SIMULATING AND PROTOTYPING SOFTWARE DEFINED NETWORKING (SDN) USING MININET APPROACH TO OPTIMISE HOST COMMUNICATION IN REALISTIC PROGRAMMABLE NETWORKING ENVIRONMENT.

by

LINDINKOSI LETHUKUTHULA ZULU
61366935

submitted in accordance with the requirements
for the degree of

MAGISTER TECHNOLOGIAE: ELECTRICAL ENGINEERING (98988)

FIELD OF SPECIALIZATION: COMPUTER NETWORKS MODELLING, TELECOMMUNICATIONS

in the

Department of Electrical and Mining Engineering
COLLEGE OF SCIENCE ENGINEERING AND TECHNOLOGY
UNIVERSITY OF SOUTH AFRICA (UNISA)

SUPERVISOR: DR KINGSLEY OGUDO

CO-SUPERVISOR: DR PATRICE UMENNE

NOVEMBER 2018

Open Rubric

DECLARATION

I declare that **SIMULATING AND PROTOTYPING SOFTWARE DEFINED NETWORKING (SDN) USING MININET APPROACH TO OPTIMISE HOST COMMUNICATION IN REALISTIC PROGRAMMABLE NETWORKING ENVIRONMENT** is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

I further declare that I submitted the thesis/dissertation to originality checking software and that it falls within the accepted requirements for originality.

I further declare that I have not previously submitted this work, or part of it, for examination at Unisa for another qualification or at any other higher education institution.

[illegible]
SIGNATURE

November 8, 2018
DATE

© University of South Africa November 2018

COPYRIGHT CLASSIFICATION

© Copyright resides in the University of South Africa (UNISA) and Lindinkosi Zulu. In terms of the Copyright Act 98 of 1978, no part of this material may be reproduced, stored in any retrieval system, be transmitted in any form or be published, redistributed or screened by any means (electronic, mechanical, photocopying, recording or otherwise) without prior written permission from the University of South Africa. However, permission to use any material in this work that is derived from other sources must be obtained from the original source.

© University of South Africa 2018

DEDICATION

I dedicate this dissertation to my son Nkazimulo Zulu whose birth gave me the courage to pursue these studies. This dissertation is also dedicated to my wife Thabisile Zulu who always believe in me even when I doubted myself.

© University of South Africa November 2018

ACKNOWLEDGEMENTS

Praises to God Almighty who has made it possible for me to reach this point in my life and gave me strengths to press on even on difficult times during my studies.
I wish to express my deep gratitude to my wife Thabisile for her constant support and believing in me. I would have not been able to complete these studies if it wasn't for her time that was sacrificed when I was doing these studies.

Specifically, I would like to acknowledge the following people.

1. To my supervisors, Dr Kingsley Ogudo and Dr Patrice Umenne, thank you for your time, wisdom, support and guidance. I'm also humble by your support and involvement and the roles you played on the two conference papers that emanated from these studies. I will always be grateful to your contribution and support.

2. To the University of South Africa, thank you for giving me an opportunity to pursue these studies and the financial assistance with the conference papers presentations.

3. To Ian Kruger, thank you for believing in my dream and providing me with access to resources and tools used for these studies more especially EVE-NG.

4. I also like to express my deep gratitude to everyone who has contributed in various ways to the successful completion of these studies. From my friends, colleagues at NEC XON and my family at large, Thank you so much.

© University of South Africa November 2018

ABSTRACT

In this project, two tests were performed. On the first test, Mininet-WiFi was used to simulate a Software Defined Network to demonstrate Mininet-WiFi' s ability to be used as the Software Defined Network emulator which can also be integrated to the existing network using a Network Virtualized Function (NVF). A typical organization's computer network was simulated which consisted of a website hosted on the LAMP (Linux, Apache, MySQL, PHP) virtual machine, and an F5 application delivery controller (ADC) which provided load balancing of requests sent to the web applications. A website page request was sent from the virtual stations inside Mininet-WiFi. The request was received by the application delivery controller, which then used round robin technique to send the request to one of the web servers on the LAMP virtual machine. The web server then returned the requested website to the requesting virtual stations using the simulated virtual network. The significance of these results is that it presents Mininet-WiFi as an emulator, which can be integrated into a real programmable networking environment offering a portable, cost effective and easily deployable testing network, which can be run on a single computer. These results are also beneficial to modern network deployments as the live network devices can also communicate with the testing environment for the data center, cloud and mobile provides.

On the second test, a Software Defined Network was created in Mininet using python script. An external interface was added to enable communication with the network outside of Mininet. The amazon web services elastic computing cloud was used to host an OpenDaylight controller. This controller is used as a control plane device for the virtual switch within Mininet. In order to test the network, a webserver hosted on the Emulated Virtual Environment – Next Generation (EVE-NG) software is connected to Mininet. EVE-NG is the Emulated Virtual Environment for networking. It provides tools to be able to model virtual devices and interconnect them with other virtual or physical devices. The OpenDaylight controller was able to create the flows to facilitate communication between the hosts in Mininet and the webserver in the real-life network.

© University of South Africa November 2018

KEY TERMS

Software Defined Networking (SDN); Network Functions Virtualization (NFV); OpenDaylight Controller; Mininet; Linux Web Server; Mininet Wi-Fi; Application Delivery Controller (F5); Cloud Computing (Amazon Web Services); Python Script; Emulated Virtual Environment – Next Generation (EVE-NG)

© University of South Africa November 2018

LIST OF PUBLICATIONS

1. L. Zulu, K. Ogudo and P. Umenne, "Simulating Software Defined Networking Using Mininet to Optimize Host Communication in a Realistic Programmable Network". Proceedings of the International Conference on Advances in Big Data, Computing and Data Communication Systems, 6-7th August 2018, Durban, South Africa, ISBN 978-1-5386-3059-4, IEEE Mauritius sub-section.

2. L. Zulu, K. Ogudo and P. Umenne, "Emulating Software Defined Network Using Mininet and OpenDaylight Controller Hosted on Amazon Web Services Cloud Platform to Demonstrate a Realistic Programmable Network". Accepted for publication by the International Conference on Intelligent and Innovative Computing Applications (ICONIC 2018), Plaine Magnien, IEEE Mauritius sub-section, December 6-7. 2018

© University of South Africa November 2018

TABLE OF CONTENTS

© University of South Africa November 2018

© University of South Africa November 2018

TABLE OF FIGURES

© University of South Africa November 2018

LIST OF ABBREVIATIONS

ADC:	Application Delivery Controller
AP:	Access Point
API:	Application Programming Interface
AWS:	Amazon Web Services
BSS:	Business Support Systems
CAPEX:	Capital Expenditure
CLI:	Command Line Interface
DoS:	Denial of Service
ETSI:	European Telecommunications Standards Institute
EVE-NG:	Emulated Virtual Environment – Next Generation
Fs-SDN	Flow simulator- Software Defined Networking
GUI:	Graphical User Interface
HA:	High Availability
HTTP:	Hypertext Transfer Protocol
IEEE:	Institute of Electrical and Electronics Engineers
IETF:	Internet Engineering Task Force
IP:	Internet Protocol
LAMP:	Linux Apache MySQL PHP
LTM:	Local Traffic Manager
MAC:	Media Access Control
MD-SAL:	Model-driven Service Abstraction Layer
NaaS:	Network as a Service
NAT:	Network Address Translation
NETCONF:	Network Configuration Protocol
NFV:	Network Functions Virtualization
NMS:	Network Management System

© University of South Africa November 2018

NOS:	Network Operating System
ODL:	OpenDaylight
ONF:	Open Networking Foundation
OPEX:	Operating Expenditure
OSGI:	Open Services Gateway initiative
OSS:	Operations Support Systems
OVS:	Open Virtual Switch
OVSDB:	Open Virtual Switch Database
PoC:	Proof of concept
QoS:	Quality of Service
REST API:	Representational State Transfer Application Programming Interface
SAL:	Service Abstraction Layer
SDN:	Software Defined Networking
TCP:	Transmission Control Protocol
TCP/IP:	Transmission Control Protocol/Internet Protocol
UDP:	User Datagram Protocol
VE:	Virtual Edition
VLAN:	Virtual Local Area Network
VM:	Virtual Machine
VTN:	Virtual Tenant Network
WiFI:	Wireless Fidelity
YANG:	Yet Another Next Generation

© University of South Africa November 2018

CHAPTER 1

1.1 Introduction

Moving to Software Defined-based networking is not without its challenges. Converting from a proprietary to an open system involves more moving parts, including controllers, clients, orchestration systems, and business applications. In many cases, Software Defined Networking (SDN) components must interact with legacy components introducing further complexity. This has created a room for testbeds, which can be used to test the Software Defined Network before being implemented in real network to minimize down time and to provide a test environment, which is close to real world where issues can be identified and rectified. A suitable tool or emulator, which must produce close to real live situation, must be used to obtain outcomes, which can be implemented as is in real world.

This study is based on simulating a Software Defined Network using Mininet, which communicates with the virtualized network. The virtual network is hosted on Emulated Virtual Environment – Next Generation (EVE-NG) which is a software that provides tools to model real life virtual devices and interconnect them with other virtual or physical devices. OpenDaylight controller was used as the control plane device. This controller was then hosted on Amazon Web Services (AWS), which is a networking cloud platform.

© University of South Africa November 2018

1.2 Problem statement

1.2.1 Knowledge gap

PROBLEM STATEMENT 1

Software Defined Networking is currently being actively researched in communications networks today and Mininet has emerged as the most preferred tool to emulate SDN networks. However, most of these activities and studies focuses on interpretation, standardization and architecture of SDN and these emulated networks are only confined to the hypervisor, such as Mininet but they do not communicate with the outside network. **So the first problem statement is to design a Software Defined network (SDN) that can communicate with an outside network.**

PROBLEM STATEMENT 2

The second problem statement is to design a Controller that can facilitate the Flow of communication between the SDN network and the outside network.

PROBLEM STATEMENT 3

For any organization with an existing network or which is looking into incremental deployment of SDN, it is important for it that when changes are to be implemented on the live network, proper care and measures are applied in eliminating or minimizing down time. To achieve this, a simulated network must be able to communicate with existing network so that real world issues can be identified and rectified in the simulated environment. Then minimal change will be required when moving from simulation to live deployment of the simulated solution. **Problem statement 3 is to be able to use a software defined network to identify changes or real life issues in an outside real life network.**

1.3 Study Framework

1.3.1 Research objectives and aims

The specific objectives of this project are-:

1. To explore Software Defined Networking and get deep understanding of it as a networking architecture through literature reviews
2. To validate Mininet and it wireless extension called Mininet-WiFI as a suitable emulator for Software Defined Networks.

© University of South Africa November 2018

3. To implement a simple Software Defined Network which will be simulated using Mininet and communicate with a virtualized network hosted on Emulated Virtual Environment – Next Generation (EVE-NG). (EVE-NG) is a software that provides tools to model virtual devices and interconnect them with other virtual or physical devices via an OpenDaylight controller hosted on Amazon Web Services (AWS) cloud platform to demonstrate a real-world network scenario.
4. To Formulate and compile a simplified paper which will summarize the research findings.
5. Simulate an SDN network using Python script which can be used for future studies on the integration of Mininet emulated SDN network with real-world network.

1.3.2 Research/Core Questions

This project is based on the **hypothesis, which says Mininet is the suitable emulator for Software defined network (SDN) and addresses the following question -:**

- Are Software Defined Networks (SDN) emulated in Mininet able to communicate with real world network using current and future technologies like network virtualization function (NVF) and cloud networking?

1.3.3 Benefit of the study

This project aims to contribute in the body of knowledge in the field of Software Defined networks simulations, network virtualization and cloud networking. The findings and the knowledge gained from this project is beneficial to organizations and individuals planning on introducing SDN on their networks. This project is also of great benefit in integrating emulated networks with real-world networks to have a feel of the effect the emulated network will have on live network once deployed.

© University of South Africa November 2018

1.3.4 Delimitation of the study

1. Software defined networking is an architecture which has not yet reached it full maturity, development process and more research is still being done on this subject. This study will be limited to current available data on SDN.

2. Although the number is growing but there are still very few companies which have already deployed SDN in their network therefore it is expected that bulk of the information and data which will be collected for this study will be of theoretical nature.

3. This study will be limited to SDN architecture definition and explanation and will not go deep into each components' standards, detailed functionalities and deployment.

4. There have since been many definitions of SDN, this project is based on the original definition where SDN is defined as the physical separation of the network control plane from the forwarding plane, where a control plane can control several devices

5. The network which will be developed and tested will be a simple network with core network components to perform basic network functions and demonstrate key points of the project.

© University of South Africa November 2018

1.4 Literature Review

1.4.1 Software Defined Networking

1.4.1.1 Background

Traditional Internet Protocol (IP) networks are currently very decentralized. Each network device has its own control plane, management plane and forwarding plane as seen in Figure 1. These networks are complex, and it requires manual configuration of each device on the network if there are changes to be implemented. The hardware and the software of the traditional networking architecture are proprietary and specifically designed to work together. To configure the devices, vendor-specific commands must be used. This current setup makes it difficult for the network to be flexible and scalable to meet the high demand of modern applications and requirements.

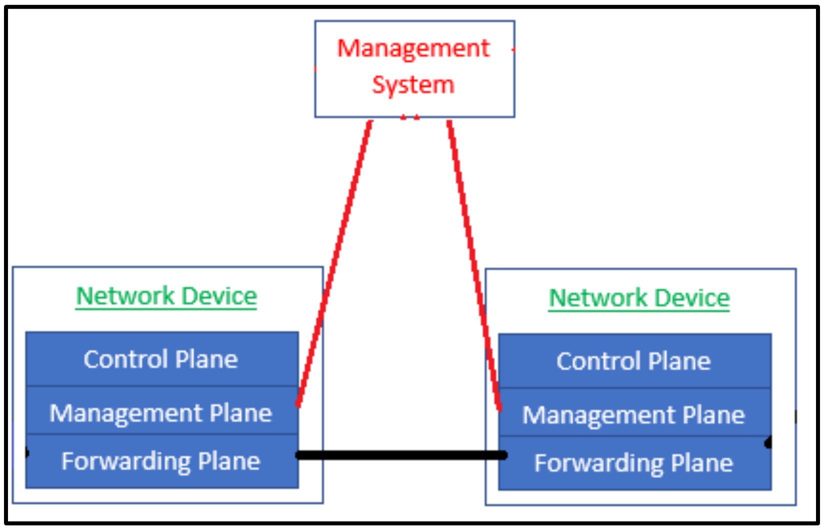


Figure 1. Traditional networking architecture

Software Defined networking (SDN) was developed to address these challenges the current network model is failing to address. It does so by separating the network's control logic from routers and switches that forward the traffic. It also separates the control and data planes leaving network switches to become simple forwarding devices and the control logic is implemented in a logically centralized controller [1] located on a cloud network.

© University of South Africa November 2018

Open Networking Foundation (ONF) defines Software Defined Networking as the physical separation of the network control plane from the forwarding plane, whereby the control plane controls several devices externally [2]. It is an architecture that decouples the network control and forwarding functions. This creates a three-layer architecture as seen in Figure 2, which are infrastructure, control and application layers [2]. This allows the network to be dynamic, adaptable, cost-effective, software programmable and easily manageable [3]. OpenFlow was developed as the first standard communications interface defined between the control and forwarding planes of an SDN architecture [4].

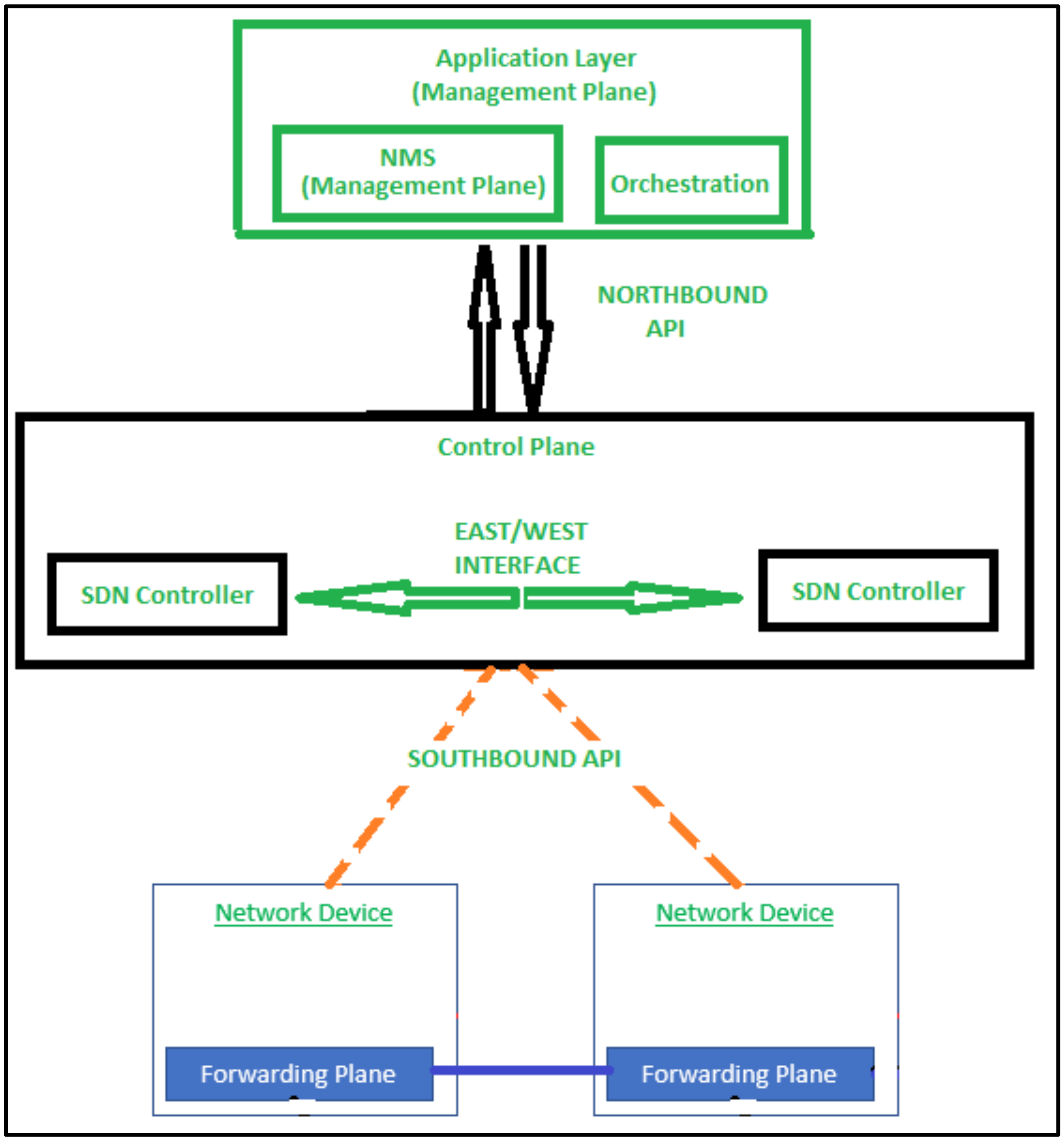


Figure 2. Software Defined Networking architecture

© University of South Africa November 2018

1.4.1.2 Need for Software Defined Networking

Software Defined Networking makes it easier for network operators to evolve network capabilities. A single software program can control the behavior of the entire network [5]. This intelligence makes it possible to offer Networking-as-a-Service (NaaS). This significantly reduces expenses both Capital Expenditures (CAPEX) and Operational Expenses (OPEX) and enables fast service architecture. This is because the data plane is highly programmable from the remote-control plane at the controlling application [6]. Software Defined Networking also offers enhanced configuration, improved performance and encouraged innovation [7].

The high demand for data has affected the telecommunication industry, more specially the mobile network providers. This hunger for data is one of the catalysts for connectivity speeds of 5G networks. Service providers are facing challenges in complying with connectivity demands without substantial financial investments [8]. To address this issue, the industry had to look for initiatives aiming at cost reduction, increase of network scalability and service flexibility. The two networking architectures introduced to meet these requirements are Network Functions Virtualization and Software Defined Networking [9].

1.4.1.3 Software Defined Networking Architecture

The control plane is the centrally located control unit. It is called Software Defined Networking controller. It acts as the Network Operating Systems (NOS). The data plane resides inside the network core devices and is only responsible for forwarding data packets controlled by the central Software Defined Networking controller. These separated planes use protocols and an Application Programmable Interface (API) to communicate [10].

OpenFlow is one of the protocols used by Software defined networks and was started by Stanford University in 2008 [11]. Different companies came together in 2011 and formed Open Networking Foundation (ONF) to further develop OpenFlow and Software Defined Networking [12]. With the separation of the control and data planes, the data plane only performs the data packet forwarding action and it resides in the network device. The control plane is logically positioned on top of the

© University of South Africa November 2018

data plane and acts as the brains of the network [13].

Software Defined networks makes it possible to consolidate in one place complex software used to configure and control several devices making the process less expensive. A centralized controller gives a benefit of having a view of the network, which then enables it to make decisions on how data planes must move the traffic [14]. SDN makes it possible to dynamically provision the network. It improves network resources utilization and simplifies traffic engineering [15]. It makes it possible to use external applications to program the network. Communication between the devices in SDN uses open interfaces making it to be vendor neutral [16]. To test Software Defined Networks, an emulator called Mininet is amongst the popularly used tools [17].

1.4.1.4 Application Plane

Application Plane is logically on top of the Software Defined Networking architecture. It consists of applications and services that make requests for network functions from the Control Plane and the Data plane [18]. This can be any third-party application. Application layer through Software Defined Networking openness provides application developers with easy development and deployment of network applications. Application layer communicates with the controller layer using a Northbound Application Programming Interface (API).

1.4.1.5 Control Plane

Control Plane consists of control applications or programs. It operates on view of network and performs different functions like routing, traffic engineering, quality of service (QoS), security. Control plane has a global view of the network [19]. It performs configuration of each network device. Network Operating System (NOS) resides on the control plane. It is a distributed system that creates a consistent, updated network view and is executed as controllers in the network. The controller is responsible for making decisions on how packets should be forwarded. It does this by pushing flows instructions down to the network devices for execution. The control plane populates the forwarding tables that reside in the forwarding plane with flows based on the network topology

© University of South Africa November 2018

or external service requests.

1.4.1.6 Data Plane

Data Plane is responsible for handling data packets based on the instructions received from the controller. It is also known as forwarding plane. Forwarding decisions are flow-based. A flow in Software Defined Networking is a sequence of packets between a source and a destination. Flow decisions from the controller informs the data plane devices on how to process the packets [20]. Example of flow actions are forwarding, dropping and changing of packets. Data plane device are network devices like switches and routers.

1.4.2 Network Function Virtualization

Although Software Defined Networking is a complete standalone networking architecture, when used with Network Functions Virtualization, it provides end-to-end network automation. Software Defined Networking is classically defined as the separation of the control plane from the forwarding plane where the control plane is centralized while Network Functions Virtualization is the virtualization of services instead of using the hardware purposefully built to provide that service in a real-life network.

Network Functions Virtualization is a framework defined by the European Telecommunications Standards Institute (ETSI) that specifies the virtualization of various network services such as firewalls, load balancers and any other services typically associated with dedicated purpose-built hardware. In the telecommunication industry, NFV proposes to run the mobile network functions as software instances on commodity servers or datacenters, while SDN supports a decomposition of the mobile network into control-plane and data-plane functions. The combination of both SDN and NFV is considered as a very promising combination in achieving a cost-efficient mobile network architecture within the mobile network environment [21].
NFV has been proposed as a model that resolves the functions of placement and aims at minimizing

© University of South Africa November 2018

the transport network load overhead against several parameters such as data-plane delay, number of potential data centers and SDN control overhead. By moving network appliance functionality from proprietary hardware to software, Network Function Virtualization promises to bring the advantages of cloud computing to network packet processing [22]. It is for this reason that this paper looks to Mininet and its wireless extension Mininet-WiFi as the emulator, which can be used to emulate a Software Defined Network, intergraded on the network using a Network Virtualized Function (NVF) in the form of an application delivery (load balancer).

1.4.3 Mininet

1.4.3.1 Background

Recent Software-Defined Networking (SDN) approaches propose new means for network virtualization and programmability advancing the way networks can be designed and operated, including user-defined features and customized behavior at run-time [23]. The need for fault tolerance and scalability is leading to the development of distributed Software Defined Networking operating systems and applications. These developments and innovations call for an emulator, which will be able to produce reliable results when emulating such networks [24]. Mininet provides the platform to understand how actual Software Defined Networking works by creating a virtual network similar to the real network. This can be applied to run on small as well as very large-scale networks. One of the advantages of using Mininet is that, an application that works on it can be easily deployed to or integrated with a real network [25].

© University of South Africa November 2018

1.4.3.2 What is Mininet

Mininet is the container-based emulator [26]. It allows the running of unmodified code interactively on virtual hardware on a regular computer. It provides convenience and realism at low cost compared to running on a hardware. Programs run on emulators require none or minimum modification when applied to real live networks [27]. Mininet runs unmodified code of network applications in lightweight Linux containers to achieve its scalability and accuracy. The greatest value of Mininet is supporting collaborative network research by enabling self-contained Software Defined Network prototypes, which anyone with a personal computer (PC) or laptop can download, and use [28] as seen in Figure 3. To achieve this, Mininet uses lightweight approach of OS-level virtualization features ranging from processes and network namespaces, which make it possible to scale to hundreds of nodes and represents a qualitative change in workflow through its ability to run and debug in real-time [29].

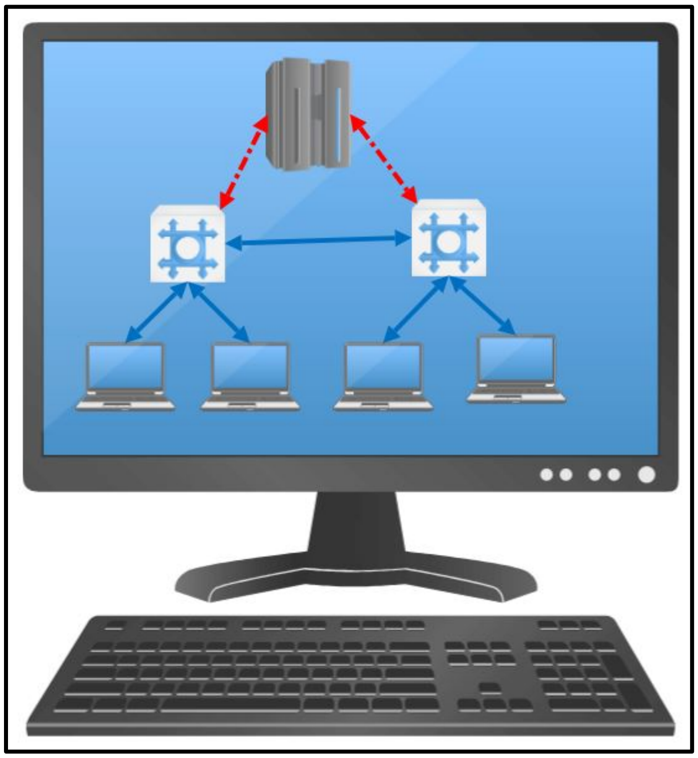


Figure 3. Network emulated in a single computer

© University of South Africa November 2018

Among the main reasons for emulating a network is to be able to test and prove concepts. Current information on Software Defined Networking can be found in research papers and in white papers [30]. In the case that an organization, needs to prove these concepts or plans to deploy Software Defined Networks, the results can be easily reproduced. Mininet enables the implementation of virtual network systems, where an environment of virtual hosts, switches, and links runs on a modern multicore server, using real application and kernel code with software-emulated network elements. An experiment has been conducted using Mininet to reproduce key results from published network experiments such as DCTCP, Hedera, and router buffer sizing which were successfully reproduced highlighting another important ability of Mininet [31].

