><a href="vlanhost.py">vlanhost.py</a>
</ul>
<hr>
</body>
</html>
100%[=====] 2,298      --.-K/s   in 0s

2017-10-22 12:33:04 (5.30 MB/s) - written to stdout [2298/2298]
mininet> internet ping server -c 1
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
0 packets transmitted, 0 received, 100% packet loss, time 0ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
--- 10.0.0.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.071/118.821/355.513/167.366 ms
mininet> internet ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1001ms
mininet> server ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1002ms
mininet> server ping internet
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1000ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
*** Starting CLI:
mininet> server python -m SimpleHTTPServer 80 &
mininet> internet python -m SimpleHTTPServer 80 &
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> server wget -O - - internet
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 07:37:24-- http://10.0.0.2/
Connecting to 10.0.0.2:80...
^C
mininet> internet wget -O - h1
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 07:39:01-- http://10.0.0.1/
Connecting to 10.0.0.1:80... failed: Connection timed out.
Retrying.
--2017-10-23 07:41:10-- (try: 2) http://10.0.0.1/
Connecting to 10.0.0.1:80... failed: No route to host.
mininet> server ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
^C
--- 10.0.0.1 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2017ms
mininet>
```

```
mininet@mininet-vm: ~/mininet/examples
3 packets transmitted, 3 received, 0% packet loss, time 2022ms
rtt min/avg/max/mdev = 0.063/80.372/240.369/113.135 ms
mininet> internet python -m SimpleHTTPServer 80 &
mininet> server python -m SimpleHTTPServer 80 &
mininet> internet wget -O - server
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-22 12:33:04-- http://10.0.0.3/
Connecting to 10.0.0.3:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2298 (2.2K) [text/html]
Saving to: 'STDOUT'

 0% [ ] 0 --K/s
!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="3leg.mn">3leg.mn</a>
<li><a href="__init__.py">__init__.py</a>
<li><a href="__init__.pyc">__init__.pyc</a>
<li><a href="baresshd.py">baresshd.py</a>
<li><a href="bind.py">bind.py</a>
<li><a href="cluster.py">cluster.py</a>
<li><a href="cluster.pyc">cluster.pyc</a>
<li><a href="clustercli.py">clustercli.py</a>
<li><a href="clustercli.pyc">clustercli.pyc</a>
```

```
mininet@mininet-vm: ~/mininet/examples
mininet@mininet-vm:~/mininet/examples$ sudo python ftopology.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
internet h1 server
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> server python -m SimpleHTTPServer 80 &
mininet> internet python -m SimpleHTTPServer 80 &
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> server wget -O - internet
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 07:37:24-- http://10.0.0.2/
Connecting to 10.0.0.2:80...
^C
mininet> internet wget -O - h1
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 07:39:01-- http://10.0.0.1/
Connecting to 10.0.0.1:80...
```



```
mininet@mininet-vm: ~/mininet/examples
*** Stopping 1 switches
s1
*** Stopping 3 hosts
server internet h1
*** Done
mininet@mininet-vm:~/mininet/examples$ sudo python ftopology.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
internet h1 server
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> server python -m SimpleHTTPServer 80 &
mininet> internet python -m SimpleHTTPServer 80 &
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> server wget -O - internet
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-23 07:37:24-- http://10.0.0.2/
Connecting to 10.0.0.2:80...
```