1.4.3.3 Advantages of using Mininet

Mininet is not the only simulator which can be used to simulate SDN networks. Other specialized hardware network devices require specialized programming languages [32] to run. Other notable simulators available include the used of Raspberry-Pi [33] to develop a cost-effective OpenFlow testbed for a small scale SDN networking and Fs-SDN [34]. Although some of these other simulators do have advantages over Mininet on some aspects of simulation [35], Mininet remains the simulator of choice for SDN networks due to its flexibility and many advantages. Other simulators use full system virtualization, heavyweight containers with increasing complexity and overheads while decreasing usability. Mininet support the development of SDN systems and applications reliably without access to an expensive testbed [36].

Networks emulated in Mininet have produced reliable results, which has made Mininet to be used as a reference system when other emulators like Fs-SDN were being developed or tested. Fs-SDN is a Python-based tool developed for generating network flows records and interface counters. To evaluate Fs-SDN accuracy, scalability, and speed, a side by side setting up between Fs-SDN and Mininet was done with a series of identical network and traffic configurations. After which network traffic and system-level measurements where compared between the two. In this investigation, Mininet was discovered to be a better tool.

© University of South Africa November 2018

Mininet supports OpenFlow-based Software-defined Networking (SDN). It provides a flexible and cost-efficient experimental platform to develop, test, and evaluate OpenFlow applications. In Mininet, the processes of the virtual hosts and their application processes run inside the container. This allows them to have an independent view of system resources but still share the kernel with other containers [37].

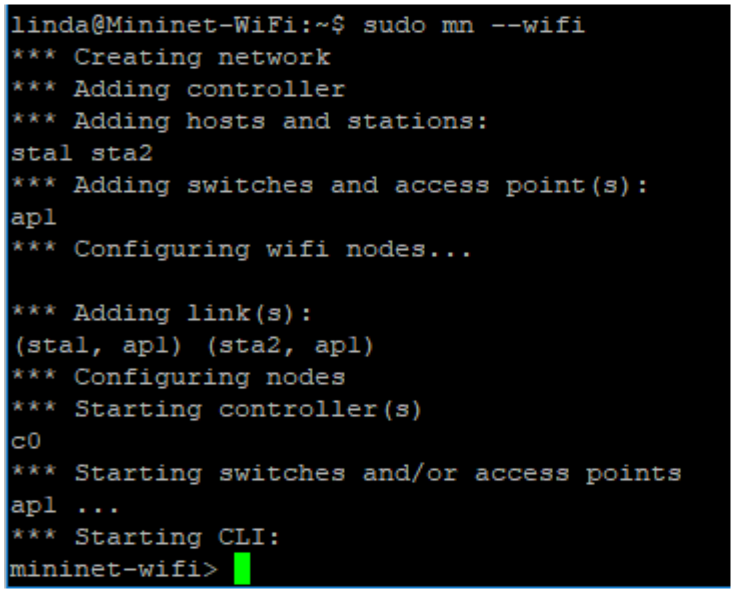
1.4.3.4 Mininet Installation

Mininet can be installed in three (3) different ways. The first one which is the easiest way is to download a pre-packed Mininet Virtual Machine (VM). The pre-packed Mininet Virtual Machine (VM) comes with Mininet and all required OpenFlow binaries and tools already pre-installed. A complete, compressed Mininet VM is about 1GB and can be used in most common hyper visors such as VMware, Xen and VirtualBox. The second option is to natively install Mininet from source. This option is suitable for local Virtual Machines and remote or cloud storage. The third option is to install Mininet from packages. This option may give an older version of Mininet and would require an upgrade once the installation has been completed. The latest Mininet release as of August 2018 is Mininet 2.2.2.

1.4.3.5 How does Mininet work

Mininet network is made of isolated hosts, which are a group of user-level processes moved into a network namespace that provide exclusive ownership of interfaces, ports and routing tables. The emulated links uses Linux Traffic Control (tc) which enforces the data rate of each link to shape traffic to a configured rate. Each emulated host has its own virtual Ethernet interface(s). The emulated switches are Open virtual Switches (OvSwitch) running in kernel mode or default Linux Bridge. These devices can run in ether kernel or user space to switch packets across interfaces. The process and code for creating a simple wireless network in Mininet is shown in Figure 4.

© University of South Africa November 2018



```
linda@Mininet-WiFi:~$ sudo mn --wifi
*** Creating network
*** Adding controller
*** Adding hosts and stations:
sta1 sta2
*** Adding switches and access point(s):
ap1
*** Configuring wifi nodes...

*** Adding link(s):
(sta1, ap1) (sta2, ap1)
*** Configuring nodes
*** Starting controller(s)
c0
*** Starting switches and/or access points
ap1 ...
*** Starting CLI:
mininet-wifi>
```

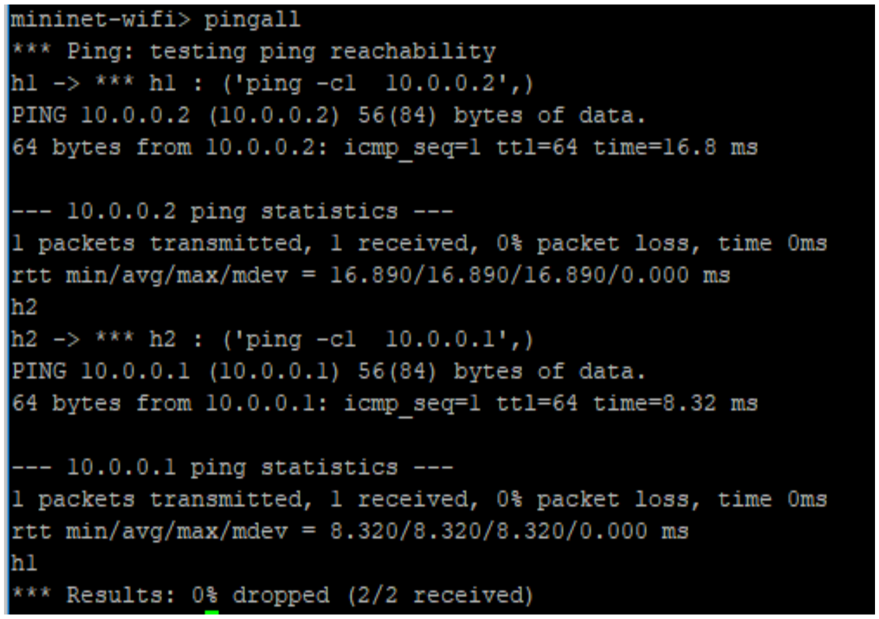
Figure 4. Basic wireless network created in Mininet

The study by Keti and Askar in [38] highlights Mininet's characteristics as being flexible, applicable, intractable, scalable, realistic and share-able. This is because in Mininet, new topologies and new features can be set in software using programming languages and common operating systems. Networks emulated in Mininet are usable with real life networks based on hardware without the need to make changes in source codes. To manage and run the simulated network in Mininet occurs in real time as it happens in a real-life network. Mininet can be scaled to large networks with hundreds or thousands of nodes. Networks implemented on Mininet can be easily shared, as it is share-able [39].

Software Defined Networking switches, hosts, controllers and links can be created by typing commands through Mininet's command line interface. The command line interface (CLI) in Mininet supports most Linux commands. The most commonly used commands are-: nodes: which lists all created nodes, dump: which displays the information about the network and created nodes, net: which shows how network elements are connected to each other.

© University of South Africa November 2018

The command line interface (CLI) also supports the day-to-day troubleshooting commands used in computer networking. These commands include "*pingall*", which output the results of the connectivity test among all nodes as seen in Figure 5. "*Ifconfig*" is also supported which displays the internet protocol (IP) information of the node. "*Iperf*" is also supported which is a tool used to test network performance. Iperf uses a client/server model, where traffic is initiated from the client and traverses the network to the server. "Iperf" creates data test streams supported by the network with a time-stamp and report the amount of data transferred and the throughput measured.



```
mininet-wifi> pingall
*** Ping: testing ping reachability
h1 -> *** h1 : ('ping -c1  10.0.0.2',)
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=16.8 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 16.890/16.890/16.890/0.000 ms
h2
h2 -> *** h2 : ('ping -c1  10.0.0.1',)
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=8.32 ms

--- 10.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 8.320/8.320/8.320/0.000 ms
h1
*** Results: 0% dropped (2/2 received)
```

Figure 5. Execution of "*pingall*" command

"Iperf" supports two types of transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Many applications like File Transfer Program (FTP), Simple Mail Transfer Protocol (SMTP) and Hypertext Transfer Protocol (HTTP) use TCP as the transport protocol. Using TCP mode, "Iperf" tests the maximum TCP bandwidth at the transport layer. In UDP mode, "Iperf" tests the jitter, packet loss and bandwidth. UDP mode is ideally for testing quality of service for applications like voice and video streaming. To see the list of all available commands, one can use "*help*" command, which is also supported in Mininet [40].

© University of South Africa November 2018

1.4.3.6 Mininet Topologies

Mininet supports five built-in network topologies. These built-in topologies are Minimal, Single, Linear, Tree and Reversed. Network topologies in Mininet can be modified using the command-line interface (CLI) [41]. The default topology is the minimal topology, which includes one OpenFlow kernel switch connected to two hosts, plus the OpenFlow reference controller.

- Minimal Topology: Minimal is the basic topology with one switch and two hosts.
- Single Topology: Single switch connected to k hosts where k is the number of hosts.
- Linear Topology: In Mininet, linear topology support k switches with k hosts where each host connects to one switch and all switches connects in a line. Linear topology can be created using "*sudo mn -topo linear, k*" where k is the number of host and switches.
- Tree Topology: This is a topology for a tree network with a given depth and fanout. This is a multilevel topology, which adds a subtree starting with node n.
- Reversed Topology: Reverse topology is the single switch connected to k hosts, with reversed ports where k is the number of hosts. The lowest-numbered host is connected to the highest-numbered port. Useful to verify that Mininet properly handles custom port numberings.

1.4.3.7 Mininet Python Script

Mininet Python Application Programming Interface (API) can also be used to create custom network topologies [44]. Python is an interpreted, interactive, object-oriented programming language. It provides high-level data structures. Python is modular by nature. The kernel is very small and can be extended by importing extension modules. A python program is compiled automatically by the interpreter and can be installed in any computer running any operating system [45]. The main classes used in Mininet Python script are mininet (net), link, node and topo. The complete list of classes includes clean, cli, log, nodelib, and topolib.

© University of South Africa November 2018

The Net class provides network emulation with hosts spawned in the namespace. This class has the following methods-:

- addHost (): This method adds a host to the network.
- addSwitch (): This method adds a switch to the network.
- addLink (): This method links two nodes together.
- addController (): This method adds a controller to the network
- getNodeByName (): This method return node with given name
- start (): This method starts the controller, switches and hosts
- stop (): This method stops the controller, switches and hosts
- ping (): This method ping between all specified hosts and return all data.

The Node class provides a virtual network node in a network namespace. In Mininet, a node can be either a host, switch or the controller. A host is a simple network node used to emulate the end computer or server. A switch is a node that is running an OpenFlow switch. The supported switches subclasses are OVSBridge, which is the Open Virtual switch in standalone or bridge mode. OVSKernelSwitch, which is the Open Virtual Switch that run-in kernel space. UserSwitch, which is the Open Virtual Switch that runs in user space and is slower that the OVSKernelSwitch and LinuxBridge, which is a normal Linux bridge. A Controller is a Node that is running an OpenFlow controller. Mininet support NOX, OVSController, RemoteController, Ryu subclasses for creating a controller.

The node class in Mininet supports the following methods-:

- MAC/setMAC: This method returns/assign the MAC address of a node or specific interface.
- IP/setIP: This method returns/assign IP address of a node or specific interface
- cmd: This method sends a command then waits for an output and return in.
- terminate: This method sends kill signal to Node and clean up after it.

© University of South Africa November 2018

The link class provides a basic link, which is represented as a pair of nodes. This class uses a link method, which creates a link to another node and makes two new interfaces. Node class inheritance diagram is shown in Figure 6.

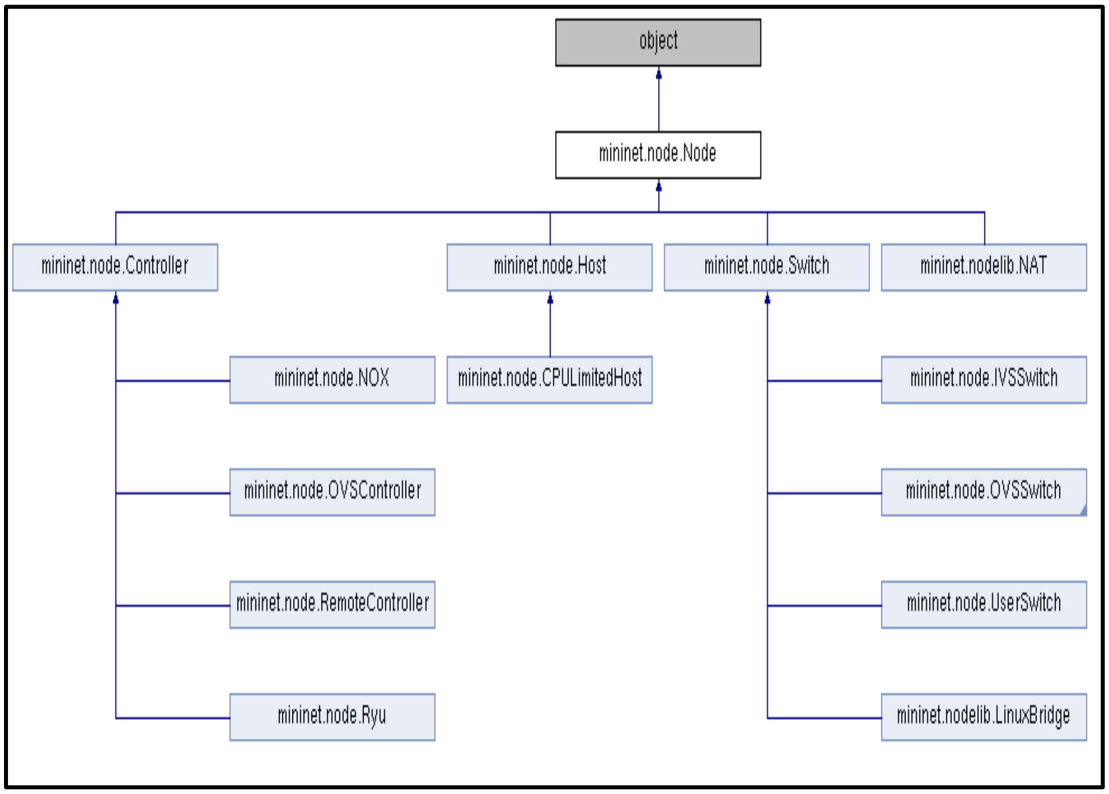


Figure 6. Mininet NODE class

The topo class provides the data center network representation for structured multi-trees. This class uses the following methods-:

- Methods similar to net e.g addHost, addSwitch, addLink
- addNode: This method adds nodes to the graph
- addPort: This method generates port mapping for new edge
- switches: This method return all switches
- hosts/nodes/switches/links: return all hosts
- isSwitch: Return true if node is a switch and return false if otherwise.

© University of South Africa November 2018

1.4.4 Mininet-WiFi

Mininet can also be used to emulate Software Defined Wireless Networks. To achieve this, the base code of Mininet must be extended by modifying classes and scripts to support wireless functionalities while also keeping all Software Defined Networking capabilities from the standard Mininet network emulator. Mininet-Wi-Fi is a fork of Mininet emulator, which extends its functionality by adding virtualized Wi-Fi stations and access points based on the standard Linux wireless drivers and the 802.11_hwsim wireless simulation driver. It adds classes to support the addition of wireless devices in a Mininet network scenario and to emulate the attributes of a mobile station such as position and movement relative to the access points. The 802.11 is the wireless standard by Institute of Electrical and Electronics (IEEE), which provides specifications for implementing wireless communications using the Wi-Fi (Wireless Fidelity) [42].

Mininet-WiFi developers have showcased it in a scenario with ad hoc and infrastructure wireless modes using a single experimental platform integrating virtual and physical nodes. This demonstration featured Mininet-WiFi as an emulator with the ability to run realistic experiments in hybrid physical-virtual environments, where users were able to experience it first hand by connecting their devices and interacting with virtual Wi-Fi stations in a wireless mesh network. They were able to connect to the internet through the emulated Software Defined Wireless Network infrastructure. Mininet-WiFi enhances Mininet emulator with virtual wireless stations and access points while keeping the original SDN capabilities and the lightweight virtualization software architecture [43].

1.4.5 Other Related Work

One of the latest studies which has been conducted is the performance of the three controllers named, Open Network Operating System (ONOS) [46], OpenMUL [47] and POX [48]. This was done by implementing the topology in Mininet and analysing real packet-in /packet-out messages between the data and control plane [49]. ONOS is a Java-based controller while POX is a python-

© University of South Africa November 2018

based centralised controller and OpenMUL is a centralised controller based on C programming language. Mininet and OpenvSwitch were used to create a linear topology and each switch in the topology was connected to a host. The performance of the controllers were then compared and measured using Mininet-Wireshark packet analysis and also Cbench which is a benchmarking tool that uses fake control packets generated from switch instances. This study showed that using Cbench [50] greatly under-estimates the performance by up to 96% for controller latency and 98% for controller throughput.

Mininet and OpenDaylight were suitable tools to conduct scalability analysis and flow admission in a Software Defined Network [51]. This study goes into length in explaining why these tools were suitable to conduct the study. Among the things mentioned is that Mininet is freely available and has already built in Open-Flow switches and virtual controllers [52]. It is also effortless because of easily building topology via drag and drop capability. Using OpenDaylight helped with flow performance and overall network scalability study [53]. Their conclusion state that the performance of the tool is strongly devoted to real environment results. During this study, they also observed the effectiveness of Mininet especially on time and resources according to prototyping, deployment and sharing.

Another groundbreaking implementation of Software Defined Networking using Mininet and OpenDaylight controller is the one that was done in the study to develop a resilient network between Philippines and Vietnam [54]. The aim of the study was to integrate Internet of Things (IoT) application (temperature and humidity sensor) applied to Software Defined Network using Mininet and OpenDaylight controller between Mapua University in Philippines and Vietnam National University in Vietnam. Data from the temperature and humidity sensors was successfully passed through the Software Defined Network having OpenFlows as it routing protocol. The results from this study also shows that Software Defined Networks has better latency, packet loss, bandwidth and convergence time compared to traditional network.

© University of South Africa November 2018

CHAPTER 2

2.1 Methodology

The design of this study is a combination of qualitative, explorative, descriptive and quantitative design as shown in Figure 7. The purpose of its combination is to gain a richer understanding of the existing studies and the findings. The presentation will be of a descriptive nature. The purpose of the quantitative nature is to get a deeper understanding of the network design and show the results on the first-hand bases through emulation of a network.

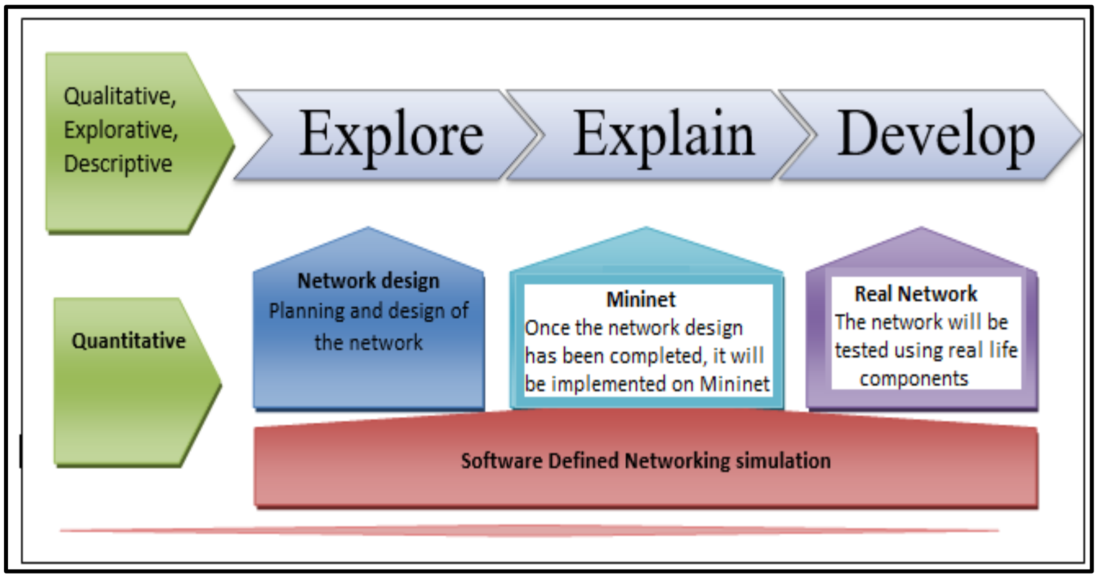

Qualitative, Explorative, Descriptive
Explore
Explain
Develop
Quantitative
Network design
Planning and design of the network
Mininet
Once the network design has been completed, it will be implemented on Mininet
Real Network
The network will be tested using real life components
Software Defined Networking simulation

Figure 7. Methodology framework

© University of South Africa November 2018

2.1.1 Logical Network

This project is based on simulating a Software Defined Network using Mininet, which communicates with the virtualized network hosted on the Emulated Virtual Environment – Next Generation (EVE-NG) which is a software that provides tools to model virtual devices and interconnect them with other virtual or physical devices. The OpenDaylight controller used in this study is hosted on Amazon Web Services (AWS) cloud platform. The logical network consists of a website hosted on the LAMP (Linux, Apache, MySQL, PHP) virtual machine. The F5 application delivery controller (ADC) provide load balancing of requests sent to the web applications. The LAMP server and the F5 application delivery controller are hosted on EVE-NG.

Two logical networks were used in this study to test different scenarios. The first scenario emulated a wireless Software Defined Network while the second scenario emulated the wired Software Defined Network. Mininet-WiFi was used to emulate wireless Software Defined Network. These network configurations were selected based on the industy best practice of Software Defined Networking. The logical network used to test wireless Software Defined Network is seen in Figure 8. Figure 9 shows the logical network used to test wired Software Defined Network.

© University of South Africa November 2018

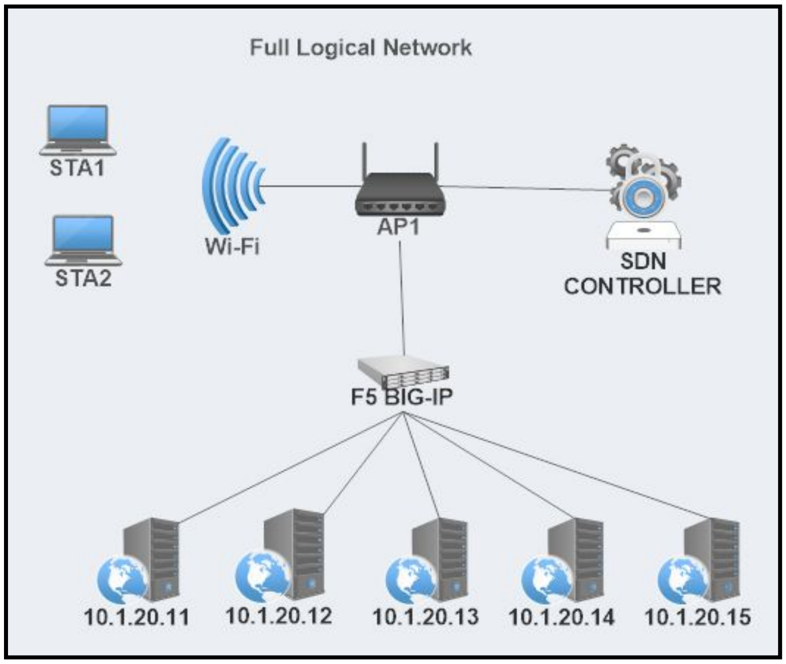


Figure 8. Logical network to test wireless SDN

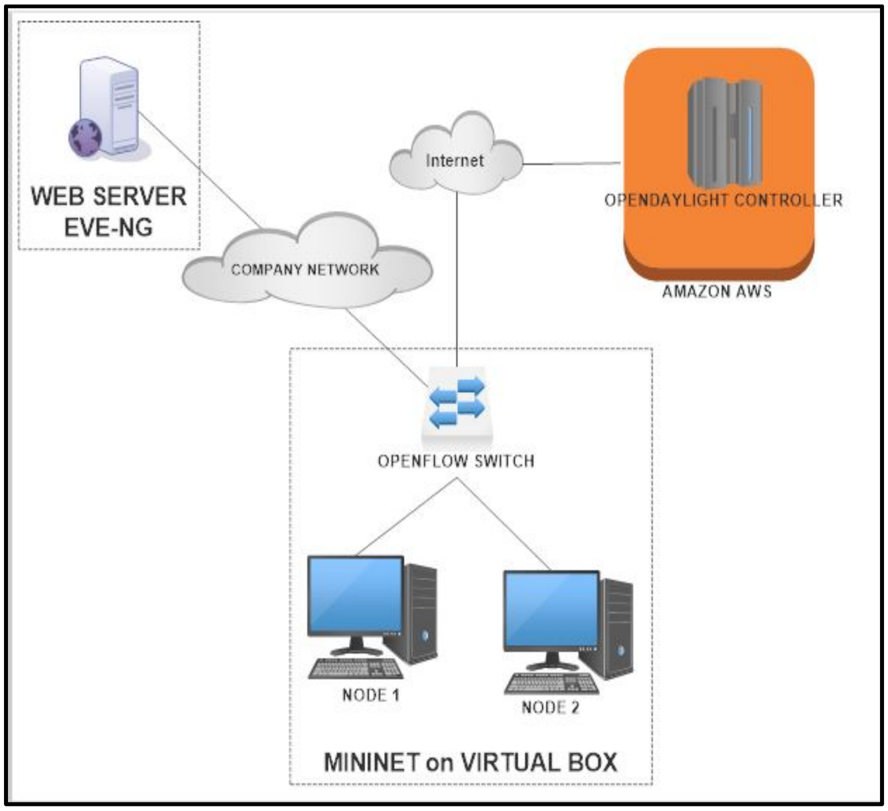


Figure 9. Logical network to test wired SDN

© University of South Africa November 2018

2.1.2 Virtualized network on EVE-NG (Emulated Virtual Environment – Next Generation)

EVE-NG is the Emulated Virtual Environment software as can be seen in Figure 10 used for Network, Security and DevOps professionals and to implement the real life network. It provides tools to be able to model virtual devices and interconnect them with other virtual or physical devices. It can be used in many ways, but it is mostly commonly used for testing modern technologies like network automation, proof of concepts (POC), network troubleshooting, test software in a simulated network, test out security vulnerabilities of any kind, system engineering etc.

The network part used in this project which is on EVE-NG is the network used on daily basis to test networks and application solutions, perform proof of concepts, network designs and troubleshooting. Some of the network devices seen in Figure 10 like the DoS_Tool, Winserver, Win7internal, Win7external and Tachyonic2 had no role in this project. The aim of using this already existing network topology was to test if indeed Mininet can be integrated and form part of the already existing real-life network that was built to perform other tasks not only for testing Software Defined Networking.

2.1.3 LAMP Server

LAMP is a group of open source software used to setup and run webservers. This is an acronym that stands for Linux, Apache, MySQL, and PHP. In this case, it uses Linux as the operating system, Apache as the Web server, MySQL as the relational database management system and PHP as the object-oriented scripting language. LAMP is mostly referred to as a LAMP stack because it has four layers. This stack can be built on different operating systems and the acronym changes to reflect that operating system. For example, when used with the Windows operating system, it is called WAMP; with Macintosh operating system, it is called MAMP; and with a Solaris system, it is called SAMP.

© University of South Africa November 2018

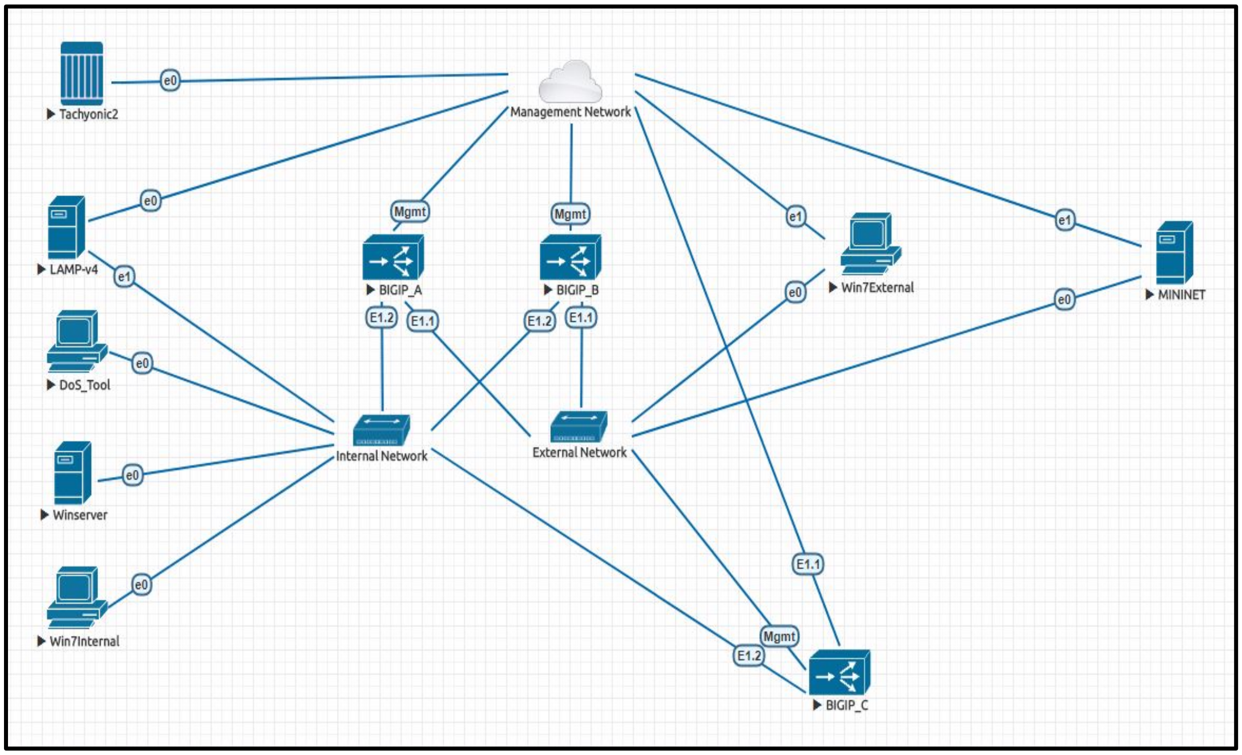


Figure 10. Network on EVE-NG

Five (5) servers were created on LAMP virtual machine as seen in Figure 10. Each of these servers hosted a test website. This is the typical scenario used by companies to host their applications. Same applications are hosted on different servers so that the servers can share the load as well as to offer backup and redundancy. Using LAMP, many applications can be hosted on one machine as if they were hosted on different machines. For this project, LAMP was used to test if hosts created on Mininet will be able to access websites hosted on it.

The configured servers were from server 1 to server 5 with Internet Protocol (IP) addresses 10.1.20.11 up to 10.1.20.15. All these configured servers deliver the same web application. The web application hosted by these servers is the same application that is used by the F5 Company which has most features used by companies in their applications. The configuration of these servers is not part of the scope for this project. This is because they are configured to do more than what is required for this project as this is a network used for many scenarios for real network environments.

© University of South Africa November 2018

2.1.4 F5 BIG-IP Virtual Edition (VE)

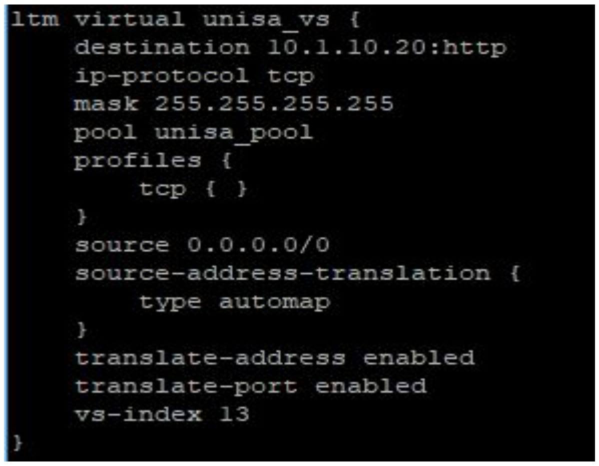
F5 is a company that specializes in application delivery networking (ADN) technology. The company name F5 was inspired by the 1996 movie Twister in which reference was made to the fastest and most powerful tornado on the Fujita Scale: F5. The company F5 founders believed that this company will cause the most powerful change in the networking field. F5 is involved in the delivery of web applications, security, performance, availability of servers, data storage devices, and other network and cloud resources. To do this, F5 uses BIG-IP platform, which is a blend of software and hardware that perform the function of a load balancer and a full proxy. BIG-IP is not an acronym for anything either, F5 got the concept from TCP/IP (Transmission Control Protocol/ Internet Protocol) notation. The BIG part of the name is from F5 view of this technology as being a full proxy, which presents a virtual IP on behalf of many devices that are behind it. This makes it according to F5 an IP bigger that a normal IP address, hence it is called BIG-IP.

BIG-IP gives the ability to control the traffic that passes through the network. These devices come in two (2) flavors, which are physical hardware and virtual Edition (VE). The virtual editions of BIG-IP products offer the same variety of features available in hardware solutions and can be deployed on a public or private cloud. Virtual Edition is the flavor of BIG-IP application delivery controller used in this project.

Three devices (BIGIP_A, BIGIP_B, and BIGIP_C) were installed and Local Traffic Manager (LTM) module was provisioned. The Local Traffic Manager (LTM) module is BIG-IP load balancer module. The devices are configured in a High Availability (HA) mode. This means that when one device fails, the other devices would take over and process the traffic that was being process by the failed device. The configured network on the devices include the internal VLAN (Virtual Local Area Network), which is the network part that communicates with backend servers running on the LAMP stack and the external VLAN (Virtual Local Area Network). The external VLAN is the network part that communicates with the company external network. Mininet used this external virtual local area network to communicate with web applications via the BIG-IP application delivery controller.

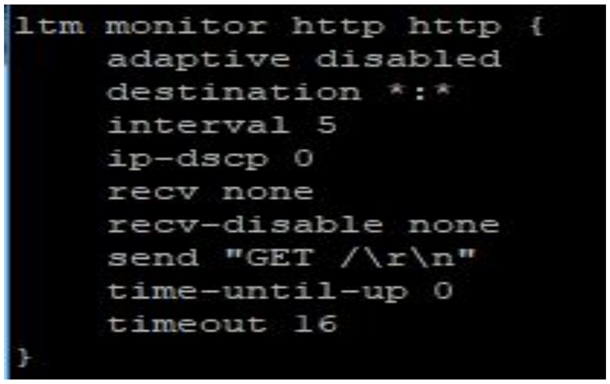
© University of South Africa November 2018

In the BIG-IP application controller, a virtual server called "unisa_vs" was created and configured as seen in Figure 11. Health monitor was enabled to check the status of the servers in the LAMP virtual machine as seen in Figure 12. This virtual sever listen to incoming traffic from the external virtual local area network. The pool of servers called "unisa_pool" was also created as seen in Figure 13. This pool points to the backend servers, which are configured on the LAMP server. These servers were enabled to listen for hypertext transmission protocol (http) port 80 requests, which is the port for web services. The "unisa_vs" virtual server uses the round-robin algorithm to load balance the requests. In round-robin algorithm, all requests are evenly distributed across pool members. In this case the 1st request goes to 1st node, 2nd request to 2nd node and 3rd to 3rd node, 4th request to 4th node, 5th request to 5th node and 6th request to 1st node and continue in that sequence through all five configured nodes.



```
ltm virtual unisa_vs {
    destination 10.1.10.20:http
    ip-protocol tcp
    mask 255.255.255.255
    pool unisa_pool
    profiles {
        tcp { }
    }
    source 0.0.0.0/0
    source-address-translation {
        type automap
    }
    translate-address enabled
    translate-port enabled
    vs-index 13
}
```

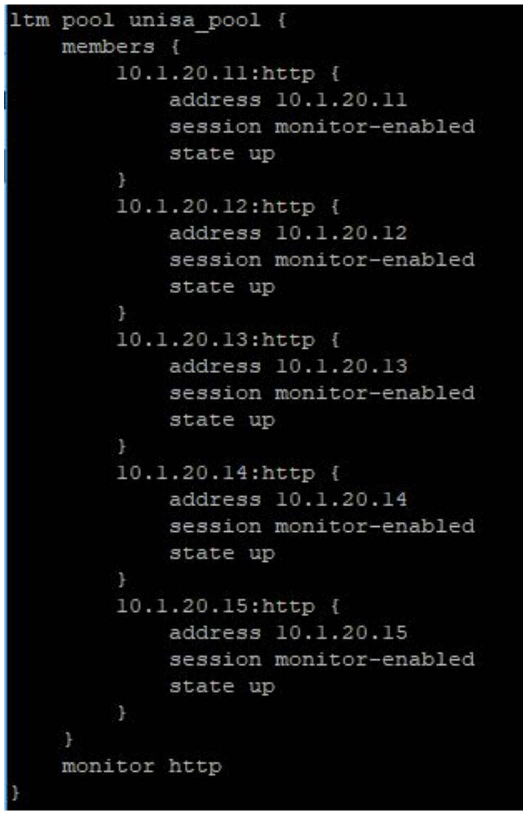
Figure 11. Virtual server configuration



```
ltm monitor http http {
    adaptive disabled
    destination *:*
    interval 5
    ip-dscp 0
    recv none
    recv-disable none
    send "GET /\r\n"
    time-until-up 0
    timeout 16
}
```

Figure 12. Monitor configuration

© University of South Africa November 2018

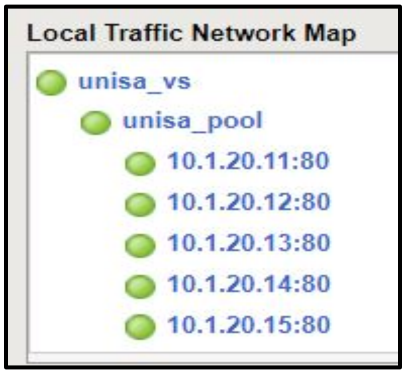


```
ltm pool unisa_pool {
    members {
        10.1.20.11:http {
            address 10.1.20.11
            session monitor-enabled
            state up
        }
        10.1.20.12:http {
            address 10.1.20.12
            session monitor-enabled
            state up
        }
        10.1.20.13:http {
            address 10.1.20.13
            session monitor-enabled
            state up
        }
        10.1.20.14:http {
            address 10.1.20.14
            session monitor-enabled
            state up
        }
        10.1.20.15:http {
            address 10.1.20.15
            session monitor-enabled
            state up
        }
    }
    monitor http
}
```

Figure 13. Pool configuration

These configured Local Traffic Manager (LTM) components works as follows-:

- The monitor keeps on monitoring the status of the nodes in the LAMP and return a green circle is the node is up as seen in Figure 14.
- The virtual server listens for incoming traffic on port 80 of 10.1.10.20 and use round-robin algorithm to load balance the traffic to the nodes on the pool.
- The pool contains the list of nodes which servers the same web application.



Local Traffic Network Map

- ● unisa_vs
  - ● unisa_pool
    - ● 10.1.20.11:80
    - ● 10.1.20.12:80
    - ● 10.1.20.13:80
    - ● 10.1.20.14:80
    - ● 10.1.20.15:80

Figure 14. Monitor results

© University of South Africa November 2018

2.1.5 Listening tools to network parameters

Wireshark is the packet monitoring and analyzing tool that provides the ability to analyze data traffic interactively on a network. Wireshark display packets with detailed protocol information. It can be used to analyze different types of traffic in all layers of the Open Systems Interconnection (OSI). Wireshark puts the network card into promiscuous mode (a mode that tells it to read all packets sent to it) and uses PCAP (packet capture) which consists of an application programming interface for capturing network traffic. From the captured packets, Wireshark then generates many reports such as round-trip time, throughput, flow graph and many more reports of Open Systems Interconnection protocol. Wireshark supports several plugins which can be enabled to give it more capabilities. One of such plugins is Transum.

Transum is a Wireshark plug-in for dynamically calculating various response time latencies within sessions. It uses Response Time Element (RTE) model to provide a breakdown of the overall response time for a service. Transum's application messages are called Application Protocol Data Units (APDU). APDU is the flow between the client and services. APDU response time is the total time the user must wait for a completion of a request. Service time is the time it takes for the service to process the request. Request spread is the time needed to transmit the whole request from the client to the service. Response spread is the time transmit the whole response from the service to the client.

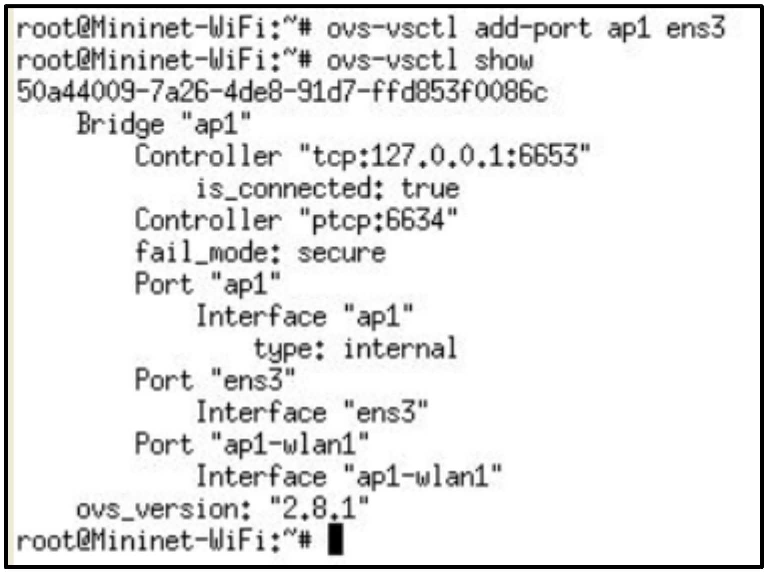
Transum is an ideal tool for troubleshooting latency on the multi-tiered networks. The example of such networks is Client -->Webserver --> Database services. When an application is experiencing an overall high latency issue, using Transum can help in identifying the part of the service that is responsible for the high latency issue. This is done by first running packet capture with Transum enable between the client and the webserver, and then between the webserver and the Database. By comparing the overall service latency against the two run Transum captures, the part responsible for the latency can then be easily identified. For this project, since the topology was just client --> Webserver, Transum only produced one Response Time Element data with the service time equal to the APDU response time. For this reason, Transum capture was not included on the result section of this project.

© University of South Africa November 2018

2.1.6 Wireless Software Defined Network on Mininet-WiFi

A Mininet-WiFi virtual machine was created by installing Mininet-WiFi on an Ubuntu Server virtual machine. Two (2) network interfaces were configured and associated with Mininet-WiFi virtual machine (VM). The first attached network interface was a host-only adapter. This allows host computer to be able to connect to the virtual box using terminal emulator. The terminal emulator such as Putty offers better command line interface as compared to Mininet command prompt. The second network interface that was attached was the Network Address Translation (NAT) adaptor. This is the connection that Mininet use to communicate with the external networks.

Mininet-WiFi basic network topology was used which consist of a wireless access point (AP1) with two wireless stations (Sta1, Sta2) and the Mininet Software Defined Networking controller. The access point connected to the controller (C0) using a virtual connection and the two (2) stations are attached to the access point (AP) using the simulated wireless interface. Once the basic network was created, the external Virtual Local Area Network interface was then added to the access point (AP) interface using OpenFlow commands as seen in Figure 15. Both stations (Sta1 and Sta2) internet protocol (IP) addresses were modified such that they are also on the same external VLAN subnet.



```
root@Mininet-WiFi:~# ovs-vsctl add-port ap1 ens3
root@Mininet-WiFi:~# ovs-vsctl show
50a44009-7a26-4de8-91d7-ffd853f0086c
    Bridge "ap1"
        Controller "tcp:127.0.0.1:6653"
            is_connected: true
        Controller "ptcp:6634"
        fail_mode: secure
        Port "ap1"
            Interface "ap1"
                type: internal
        Port "ens3"
            Interface "ens3"
        Port "ap1-wlan1"
            Interface "ap1-wlan1"
    ovs_version: "2.8.1"
root@Mininet-WiFi:~# █
```

Figure 15. Adding external interface on the Access Point (AP)

© University of South Africa November 2018

The logical emulated wireless Software in Mininet-WiFi can be seen in Figure 16 below.

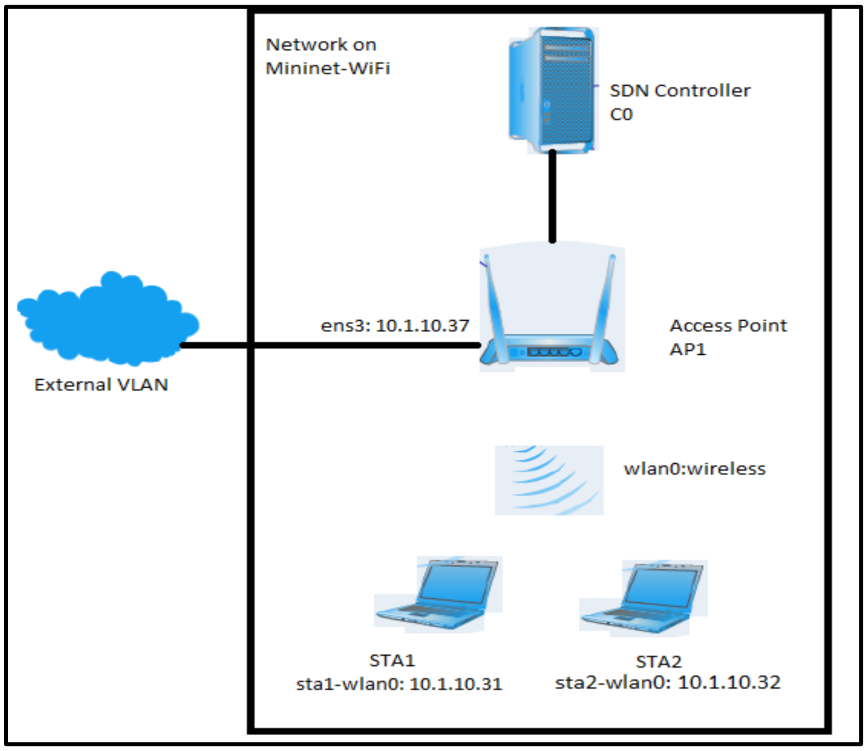


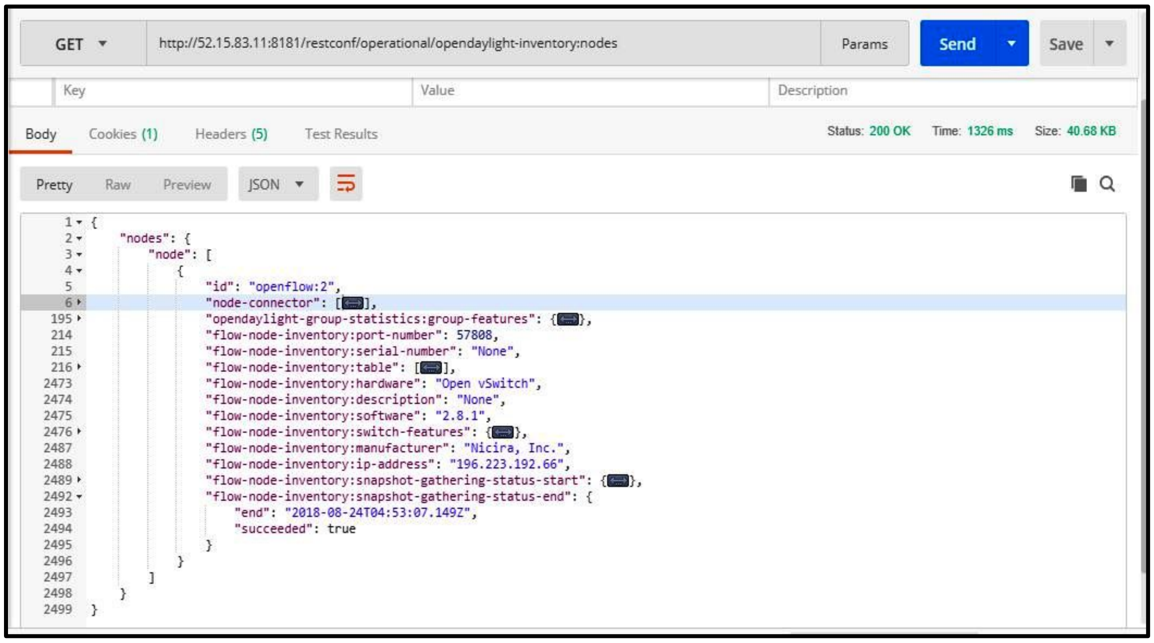
Figure 16. Logical Mininet-WiFi network

2.1.7 Wired Software Defined Network on Mininet

The Mininet Virtual Machine (VM) used in this paper is hosted on Oracles' Virtual Box. Mininet was installed on Ubuntu 17 operating system. The emulated Software Defined Network in Mininet was created using a Python script to enable communication with the controller using the Application Programmable Interface (API). Application program interface (API) is a set of routines, protocols, and tools for building software applications. It specifies how software components should interact. They are used when programming graphical user interface (GUI) components. API makes it easier to develop a program by providing all the building blocks. One of the commonly used API in the field of networking is REST API.
Representational State Transfer (REST) API is an architectural style and an approach for

© University of South Africa November 2018

communication used in the development of Web Services. It enables users to connect and interact with cloud services efficiently. To test the API, the program called “Postman” can be used. “Postman” is an application for testing APIs by sending request to the web server and getting the response back. “Postman” makes it easy to test, develop and document APIs. It allows users to set up all the headers and cookies the API expects and checks the response. “Postman” was used to send RESTCONF GET API to retrieve node inventory and topology as created by Mininet and seen by OpenDaylight controller as seen in Figure 17 below.



http://52.15.83.11:8181/restconf/operational/opendaylight-inventory:nodes

Status: 200 OK Time: 1326 ms Size: 40.68 KB

```
{
    "nodes": {
        "node": [
            {
                "id": "openflow:2",
                "node-connector": [↔],
                "opendaylight-group-statistics:group-features": {↔},
                "flow-node-inventory:port-number": 57808,
                "flow-node-inventory:serial-number": "None",
                "flow-node-inventory:table": [↔],
                "flow-node-inventory:hardware": "Open vSwitch",
                "flow-node-inventory:description": "None",
                "flow-node-inventory:software": "2.8.1",
                "flow-node-inventory:switch-features": {↔},
                "flow-node-inventory:manufacturer": "Nicira, Inc.",
                "flow-node-inventory:ip-address": "196.223.192.66",
                "flow-node-inventory:snapshot-gathering-status-start": {↔},
                "flow-node-inventory:snapshot-gathering-status-end": {
                    "end": "2018-08-24T04:53:07.149Z",
                    "succeeded": true
                }
            }
        ]
    }
}
```

Figure 17. Postman "GET" node inventory API

For the Mininet script to work, three (3) main Mininet API classes were used. Those classes are Topo, Switch and Controller.

A function, which is used to create a custom network was created using Python. For this function to create the network, Mininet was prevented from creating the network using the default values. This was achieved by setting the topo class to none and the build class to false. Using the controller class, a remote controller was defined and given values for the name and an IP address, which in this case is the IP address of the OpenDaylight controller hosted on Amazon cloud. The connection port was set to port 6633.

Using OVSKernelSwitch sub class, an Open Virtual Switch (OVS) was created. Two (2) network

© University of South Africa November 2018

hosts were also defined and given networking properties. The script defines the network subnet that the controller must use together with the links between the switch and the hosts. As part of the program, the script programs the controller to add the external interface to the switch after creating the network. This interface is used by Mininet to reach the Linux server inside the company domain.

To start the program, we loaded the saved python script from the directory that it was saved on. Mininet created the network as defined by the script. The created network consists of two (2) hosts and the Open Virtual Switch (OVS). The created switch has a control channel, which it uses to communicate with the controller. It has the pipeline, which consists of flow tables. There is also data path, which is the forwarding plane as seen in Figure 18.

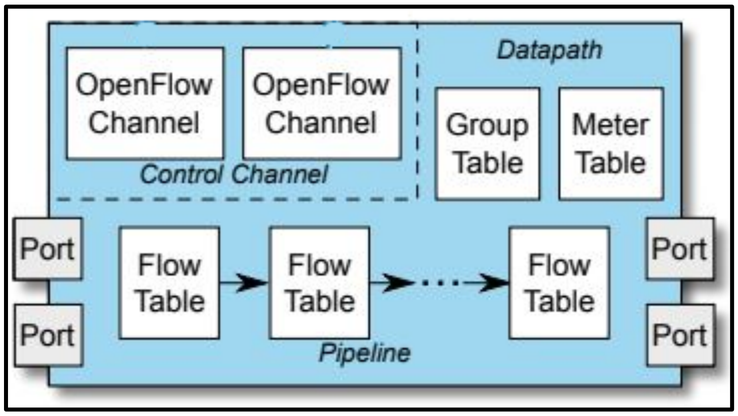


Figure 18. OpenFlow virtual Switch (OvSwitch)

2.1.8 OpenDaylight controller

Mininet support extensible Python API for network creation and experimentation. Depending on the choice of the controller used, Mininet also supports programs written in C, JAVA and C++. In this project the network is created on Python which is the interpreted high-level programming language for general purpose programming and the controller used on this project is the OpenDaylight controller. OpenDaylight (ODL) is a modular open platform for customizing and automating networks of any size and scale. It is created and managed by OpenDaylight Foundation as part of the Linux foundation project.

OpenDaylight (ODL) controller is the Software Defined Networking controller used in this project.

© University of South Africa November 2018

It is based on Services Abstraction Layer (SAL), which allows it to support other protocols and not only OpenFlow. It is implemented in Java and can be deployed in any system supporting Java. OpenDaylight controller was developed by the OpenDaylight consortium in 2013. OpenDaylight project is supported by Cisco, Juniper, VMWare and many other vendors and companies operating in the networking environment. This support by many organizations enables OpenDaylight to be vendor neutral [55].

The controller uses Application Programmable Interfaces (API) like Representational State Transfer (REST) technology to communicate with the Network Applications orchestrations and services layer. This can include OpenStack Neutron, Virtual Tenants Network (VTN) coordinator [56-57]. The controller layer itself run several services which includes service abstraction layer (SAL), OpenStack service, base network service and many more. To communicate with data plane elements, the controller uses southbound interfaces and protocol plugins such as OpenFlow, the Open vSwitch Database Management Protocol (OVSDB) [58], the Network Configuration Protocol (NETCONF) [59] and many more [60].

© University of South Africa November 2018

The OpenDaylight controller architecture is shown on Figure 19 below.