```
mininet@mininet-vm: ~/mininet/examples
*** Stopping 3 switches
dmz ifire ofire
*** Stopping 3 hosts
internet h1 server
*** Done
mininet@mininet-vm:~/mininet/examples$ sudo python ftopology.py
*** Adding controller
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
internet h1 server
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet> internet python -m SimpleHTTPServer 80 &
mininet> server python -m SimpleHTTPServer 80 &
mininet> server wget -O - h1
Serving HTTP on 0.0.0.0 port 80 ...
--2017-10-24 06:14:04-- http://10.0.0.1/
Connecting to 10.0.0.1:80... failed: Connection refused.
mininet>
```

Figure 4-3 three-part rules tests

## 4.2 Comparison

The Comparison carried out considering different aspects.

i. Performance:

In the three-legged topology, firewall use multiple interfaces to control the access to three- different subnets this allows the internal network host to communicate with other part of the network with a relatively fair speed, throughput and bandwidth.

While the three-part topology provide less performance compared to the three-legged as the packets has to transfer through two stages.

ii. Complexity:

Three-legged controller rule-base that used to implement access restrictions on the firewall is complicated. This increases the likelihood that the controller will be misconfigured, introducing risks into this design.

While three-legged has less complex configuration when considering each firewall on its own. Though as a whole complexity is distributed among the two firewalls.

iii. Reliability:

Three-legged has single firewall, even if redundant in hardware, presents a single point of failure for the design. If it is compromised then the whole network will be down.

As three-part has two firewall, each protecting a specific area of the network, so if one is compromised the other still has a good choice to survive and save the contained data.

iv. Scalability:

Having a programmable network makes it easier for both topologies to expand and scale up without requiring a great effort in changing the configuration and installing the new devices.

v. . QoS:

QoS is fair in both topologies.

vi. Management:

Both topologies are managed well, as the SDN provide a global view of the network on the controller to monitor and supervise the network.

Three-legged may be slightly easier to be managed as it has a single point management.

vii. Cost:

Three-legged has less components (a single firewall) than the three-part that has two firewalls.

This may be a concern in traditional networks, but SDN offers OpenFlow switches with suitable cost for enterprise and others.

The only cost concern is when it comes to afford the SDN controller.



## **Chapter Five**

### **Conclusion**

## **Chapter Five**

### **Conclusion**

5.1 Conclusion

5.2 Recommendations

## **5.1 Conclusion**

As networks become an essential part of our daily life and they are involved in almost every single detail of our lives, security of the network becomes a crucial aspect.

We believe that the network topology plays a major role in the security, performance, reliability, availability, Etc. of the network so implying a robust topology in ourhouse, companies,hospitals, and schools is necessary.

SDN is considered as a great step towards achieving this goal as it offers many features that helps to have sustainable and robust network.

Both three-legged and three- part offers high security to the network to stand against maliciousactions. Each topology has its own operating circumstances.

Depending on the comparison provided in this thesis a one can choose the suitable configuration and layout of their network that meets their design and work goals.

## **5.2 Recommendation**

- We recommend the emphasis on doing further investigation on SDN network topologies to achieve additional design and work goals to match different scenarios and environment starting from simple environments to complex ones.
- Also we suggest that future workers put additional effort on enhancing the security of the SDN network.So the world can get both programmable and secure networks.

## References:

- [1] ONF, “Software-Defined Networking: The New Norm for Networks,” white paper, <https://www.opennetworking.org>.
- [2] (SakirSezer, Sandra Scott-Hayward, P. K. Chouhan, et al July 2013). “Are we ready for SDN?” Implementation challenges for software-defined networks Communications Magazine, IEEE, Vol. 51, No. 7.
- [3] ONF, “Software-Defined Networking: The New Norm for Networks,” white paper, <https://www.opennetworking.org>.
- [4] (Nick Feamster, Jennifer Rexford Ellen Zegura 2013).The Road to SDN: An Intellectual History of Programmable Networks,.
- [5] (N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner 2008), “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Computer Communication Review, vol. 38, no. 2, pp. 69–74,.
- [6] (Sandra Scott-Hayward, Member, IEEE, Sriram Natarajan, and SakirSezer Member 2015), IEEE,A:Survey of Security in Software Defined Networks ,.
- [7] M. Jarschel, T. Zinner, T. Hofeld, P. Tran-Gia, and W. Kellerer, “Interfaces, attributes, and use cases: A compass for sdn,” Communications Magazine, IEEE, vol. 52, no. 6, pp. 210–217, 2014.
- [8] (Scott-Hayward, S., Natarajan, S., &Sezer, S. 2016). A Survey of Security in Software Defined Networks. IEEE Communications Surveys and Tutorials, 18(1), 623-654. DOI: 10.1109/COMST.2015.2453114
- [9] (Nick McKeown, Tom Anderson, HariBalakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner.

2008) Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review;38(2):6974

[10] <http://pakiti.com/datapath-ids/David Bombal in Comware, HP, HP Switch, HP VAN SDN Controller, OpenFlow, ProCurve, SDN.>

## Appendix A

Python code of three-legged topology:

```
#!/usr/bin/python

from mininet.net import Mininet

from mininet.node import Controller, RemoteController, OVSController

from mininet.node import CPULimitedHost, Host, Node

from mininet.node import OVSKernelSwitch, UserSwitch

from mininet.node import IVSSwitch

from mininet.cli import CLI

from mininet.log import setLogLevel, info

from mininet.link import TCLink, Intf

from subprocess import call

def myNetwork():

    net = Mininet( topo=None,

    build=False,

    ipBase='10.0.0.0/8')
```

```

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

    c0=net.addController(name='c0',

controller=RemoteController,

ip='127.0.0.1',

protocol='tcp',

port=6633)

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

    s1 = net.addSwitch('s1', cls=OVSKernelSwitch, dpid='1')

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

server      =      net.addHost('server',      cls=Host,      ip='10.0.0.3',
defaultRoute=None)

internet    =      net.addHost('internet',    cls=Host,    ip='10.0.0.2',
defaultRoute=None)

    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)

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

net.addLink(server, s1)

net.addLink(h1, s1)

net.addLink(internet, s1)

```

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

net.build()

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

for controller in net.controllers:

    controller.start()

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

net.get('s1').start([c0])

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

CLI(net)

net.stop()

if __name__ == '__main__':

    setLogLevel( 'info' )

    myNetwork()
```



## Appendix B

Python code of three-part topology:

```
#!/usr/bin/python
```

```
from mininet.net import Mininet
```

```
from mininet.node import Controller, RemoteController, OVSController
```

```
from mininet.node import CPULimitedHost, Host, Node
```

```
from mininet.node import OVSKernelSwitch, UserSwitch
```

```
from mininet.node import IVSSwitch
```

```
from mininet.cli import CLI
```

```
from mininet.log import setLogLevel, info
```

```
from mininet.link import TCLink, Intf
```

```
from subprocess import call
```

```
def myNetwork():
```

```
    net = Mininet( topo=None,
```

```
                  build=False,
```

```
                  ipBase='10.0.0.0/8')
```

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

```
    c0=net.addController(name='c0',  
  
controller=RemoteController,  
  
ip='127.0.0.1',  
  
protocol='tcp',  
  
port=6633)
```

```
info( '*** Add switches\n' )
```

```
dmz = net.addSwitch('dmz', cls=OVSKernelSwitch, dpid='2')  
  
ifire = net.addSwitch('ifire', cls=OVSKernelSwitch, dpid='1')  
  
ofire = net.addSwitch('ofire', cls=OVSKernelSwitch, dpid='3')
```

```
info( '*** Add hosts\n' )
```

```
internet    =    net.addHost('internet',    cls=Host,    ip='10.0.0.2',  
defaultRoute=None)  
  
    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)  
  
server      =      net.addHost('server',      cls=Host,      ip='10.0.0.3',  
defaultRoute=None)
```

```
info( '*** Add links\n' )
```

```
net.addLink(ifire, dmz)

net.addLink(dmz, ofire)

net.addLink(server, dmz)

net.addLink(h1, ifire)

net.addLink(internet, ofire)

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

net.build()

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

for controller in net.controllers:

    controller.start()

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

net.get('dmz').start([c0])

net.get('ifire').start([c0])

net.get('ofire').start([c0])

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

CLI(net)
```

```
net.stop()
```

```
if __name__ == '__main__':
```

```
    setLogLevel( 'info' )
```

```
    myNetwork()
```

## Appendix C

Python code of the controller rules :

```
#!/usr/bin/python

# Copyright 2012 William Yu

# wyu@ateneo.edu

#

# This file is part of POX.

#

# POX is free software: you can redistribute it and/or modify

# it under the terms of the GNU General Public License as published by

# the Free Software Foundation, either version 3 of the License, or

# (at your option) any later version.

#

# POX is distributed in the hope that it will be useful,

# but WITHOUT ANY WARRANTY; without even the implied

# warranty of

# MERCHANTABILITY or FITNESS FOR A PARTICULAR

# PURPOSE. See the

# GNU General Public License for more details.

#
```

```
# You should have received a copy of the GNU General Public License
# along with POX. If not, see <http://www.gnu.org/licenses/>.
#
# NOISY FIREWALL. This is a simple firewall implementation for
demonstration
# purposes. This firewall is bad because only flows for valid packets are
# installed. Non-matches always trigger a PacketIn().
#
# This is a demonstration file aims to build a firewall. In this demo,
# firewall rules are applied to specific ports in the switch using the
# following commands:
#
#     AddRule (event, dl_type=0x800, nw_proto=1, port=0,
src_port=of.OFPP_ALL)
#
#     DeleteRule (event, dl_type=0x800, nw_proto=1, port=0,
src_port=of.OFPP_ALL):
#
# ShowRule ()
#
# Mininet Command Line: sudomn --topo single,3 --mac --switch ovsk -
-controller remote
#
# Command Line: ./pox.py pylog.level --DEBUG samples.of_firewall
#
```

```
# THIS VERSION SUPPORT resend() functionality in the betta branch  
POX.
```

```
#
```

```
# These next two imports are common POX convention
```

```
frompox.core import core
```

```
frompox.lib.util import dpidToStr
```

```
import pox.openflow.libopenflow_01 as of
```

```
frompox.lib.packet.ethernet import ethernet
```

```
# Even a simple usage of the logger is much nicer than print!
```

```
log = core.getLogger()
```

```
# This table maps (switch,MAC-addr) pairs to the port on 'switch' at
```

```
# which we last saw a packet *from* 'MAC-addr'.
```

```
# (In this case, we use a Connection object for the switch.)
```

```
table = { }
```

```
# This table contains the firewall rules:
```

```
# firewall[(switch, dl_type, nw_proto, port, src_port)] = TRUE/FALSE
```

```
#
```

```

# Our firewall only supports inbound rule enforcement per port only.

# By default, this is empty.

# Sample dl_type(s): IP (0x800)

# Sample nw_proto(s): ICMP (1), TCP (6), UDP (17)

#

firewall = {}

# function that allows adding firewall rules into the firewall table

defAddRule          (event,          dl_type=0x800,
nw_proto=1,srcip="0.0.0.0",dstip="0.0.0.0"):

firewall[(event.connection,dl_type,nw_proto,IPAddr(srcip),IPAddr(dstip
))]=True

# function that allows deleting firewall rules from the firewall table

defDeleteRule  (event,  dl_type=0x800,  nw_proto=1,  port=0,
src_port=of.OFPP_ALL):

try:

del firewall[(event.connection,dl_type,nw_proto,port,src_port)]

log.debug("Deleting firewall rule in %s: %s %s %s %s" %

dpidToStr(event.connection.dpid), dl_type, nw_proto, port, src_port)

exceptKeyError:

```



```

log.error("Cannot find in %s: %s %s %s %s" %
dpidToStr(event.connection.dpid), dl_type, nw_proto, port, src_port)

# function to display firewall rules

def ShowRules ():

for key in firewall:

log.info("Rule %s defined" % str(key))

# function to handle all housekeeping items when firewall starts

def _handle_StartFirewall (event):

ifevent.connection.dpid==1:

AddRule (event,0x800, 1,"10.0.0.1","10.0.0.3")

AddRule (event,0x800, 1,"10.0.0.1","10.0.0.2")

AddRule (event,0x800, 80,"10.0.0.1","10.0.0.3")

AddRule (event,0x800, 6,"10.0.0.1","10.0.0.3")

AddRule (event,0x800, 6,"10.0.0.3","10.0.0.1")

AddRule (event,0x800, 80,"10.0.0.1","10.0.0.2")

AddRule (event,0x800, 6,"10.0.0.1","10.0.0.2")

AddRule (event,0x800, 6,"10.0.0.2","10.0.0.3")

AddRule (event,0x800, 80,"10.0.0.2","10.0.0.3")

```

```

if event.connection.dpid==3:

    AddRule (event,0x800, 80,"10.0.0.2","10.0.0.3")

    AddRule (event,0x800, 6,"10.0.0.2","10.0.0.3")

    AddRule (event,0x800, 6,"10.0.0.1","10.0.0.2")

    AddRule (event,0x800, 1,"10.0.0.1","10.0.0.2")

ShowRules()

core.openflow.addListenerByName("PacketIn", _handle_ifire_PacketIn)

#core.openflow.addListenerByName("PacketIn", _handle_PacketIn)

# function to handle all PacketIns from switch/router

from pox.lib.addresses import IPAddr

def _handle_ifire_PacketIn (event):

    #if packet.type == packet.ARP_TYPE:

        # arpp = packet.find('arp')

        #if arpp.opcode == arpp.REPLY:

if event.connection.dpid==1:

    _handle_innerfirewall(event)