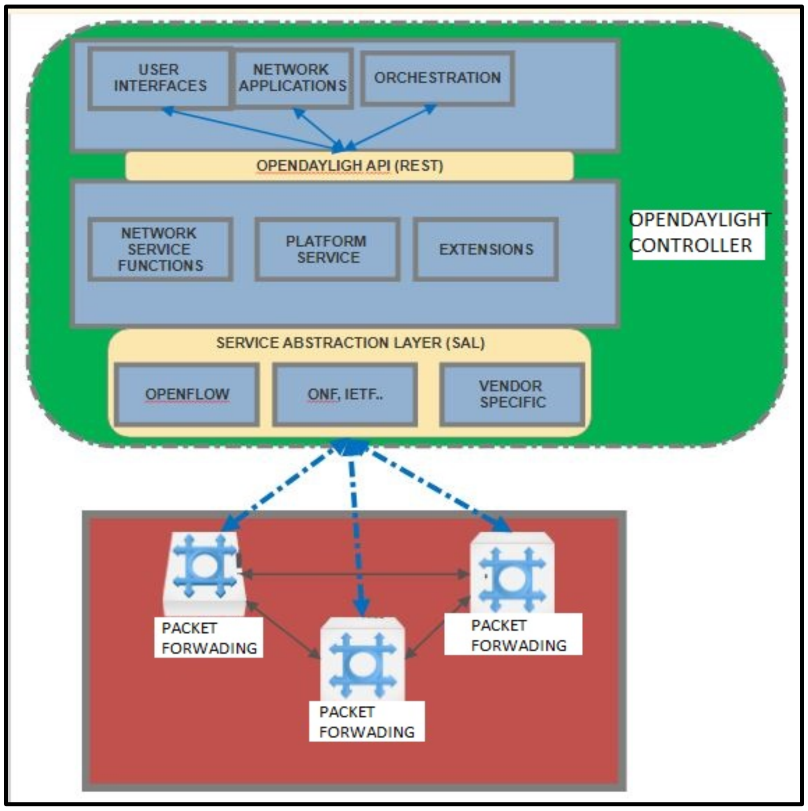


Figure 19. OpenDayligh controller architecture

In this project we used the eighth release of the OpenDaylight controller, which is called Oxygen. We downloaded the software from the OpenDaylight software download page and installed the controller on the Ubuntu 17 server. To install and enable required features that the OpenDaylight controller must use, an open source application called Apache Karaf is used. Karaf as it is normally called is a modular Open Services Gateway Initiative (OSGI) that provides tools and features required to deploy an application. An Open Services Gateway Initiative (OSGI) is a set of specifications for developing and deploying modular software programs and libraries, which are packed in bundles. Karaf enables modules to be installed, started, stopped, updated, and uninstalled without requiring a reboot.

© University of South Africa November 2018

By default, the OpenDaylight controller has no features enabled. We have installed and enabled the following features in this project on the controller (but there are many features, which can be installed and enabled)-:

- odl-restconf – Representational State (REST) like protocol that provides a programmatic interface over Hyper Text Transfer Protocol (HTTP) for accessing data on port 8080 for HTTP requests.
- odl-l2switch-all – Layer2 switch functionality.
- odl-mdsal-apidocs - Model Driven Service Abstraction Layer (MD-SAL) Application Programmable Interface (API) Documentation.
- odl-dlux-all - Graphical user interface for OpenDaylight based on the AngularJS Framework.

The OpenDaylight controller used on this project is hosted on Amazon Web Services (AWS) cloud platform. We have used the Elastic Compute Cloud (EC2), which is a secure and resizable compute node. It allowed us to obtain and configure capacity in minutes. The Elastic Compute Cloud (EC2) can scale both up and down allowing us to increase or decrease capacity as per our need. Wireshark, the network packet analyzer was used to analyze communication between the Open Virtual Switch in Mininet and the OpenDaylight controller.

When the communication between the switch and the controller is established. The controller adds flows to enable the switch to behave like a learning switch. When the switch receives a packet, it starts by performing a table lookup in the first flow table called table 0. In pipeline, each flow table contains one or more flow entries. Matching starts with the first flow table. If a Match is found, instructions associated with flow entry are executed. Instruction may direct the packet to next flow table in pipeline. When processing stops, the associated action set is applied, and packet forwarded. Instructions describe packet forwarding, packet modification, group table processing and pipeline processing. The summary flow chart is shown in Figure 20. This flow chart represents the program used to program the controller to establish communication between the controller in the cloud and the switch within Mininet.

© University of South Africa November 2018

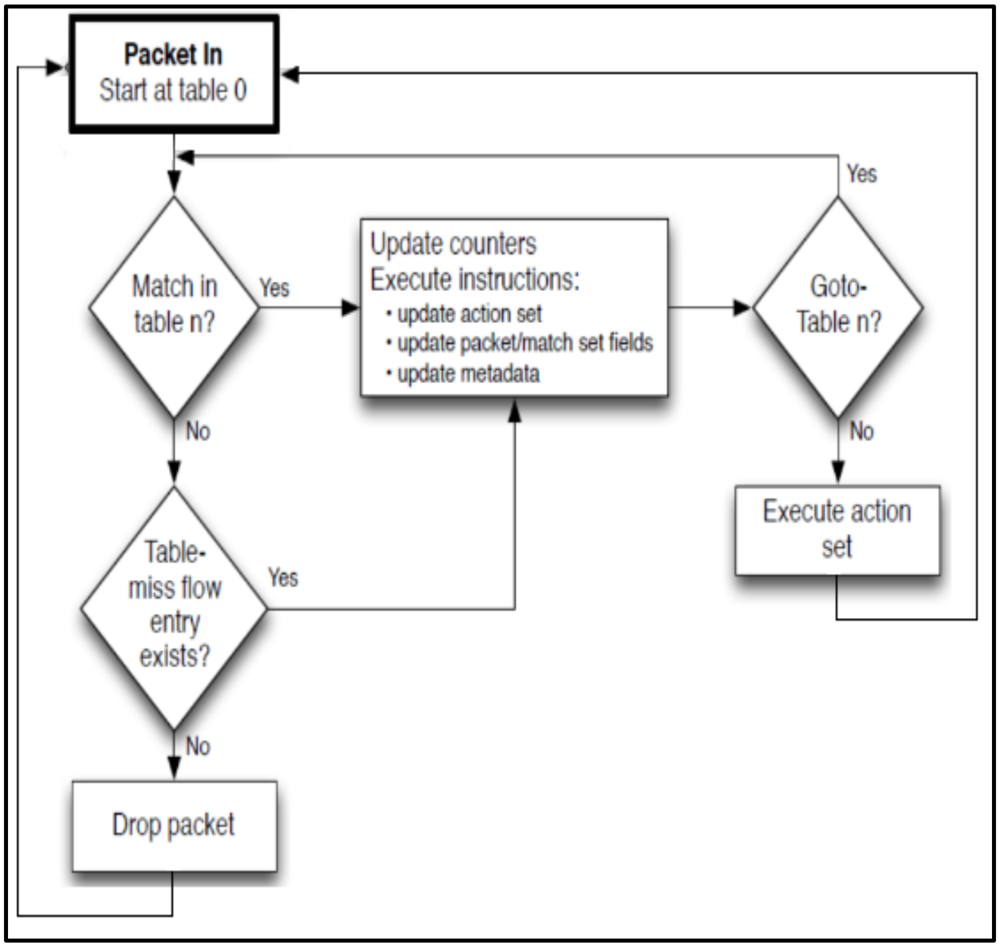


Figure 20. OpenFlow virtual switch flow chart

© University of South Africa November 2018

CHAPTER 3

3.1 Implementation of Mininet emulated network

To demonstrate that the network emulated on Mininet can be deployed in real life, the wired Mininet network seen on Figure 21 was deployed.

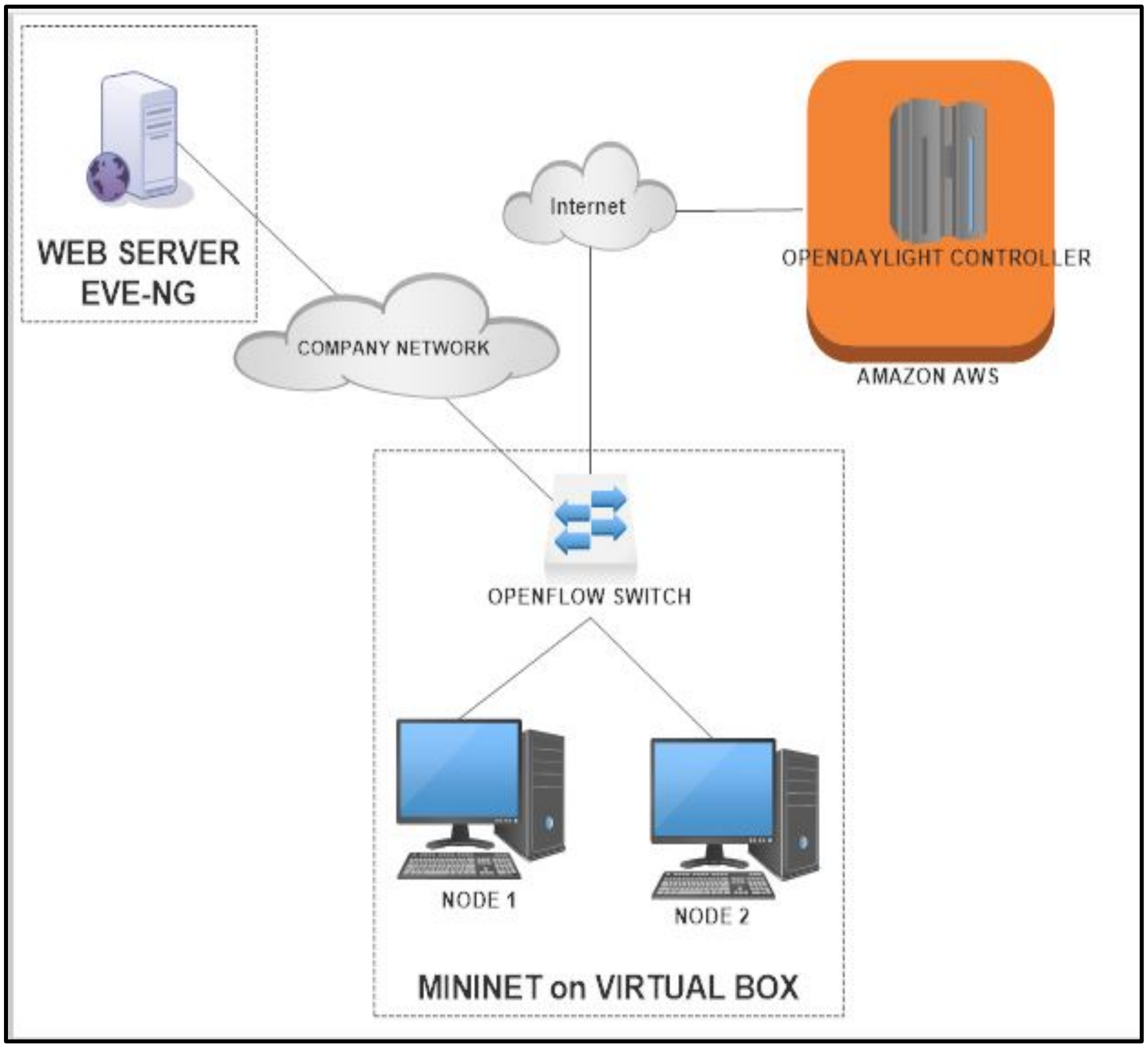


Figure 21.. Logical network to test wired SDN

© University of South Africa November 2018

The emulated network in Mininet consisted of the Openflow virtual switch and two Linux hosts. The deployed networked consisted of a Juniper vMX device and two Ubuntu Personal Computers (PC). The images used on this test were hosted on EVE-NG. Figure 22 shows the network as implemented on EVE-NG.

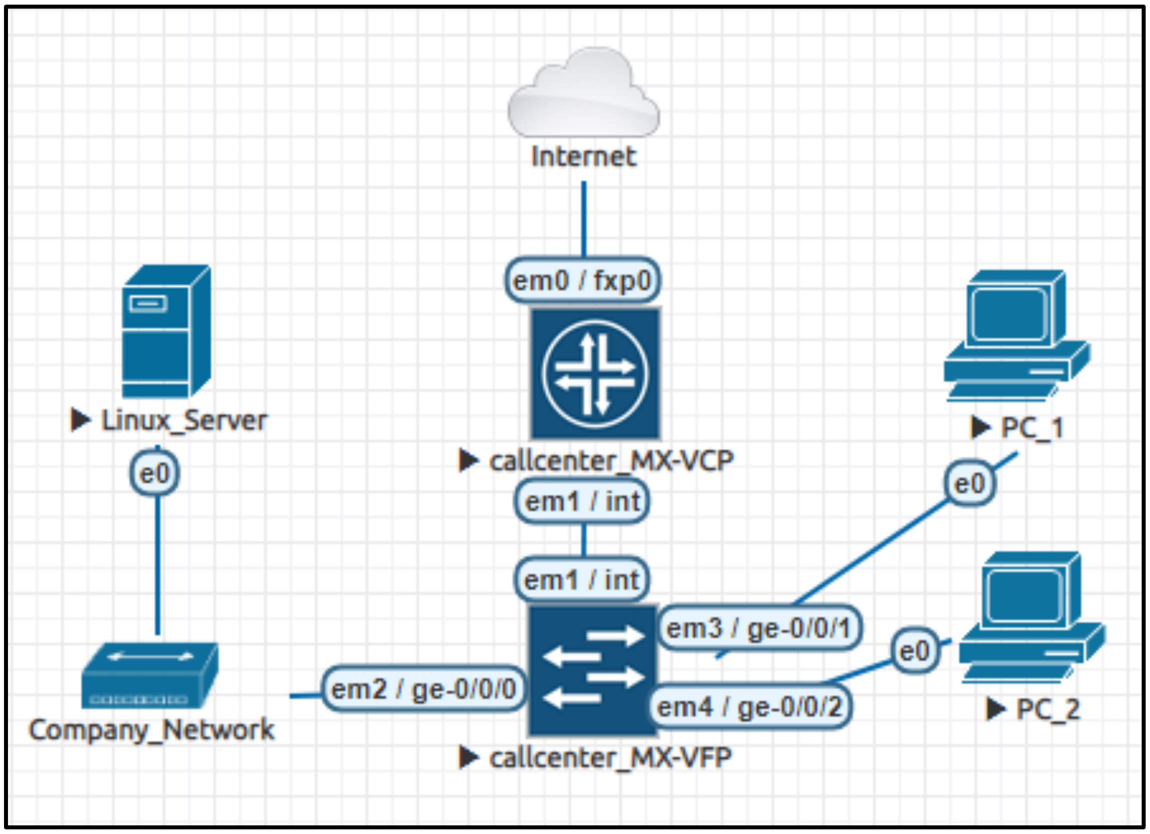


Figure 22. Deployed network on EVE-NG

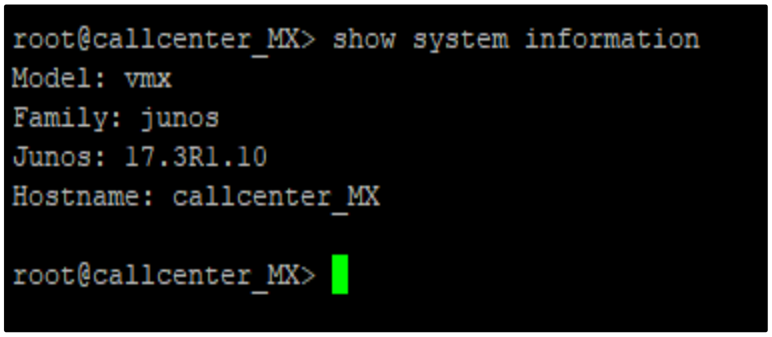
© University of South Africa November 2018

3.1.1 Preparation of Juniper device

OpenDaylight controller uses Network Configuration Protocol (NETCONF) connector to communicate with Juniper devices. Network Configuration Protocol (NETCONF) provides a mechanism to install, manipulate and delete the configuration of network devices. It uses an Extensible Markup Language (XML) based data encoding for the configuration data as well as the protocol messages. The NETCONF protocol operations are realized as remote procedure calls (RPCs).

Interoperability between Juniper MX series routers and OpenDaylight controller is supported starting from the Junos OS Release 17.3R1 onwards. This interoperability is made possible by the OpenDaylight Southbound Network Configuration Protocol (NETCONF) connector Application Programming Interface (API). The NETCONF connector uses YANG models to interact with the network device using OPENFLOW or any other supported protocol. YANG (Yet Another Next Generation) is a data modelling language by Internet Engineering Task Force (IETF). It was published as Request for Comment (RFC) RFC 6020.

Juniper vMX is a full-featured device that run on Junos operating system. It has trio microcode for x86 chipsets which allow it to have full features and operations as a physical MX series universal edge router. Using this setup, OpenDaylight can then be used to carry out any configurations, orchestrations, maintenance, provisioning and support of Juniper MX router and can execute Remote Procedure Calls (RPC). Figure 23 shows the information of the Juniper devices used on this project.



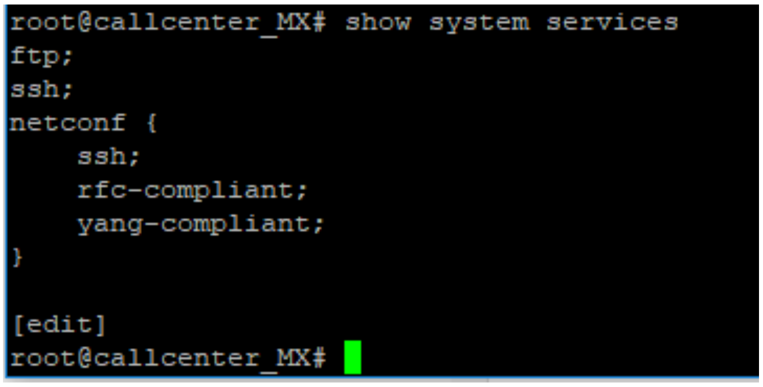
```
root@callcenter_MX> show system information
Model: vmx
Family: junos
Junos: 17.3R1.10
Hostname: callcenter_MX

root@callcenter_MX>
```

Figure 23. Juniper device information

© University of South Africa November 2018

As it has been stated before, NETCONF is used by the OpenDaylight controller to interact with southbound devices, therefore it must be configured on the Juniper device. For this project, NETCONF was configured as follows in Figure 24;

1. Access to NETCONF SSH subsystem was enabled by running command "*set system services netconf ssh*"
2. Device was configured to be compliant with NETCONF configuration. This was done by running the command "*set system services netconf rfc-compliant*"
3. The router was also configured to be YANG compliant to prevent exporting any hidden or unsupported configurations hierarchies' output. This was done by running the command "*set system services netconf yang-compliant*"
4. The settings were then committed which is Juniper's way of permanently storing the configs by running the command "*commit*".



```
root@callcenter_MX# show system services
ftp;
ssh;
netconf {
    ssh;
    rfc-compliant;
    yang-compliant;
}

[edit]
root@callcenter_MX#
```

Figure 24. Juniper device netconf settings

© University of South Africa November 2018

3.1.2 Preparation of OpenDaylight controller

OpenDaylight supports the NETCONF protocol as a northbound server as well as a southbound plugin. It also includes a set of test tools for simulating NETCONF devices and clients. For southbound plugin, OpenDaylight uses netconf-connector. Netconf-connector connects to remote NETCONF devices and expose it configuration, operational datastores and notifications as Model-Driven Service Abstraction Layer (MD-SAL) mount points. This enables applications and remote users to interact with the mounted devices.

OpenDaylight Release Oxygen 0.8.2 was used on this project. Netconf-connector was configured directly through Model-Driven Service Abstraction Layer (MD-SAL) with the usage of the network-topology model. OpenDaylight controller was started by running the command "*./karaf-0.8.2/bin/karaf*". Once the controller was running, netconf-topology and restconf featured were installed by using the command "*feature:install odl-netconf-topology odl-restconf odl-netconf-connector-all"*

To create a new netconf-connector, "Postman" was used to send PUT request which had the following values-:

- URL:

http://<controller IP>:8181/restconf/config/network-topology:network-opology/topology /topology-netconf/node/<device name>

For this project, controller IP was 10.10.204.85 and the device name was callcenter_MX

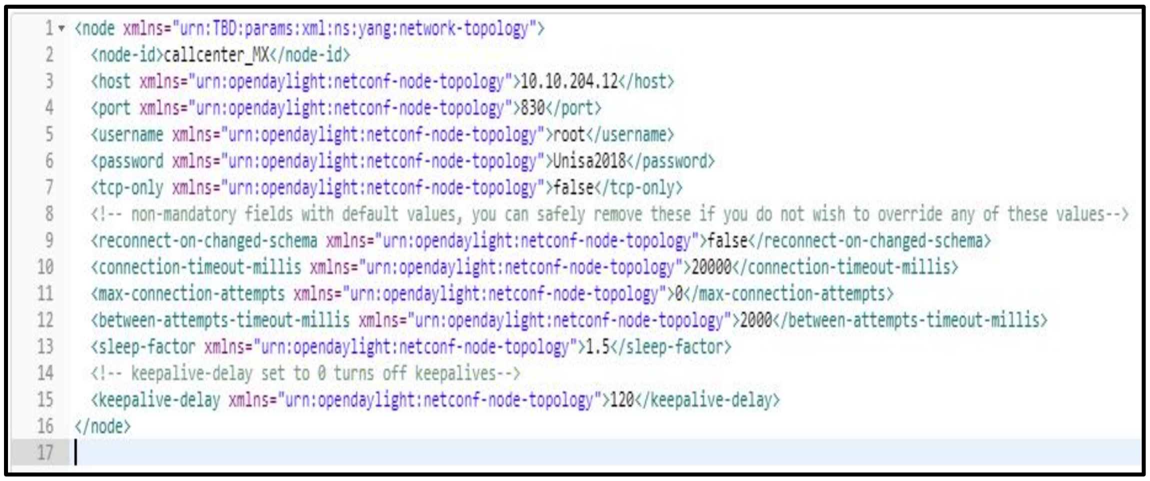
- Headers:

Accept:application/xml
Content-Type:application/xml

© University of South Africa November 2018

- Payload:

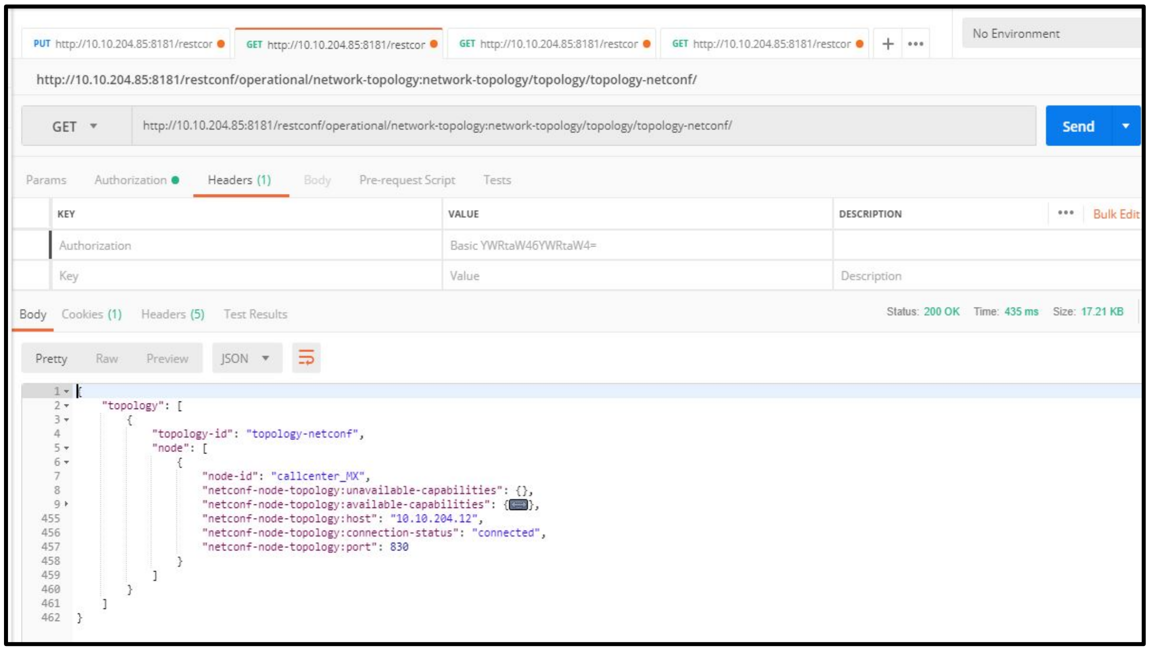
The payload information used is as seen on Figure 25.



```xml
<node xmlns="urn:TBD:params:xml:ns:yang:network-topology">
  <node-id>callcenter_MX</node-id>
  <host xmlns="urn:opendaylight:netconf-node-topology">10.10.204.12</host>
  <port xmlns="urn:opendaylight:netconf-node-topology">830</port>
  <username xmlns="urn:opendaylight:netconf-node-topology">root</username>
  <password xmlns="urn:opendaylight:netconf-node-topology">Unisa2018</password>
  <tcp-only xmlns="urn:opendaylight:netconf-node-topology">false</tcp-only>
  <!-- non-mandatory fields with default values, you can safely remove these if you do not wish to override any of these values-->
  <reconnect-on-changed-schema xmlns="urn:opendaylight:netconf-node-topology">false</reconnect-on-changed-schema>
  <connection-timeout-millis xmlns="urn:opendaylight:netconf-node-topology">20000</connection-timeout-millis>
  <max-connection-attempts xmlns="urn:opendaylight:netconf-node-topology">0</max-connection-attempts>
  <between-attempts-timeout-millis xmlns="urn:opendaylight:netconf-node-topology">2000</between-attempts-timeout-millis>
  <sleep-factor xmlns="urn:opendaylight:netconf-node-topology">1.5</sleep-factor>
  <!-- keepalive-delay set to 0 turns off keepalives-->
  <keepalive-delay xmlns="urn:opendaylight:netconf-node-topology">120</keepalive-delay>
</node>
```

Figure 25. Creating a netconf-connector payload

To effectively apply the configurations, the controller was then rebooted by running the command "*system:shutdown*" and then restarting the controller again using the command *"./karaf-0.8.2/bin/karaf"*. The successful connection is seen in Figure 26 below.



PUT http://10.10.204.85:8181/restcor
GET http://10.10.204.85:8181/restcor
GET http://10.10.204.85:8181/restcor
GET http://10.10.204.85:8181/restcor
No Environment
Send
Params
Authorization
Headers (1)
Body
Pre-request Script
Tests
Bulk Edit
Cookies (1)
Headers (5)
Test Results
Pretty
Raw
Preview
JSON

http://10.10.204.85:8181/restconf/operational/network-topology:network-topology/topology/topology-netconf/

GET http://10.10.204.85:8181/restconf/operational/network-topology:network-topology/topology/topology-netconf/

KEY	VALUE	DESCRIPTION
Authorization	Basic YWRtaW46YWRtaW4=	
Key	Value	Description

Status: 200 OK Time: 435 ms Size: 17.21 KB

```json
{
    "topology": [
        {
            "topology-id": "topology-netconf",
            "node": [
                {
                    "node-id": "callcenter_MX",
                    "netconf-node-topology:unavailable-capabilities": {},
                    "netconf-node-topology:available-capabilities": {},
                    "netconf-node-topology:host": "10.10.204.12",
                    "netconf-node-topology:connection-status": "connected",
                    "netconf-node-topology:port": 830
                }
            ]
        }
    ]
}
```

Figure 26. Connected netconf-connecter

© University of South Africa November 2018

CHAPTER 4

4.1 Results

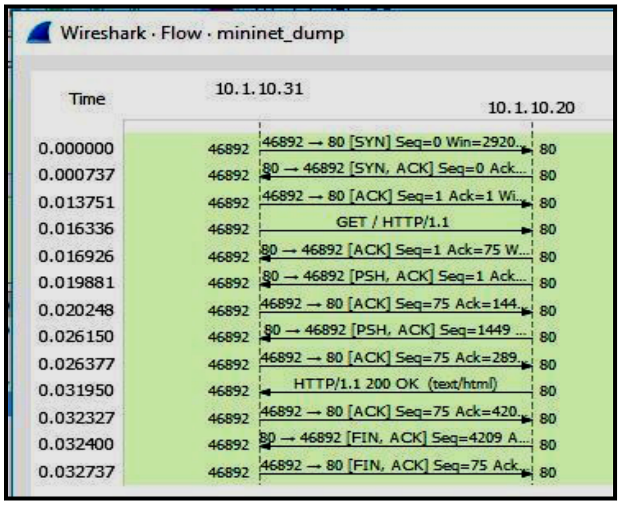
4.1.1 Results for Emulated Wireless Software Defined Networking

To test our simulated network functionality, a hypertext terminal protocol (http) website page request was sent to the virtual server on the BIG-IP application delivery controller from the virtual stations (Sta1 and Sta2) in Mininet connected to the access point (AP1) via simulated wireless interface inside Mininet-WiFi. The controller (C0) was able to create flow entries on the access point directing traffic to the external VLAN interface of the access point.

The request was received by the virtual server, which then used round robin to load balance the request to one of the web servers on the LAMP virtual machine. The web server returned the requested website in the hypertext terminal protocol (http) format to the requesting virtual stations using the simulated virtual network.

Figure 27 shows that from 0.016 s to about 0.032 s, a command to "GET" an (http) packet was issued from the virtual stations to the virtual server and then the (http) packet was received at the virtual stations signified by the "OK" message. If this is related to the information in Figure 28 we notice that the round-trip time initially decreases from (0.016 – 0.027 s) indicating that no data is travelling during this time and then increases from (0.027 – 0.033 s), indicating that the (http) packet is being transmitted relative to time. Figure 29 shows that the average data throughput remains constant at 600 bits/s over a time period of (0.016 – 0.032 s) which means that there are no dropped packets during this period and that the data travels over the round trip without dropping any packets.

© University of South Africa November 2018



```
Wireshark · Flow · mininet_dump

Time       10.1.10.31                                              10.1.10.20
0.000000   46892  46892 → 80 [SYN] Seq=0 Win=2920...               80
0.000737   46892  80 → 46892 [SYN, ACK] Seq=0 Ack...               80
0.013751   46892  46892 → 80 [ACK] Seq=1 Ack=1 Wi...               80
0.016336   46892  GET / HTTP/1.1                                   80
0.016926   46892  80 → 46892 [ACK] Seq=1 Ack=75 W...               80
0.019881   46892  80 → 46892 [PSH, ACK] Seq=1 Ack...               80
0.020248   46892  46892 → 80 [ACK] Seq=75 Ack=144...               80
0.026150   46892  80 → 46892 [PSH, ACK] Seq=1449 ...               80
0.026377   46892  46892 → 80 [ACK] Seq=75 Ack=289...               80
0.031950   46892  HTTP/1.1 200 OK (text/html)                      80
0.032327   46892  46892 → 80 [ACK] Seq=75 Ack=420...               80
0.032400   46892  80 → 46892 [FIN, ACK] Seq=4209 A...              80
0.032737   46892  46892 → 80 [FIN, ACK] Seq=75 Ack...              80
```

Figure 27. TCP flow

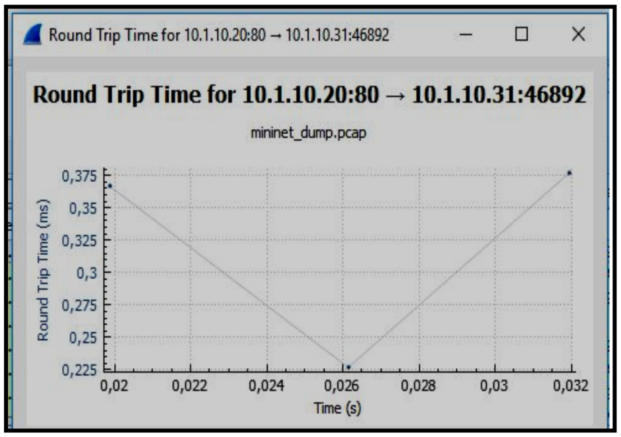


Figure 28. Round Trip Time

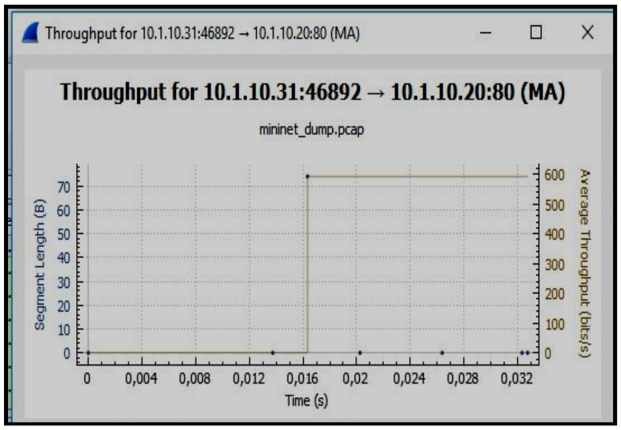


Figure 29. Throughput

© University of South Africa November 2018

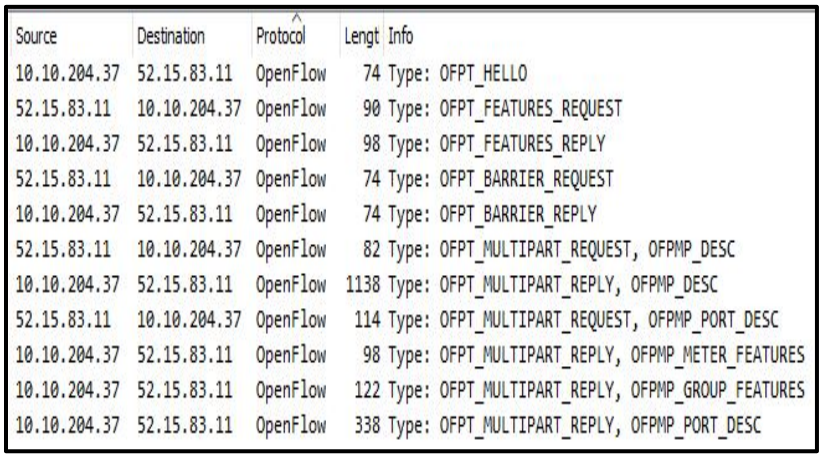
The result of this project is a successful communication between the Software Defined network devices inside the Mininet-WiFi emulator with the virtualized application delivery controller (load balancer) which is a Network Virtualized Function. This result shows Mininet and its wireless extension Mininet-WiFi as the suitable emulator which can be integrated into a realistic programmable networking environment using the combination of both SDN and NFV. These offer a portable, cost effective and easily deployable testing network, which can be run on a single computer.

© University of South Africa November 2018

4.1.2 Results for Emulated Wired Software Defined Networking

"Iperf" is a tool used to test the maximum bandwidth that can be achieved between two (2) network devices. "Iperf" sends test data between the defined network devices and measures the throughput, bitrate, loss and other parameters. To test the functionality of the created network, an TCP "Iperf" test between the host and the Linux server was performed.

Using Wireshark, communication between the switch and the controller was captured. In the beginning of the communication, OpenFlow Channel messages between the switch and the controller are observed. The OpenDaylight requested the identity and basic capabilities of the switch. The switch responded with the requested information as seen with the OFPT_HELLO, OFPT_FEATURES_REQUEST and OFPT_FEATURES_REPLY packets as seen in Figure 30 this communication can also be used to establish reliability of the network.



Source	Destination	Protocol	Lengt	Info
10.10.204.37	52.15.83.11	OpenFlow	74	Type: OFPT_HELLO
52.15.83.11	10.10.204.37	OpenFlow	90	Type: OFPT_FEATURES_REQUEST
10.10.204.37	52.15.83.11	OpenFlow	98	Type: OFPT_FEATURES_REPLY
52.15.83.11	10.10.204.37	OpenFlow	74	Type: OFPT_BARRIER_REQUEST
10.10.204.37	52.15.83.11	OpenFlow	74	Type: OFPT_BARRIER_REPLY
52.15.83.11	10.10.204.37	OpenFlow	82	Type: OFPT_MULTIPART_REQUEST, OFPMP_DESC
10.10.204.37	52.15.83.11	OpenFlow	1138	Type: OFPT_MULTIPART_REPLY, OFPMP_DESC
52.15.83.11	10.10.204.37	OpenFlow	114	Type: OFPT_MULTIPART_REQUEST, OFPMP_PORT_DESC
10.10.204.37	52.15.83.11	OpenFlow	98	Type: OFPT_MULTIPART_REPLY, OFPMP_METER_FEATURES
10.10.204.37	52.15.83.11	OpenFlow	122	Type: OFPT_MULTIPART_REPLY, OFPMP_GROUP_FEATURES
10.10.204.37	52.15.83.11	OpenFlow	338	Type: OFPT_MULTIPART_REPLY, OFPMP_PORT_DESC

Figure 30. OpenFlow Channel messages captured with Wireshark

Once the flows were added, the TCP "Iperf" test between host and the Linux server was successful and the OpenDaylight controller was able to create the network topology as seen in Figure 31 below on the ODL-DLUX network topology using "Postman".

© University of South Africa November 2018

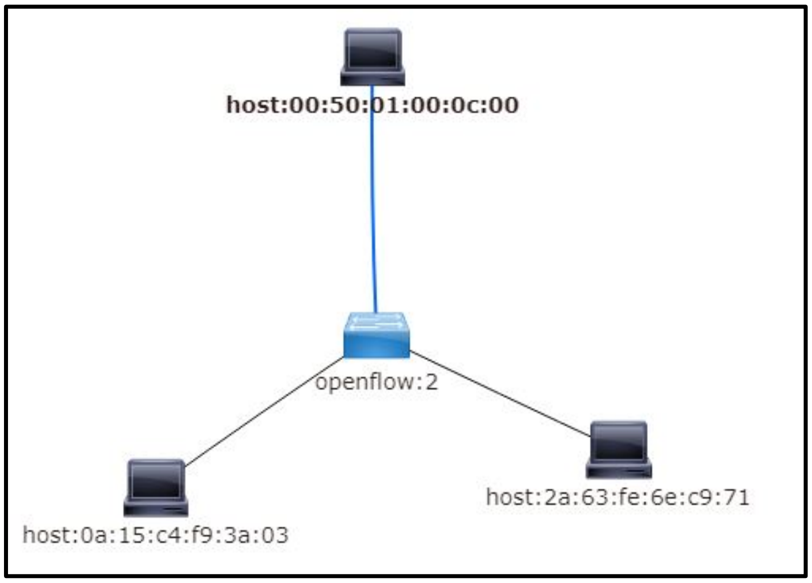
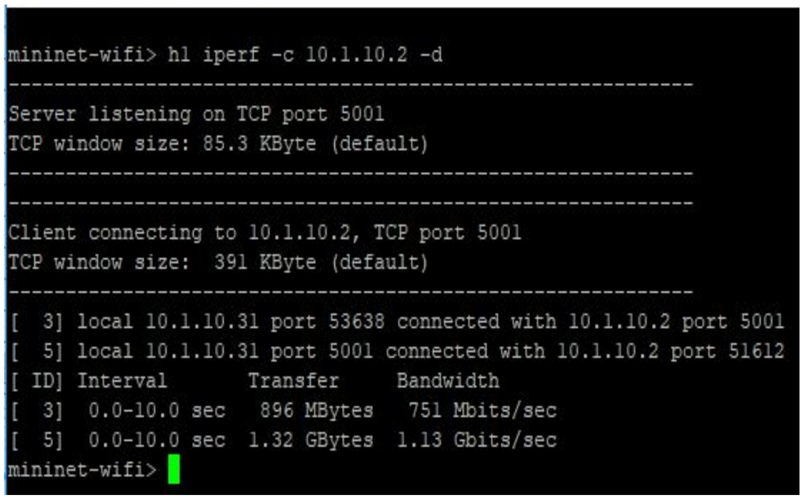


Figure 31. ODL-DLUX network topology

With the server using default TCP window size of 85.3 KBytes and the host using the default window size of 391 KByte, we were able to transfer 896 MBytes at a rate of 751 Mbits/sec from host to Linux server. From Linux server to host, 1.32 GBytes was transferred at a rate of 1.13 Gbits/sec as seen in the TCP "Iperf" results in Figure 32.



```
mininet-wifi> h1 iperf -c 10.1.10.2 -d
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.1.10.2, TCP port 5001
TCP window size:  391 KByte (default)
------------------------------------------------------------
[  3] local 10.1.10.31 port 53638 connected with 10.1.10.2 port 5001
[  5] local 10.1.10.31 port 5001 connected with 10.1.10.2 port 51612
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   896 MBytes   751 Mbits/sec
[  5]  0.0-10.0 sec  1.32 GBytes  1.13 Gbits/sec
mininet-wifi> 
```

Figure 32. "Iperf" test results

© University of South Africa November 2018

Another way of representing the Iperf test results using Wireshark is seen in Figure 33 below. Figure 27 shows the average throughput which is the actual throughput about 42 Mbits/s and the maximum bandwidth during the 90ms period of the Iperf test on the uplink and downlink.

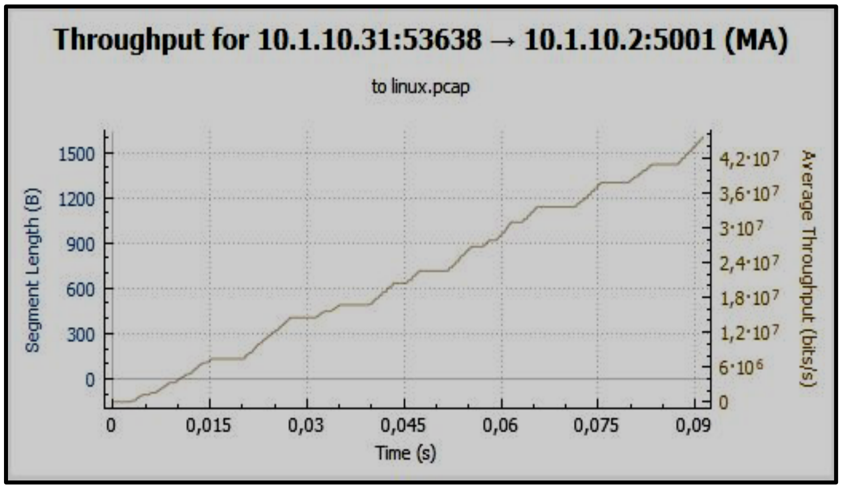


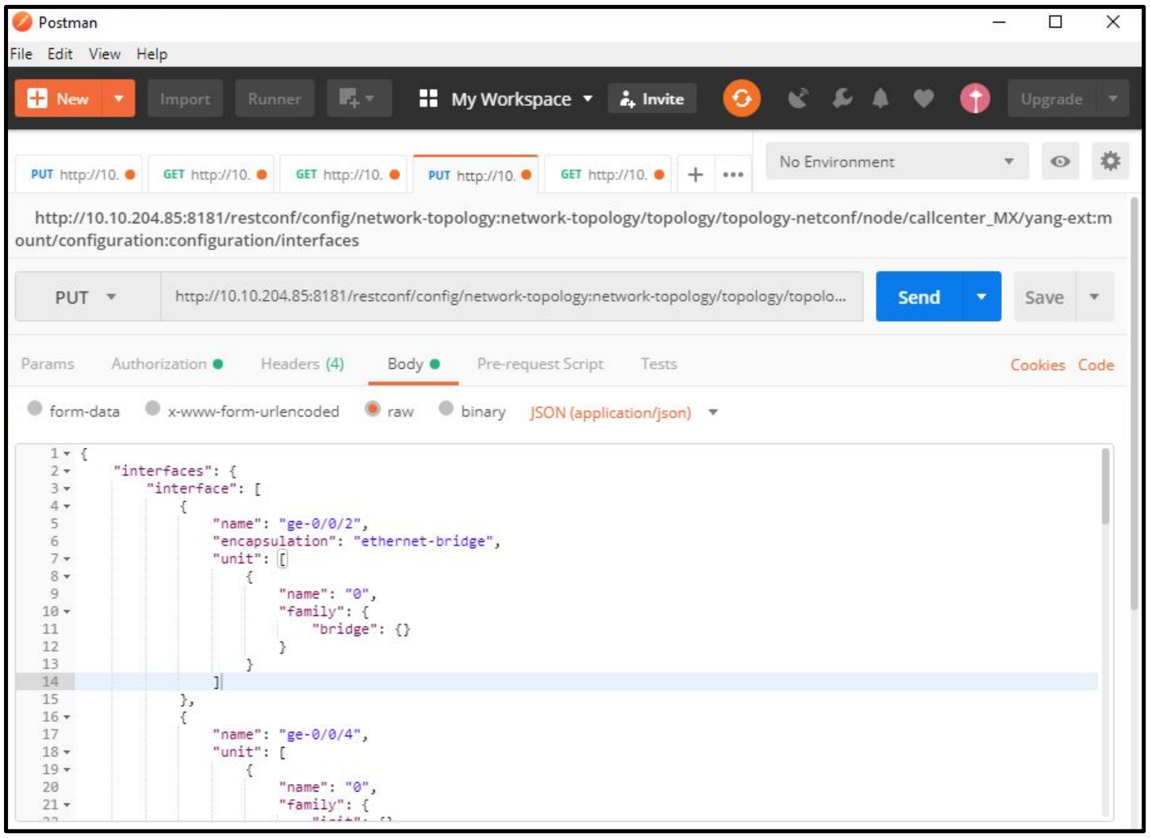
Figure 33. Average throughput over 90ms

© University of South Africa November 2018

4.1.3 Results for Intergrated Softwared Defined Network in Real-Life (EVE-NG)

4.1.3.1 Network Testing

With the communication between Juniper vMX and OpenDaylight controller established, the OpenDaylight can now program the device. To configure the interfaces, a “PUT” REST API request was sent with the interfaces values as shown in Figure 34.



http://10.10.204.85:8181/restconf/config/network-topology:network-topology/topology/topology-netconf/node/callcenter_MX/yang-ext:mount/configuration:configuration/interfaces

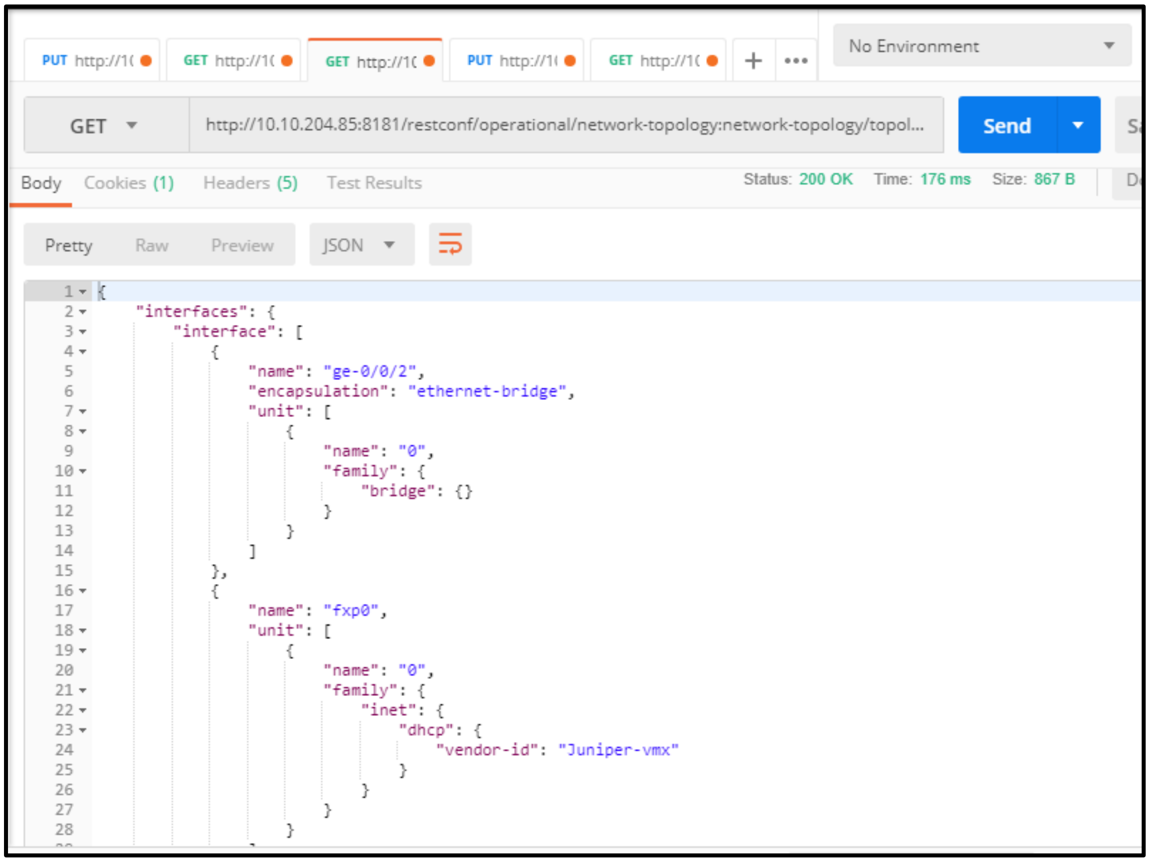
PUT http://10.10.204.85:8181/restconf/config/network-topology:network-topology/topology/topolo...

```
{
    "interfaces": {
        "interface": [
            {
                "name": "ge-0/0/2",
                "encapsulation": "ethernet-bridge",
                "unit": [
                    {
                        "name": "0",
                        "family": {
                            "bridge": {}
                        }
                    }
                ]
            },
            {
                "name": "ge-0/0/4",
                "unit": [
                    {
                        "name": "0",
                        "family": {
```

Figure 34. Configuring interfaces using netconf on "Postman"

© University of South Africa November 2018

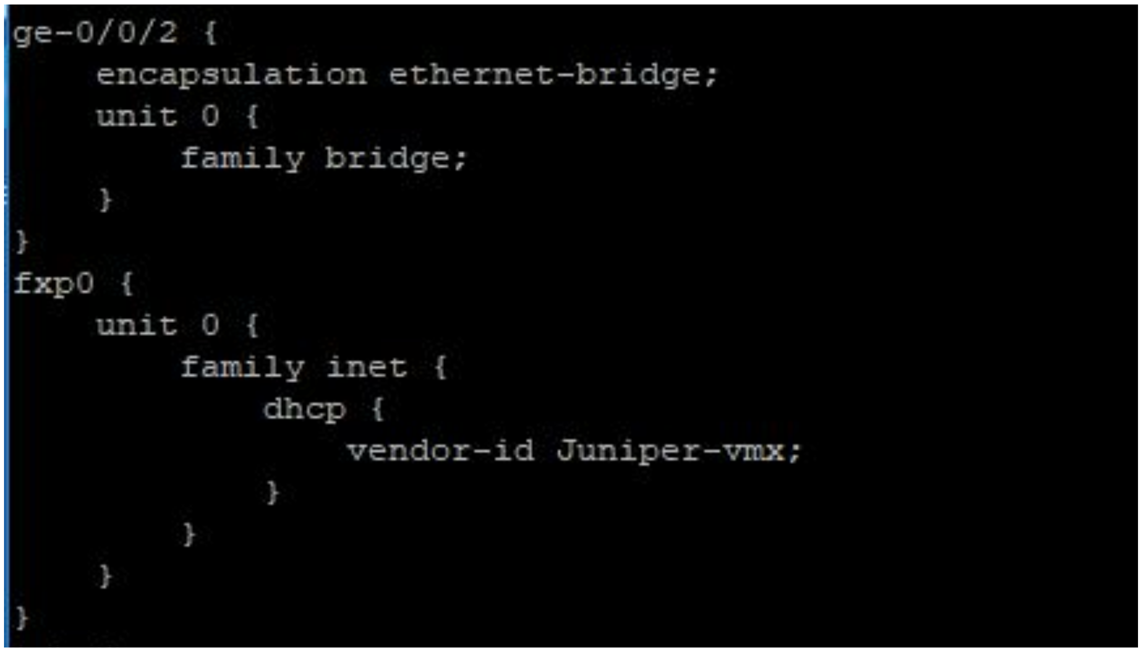
To see the configured interfaces, a “GET” REST API request was sent as seen in Figure 35. The results in Figure 35 where the same as the results obtained when running the command “*show interfaces*” direct from the Juniper device command line interface (CLI) as shown in Figure 36.



```
{
    "interfaces": {
        "interface": [
            {
                "name": "ge-0/0/2",
                "encapsulation": "ethernet-bridge",
                "unit": [
                    {
                        "name": "0",
                        "family": {
                            "bridge": {}
                        }
                    }
                ]
            },
            {
                "name": "fxp0",
                "unit": [
                    {
                        "name": "0",
                        "family": {
                            "inet": {
                                "dhcp": {
                                    "vendor-id": "Juniper-vmx"
                                }
                            }
                        }
                    }
```

Figure 35. Some of configured interfaces from restconf API

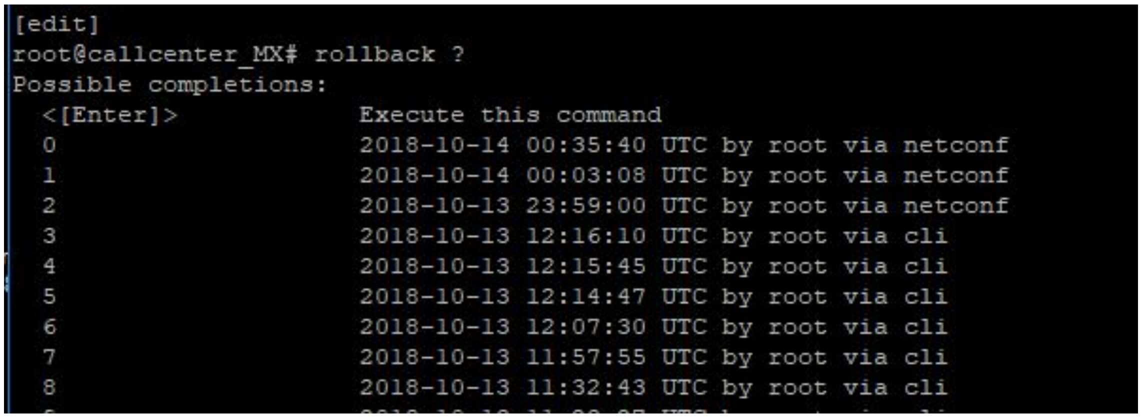
© University of South Africa November 2018



```
ge-0/0/2 {
    encapsulation ethernet-bridge;
    unit 0 {
        family bridge;
    }
}
fxp0 {
    unit 0 {
        family inet {
            dhcp {
                vendor-id Juniper-vmx;
            }
        }
    }
}
```

Figure 36. Some of configured interfaces from CLI

Juniper devices keeps tracks of every configuration change made. Among the information captured is the time and date of the change, the user making that change and the platform used. When checking the changes made on the device, we could see on Figure 37 that the last three (3) configurations made on the device, we done via netconf which is another proof that the OpenDaylight controller is indeed able to communicate and program the device.