```

```
if event.connection.dpid==2:
```

```
    _handle_switch(event)
```

```
if event.connection.dpid==3:
```

```
    _handle_outterfirewall(event)
```

```
def resend_packet(event, dst_port = of.OFPP_ALL):
```

```
    msg = of.ofp_packet_out(data = event.ofp)
```

```
    msg.actions.append(of.ofp_action_output(port = dst_port))
```

```
    event.connection.send(msg)
```

```
def drop (event,duration = None):
```

```
    """
```

```
        Drops this packet and optionally installs a flow to continue
```

```
dropping similar ones for a while
```

```
    """
```

```
    packet=event.parsed
```

```
    if duration is not None:
```

```
        if not isinstance(duration, tuple):
```

```

duration = (duration,duration)

msg = of.ofp_flow_mod()

msg.match = of.ofp_match.from_packet(packet)

msg.idle_timeout = duration[0]

msg.hard_timeout = duration[1]

msg.buffer_id = event.ofp.buffer_id

event.connection.send(msg)

elif event.ofp.buffer_id is not None:

msg = of.ofp_packet_out()

msg.buffer_id = event.ofp.buffer_id

msg.in_port = event.port

event.connection.send(msg)

def _handle_outterfirewall(event):

packet = event.parsed

# only process Ethernet packets

if packet.type == packet.ARP_TYPE:

```

```

print("arp")

arpp = packet.find('arp')

    #if IPAddr(arpp.protodst)==IPAddr('10.0.0.1')and arpp.opcode ==
arpp.REQUEST:

    # drop(event)

    # return

# only process Ethernet packets

elifpacket.type == ethernet.IP_TYPE:

print("hi3")

if          (firewall[(event.connection,          packet.type,
packet.payload.protocol,packet.next.srcip,packet.next.dstip)] == True):

log.debug("Rule (%s %s %s )" %

        (str(event.connection.dpid),          str(packet.type),
str(packet.payload.protocol)))

else:

log.debug("Rule (%s %s %s )" %

        (str(event.connection),          str(packet.type),
str(packet.payload.protocol)))

return

else:

print("return")

```

```
return
```

```
# Learn the source and fill up routing table
```

```
table[(event.connection,packet.src)] = event.port
```

```
dst_port = table.get((event.connection,packet.dst))
```

```
ifdst_port == event.port: # 5
```

```
    # 5a
```

```
log.warning("Same port for packet from %s -> %s on %s.%s. Drop.")
```

```
    % (packet.src, packet.dst, dpid_to_str(event.dpid), port))
```

```
drop(event,10)
```

```
return
```

```
ifdst_port is None:
```

```
    # We don't know where the destination is yet. So, we'll just
```

```

    # send the packet out all ports (except the one it came in on!)

resend_packet(event)

log.debug("Broadcasting %s.%i -> %s.%i" %
         (packet.src, event.ofp.in_port, packet.dst, of.OFPP_ALL))

else:

    # Since we know the switch ports for both the source and dest
    # MACs, we can install rules for both directions.

msg = of.ofp_flow_mod()

msg.idle_timeout = 2

msg.hard_timeout = 0

    #msg.buffer_id = None

msg.match.dl_dst = packet.src

msg.match.dl_src = packet.dst

msg.match.dl_type = packet.type

msg.actions.append(of.ofp_action_output(port = event.port))

event.connection.send(msg)

    # This is the packet that just came in -- we want to
    # install the rule and also resend the packet.