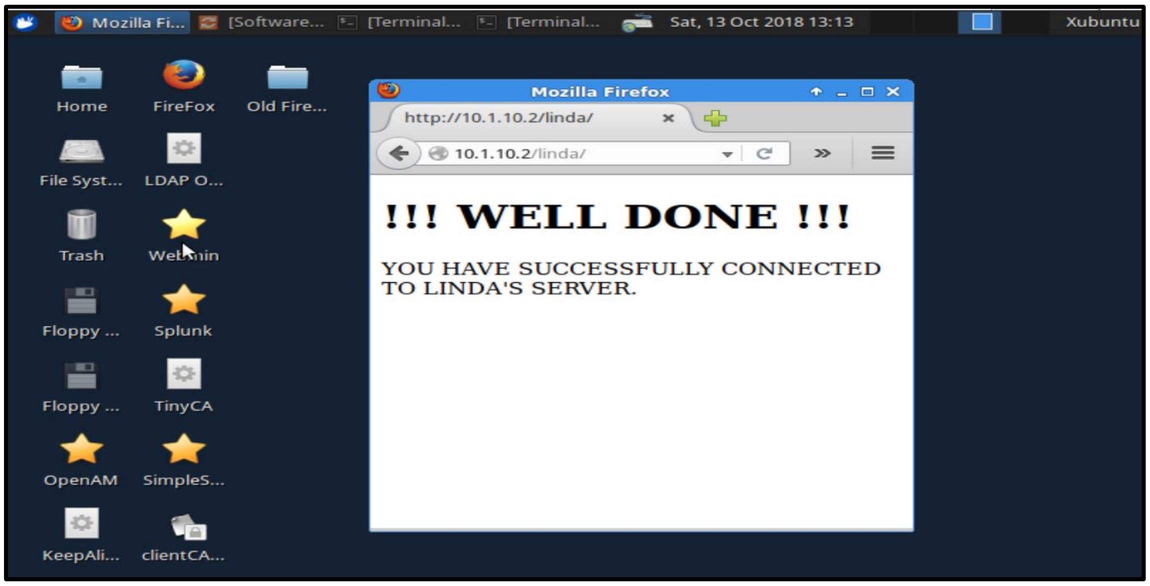


```
[edit]
root@callcenter_MX# rollback ?
Possible completions:
  <[Enter]>            Execute this command
  0                    2018-10-14 00:35:40 UTC by root via netconf
  1                    2018-10-14 00:03:08 UTC by root via netconf
  2                    2018-10-13 23:59:00 UTC by root via netconf
  3                    2018-10-13 12:16:10 UTC by root via cli
  4                    2018-10-13 12:15:45 UTC by root via cli
  5                    2018-10-13 12:14:47 UTC by root via cli
  6                    2018-10-13 12:07:30 UTC by root via cli
  7                    2018-10-13 11:57:55 UTC by root via cli
  8                    2018-10-13 11:32:43 UTC by root via cli
```

Figure 37. Configuration commit record

© University of South Africa November 2018

With the Juniper vMX configured, the connected ubuntu Personal Computers (PC) where able to communicate with the webserver as emulated on Mininet. Using the browser on PC1, we were able to receive the test web page hosted on the Linux server as seen in Figure 38.



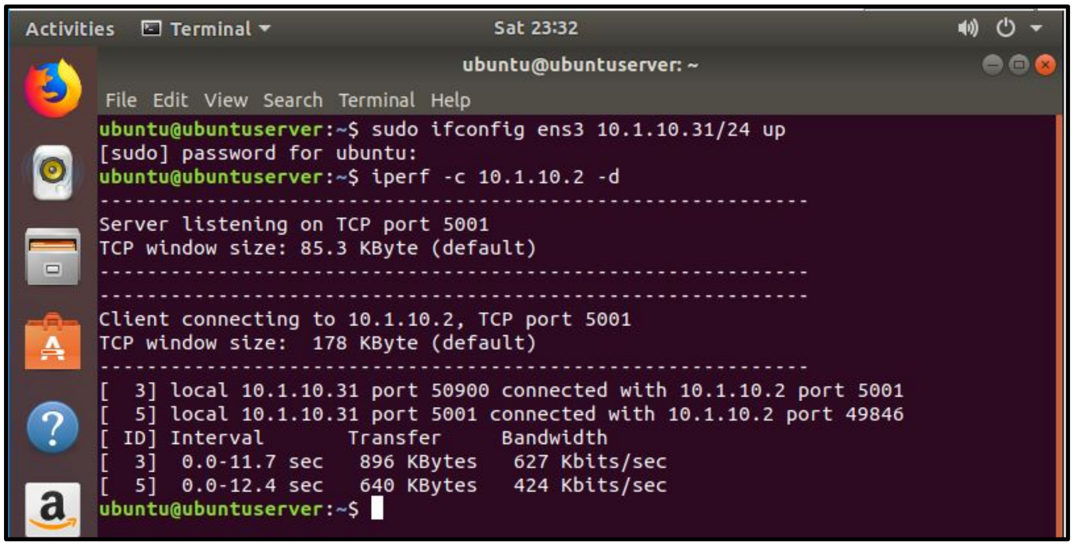
!!! WELL DONE !!!

YOU HAVE SUCCESSFULLY CONNECTED TO LINDA'S SERVER.

Figure 38. PC1 accessing webpage from Linux server

“Iperf” test was also performed between PC1 and Linux server. With the TCP window size of 178 KByte on PC1 and TCP window size of 85,3 KByte on Linux server, 896 Kbytes were successfully transferred from PC1 to Linux server at the rate of 627 Kbits/sec. For the reverse direction from Linux server to PC1, 640 KBytes was successfully transferred at the rate of 424 Kbits/sec as shown in Figure 39. Wireshark was used to capture the packets during the “Iperf” test. The throughput graph shows that the average throughput during the test was between 600000 bits/s and 800000 bits/s. Figure 40 below shows the throughput graph.

© University of South Africa November 2018



```
ubuntu@ubuntuserver:~$ sudo ifconfig ens3 10.1.10.31/24 up
[sudo] password for ubuntu:
ubuntu@ubuntuserver:~$ iperf -c 10.1.10.2 -d
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.1.10.2, TCP port 5001
TCP window size:  178 KByte (default)
------------------------------------------------------------
[  3] local 10.1.10.31 port 50900 connected with 10.1.10.2 port 5001
[  5] local 10.1.10.31 port 5001 connected with 10.1.10.2 port 49846
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-11.7 sec   896 KBytes   627 Kbits/sec
[  5]  0.0-12.4 sec   640 KBytes   424 Kbits/sec
ubuntu@ubuntuserver:~$
```

Figure 39. "Iperf test results between PC1 and Linux server

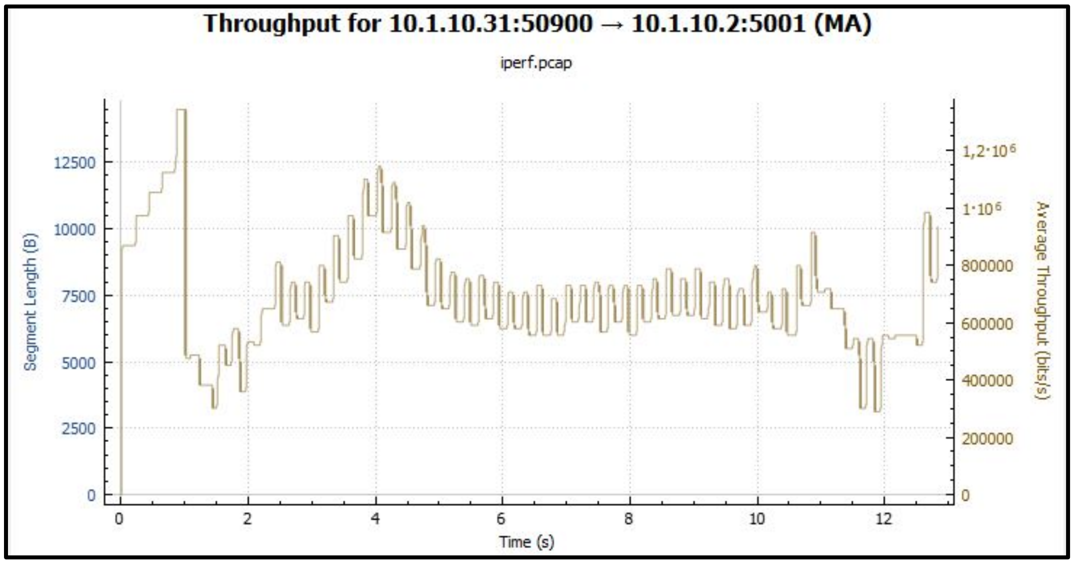


Figure 40. Throughput results during "Iperf" test

© University of South Africa November 2018

CHAPTER 5

5.1 Conclusion

1. Modern network technologies are moving away from traditional architecture, which is proprietary, and opting for more flexible and automated architectures, which support intelligent business models by allowing network programmability. The advent of Software Defined Networking and Network Functions Virtualization resulted in a need for an emulator, which can be able to emulate networks even in realistic programmable networking environments.
2. This project was able to answer the research question which was "Are Software Defined Networks emulated in Mininet able to communicate with real world network using current and future technologies like network virtualization and cloud networking?" as seen on the results of this project. The answer is "Yes, it was possible to make a Software Defined Network in Mininet to communicate with a real-life virtualized network using both wireless and wired means".

   Although developments of Mininet and it wireless extension Mininet-WiFi is an ongoing process, but results from this project are encouraging as it demonstrates that this emulator can be of benefit to even wireless network providers and organizations with virtualized network functions and cloud solutions. This project open doors for Mininet integration research with other technologies like Orchestrators, datacenters, Business Support Systems (BSS) and Operations Support Systems (OSS).
3. Mininet is a very useful tool to emulate Software Defined Networks. Combined with cloud networking technology, we were able to use it to model a realistic programmable network. Basic Python script was used to generate a Software Defined network in Mininet and OpenFlow protocol was used to facilitate communication between the virtual switch and the OpenDaylight controller in the wired scenario. This project lays the foundation for further studies on network automation and Software Defined Networking communication protocols such as the Northbound, Southbound, East and West bound Application Programmable Interfaces (APIs). Results were obtained for the communication in the form of data throughput in the wireless case and bandwidth throughput in the wired case.

© University of South Africa November 2018

5.2 Recommendations

1. To get full benefits of this study, it is recommended that the network being emulated in Mininet be fully defined on a python script and saved as a .py file. This will make the network portable and easily deployable in any computer with Mininet.

2. It is recommended to keep the python script up to date with the changes that takes place on the live network. This will make it easy when Mininet is used to test for live network expansion in a lab environment. This will also be useful for isolated real network troubleshooting as the network fault will be replicated in the emulated network in Mininet.

3. Mininet comes with the folder of example python scripts for different common networking tasks which can be used as the base of the network being emulated. Among the programs in this example folder is MiniEdit.py file. MiniEdit is a simple Graphical User Interface (GUI) editor which can be used to create network topologies. For a person who is not familiar with Mininet topologies, MiniEdit is highly recommended as a tool to create the topology. The topology created using MiniEdit can be saved as a .py file which makes it potable and easily deployable in any computer with Mininet.

4. Wireshark was used in this study to help with packets and traffic flow analysis between Mininet and the external network. To visually see how traffic flows within Mininet network components, I recommend the use of MiniNAM which was not part of the scope for this study. MiniNAM is a network animator for visualizing real time packets flows in Mininet. Example on how MiniNAM works can be found on http://www.cs.ucc.ie/~ak18/MiniNAM/ (link accessed in November 2018).

© University of South Africa November 2018

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker & J. Turner. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[2] Open Networking Foundation (ONF). The SDN Solutions Showcase: Technical Report & Analysis version Number 1.0. Paper presented at SDN & OpenFlow world congress, Dusseldorf, Germany. 2016.

[3] Open Networking Foundation (ONF). Software defined networking: The new norm for networks. White Paper. From https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf. 2012

[4] D. Kreutz, M. Fernando, V. Ramos, P. Esteves Verı´ssimo, C. Rothenberg, S. Azodolmolky & S. Uhlig. Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE Vol. 103, No. 1, January 2015

[5] N. Feamster, S. Woodrow, S. Sundaresan, H. Kim, R. Clark & A. Voellmy. The Past, Present, and Future of Software Defined Networking. 2012

[6] Y. Lin, D. Pitt, D. Hausheer, E. Johnson & Y. Lin. Software-Defined Networking: Standardization for Cloud Computing Second Wave. 2014

[7] W. Xia, Y. Wen, C. Foh, D. Niyato & H. Xie H. A Survey on Software-Defined Networking. 2015

[8] H. Hawilo, A. Shami & M. Mirahmadi. NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC). 2014

[9] A. Basta, W. Kellerer, M. Hoffmann, H. Morper & K. Hoffmann. Applying NFV and SDN to LTE mobile core gateways, the functions placement problem. 2014

[10] A. Lara, A. Kolasani, & B. Ramamurthy. Network innovation using openflow: A survey. IEEE Communications Surveys and Tutorials, Vol 16 No 1 pp 1-20. 2013.

[11] H. Shimonishi, Y. Takamiya, Y. Chiba, K. Sugyo, Y. Hatano, K. Sonoda, K. Suzuki, D Kotani, & I. Akiyoshi. Programmable network using OpenFlow for network researches and experiments. Proceedings of the Sixth International Conference on Mobile Computing and Ubiquitous Networking (pp.164-171). Okinawa, Japan. 2012

© University of South Africa November 2018

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, & J. Turner. OpenFlow: Enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 69–74. 2008

[13] A. Lara, A. Kolasani, & B. Ramamurthy. Network innovation using OpenFlow: A survey. IEEE Communications Surveys and Tutorials, 16(1), 493–512. 2014

[14] C. C. Machado, L. Z. Granville, A. Schaeffer-Filho, & J. A. Wickboldt, Towards SLA Policy Refinement for QoSManagement in Software-Defined Networking. IEEE 28th International Conference on Advanced Information Networking and Applications. 2014

[15] J. Mambretti, J. Chen & F. Yeh. Software-Defined Network Exchanges (SDXs): Architecture, Services, Capabilities, and Foundation Technologies. Proceedings of the 2014 26th International Teletraffic Congress (ITC). 2014

[16] B. Astuto, A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, & T Turletti. A Survey of Software-Defined Networking: Past,Present, and Future of Programmable Networks. hal-00825087, version 5- 19 Jan. 2014.

[17] I. Z. Bholebawa & U. D. Dalal. Design and Performance Analysis of OpenFlow-Enabled Network Topologies Using Mininet. International Journal of Computer and Communication Engineering. Volume 5, Number 6, November 2016.

[18] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan,N. Viljoen, M. Miller, & N. Rao. Are we ready for sdn? Implementation challenges for software-defined networks. Communications Magazine, IEEE, vol. 51, no. 7, pp. 36–43, 2013.

[19] O. W. Paper. Software-Defined Networking: The New Norm for Networks. Open Networking Foundation, Tech. Rep., April 2012.

[20] D. Thomas & N. K. Gray. SDN: Software Defined Networks, Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, First Edition, August 2013.

[21] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo & S. Shenker. A Framework for NFV Applications. 2015

[22] J. Matias, J. Garay, N. Toledo. Toward an SDN-enabled NFV architecture. 2015

[23] V. Antonenko. Global Network Modelling Based on Mininet Approach. 2013

[24] M. Handigol, B. Heller, V. Jeyakumar, B. Lantz & N. McKeown. Reproducible Network Experiments Using Container-Based Emulation. 2012

© University of South Africa November 2018

[25] B. Lantz & B. O'Connor B. A Mininet-based Virtual Testbed for Distributed SDN Development. 2015

[26] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz & N. McKeown. Reproducible Network Experiments Using Container-Based Emulation. CoNEXT'12, December 10–13, Nice, France. 2012

[27] Y. Huang, V. Jeyakumar, B. Lantz, B. O'Connor, N. Feamster, K. Winstein & A. Siyaraman. Teaching Computer Networking with Mininet. Wi-Fi: HHonors, SGC. 2014

[28] B. Lantz, B. Heller & N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. 2010

[29] D. Syrivelis, G. Parisis, D. Trosse, P. Flegkas, V. Sourlas, T. Korakis & L. Tassiulas. Pursuing a Software Defined Information-Centric Network. Paper presented at the European workshop on Software Defined Networks in Darmstadt, Germany from 25-26 October 2012

[30] A. Frömmgeny, D. Stohr, J. Fornoffy, W. Effelsberg & A. Buchmanny. Capture and Replay: Reproducible Network Experiments in Mininet. 2016

[31] D. Kumar & M. Sood. Software Defined Networks (S.D.N): Experimentation with Mininet Topologies. 2016

[32] S. Wang. Comparison of SDN OpenFlow network simulator and emulators: EstiNet vs Mininet. 2014

[33] J. Weerawardhana, N. Chandimal & A. Bandaranayake. SDN Testbed for Undergraduate Education. 2015

[34] H. Kim, J. Kim & Y. Ko. Developing a Cost-Effective OpenFlow Testbed for Small-Scale Software Defined Networking. 2014

[35] M. Gupta, J. Sommers & P. Barford. Fast, Accurate Simulation for SDN Prototyping. 2013

[36] B. Lantz & B. O'Connor. A Mininet-based Virtual Testbed for Distributed SDN Development. 2015

[37] J. Yan & D. Jin. VT-Mininet: Virtual-time-enabled Mininet for Scalable and Accurate Software-Define Network Emulation. SOSR2015, Santa Clara, CA, USA. June 17–18, 2015.

[38] F. Keti & S. Askar. Emulation of Software Defined Networks Using Mininet in Different Simulation Environments. 6th International Conference on Intelligent Systems, Modelling

© University of South Africa November 2018

and Simulation, Kuala Lumpur, 2015, pp. 205-210. 2016

[39] B. Lantz, B. Heller, & N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. in Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. ACM, 2010.

[40] K. K. Sharma & M. Sood. Mininet as a Container Based Emulator for Software Defined Networks. International Journal of Advanced Research in Computer Science and Software Engineering. Volume 4, Issue 12, December 2014

[41] R. Kharga, P. Bholebawa, I. Satyarthi, S. Gupta, & S. Kumari. OpenFlow technology: Ajourney of simulation tools. International Journal of Computer Network and Information Security, 6(11), 49–55. 2014

[42] R. Fontes, S. Afzal, S. Brito, M. Santos & C. Rothenberg. Mininet-WiFi: Emulating software-defined wireless network. 2015

[43] R. Fontes & C. Rothenberg. Mininet-WiFi: A Platform for Hybrid Physical-Virtual Software-Defined Wireless Networking Research. 2016

[44] Python at https://www.python.org/

[45] M. Sanner. Python: A Programming Language for Software Integration and Development. Article in Journal of Molecular Graphics and Modelling. November 1998

[46] Y. Han, T. Vachuska, A. Al-Shabibi, A, et al. ONVisor: Towards a scalable and flexible SDN-based network virtualization platform on ONOS. Int J Network Mgmt. 2018.

[47] X. Li, D. Li, J. Wan, C. Liu and M. Imran, "Adaptive Transmission Optimization in SDN-Based Industrial Internet of Things With Edge Computing," in IEEE Internet of Things Journal, vol. 5, no. 3, pp. 1351-1360, June 2018.

[48] V. Jara and Y. Shayan, "Latency Measurement in an SDN Network Using a POX Controller," 2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE), Quebec City, QC, 2018, pp. 1-5.

[49] R. Jawaharan, P. M. Mohan, T. Das and M. Gurusamy, "Empirical Evaluation of SDN Controllers Using Mininet/Wireshark and Comparison with Cbench," 2018 27th International Conference on Computer Communication and Networks (ICCCN), Hangzhou, 2018, pp. 1-2.

© University of South Africa November 2018

[50] K. Wang, S. Kao and M. Kao, "An efficient load adjustment for balancing multiple controllers in reliable SDN systems," 2018 IEEE International Conference on Applied System Invention (ICASI), Chiba, 2018, pp. 593-596.

[51] M. Erel, E. Teoman, Y. Özçevik, G. Seçinti and B. Canberk, "Scalability analysis and flow admission control in mininet-based SDN environment," 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN), San Francisco, CA, 2015, pp. 18-19.

[52] C. Fancy and M. Pushpalatha, "Performance evaluation of SDN controllers POX and floodlight in mininet emulation environment," 2017 International Conference on Intelligent Sustainable Systems (ICISS), Palladam, 2017, pp. 695-699.

[53] M. T.BAH, A. Azzouni, M. T. Nguyen and G. Pujolle, "Topology Discovery Performance Evaluation of OpenDaylight and ONOS Controllers," 2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN), Paris, France, 2019, pp. 285-291.

[54] A. C. P. Bonifacio, P. J. C. Galo, J. M. D. Lopena, J. F. Villaverde and G. V. Magwili, "Resilient Network and Data Transmission between Philippines and Vietnam via Software-Defined Network using OpenDaylight Controller and Mininet," 2018 IEEE 10th International Conference on Humanoid, Nanotechnology, Information Technology,Communication and Control, Environment and Management (HNICEM), Baguio City, Philippines, 2018, pp. 1-5.

[55] OpenDaylight Consortium. http://www.opendaylight.org.

[56] W. Zhou, L. Li, M. Luo & W. Chou. REST API Design Patterns for SDN Northbound API. 28th International Conference on Advanced Information Networking and Applications Workshops, Victoria, BC, 2014, pp. 358-365. 2014

[57] L. Li, W. Chou, W. Zhou & M. Luo. Design Patterns and Extensibility of REST API for Networking Applications. IEEE Transactions on Network and Service Management, vol. 13, no. 1, pp. 154-167, March 2016.

[58] M. Brandt, R. Khondoker, R. Marx & K. Bayarou. Security analysis of software defined networking protocols - OpenFlow, OF-Config and OVSDB. Paper presented at Fifth IEEE International Conference on Communications and Electronics, ICCE 2014, July 30 - August 1, 2014

© University of South Africa November 2018

[59] J. Schonwalder, M. Bjorklund & P. Shafer. Network configuration management using NETCONF and YANG. IEEE Communications Magazine, vol. 48, no. 9, pp. 166-173, Sept. 2010

[60] Z. K. Khattak, M. Awais & A. Iqbal. Performance evaluation of OpenDaylight SDN controller. 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, 2014, pp. 671-676. 2014

© University of South Africa November 2018

APPENDICES

Appendix A: MININET PYTHON SCRIPT

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

"""
This program creates a network consisting of one (1) remote controller, one
(1) Open Virtual Switch (OVS) in kernel mode and two (2) network hosts.

The created switch has the learning switch behaviour. It learns which MAC
address is associated with which port and forwards traffic destined for a MAC
address to the associated port once it has learned which port is associated
with that MAC address.

"""


# We start by importing the clases we use on this program.

import os
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.link import TCLink, Intf
from mininet.log import setLogLevel, info
from subprocess import call

# Define the function to use to create custom topology.
def myNetwork():

# Inform Mininet not to create default topology but a network on the
10.1.10.0 subnetwork.
    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.1.10.0/24')

# Creates a controller instance pointing to the OpenDaylight controller
    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',
                         controller=RemoteController ,
                         ip='52.15.83.11',
                         protocol='tcp',
                         port=6633)
```

© University of South Africa November 2018

```
# Creating a learning Open Virtual Switch in kernel mode
    info( '*** Add switches\n')
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)


# Creating two network hosts
    info( '*** Add hosts\n')
    h1 = net.addHost('h1', ip='10.1.10.31/24', defaultRoute='via
10.1.10.10.1/24')
    h2 = net.addHost('h2', ip='10.1.10.32/24', defaultRoute='via
10.1.10.10.1/24')

# Creating links which the controller adds to the switch as flows
    info( '*** Add links\n')
    net.addLink(h1, s2)
    net.addLink(h2, s2)

# Building and starting the network
    info( '*** Starting network\n')
    net.build()
    info( '***Starting controllers\n')
    for controller in net.controllers:
        controller.start()
    info( '*** Starting switches\n')
    net.get('s2').start([c0])
    os.popen('ovs-vsctl add-port s2 ens3')

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

    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```

© University of South Africa November 2018

Appendix B: MININET NET CLASS

```
"""

    Mininet: A simple networking testbed for OpenFlow/SDN!

author: Bob Lantz (rlantz@cs.stanford.edu)
author: Brandon Heller (brandonh@stanford.edu)

Mininet creates scalable OpenFlow test networks by using
process-based virtualization and network namespaces.

Simulated hosts are created as processes in separate network
namespaces. This allows a complete OpenFlow network to be simulated on
top of a single Linux kernel.

Each host has:

A virtual console (pipes to a shell)
A virtual interfaces (half of a veth pair)
A parent shell (and possibly some child processes) in a namespace

Hosts have a network interface which is configured via ifconfig/ip
link/etc.

This version supports both the kernel and user space datapaths
from the OpenFlow reference implementation (openflowswitch.org)
as well as OpenVSwitch (openvswitch.org.)

In kernel datapath mode, the controller and switches are simply
processes in the root namespace.

Kernel OpenFlow datapaths are instantiated using dpctl(8), and are
attached to the one side of a veth pair; the other side resides in the
host namespace. In this mode, switch processes can simply connect to the
controller via the loopback interface.

In user datapath mode, the controller and switches can be full-service
nodes that live in their own network namespaces and have management
interfaces and IP addresses on a control network (e.g. 192.168.123.1,
currently routed although it could be bridged.)

In addition to a management interface, user mode switches also have
several switch interfaces, halves of veth pairs whose other halves
reside in the host nodes that the switches are connected to.

Consistent, straightforward naming is important in order to easily
identify hosts, switches and controllers, both from the CLI and
from program code. Interfaces are named to make it easy to identify
which interfaces belong to which node.

The basic naming scheme is as follows:

    Host nodes are named h1-hN
    Switch nodes are named s1-sN
    Controller nodes are named c0-cN
    Interfaces are named {nodename}-eth0 .. {nodename}-ethN

Note: If the network topology is created using mininet.topo, then
node numbers are unique among hosts and switches (e.g. we have
h1..hN and SN..SN+M) and also correspond to their default IP addresses
of 10.x.y.z/8 where x.y.z is the base-256 representation of N for
hN. This mapping allows easy determination of a node's IP
address from its name, e.g. h1 -> 10.0.0.1, h257 -> 10.0.1.1.

Note also that 10.0.0.1 can often be written as 10.1 for short, e.g.
"ping 10.1" is equivalent to "ping 10.0.0.1".

Currently we wrap the entire network in a 'mininet' object, which
constructs a simulated network based on a network topology created
using a topology object (e.g. LinearTopo) from mininet.topo or
mininet.topolib, and a Controller which the switches will connect
to. Several configuration options are provided for functions such as
automatically setting MAC addresses, populating the ARP table, or
even running a set of terminals to allow direct interaction with nodes.

After the network is created, it can be started using start(), and a
variety of useful tasks maybe performed, including basic connectivity
and bandwidth tests and running the mininet CLI.

Once the network is up and running, test code can easily get access
to host and switch objects which can then be used for arbitrary
experiments, typically involving running a series of commands on the
hosts.

After all desired tests or activities have been completed, the stop()
method may be called to shut down the network.

"""

import os
import re
import select
import signal
import random

from time import sleep
from itertools import chain, groupby
from math import ceil

from mininet.cli import CLI
from mininet.log import info, error, debug, output, warn
from mininet.node import ( Node, Host, OVSKernelSwitch, DefaultController,
                           Controller )
from mininet.nodelib import NAT
from mininet.link import Link, Intf
from mininet.util import ( quietRun, fixLimits, numCores, ensureRoot,
                           macColonHex, ipStr, ipParse, netParse, ipAdd,
                           waitListening )
from mininet.term import cleanUpScreens, makeTerms
```

© University of South Africa November 2018

```
# Mininet version: should be consistent with README and LICENSE
VERSION = "2.3.0d1"

class Mininet( object ):
    "Network emulation with hosts spawned in network namespaces."

    def __init__( self, topo=None, switch=OVSKernelSwitch, host=Host,
                  controller=DefaultController, link=Link, intf=Intf,
                  build=True, xterms=False, cleanup=False,
ipBase='10.0.0.0/8',
                  inNamespace=False,
                  autoSetMacs=False, autoStaticArp=False, autoPinCpus=False,
                  listenPort=None, waitConnected=False ):
        """Create Mininet object.
           topo: Topo (topology) object or None
           switch: default Switch class
           host: default Host class/constructor
           controller: default Controller class/constructor
           link: default Link class/constructor
           intf: default Intf class/constructor
           ipBase: base IP address for hosts,
           build: build now from topo?
           xterms: if build now, spawn xterms?
           cleanup: if build now, cleanup before creating?
           inNamespace: spawn switches and controller in net namespaces?
           autoSetMacs: set MAC addrs automatically like IP addresses?
           autoStaticArp: set all-pairs static MAC addrs?
           autoPinCpus: pin hosts to (real) cores (requires CPULimitedHost)?
           listenPort: base listening port to open; will be incremented for
               each additional switch in the net if inNamespace=False"""
        self.topo = topo
        self.switch = switch
        self.host = host
        self.controller = controller
        self.link = link
        self.intf = intf
        self.ipBase = ipBase
        self.ipBaseNum, self.prefixLen = netParse( self.ipBase )
        hostIP = ( 0xffffffff >> self.prefixLen ) & self.ipBaseNum
        # Start for address allocation
        self.nextIP = hostIP if hostIP > 0 else 1
        self.inNamespace = inNamespace
        self.xterms = xterms
        self.cleanup = cleanup
        self.autoSetMacs = autoSetMacs
        self.autoStaticArp = autoStaticArp
        self.autoPinCpus = autoPinCpus
        self.numCores = numCores()
        self.nextCore = 0  # next core for pinning hosts to CPUs
        self.listenPort = listenPort
        self.waitConn = waitConnected

        self.hosts = []
        self.switches = []
        self.controllers = []
        self.links = []
```

© University of South Africa November 2018

```
        self.nameToNode = {}  # name to Node (Host/Switch) objects

        self.terms = []  # list of spawned xterm processes

        Mininet.init()  # Initialize Mininet if necessary

        self.built = False
        if topo and build:
            self.build()

    def waitConnected( self, timeout=None, delay=.5 ):
        """wait for each switch to connect to a controller,
           up to 5 seconds
           timeout: time to wait, or None to wait indefinitely
           delay: seconds to sleep per iteration
           returns: True if all switches are connected"""
        info( '*** Waiting for switches to connect\n' )
        time = 0
        remaining = list( self.switches )
        while True:
            for switch in tuple( remaining ):
                if switch.connected():
                    info( '%s ' % switch )
                    remaining.remove( switch )
            if not remaining:
                info( '\n' )
                return True
            if timeout is not None and time > timeout:
                break
            sleep( delay )
            time += delay
        warn( 'Timed out after %d seconds\n' % time )
        for switch in remaining:
            if not switch.connected():
                warn( 'Warning: %s is not connected to a controller\n'
                      % switch.name )
            else:
                remaining.remove( switch )
        return not remaining

    def addHost( self, name, cls=None, **params ):
        """Add host.
           name: name of host to add
           cls: custom host class/constructor (optional)
           params: parameters for host
           returns: added host"""
        # Default IP and MAC addresses
        defaults = { 'ip': ipAdd( self.nextIP,
                                  ipBaseNum=self.ipBaseNum,
                                  prefixLen=self.prefixLen ) +
                                  '/%s' % self.prefixLen }
        if self.autoSetMacs:
            defaults[ 'mac' ] = macColonHex( self.nextIP )
        if self.autoPinCpus:
            defaults[ 'cores' ] = self.nextCore
```

© University of South Africa November 2018

```
            self.nextCore = ( self.nextCore + 1 ) % self.numCores
        self.nextIP += 1
        defaults.update( params )
        if not cls:
            cls = self.host
        h = cls( name, **defaults )
        self.hosts.append( h )
        self.nameToNode[ name ] = h
        return h

    def delNode( self, node, nodes=None):
        """Delete node
           node: node to delete
           nodes: optional list to delete from (e.g. self.hosts)"""
        if nodes is None:
            nodes = ( self.hosts if node in self.hosts else
                      ( self.switches if node in self.switches else
                        ( self.controllers if node in self.controllers else
                          [] ) ) )
        node.stop( deleteIntfs=True )
        node.terminate()
        nodes.remove( node )
        del self.nameToNode[ node.name ]

    def delHost( self, host ):
        "Delete a host"
        self.delNode( host, nodes=self.hosts )

    def addSwitch( self, name, cls=None, **params ):
        """Add switch.
           name: name of switch to add
           cls: custom switch class/constructor (optional)
           returns: added switch
           side effect: increments listenPort ivar ."""
        defaults = { 'listenPort': self.listenPort,
                     'inNamespace': self.inNamespace }
        defaults.update( params )
        if not cls:
            cls = self.switch
        sw = cls( name, **defaults )
        if not self.inNamespace and self.listenPort:
            self.listenPort += 1
        self.switches.append( sw )
        self.nameToNode[ name ] = sw
        return sw

    def delSwitch( self, switch ):
        "Delete a switch"
        self.delNode( switch, nodes=self.switches )

    def addController( self, name='c0', controller=None, **params ):
        """Add controller.
           controller: Controller class"""
        # Get controller class
        if not controller:
            controller = self.controller
```

© University of South Africa November 2018

```
        # Construct new controller if one is not given
        if isinstance( name, Controller ):
            controller_new = name
            # Pylint thinks controller is a str()
            # pylint: disable=maybe-no-member
            name = controller_new.name
            # pylint: enable=maybe-no-member
        else:
            controller_new = controller( name, **params )
        # Add new controller to net
        if controller_new:  # allow controller-less setups
            self.controllers.append( controller_new )
            self.nameToNode[ name ] = controller_new
        return controller_new

    def delController( self, controller ):
        """Delete a controller
           Warning - does not reconfigure switches, so they
           may still attempt to connect to it!"""
        self.delNode( controller )

    def addNAT( self, name='nat0', connect=True, inNamespace=False,
                **params):
        """Add a NAT to the Mininet network
           name: name of NAT node
           connect: switch to connect to | True (s1) | None
           inNamespace: create in a network namespace
           params: other NAT node params, notably:
               ip: used as default gateway address"""
        nat = self.addHost( name, cls=NAT, inNamespace=inNamespace,
                            subnet=self.ipBase, **params )
        # find first switch and create link
        if connect:
            if not isinstance( connect, Node ):
                # Use first switch if not specified
                connect = self.switches[ 0 ]
            # Connect the nat to the switch
            self.addLink( nat, connect )
            # Set the default route on hosts
            natIP = nat.params[ 'ip' ].split('/')[ 0 ]
            for host in self.hosts:
                if host.inNamespace:
                    host.setDefaultRoute( 'via %s' % natIP )
        return nat

    # BL: We now have four ways to look up nodes
    # This may (should?) be cleaned up in the future.
    def getNodeByName( self, *args ):
        "Return node(s) with given name(s)"
        if len( args ) == 1:
            return self.nameToNode[ args[ 0 ] ]
        return [ self.nameToNode[ n ] for n in args ]

    def get( self, *args ):
        "Convenience alias for getNodeByName"
        return self.getNodeByName( *args )
```

© University of South Africa November 2018

```
    # Even more convenient syntax for node lookup and iteration
    def __getitem__( self, key ):
        "net[ name ] operator: Return node with given name"
        return self.nameToNode[ key ]

    def __delitem__( self, key ):
        "del net[ name ] operator - delete node with given name"
        self.delNode( self.nameToNode[ key ] )

    def __iter__( self ):
        "return iterator over node names"
        for node in chain( self.hosts, self.switches, self.controllers ):
            yield node.name

    def __len__( self ):
        "returns number of nodes in net"
        return ( len( self.hosts ) + len( self.switches ) +
                 len( self.controllers ) )

    def __contains__( self, item ):
        "returns True if net contains named node"
        return item in self.nameToNode

    def keys( self ):
        "return a list of all node names or net's keys"
        return list( self )

    def values( self ):
        "return a list of all nodes or net's values"
        return [ self[name] for name in self ]

    def items( self ):
        "return (key,value) tuple list for every node in net"
        return zip( self.keys(), self.values() )

    @staticmethod
    def randMac():
        "Return a random, non-multicast MAC address"
        return macColonHex( random.randint(1, 2**48 - 1) & 0xfeffffffffff |
                            0x020000000000 )

    def addLink( self, node1, node2, port1=None, port2=None,
                 cls=None, **params ):
        """"Add a link from node1 to node2
            node1: source node (or name)
            node2: dest node (or name)
            port1: source port (optional)
            port2: dest port (optional)
            cls: link class (optional)
            params: additional link params (optional)
            returns: link object"""
        # Accept node objects or names
        node1 = node1 if not isinstance( node1, basestring ) else self[ node1
]
```

© University of South Africa November 2018

```
        node2 = node2 if not isinstance( node2, basestring ) else self[ node2
]
        options = dict( params )
        # Port is optional
        if port1 is not None:
            options.setdefault( 'port1', port1 )
        if port2 is not None:
            options.setdefault( 'port2', port2 )
        if self.intf is not None:
            options.setdefault( 'intf', self.intf )
        # Set default MAC - this should probably be in Link
        options.setdefault( 'addr1', self.randMac() )
        options.setdefault( 'addr2', self.randMac() )
        cls = self.link if cls is None else cls
        link = cls( node1, node2, **options )
        self.links.append( link )
        return link

    def delLink( self, link ):
        "Remove a link from this network"
        link.delete()
        self.links.remove( link )

    def linksBetween( self, node1, node2 ):
        "Return Links between node1 and node2"
        return [ link for link in self.links
                 if ( node1, node2 ) in (
                    ( link.intf1.node, link.intf2.node ),
                    ( link.intf2.node, link.intf1.node ) ) ]

    def delLinkBetween( self, node1, node2, index=0, allLinks=False ):
        """Delete link(s) between node1 and node2
           index: index of link to delete if multiple links (0)
           allLinks: ignore index and delete all such links (False)
           returns: deleted link(s)"""
        links = self.linksBetween( node1, node2 )
        if not allLinks:
            links = [ links[ index ] ]
        for link in links:
            self.delLink( link )
        return links

    def configHosts( self ):
        "Configure a set of hosts."
        for host in self.hosts:
            info( host.name + ' ' )
            intf = host.defaultIntf()
            if intf:
                host.configDefault()
            else:
                # Don't configure nonexistent intf
                host.configDefault( ip=None, mac=None )
            # You're low priority, dude!
            # BL: do we want to do this here or not?
            # May not make sense if we have CPU lmiting...
            # quietRun( 'renice +18 -p ' + repr( host.pid ) )
            # This may not be the right place to do this, but
            # it needs to be done somewhere.
        info( '\n' )

    def buildFromTopo( self, topo=None ):
        """Build mininet from a topology object
           At the end of this function, everything should be connected
           and up."""

        # Possibly we should clean up here and/or validate
        # the topo
        if self.cleanup:
            pass

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

        if not self.controllers and self.controller:
            # Add a default controller
            info( '*** Adding controller\n' )
            classes = self.controller
            if not isinstance( classes, list ):
                classes = [ classes ]
            for i, cls in enumerate( classes ):
                # Allow Controller objects because nobody understands
partial()
                if isinstance( cls, Controller ):
                    self.addController( cls )
                else:
                    self.addController( 'c%d' % i, cls )

        info( '*** Adding hosts:\n' )
        for hostName in topo.hosts():
            self.addHost( hostName, **topo.nodeInfo( hostName ) )
            info( hostName + ' ' )

        info( '\n*** Adding switches:\n' )
        for switchName in topo.switches():
            # A bit ugly: add batch parameter if appropriate
            params = topo.nodeInfo( switchName)
            cls = params.get( 'cls', self.switch )
            if hasattr( cls, 'batchStartup' ):
                params.setdefault( 'batch', True )
            self.addSwitch( switchName, **params )
            info( switchName + ' ' )

        info( '\n*** Adding links:\n' )
        for srcName, dstName, params in topo.links(
                sort=True, withInfo=True ):
            self.addLink( **params )
            info( '(%s, %s) ' % ( srcName, dstName ) )

        info( '\n' )

    def configureControlNetwork( self ):
        "Control net config hook: override in subclass"
        raise Exception( 'configureControlNetwork: '
```

© University of South Africa November 2018

```
                         'should be overriden in subclass', self )

    def build( self ):
        "Build mininet."
        if self.topo:
            self.buildFromTopo( self.topo )
        if self.inNamespace:
            self.configureControlNetwork()
        info( '*** Configuring hosts\n' )
        self.configHosts()
        if self.xterms:
            self.startTerms()
        if self.autoStaticArp:
            self.staticArp()
        self.built = True

    def startTerms( self ):
        "Start a terminal for each node."
        if 'DISPLAY' not in os.environ:
            error( "Error starting terms: Cannot connect to display\n" )
            return
        info( "*** Running terms on %s\n" % os.environ[ 'DISPLAY' ] )
        cleanUpScreens()
        self.terms += makeTerms( self.controllers, 'controller' )
        self.terms += makeTerms( self.switches, 'switch' )
        self.terms += makeTerms( self.hosts, 'host' )

    def stopXterms( self ):
        "Kill each xterm."
        for term in self.terms:
            os.kill( term.pid, signal.SIGKILL )
        cleanUpScreens()

    def staticArp( self ):
        "Add all-pairs ARP entries to remove the need to handle broadcast."
        for src in self.hosts:
            for dst in self.hosts:
                if src != dst:
                    src.setARP( ip=dst.IP(), mac=dst.MAC() )

    def start( self ):
        "Start controller and switches."
        if not self.built:
            self.build()
        info( '*** Starting controller\n' )
        for controller in self.controllers:
            info( controller.name + ' ')
            controller.start()
        info( '\n' )
        info( '*** Starting %s switches\n' % len( self.switches ) )
        for switch in self.switches:
            info( switch.name + ' ')
            switch.start( self.controllers )
        started = {}
        for swclass, switches in groupby(
                sorted( self.switches, key=type ), type ):
```

© University of South Africa November 2018

```
            switches = tuple( switches )
            if hasattr( swclass, 'batchStartup' ):
                success = swclass.batchStartup( switches )
                started.update( { s: s for s in success } )
        info( '\n' )
        if self.waitConn:
            self.waitConnected()

    def stop( self ):
        "Stop the controller(s), switches and hosts"
        info( '*** Stopping %i controllers\n' % len( self.controllers ) )
        for controller in self.controllers:
            info( controller.name + ' ' )
            controller.stop()
        info( '\n' )
        if self.terms:
            info( '*** Stopping %i terms\n' % len( self.terms ) )
            self.stopXterms()
        info( '*** Stopping %i links\n' % len( self.links ) )
        for link in self.links:
            info( '.' )
            link.stop()
        info( '\n' )
        info( '*** Stopping %i switches\n' % len( self.switches ) )
        stopped = {}
        for swclass, switches in groupby(
                sorted( self.switches, key=type ), type ):
            switches = tuple( switches )
            if hasattr( swclass, 'batchShutdown' ):
                success = swclass.batchShutdown( switches )
                stopped.update( { s: s for s in success } )
        for switch in self.switches:
            info( switch.name + ' ' )
            if switch not in stopped:
                switch.stop()
            switch.terminate()
        info( '\n' )
        info( '*** Stopping %i hosts\n' % len( self.hosts ) )
        for host in self.hosts:
            info( host.name + ' ' )
            host.terminate()
        info( '\n*** Done\n' )

    def run( self, test, *args, **kwargs ):
        "Perform a complete start/test/stop cycle."
        self.start()
        info( '*** Running test\n' )
        result = test( *args, **kwargs )
        self.stop()
        return result

    def monitor( self, hosts=None, timeoutms=-1 ):
        """Monitor a set of hosts (or all hosts by default),
           and return their output, a line at a time.
           hosts: (optional) set of hosts to monitor
           timeoutms: (optional) timeout value in ms
```

© University of South Africa November 2018

```
           returns: iterator which returns host, line"""
        if hosts is None:
            hosts = self.hosts
        poller = select.poll()
        h1 = hosts[ 0 ]  # so we can call class method fdToNode
        for host in hosts:
            poller.register( host.stdout )
        while True:
            ready = poller.poll( timeoutms )
            for fd, event in ready:
                host = h1.fdToNode( fd )
                if event & select.POLLIN:
                    line = host.readline()
                    if line is not None:
                        yield host, line
            # Return if non-blocking
            if not ready and timeoutms >= 0:
                yield None, None

    # XXX These test methods should be moved out of this class.
    # Probably we should create a tests.py for them

    @staticmethod
    def _parsePing( pingOutput ):
        "Parse ping output and return packets sent, received."
        # Check for downed link
        if 'connect: Network is unreachable' in pingOutput:
            return 1, 0
        r = r'(\d+) packets transmitted, (\d+)( packets)? received'
        m = re.search( r, pingOutput )
        if m is None:
            error( '*** Error: could not parse ping output: %s\n' %
                   pingOutput )
            return 1, 0
        sent, received = int( m.group( 1 ) ), int( m.group( 2 ) )
        return sent, received

    def ping( self, hosts=None, timeout=None ):
        """Ping between all specified hosts.
           hosts: list of hosts
           timeout: time to wait for a response, as string
           returns: ploss packet loss percentage"""
        # should we check if running?
        packets = 0
        lost = 0
        ploss = None
        if not hosts:
            hosts = self.hosts
            output( '*** Ping: testing ping reachability\n' )
        for node in hosts:
            output( '%s -> ' % node.name )
            for dest in hosts:
                if node != dest:
                    opts = ''
                    if timeout:
                        opts = '-W %s' % timeout
```

© University of South Africa November 2018

```
                if dest.intfs:
                    result = node.cmd( 'ping -c1 %s %s' %
                                       (opts, dest.IP()) )
                    sent, received = self._parsePing( result )
                else:
                    sent, received = 0, 0
                packets += sent
                if received > sent:
                    error( '*** Error: received too many packets' )
                    error( '%s' % result )
                    node.cmdPrint( 'route' )
                    exit( 1 )
                lost += sent - received
                output( ( '%s ' % dest.name ) if received else 'X ' )
            output( '\n' )
        if packets > 0:
            ploss = 100.0 * lost / packets
            received = packets - lost
            output( "*** Results: %i%% dropped (%d/%d received)\n" %
                    ( ploss, received, packets ) )
        else:
            ploss = 0
            output( "*** Warning: No packets sent\n" )
        return ploss

    @staticmethod
    def _parsePingFull( pingOutput ):
        "Parse ping output and return all data."
        errorTuple = (1, 0, 0, 0, 0, 0)
        # Check for downed link
        r = r'[uU]nreachable'
        m = re.search( r, pingOutput )
        if m is not None:
            return errorTuple
        r = r'(\d+) packets transmitted, (\d+)( packets)? received'
        m = re.search( r, pingOutput )
        if m is None:
            error( '*** Error: could not parse ping output: %s\n' %
                   pingOutput )
            return errorTuple
        sent, received = int( m.group( 1 ) ), int( m.group( 2 ) )
        r = r'rtt min/avg/max/mdev = '
        r += r'(\d+\.\d+)/(\d+\.\d+)/(\d+\.\d+)/(\d+\.\d+) ms'
        m = re.search( r, pingOutput )
        if m is None:
            if received == 0:
                return errorTuple
            error( '*** Error: could not parse ping output: %s\n' %
                   pingOutput )
            return errorTuple
        rttmin = float( m.group( 1 ) )
        rttavg = float( m.group( 2 ) )
        rttmax = float( m.group( 3 ) )
        rttdev = float( m.group( 4 ) )
        return sent, received, rttmin, rttavg, rttmax, rttdev
```

© University of South Africa November 2018

```
def pingFull( self, hosts=None, timeout=None ):
    """Ping between all specified hosts and return all data.
       hosts: list of hosts
       timeout: time to wait for a response, as string
       returns: all ping data; see function body."""
    # should we check if running?
    # Each value is a tuple: (src, dsd, [all ping outputs])
    all_outputs = []
    if not hosts:
        hosts = self.hosts
        output( '*** Ping: testing ping reachability\n' )
    for node in hosts:
        output( '%s -> ' % node.name )
        for dest in hosts:
            if node != dest:
                opts = ''
                if timeout:
                    opts = '-W %s' % timeout
                result = node.cmd( 'ping -c1 %s %s' % (opts, dest.IP()) )
                outputs = self._parsePingFull( result )
                sent, received, rttmin, rttavg, rttmax, rttdev = outputs
                all_outputs.append( (node, dest, outputs) )
                output( ( '%s ' % dest.name ) if received else 'X ' )
        output( '\n' )
    output( "*** Results: \n" )
    for outputs in all_outputs:
        src, dest, ping_outputs = outputs
        sent, received, rttmin, rttavg, rttmax, rttdev = ping_outputs
        output( " %s->%s: %s/%s, " % (src, dest, sent, received ) )
        output( "rtt min/avg/max/mdev %0.3f/%0.3f/%0.3f/%0.3f ms\n" %
                (rttmin, rttavg, rttmax, rttdev) )
    return all_outputs

def pingAll( self, timeout=None ):
    """Ping between all hosts.
       returns: ploss packet loss percentage"""
    return self.ping( timeout=timeout )

def pingPair( self ):
    """Ping between first two hosts, useful for testing.
       returns: ploss packet loss percentage"""
    hosts = [ self.hosts[ 0 ], self.hosts[ 1 ] ]
    return self.ping( hosts=hosts )

def pingAllFull( self ):
    """Ping between all hosts.
       returns: ploss packet loss percentage"""
    return self.pingFull()

def pingPairFull( self ):
    """Ping between first two hosts, useful for testing.
       returns: ploss packet loss percentage"""
    hosts = [ self.hosts[ 0 ], self.hosts[ 1 ] ]
    return self.pingFull( hosts=hosts )

@staticmethod
```

© University of South Africa November 2018

```
def _parseIperf( iperfOutput ):
    """Parse iperf output and return bandwidth.
       iperfOutput: string
       returns: result string"""
    r = r'([\d\.]+ \w+/sec)'
    m = re.findall( r, iperfOutput )
    if m:
        return m[-1]
    else:
        # was: raise Exception(...)
        error( 'could not parse iperf output: ' + iperfOutput )
        return ''

# XXX This should be cleaned up

def iperf( self, hosts=None, l4Type='TCP', udpBw='10M', fmt=None,
           seconds=5, port=5001):
    """Run iperf between two hosts.
       hosts: list of hosts; if None, uses first and last hosts
       l4Type: string, one of [ TCP, UDP ]
       udpBw: bandwidth target for UDP test
       fmt: iperf format argument if any
       seconds: iperf time to transmit
       port: iperf port
       returns: two-element array of [ server, client ] speeds
       note: send() is buffered, so client rate can be much higher than
       the actual transmission rate; on an unloaded system, server
       rate should be much closer to the actual receive rate"""
    hosts = hosts or [ self.hosts[ 0 ], self.hosts[ -1 ] ]
    assert len( hosts ) == 2
    client, server = hosts
    output( '*** Iperf: testing', l4Type, 'bandwidth between',
            client, 'and', server, '\n' )
    server.cmd( 'killall -9 iperf' )
    iperfArgs = 'iperf -p %d ' % port
    bwArgs = ''
    if l4Type == 'UDP':
        iperfArgs += '-u '
        bwArgs = '-b ' + udpBw + ' '
    elif l4Type != 'TCP':
        raise Exception( 'Unexpected l4 type: %s' % l4Type )
    if fmt:
        iperfArgs += '-f %s ' % fmt
    server.sendCmd( iperfArgs + '-s' )
    if l4Type == 'TCP':
        if not waitListening( client, server.IP(), port ):
            raise Exception( 'Could not connect to iperf on port %d'
                             % port )
    cliout = client.cmd( iperfArgs + '-t %d -c ' % seconds +
                         server.IP() + ' ' + bwArgs )
    debug( 'Client output: %s\n' % cliout )
    servout = ''
    # We want the last *b/sec from the iperf server output
    # for TCP, there are two of them because of waitListening
    count = 2 if l4Type == 'TCP' else 1
    while len( re.findall( '/sec', servout ) ) < count:
```

© University of South Africa November 2018

```
            servout += server.monitor( timeoutms=5000 )
        server.sendInt()
        servout += server.waitOutput()
        debug( 'Server output: %s\n' % servout )
        result = [ self._parseIperf( servout ), self._parseIperf( cliout ) ]
        if l4Type == 'UDP':
            result.insert( 0, udpBw )
        output( '*** Results: %s\n' % result )
        return result

    def runCpuLimitTest( self, cpu, duration=5 ):
        """run CPU limit test with 'while true' processes.
        cpu: desired CPU fraction of each host
        duration: test duration in seconds (integer)
        returns a single list of measured CPU fractions as floats.
        """
        pct = cpu * 100
        info( '*** Testing CPU %.0f%% bandwidth limit\n' % pct )
        hosts = self.hosts
        cores = int( quietRun( 'nproc' ) )
        # number of processes to run a while loop on per host
        num_procs = int( ceil( cores * cpu ) )
        pids = {}
        for h in hosts:
            pids[ h ] = []
            for _core in range( num_procs ):
                h.cmd( 'while true; do a=1; done &' )
                pids[ h ].append( h.cmd( 'echo $!' ).strip() )
        outputs = {}
        time = {}
        # get the initial cpu time for each host
        for host in hosts:
            outputs[ host ] = []
            with open( '/sys/fs/cgroup/cpuacct/%s/cpuacct.usage' %
                       host, 'r' ) as f:
                time[ host ] = float( f.read() )
        for _ in range( duration ):
            sleep( 1 )
            for host in hosts:
                with open( '/sys/fs/cgroup/cpuacct/%s/cpuacct.usage' %
                           host, 'r' ) as f:
                    readTime = float( f.read() )
                outputs[ host ].append( ( ( readTime - time[ host ] )
                                        / 1000000000 ) / cores * 100 )
                time[ host ] = readTime
        for h, pids in pids.items():
            for pid in pids:
                h.cmd( 'kill -9 %s' % pid )
        cpu_fractions = []
        for _host, outputs in outputs.items():
            for pct in outputs:
                cpu_fractions.append( pct )
        output( '*** Results: %s\n' % cpu_fractions )
        return cpu_fractions

    # BL: I think this can be rewritten now that we have
```

© University of South Africa November 2018

```
    # a real link class.
    def configLinkStatus( self, src, dst, status ):
        """Change status of src <-> dst links.
           src: node name
           dst: node name
           status: string {up, down}"""
        if src not in self.nameToNode:
            error( 'src not in network: %s\n' % src )
        elif dst not in self.nameToNode:
            error( 'dst not in network: %s\n' % dst )
        else:
            src = self.nameToNode[ src ]
            dst = self.nameToNode[ dst ]
            connections = src.connectionsTo( dst )
            if len( connections ) == 0:
                error( 'src and dst not connected: %s %s\n' % ( src, dst) )
            for srcIntf, dstIntf in connections:
                result = srcIntf.ifconfig( status )
                if result:
                    error( 'link src status change failed: %s\n' % result )
                result = dstIntf.ifconfig( status )
                if result:
                    error( 'link dst status change failed: %s\n' % result )

    def interact( self ):
        "Start network and run our simple CLI."
        self.start()
        result = CLI( self )
        self.stop()
        return result

    inited = False

    @classmethod
    def init( cls ):
        "Initialize Mininet"
        if cls.inited:
            return
        ensureRoot()
        fixLimits()
        cls.inited = True


class MininetWithControlNet( Mininet ):

    """Control network support:

       Create an explicit control network. Currently this is only
       used/usable with the user datapath.

       Notes:

       1. If the controller and switches are in the same (e.g. root)
          namespace, they can just use the loopback connection.

       2. If we can get unix domain sockets to work, we can use them
```

© University of South Africa November 2018

```
       instead of an explicit control network.

    3. Instead of routing, we could bridge or use 'in-band' control.

    4. Even if we dispense with this in general, it could still be
       useful for people who wish to simulate a separate control
       network (since real networks may need one!)

    5. Basically nobody ever used this code, so it has been moved
       into its own class.

    6. Ultimately we may wish to extend this to allow us to create a
       control network which every node's control interface is
       attached to."""

def configureControlNetwork( self ):
    "Configure control network."
    self.configureRoutedControlNetwork()

# We still need to figure out the right way to pass
# in the control network location.

def configureRoutedControlNetwork( self, ip='192.168.123.1',
                                   prefixLen=16 ):
    """Configure a routed control network on controller and switches.
       For use with the user datapath only right now."""
    controller = self.controllers[ 0 ]
    info( controller.name + ' <->' )
    cip = ip
    snum = ipParse( ip )
    for switch in self.switches:
        info( ' ' + switch.name )
        link = self.link( switch, controller, port1=0 )
        sintf, cintf = link.intf1, link.intf2
        switch.controlIntf = sintf
        snum += 1
        while snum & 0xff in [ 0, 255 ]:
            snum += 1
        sip = ipStr( snum )
        cintf.setIP( cip, prefixLen )
        sintf.setIP( sip, prefixLen )
        controller.setHostRoute( sip, cintf )
        switch.setHostRoute( cip, sintf )
    info( '\n' )
    info( '*** Testing control network\n' )
    while not cintf.isUp():
        info( '*** Waiting for', cintf, 'to come up\n' )
        sleep( 1 )
    for switch in self.switches:
        while not sintf.isUp():
            info( '*** Waiting for', sintf, 'to come up\n' )
            sleep( 1 )
        if self.ping( hosts=[ switch, controller ] ) != 0:
            error( '*** Error: control network test failed\n' )
            exit( 1 )
    info( '\n' )
```