```

```

msg = of.ofp_flow_mod()

msg.idle_timeout = 2

msg.hard_timeout = 0

    #msg.buffer_id = None

msg.data = event.ofp # Forward the incoming packet

msg.match.dl_src = packet.src

msg.match.dl_dst = packet.dst

msg.match.dl_type = packet.type

if packet.type == ethernet.IP_TYPE:

    msg.match.nw_proto = packet.payload.protocol

msg.actions.append(of.ofp_action_output(port = dst_port))

event.connection.send(msg)

log.debug("Installing %s.%i -> %s.%i AND %s.%i -> %s.%i" %

    (packet.dst, dst_port, packet.src, event.ofp.in_port,

packet.src, event.ofp.in_port, packet.dst, dst_port))

defsend_packet(event, dst_port = of.OFPP_FLOOD):

```



```

msg = of.ofp_packet_out(data = event.ofp)

msg.actions.append(of.ofp_action_output(port = all_ports))

event.connection.send(msg)

def _handle_innerfirewall(event):

    packet = event.parsed

    if packet.type == packet.ARP_TYPE:

        print("arp")

        # only process Ethernet packets

    elif packet.type == ethernet.IP_TYPE:

        print("hi1")

        if (firewall[(event.connection, packet.type,
            packet.payload.protocol, packet.next.srcip, packet.next.dstip)] == True):

            log.debug("Rule (%s %s %s)" %
                (str(event.connection.dpid), str(packet.type),
                str(packet.payload.protocol)))

        else:

            log.debug("Rule (%s %s %s)" %
                (str(event.connection), str(packet.type),
                str(packet.payload.protocol)))

    return

```

else:

print("return")

return

# Learn the source and fill up routing table

table[(event.connection,packet.src)] = event.port

dst\_port = table.get((event.connection,packet.dst))

ifdst\_port == event.port: # 5

# 5a

log.warning("Same port for packet from %s -> %s on %s.%s. Drop."

% (packet.src, packet.dst, dpid\_to\_str(event.dpid), port))

drop(event,10)

return

ifdst\_port is None:

# We don't know where the destination is yet. So, we'll just

# send the packet out all ports (except the one it came in on!)

```

resend_packet(event)

log.debug("Broadcasting %s.%i -> %s.%i" %
         (packet.src, event.ofp.in_port, packet.dst, of.OFPP_ALL))
else:
    # Since we know the switch ports for both the source and dest
    # MACs, we can install rules for both directions.

msg = of.ofp_flow_mod()

msg.idle_timeout = 2

msg.hard_timeout = 0

    #msg.buffer_id = None

msg.match.dl_dst = packet.src

msg.match.dl_src = packet.dst

msg.match.dl_type = packet.type

msg.actions.append(of.ofp_action_output(port = event.port))

event.connection.send(msg)

    # This is the packet that just came in -- we want to
    # install the rule and also resend the packet.

msg = of.ofp_flow_mod()

```

```

msg.idle_timeout = 2

msg.hard_timeout = 0

    #msg.buffer_id = None

msg.data = event.ofp # Forward the incoming packet

msg.match.dl_src = packet.src

msg.match.dl_dst = packet.dst

msg.match.dl_type = packet.type

if packet.type == ethernet.IP_TYPE:

    msg.match.nw_proto = packet.payload.protocol

msg.actions.append(of.ofp_action_output(port = dst_port))

event.connection.send(msg)

log.debug("Installing %s.%i -> %s.%i AND %s.%i -> %s.%i" %

    (packet.dst, dst_port, packet.src, event.ofp.in_port,

packet.src, event.ofp.in_port, packet.dst, dst_port))

def _handle_switch(event):

    packet = event.parsed

```

```

ifpacket.type == packet.ARP_TYPE:

print("arp")

    # only process Ethernet packets

elifpacket.type == ethernet.IP_TYPE:

print("hi2")

else:

print("return")

return

    # Learn the source and fill up routing table

table[(event.connection,packet.src)] = event.port

dst_port = table.get((event.connection,packet.dst))

ifdst_port == event.port: # 5

    # 5a

log.warning("Same port for packet from %s -> %s on %s.%s. Drop.")

```

```

        % (packet.src, packet.dst, dpid_to_str(event.dpid), port))

drop(event,10)

return

ifdst_port is None:

    # We don't know where the destination is yet. So, we'll just

    # send the packet out all ports (except the one it came in on!)

    resend_packet(event)

    log.debug("Broadcasting %s.%i -> %s.%i" %

        (packet.src, event.ofp.in_port, packet.dst, of.OFPP_ALL))

else:

    # Since we know the switch ports for both the source and dest

    # MACs, we can install rules for both directions.

    msg = of.ofp_flow_mod()

    msg.idle_timeout = 3

    msg.hard_timeout = 5

    #msg.buffer_id = None

    msg.match.dl_dst = packet.src

    msg.match.dl_src = packet.dst

    msg.match.dl_type = packet.type

```

```

msg.actions.append(of.ofp_action_output(port = event.port))

event.connection.send(msg)

# This is the packet that just came in -- we want to
# install the rule and also resend the packet.

msg = of.ofp_flow_mod()

msg.idle_timeout = 3

msg.hard_timeout = 5

#msg.buffer_id = None

msg.data = event.ofp # Forward the incoming packet

msg.match.dl_src = packet.src

msg.match.dl_dst = packet.dst

msg.actions.append(of.ofp_action_output(port = dst_port))

event.connection.send(msg)

log.debug("Installing %s.%i -> %s.%i AND %s.%i -> %s.%i" %

        (packet.dst, dst_port, packet.src, event.ofp.in_port,

        packet.src, event.ofp.in_port, packet.dst, dst_port))

def _handle_PacketIn (event):

```

```

packet = event.parsed

# only process Ethernet packets

if packet.type != ethernet.IP_TYPE:

return

# check if packet is compliant to rules before proceeding

if (firewall[(event.connection, packet.dl_type, packet.nw_proto,
packet.tp_src, event.port)] == True):

log.debug("Rule (%s %s %s %s) FOUND in %s" %

dpidToStr(event.connection.dpid), packet.dl_type, packet.nw_proto,
packet.tp_src, event.port)

else:

log.debug("Rule (%s %s %s %s) NOT FOUND in %s" %

dpidToStr(event.connection.dpid), packet.dl_type, packet.nw_proto,
packet.tp_src, event.port)

return

# Learn the source and fill up routing table

table[(event.connection,packet.src)] = event.port

```



```
dst_port = table.get((event.connection,packet.dst))
```

```
ifdst_port is None:
```

```
    # We don't know where the destination is yet. So, we'll just
```

```
    # send the packet out all ports (except the one it came in on!)
```

```
msg = of.ofp_packet_out(resend = event.ofp)
```

```
msg.actions.append(of.ofp_action_output(port = of.OFPP_ALL))
```

```
msg.send(event.connection)
```

```
log.debug("Broadcasting %s.%i -> %s.%i" %
```

```
    (packet.src, event.ofp.in_port, packet.dst, of.OFPP_ALL))
```

```
else:
```

```
    # Since we know the switch ports for both the source and dest
```

```
    # MACs, we can install rules for both directions.
```

```
msg = of.ofp_flow_mod()
```

```
msg.match.dl_type = packet.dl_type
```

```
msg.match.nw_proto = packet.nw_proto
```

```
if (nw_proto != 1):
```

```
msg.match.tp_src = packet.tp_src
```

```
msg.match.dl_dst = packet.src
```

```
msg.match.dl_src = packet.dst

msg.idle_timeout = 10

msg.hard_timeout = 30

msg.actions.append(of.ofp_action_output(port = event.port))

msg.send(event.connection)
```

```
# This is the packet that just came in -- we want to
```

```
# install the rule and also resend the packet.
```

```
msg = of.ofp_flow_mod()

msg.match.dl_type = packet.dl_type

msg.match.nw_proto = packet.nw_proto

if (nw_proto != 1):

    msg.match.tp_src = packet.tp_src

    msg.match.dl_src = packet.src

    msg.match.dl_dst = packet.dst

    msg.idle_timeout = 10

    msg.hard_timeout = 30

    msg.actions.append(of.ofp_action_output(port = dst_port))

    msg.send(event.connection, resend = event.ofp)
```

```
log.debug("Installing %s.%i -> %s.%i AND %s.%i -> %s.%i" %  
  
        (packet.dst, dst_port, packet.src, event.ofp.in_port,  
packet.src, event.ofp.in_port, packet.dst, dst_port))  
  
# main function to start module  
  
def launch ():  
  
    #from proto.arp_responder import launch as arp_launch  
  
    #arp_launch(eat_packets=False,**{str(ip):True})  
  
    core.openflow.addListenerByName("ConnectionUp",  
_handle_StartFirewall)
```