© University of South Africa November 2018

Appendix C: MININET NODE CLASS

```
"""
Node objects for Mininet.

Nodes provide a simple abstraction for interacting with hosts, switches
and controllers. Local nodes are simply one or more processes on the local
machine.

Node: superclass for all (primarily local) network nodes.

Host: a virtual host. By default, a host is simply a shell; commands
    may be sent using Cmd (which waits for output), or using sendCmd(),
    which returns immediately, allowing subsequent monitoring using
    monitor(). Examples of how to run experiments using this
    functionality are provided in the examples/ directory. By default,
    hosts share the root file system, but they may also specify private
    directories.

CPULimitedHost: a virtual host whose CPU bandwidth is limited by
    RT or CFS bandwidth limiting.

Switch: superclass for switch nodes.

UserSwitch: a switch using the user-space switch from the OpenFlow
    reference implementation.

OVSSwitch: a switch using the Open vSwitch OpenFlow-compatible switch
    implementation (openvswitch.org).

OVSBridge: an Ethernet bridge implemented using Open vSwitch.
    Supports STP.

IVSSwitch: OpenFlow switch using the Indigo Virtual Switch.

Controller: superclass for OpenFlow controllers. The default controller
    is controller(8) from the reference implementation.

OVSController: The test controller from Open vSwitch.

NOXController: a controller node using NOX (noxrepo.org).

Ryu: The Ryu controller (https://osrg.github.io/ryu/)

RemoteController: a remote controller node, which may use any
    arbitrary OpenFlow-compatible controller, and which is not
    created or managed by Mininet.

Future enhancements:

- Possibly make Node, Switch and Controller more abstract so that
  they can be used for both local and remote nodes

- Create proxy objects for remote nodes (Mininet: Cluster Edition)
"""
```

© University of South Africa November 2018

```
import os
import pty
import re
import signal
import select
from subprocess import Popen, PIPE
from time import sleep

from mininet.log import info, error, warn, debug
from mininet.util import ( quietRun, errRun, errFail, moveIntf,
isShellBuiltin,
                           numCores, retry, mountCgroups )
from mininet.moduledeps import moduleDeps, pathCheck, TUN
from mininet.link import Link, Intf, TCIntf, OVSIntf
from re import findall
from distutils.version import StrictVersion

class Node( object ):
    """A virtual network node is simply a shell in a network namespace.
       We communicate with it using pipes."""

    portBase = 0  # Nodes always start with eth0/port0, even in OF 1.0

    def __init__( self, name, inNamespace=True, **params ):
        """name: name of node
           inNamespace: in network namespace?
           privateDirs: list of private directory strings or tuples
           params: Node parameters (see config() for details)"""

        # Make sure class actually works
        self.checkSetup()

        self.name = params.get( 'name', name )
        self.privateDirs = params.get( 'privateDirs', [] )
        self.inNamespace = params.get( 'inNamespace', inNamespace )

        # Stash configuration parameters for future reference
        self.params = params

        self.intfs = {}  # dict of port numbers to interfaces
        self.ports = {}  # dict of interfaces to port numbers
                         # replace with Port objects, eventually ?
        self.nameToIntf = {}  # dict of interface names to Intfs

        # Make pylint happy
        ( self.shell, self.execed, self.pid, self.stdin, self.stdout,
            self.lastPid, self.lastCmd, self.pollOut ) = (
                None, None, None, None, None, None, None, None )
        self.waiting = False
        self.readbuf = ''

        # Start command interpreter shell
        self.startShell()
        self.mountPrivateDirs()
```

© University of South Africa November 2018

```
    # File descriptor to node mapping support
    # Class variables and methods

    inToNode = {}  # mapping of input fds to nodes
    outToNode = {}  # mapping of output fds to nodes

    @classmethod
    def fdToNode( cls, fd ):
        """Return node corresponding to given file descriptor.
           fd: file descriptor
           returns: node"""
        node = cls.outToNode.get( fd )
        return node or cls.inToNode.get( fd )

    # Command support via shell process in namespace
    def startShell( self, mnopts=None ):
        "Start a shell process for running commands"
        if self.shell:
            error( "%s: shell is already running\n" % self.name )
            return
        # mnexec: (c)lose descriptors, (d)etach from tty,
        #  (p)rint pid, and run in (n)amespace
        opts = '-cd' if mnopts is None else mnopts
        if self.inNamespace:
            opts += 'n'
        # bash -i: force interactive
        # -s: pass $* to shell, and make process easy to find in ps
        # prompt is set to sentinel chr( 127 )
        cmd = [ 'mnexec', opts, 'env', 'PS1=' + chr( 127 ),
                'bash', '--norc', '--noediting',
                '-is', 'mininet:' + self.name ]

        # Spawn a shell subprocess in a pseudo-tty, to disable buffering
        # in the subprocess and insulate it from signals (e.g. SIGINT)
        # received by the parent
        master, slave = pty.openpty()
        self.shell = self._popen( cmd, stdin=slave, stdout=slave,
stderr=slave,
                                  close_fds=False )
        self.stdin = os.fdopen( master, 'rw' )
        self.stdout = self.stdin
        self.pid = self.shell.pid
        self.pollOut = select.poll()
        self.pollOut.register( self.stdout )
        # Maintain mapping between file descriptors and nodes
        # This is useful for monitoring multiple nodes
        # using select.poll()
        self.outToNode[ self.stdout.fileno() ] = self
        self.inToNode[ self.stdin.fileno() ] = self
        self.execed = False
        self.lastCmd = None
        self.lastPid = None
        self.readbuf = ''
        # Wait for prompt
        while True:
            data = self.read( 1024 )
```

© University of South Africa November 2018

```
            if data[ -1 ] == chr( 127 ):
                break
            self.pollOut.poll()
        self.waiting = False
        # +m: disable job control notification
        self.cmd( 'unset HISTFILE; stty -echo; set +m' )

    def mountPrivateDirs( self ):
        "mount private directories"
        # Avoid expanding a string into a list of chars
        assert not isinstance( self.privateDirs, basestring )
        for directory in self.privateDirs:
            if isinstance( directory, tuple ):
                # mount given private directory
                privateDir = directory[ 1 ] % self.__dict__
                mountPoint = directory[ 0 ]
                self.cmd( 'mkdir -p %s' % privateDir )
                self.cmd( 'mkdir -p %s' % mountPoint )
                self.cmd( 'mount --bind %s %s' %
                               ( privateDir, mountPoint ) )
            else:
                # mount temporary filesystem on directory
                self.cmd( 'mkdir -p %s' % directory )
                self.cmd( 'mount -n -t tmpfs tmpfs %s' % directory )

    def unmountPrivateDirs( self ):
        "mount private directories"
        for directory in self.privateDirs:
            if isinstance( directory, tuple ):
                self.cmd( 'umount ', directory[ 0 ] )
            else:
                self.cmd( 'umount ', directory )

    def _popen( self, cmd, **params ):
        """Internal method: spawn and return a process
            cmd: command to run (list)
            params: parameters to Popen()"""
        # Leave this is as an instance method for now
        assert self
        return Popen( cmd, **params )

    def cleanup( self ):
        "Help python collect its garbage."
        # We used to do this, but it slows us down:
        # Intfs may end up in root NS
        # for intfName in self.intfNames():
        # if self.name in intfName:
        # quietRun( 'ip link del ' + intfName )
        self.shell = None

    # Subshell I/O, commands and control

    def read( self, maxbytes=1024 ):
        """Buffered read from node, potentially blocking.
           maxbytes: maximum number of bytes to return"""
        count = len( self.readbuf )
```

© University of South Africa November 2018

```
        if count < maxbytes:
            data = os.read( self.stdout.fileno(), maxbytes - count )
            self.readbuf += data
        if maxbytes >= len( self.readbuf ):
            result = self.readbuf
            self.readbuf = ''
        else:
            result = self.readbuf[ :maxbytes ]
            self.readbuf = self.readbuf[ maxbytes: ]
        return result

    def readline( self ):
        """Buffered readline from node, potentially blocking.
           returns: line (minus newline) or None"""
        self.readbuf += self.read( 1024 )
        if '\n' not in self.readbuf:
            return None
        pos = self.readbuf.find( '\n' )
        line = self.readbuf[ 0: pos ]
        self.readbuf = self.readbuf[ pos + 1: ]
        return line

    def write( self, data ):
        """Write data to node.
           data: string"""
        os.write( self.stdin.fileno(), data )

    def terminate( self ):
        "Send kill signal to Node and clean up after it."
        self.unmountPrivateDirs()
        if self.shell:
            if self.shell.poll() is None:
                os.killpg( self.shell.pid, signal.SIGHUP )
        self.cleanup()

    def stop( self, deleteIntfs=False ):
        """Stop node.
           deleteIntfs: delete interfaces? (False)"""
        if deleteIntfs:
            self.deleteIntfs()
        self.terminate()

    def waitReadable( self, timeoutms=None ):
        """Wait until node's output is readable.
           timeoutms: timeout in ms or None to wait indefinitely.
           returns: result of poll()"""
        if len( self.readbuf ) == 0:
            return self.pollOut.poll( timeoutms )

    def sendCmd( self, *args, **kwargs ):
        """Send a command, followed by a command to echo a sentinel,
           and return without waiting for the command to complete.
           args: command and arguments, or string
           printPid: print command's PID? (False)"""
        assert self.shell and not self.waiting
        printPid = kwargs.get( 'printPid', False )
```

© University of South Africa November 2018

```
        # Allow sendCmd( [ list ] )
        if len( args ) == 1 and isinstance( args[ 0 ], list ):
            cmd = args[ 0 ]
        # Allow sendCmd( cmd, arg1, arg2... )
        elif len( args ) > 0:
            cmd = args
        # Convert to string
        if not isinstance( cmd, str ):
            cmd = ' '.join( [ str( c ) for c in cmd ] )
        if not re.search( r'\w', cmd ):
            # Replace empty commands with something harmless
            cmd = 'echo -n'
        self.lastCmd = cmd
        # if a builtin command is backgrounded, it still yields a PID
        if len( cmd ) > 0 and cmd[ -1 ] == '&':
            # print ^A{pid}\n so monitor() can set lastPid
            cmd += ' printf "\\001%d\\012" $! '
        elif printPid and not isShellBuiltin( cmd ):
            cmd = 'mnexec -p ' + cmd
        self.write( cmd + '\n' )
        self.lastPid = None
        self.waiting = True

    def sendInt( self, intr=chr( 3 ) ):
        "Interrupt running command."
        debug( 'sendInt: writing chr(%d)\n' % ord( intr ) )
        self.write( intr )

    def monitor( self, timeoutms=None, findPid=True ):
        """Monitor and return the output of a command.
           Set self.waiting to False if command has completed.
           timeoutms: timeout in ms or None to wait indefinitely
           findPid: look for PID from mnexec -p"""
        ready = self.waitReadable( timeoutms )
        if not ready:
            return ''
        data = self.read( 1024 )
        pidre = r'\[\d+\] \d+\r\n'
        # Look for PID
        marker = chr( 1 ) + r'\d+\r\n'
        if findPid and chr( 1 ) in data:
            # suppress the job and PID of a backgrounded command
            if re.findall( pidre, data ):
                data = re.sub( pidre, '', data )
            # Marker can be read in chunks; continue until all of it is read
            while not re.findall( marker, data ):
                data += self.read( 1024 )
            markers = re.findall( marker, data )
            if markers:
                self.lastPid = int( markers[ 0 ][ 1: ] )
                data = re.sub( marker, '', data )
        # Look for sentinel/EOF
        if len( data ) > 0 and data[ -1 ] == chr( 127 ):
            self.waiting = False
            data = data[ :-1 ]
        elif chr( 127 ) in data:
```

© University of South Africa November 2018

```
            self.waiting = False
            data = data.replace( chr( 127 ), '' )
        return data

    def waitOutput( self, verbose=False, findPid=True ):
        """Wait for a command to complete.
           Completion is signaled by a sentinel character, ASCII(127)
           appearing in the output stream.  Wait for the sentinel and return
           the output, including trailing newline.
           verbose: print output interactively"""
        log = info if verbose else debug
        output = ''
        while self.waiting:
            data = self.monitor( findPid=findPid )
            output += data
            log( data )
        return output

    def cmd( self, *args, **kwargs ):
        """Send a command, wait for output, and return it.
           cmd: string"""
        verbose = kwargs.get( 'verbose', False )
        log = info if verbose else debug
        log( '*** %s : %s\n' % ( self.name, args ) )
        if self.shell:
            self.sendCmd( *args, **kwargs )
            return self.waitOutput( verbose )
        else:
            warn( '(%s exited - ignoring cmd%s)\n' % ( self, args ) )

    def cmdPrint( self, *args):
        """Call cmd and printing its output
           cmd: string"""
        return self.cmd( *args, **{ 'verbose': True } )

    def popen( self, *args, **kwargs ):
        """Return a Popen() object in our namespace
           args: Popen() args, single list, or string
           kwargs: Popen() keyword args"""
        defaults = { 'stdout': PIPE, 'stderr': PIPE,
                     'mncmd':
                     [ 'mnexec', '-da', str( self.pid ) ] }
        defaults.update( kwargs )
        if len( args ) == 1:
            if isinstance( args[ 0 ], list ):
                # popen([cmd, arg1, arg2...])
                cmd = args[ 0 ]
            elif isinstance( args[ 0 ], basestring ):
                # popen("cmd arg1 arg2...")
                cmd = args[ 0 ].split()
            else:
                raise Exception( 'popen() requires a string or list' )
        elif len( args ) > 0:
            # popen( cmd, arg1, arg2... )
            cmd = list( args )
        # Attach to our namespace  using mnexec -a
```

© University of South Africa November 2018

```
        cmd = defaults.pop( 'mncmd' ) + cmd
        # Shell requires a string, not a list!
        if defaults.get( 'shell', False ):
            cmd = ' '.join( cmd )
        popen = self._popen( cmd, **defaults )
        return popen

    def pexec( self, *args, **kwargs ):
        """Execute a command using popen
           returns: out, err, exitcode"""
        popen = self.popen( *args, stdin=PIPE, stdout=PIPE, stderr=PIPE,
                            **kwargs )
        # Warning: this can fail with large numbers of fds!
        out, err = popen.communicate()
        exitcode = popen.wait()
        return out, err, exitcode

    # Interface management, configuration, and routing

    # BL notes: This might be a bit redundant or over-complicated.
    # However, it does allow a bit of specialization, including
    # changing the canonical interface names. It's also tricky since
    # the real interfaces are created as veth pairs, so we can't
    # make a single interface at a time.

    def newPort( self ):
        "Return the next port number to allocate."
        if len( self.ports ) > 0:
            return max( self.ports.values() ) + 1
        return self.portBase

    def addIntf( self, intf, port=None, moveIntfFn=moveIntf ):
        """Add an interface.
           intf: interface
           port: port number (optional, typically OpenFlow port number)
           moveIntfFn: function to move interface (optional)"""
        if port is None:
            port = self.newPort()
        self.intfs[ port ] = intf
        self.ports[ intf ] = port
        self.nameToIntf[ intf.name ] = intf
        debug( '\n' )
        debug( 'added intf %s (%d) to node %s\n' % (
                intf, port, self.name ) )
        if self.inNamespace:
            debug( 'moving', intf, 'into namespace for', self.name, '\n' )
            moveIntfFn( intf.name, self  )

    def delIntf( self, intf ):
        """Remove interface from Node's known interfaces
           Note: to fully delete interface, call intf.delete() instead"""
        port = self.ports.get( intf )
        if port is not None:
            del self.intfs[ port ]
            del self.ports[ intf ]
            del self.nameToIntf[ intf.name ]
```

© University of South Africa November 2018

```
def defaultIntf( self ):
    "Return interface for lowest port"
    ports = self.intfs.keys()
    if ports:
        return self.intfs[ min( ports ) ]
    else:
        warn( '*** defaultIntf: warning:', self.name,
              'has no interfaces\n' )

def intf( self, intf=None ):
    """Return our interface object with given string name,
       default intf if name is falsy (None, empty string, etc).
       or the input intf arg.

    Having this fcn return its arg for Intf objects makes it
    easier to construct functions with flexible input args for
    interfaces (those that accept both string names and Intf objects).
    """
    if not intf:
        return self.defaultIntf()
    elif isinstance( intf, basestring):
        return self.nameToIntf[ intf ]
    else:
        return intf

def connectionsTo( self, node):
    "Return [ intf1, intf2... ] for all intfs that connect self to node."
    # We could optimize this if it is important
    connections = []
    for intf in self.intfList():
        link = intf.link
        if link:
            node1, node2 = link.intf1.node, link.intf2.node
            if node1 == self and node2 == node:
                connections += [ ( intf, link.intf2 ) ]
            elif node1 == node and node2 == self:
                connections += [ ( intf, link.intf1 ) ]
    return connections

def deleteIntfs( self, checkName=True ):
    """Delete all of our interfaces.
       checkName: only delete interfaces that contain our name"""
    # In theory the interfaces should go away after we shut down.
    # However, this takes time, so we're better off removing them
    # explicitly so that we won't get errors if we run before they
    # have been removed by the kernel. Unfortunately this is very slow,
    # at least with Linux kernels before 2.6.33
    for intf in self.intfs.values():
        # Protect against deleting hardware interfaces
        if ( self.name in intf.name ) or ( not checkName ):
            intf.delete()
            info( '.' )

# Routing support
```

© University of South Africa November 2018

```
def setARP( self, ip, mac ):
    """Add an ARP entry.
       ip: IP address as string
       mac: MAC address as string"""
    result = self.cmd( 'arp', '-s', ip, mac )
    return result

def setHostRoute( self, ip, intf ):
    """Add route to host.
       ip: IP address as dotted decimal
       intf: string, interface name"""
    return self.cmd( 'route add -host', ip, 'dev', intf )

def setDefaultRoute( self, intf=None ):
    """Set the default route to go through intf.
       intf: Intf or {dev <intfname> via <gw-ip> ...}"""
    # Note setParam won't call us if intf is none
    if isinstance( intf, basestring ) and ' ' in intf:
        params = intf
    else:
        params = 'dev %s' % intf
    # Do this in one line in case we're messing with the root namespace
    self.cmd( 'ip route del default; ip route add default', params )

# Convenience and configuration methods

def setMAC( self, mac, intf=None ):
    """Set the MAC address for an interface.
       intf: intf or intf name
       mac: MAC address as string"""
    return self.intf( intf ).setMAC( mac )

def setIP( self, ip, prefixLen=8, intf=None, **kwargs ):
    """Set the IP address for an interface.
       intf: intf or intf name
       ip: IP address as a string
       prefixLen: prefix length, e.g. 8 for /8 or 16M addrs
       kwargs: any additional arguments for intf.setIP"""
    return self.intf( intf ).setIP( ip, prefixLen, **kwargs )

def IP( self, intf=None ):
    "Return IP address of a node or specific interface."
    return self.intf( intf ).IP()

def MAC( self, intf=None ):
    "Return MAC address of a node or specific interface."
    return self.intf( intf ).MAC()

def intfIsUp( self, intf=None ):
    "Check if an interface is up."
    return self.intf( intf ).isUp()

# The reason why we configure things in this way is so
# That the parameters can be listed and documented in
# the config method.
# Dealing with subclasses and superclasses is slightly
```

© University of South Africa November 2018

```
# annoying, but at least the information is there!

def setParam( self, results, method, **param ):
    """Internal method: configure a *single* parameter
       results: dict of results to update
       method: config method name
       param: arg=value (ignore if value=None)
       value may also be list or dict"""
    name, value = param.items()[ 0 ]
    if value is None:
        return
    f = getattr( self, method, None )
    if not f:
        return
    if isinstance( value, list ):
        result = f( *value )
    elif isinstance( value, dict ):
        result = f( **value )
    else:
        result = f( value )
    results[ name ] = result
    return result

def config( self, mac=None, ip=None,
            defaultRoute=None, lo='up', **_params ):
    """Configure Node according to (optional) parameters:
       mac: MAC address for default interface
       ip: IP address for default interface
       ifconfig: arbitrary interface configuration
       Subclasses should override this method and call
       the parent class's config(**params)"""
    # If we were overriding this method, we would call
    # the superclass config method here as follows:
    # r = Parent.config( **_params )
    r = {}
    self.setParam( r, 'setMAC', mac=mac )
    self.setParam( r, 'setIP', ip=ip )
    self.setParam( r, 'setDefaultRoute', defaultRoute=defaultRoute )
    # This should be examined
    self.cmd( 'ifconfig lo ' + lo )
    return r

def configDefault( self, **moreParams ):
    "Configure with default parameters"
    self.params.update( moreParams )
    self.config( **self.params )

# This is here for backward compatibility
def linkTo( self, node, link=Link ):
    """(Deprecated) Link to another node
       replace with Link( node1, node2)"""
    return link( self, node )

# Other methods

def intfList( self ):
```

© University of South Africa November 2018

```
        "List of our interfaces sorted by port number"
        return [ self.intfs[ p ] for p in sorted( self.intfs.iterkeys() ) ]

    def intfNames( self ):
        "The names of our interfaces sorted by port number"
        return [ str( i ) for i in self.intfList() ]

    def __repr__( self ):
        "More informative string representation"
        intfs = ( ','.join( [ '%s:%s' % ( i.name, i.IP() )
                              for i in self.intfList() ] ) )
        return '<%s %s: %s pid=%s> ' % (
            self.__class__.__name__, self.name, intfs, self.pid )

    def __str__( self ):
        "Abbreviated string representation"
        return self.name

    # Automatic class setup support

    isSetup = False

    @classmethod
    def checkSetup( cls ):
        "Make sure our class and superclasses are set up"
        while cls and not getattr( cls, 'isSetup', True ):
            cls.setup()
            cls.isSetup = True
            # Make pylint happy
            cls = getattr( type( cls ), '__base__', None )

    @classmethod
    def setup( cls ):
        "Make sure our class dependencies are available"
        pathCheck( 'mnexec', 'ifconfig', moduleName='Mininet')

class Host( Node ):
    "A host is simply a Node"
    pass

class CPULimitedHost( Host ):

    "CPU limited host"

    def __init__( self, name, sched='cfs', **kwargs ):
        Host.__init__( self, name, **kwargs )
        # Initialize class if necessary
        if not CPULimitedHost.inited:
            CPULimitedHost.init()
        # Create a cgroup and move shell into it
        self.cgroup = 'cpu,cpuacct,cpuset:/' + self.name
        errFail( 'cgcreate -g ' + self.cgroup )
        # We don't add ourselves to a cpuset because you must
        # specify the cpu and memory placement first
        errFail( 'cgclassify -g cpu,cpuacct:/%s %s' % ( self.name, self.pid )
)
```

© University of South Africa November 2018

```
        # BL: Setting the correct period/quota is tricky, particularly
        # for RT. RT allows very small quotas, but the overhead
        # seems to be high. CFS has a mininimum quota of 1 ms, but
        # still does better with larger period values.
        self.period_us = kwargs.get( 'period_us', 100000 )
        self.sched = sched
        if sched == 'rt':
            self.checkRtGroupSched()
            self.rtprio = 20

    def cgroupSet( self, param, value, resource='cpu' ):
        "Set a cgroup parameter and return its value"
        cmd = 'cgset -r %s.%s=%s /%s' % (
            resource, param, value, self.name )
        quietRun( cmd )
        nvalue = int( self.cgroupGet( param, resource ) )
        if nvalue != value:
            error( '*** error: cgroupSet: %s set to %s instead of %s\n'
                   % ( param, nvalue, value ) )
        return nvalue

    def cgroupGet( self, param, resource='cpu' ):
        "Return value of cgroup parameter"
        cmd = 'cgget -r %s.%s /%s' % (
            resource, param, self.name )
        return int( quietRun( cmd ).split()[ -1 ] )

    def cgroupDel( self ):
        "Clean up our cgroup"
        # info( '*** deleting cgroup', self.cgroup, '\n' )
        _out, _err, exitcode = errRun( 'cgdelete -r ' + self.cgroup )
        # Sometimes cgdelete returns a resource busy error but still
        # deletes the group; next attempt will give "no such file"
        return exitcode == 0 or ( 'no such file' in _err.lower() )

    def popen( self, *args, **kwargs ):
        """Return a Popen() object in node's namespace
           args: Popen() args, single list, or string
           kwargs: Popen() keyword args"""
        # Tell mnexec to execute command in our cgroup
        mncmd = kwargs.pop( 'mncmd', [ 'mnexec', '-g', self.name,
                                       '-da', str( self.pid ) ] )
        # if our cgroup is not given any cpu time,
        # we cannot assign the RR Scheduler.
        if self.sched == 'rt':
            if int( self.cgroupGet( 'rt_runtime_us', 'cpu' ) ) <= 0:
                mncmd += [ '-r', str( self.rtprio ) ]
            else:
                debug( '*** error: not enough cpu time available for %s.' %
                       self.name, 'Using cfs scheduler for subprocess\n' )
        return Host.popen( self, *args, mncmd=mncmd, **kwargs )

    def cleanup( self ):
        "Clean up Node, then clean up our cgroup"
        super( CPULimitedHost, self ).cleanup()
        retry( retries=3, delaySecs=.1, fn=self.cgroupDel )
```

© University of South Africa November 2018

```
    _rtGroupSched = False   # internal class var: Is CONFIG_RT_GROUP_SCHED
set?

    @classmethod
    def checkRtGroupSched( cls ):
        "Check (Ubuntu,Debian) kernel config for CONFIG_RT_GROUP_SCHED for
RT"
        if not cls._rtGroupSched:
            release = quietRun( 'uname -r' ).strip('\r\n')
            output = quietRun( 'grep CONFIG_RT_GROUP_SCHED /boot/config-%s' %
                               release )
            if output == '# CONFIG_RT_GROUP_SCHED is not set\n':
                error( '\n*** error: please enable RT_GROUP_SCHED '
                       'in your kernel\n' )
                exit( 1 )
            cls._rtGroupSched = True

    def chrt( self ):
        "Set RT scheduling priority"
        quietRun( 'chrt -p %s %s' % ( self.rtprio, self.pid ) )
        result = quietRun( 'chrt -p %s' % self.pid )
        firstline = result.split( '\n' )[ 0 ]
        lastword = firstline.split( ' ' )[ -1 ]
        if lastword != 'SCHED_RR':
            error( '*** error: could not assign SCHED_RR to %s\n' % self.name
)
        return lastword

    def rtInfo( self, f ):
        "Internal method: return parameters for RT bandwidth"
        pstr, qstr = 'rt_period_us', 'rt_runtime_us'
        # RT uses wall clock time for period and quota
        quota = int( self.period_us * f )
        return pstr, qstr, self.period_us, quota

    def cfsInfo( self, f ):
        "Internal method: return parameters for CFS bandwidth"
        pstr, qstr = 'cfs_period_us', 'cfs_quota_us'
        # CFS uses wall clock time for period and CPU time for quota.
        quota = int( self.period_us * f * numCores() )
        period = self.period_us
        if f > 0 and quota < 1000:
            debug( '(cfsInfo: increasing default period) ' )
            quota = 1000
            period = int( quota / f / numCores() )
        # Reset to unlimited on negative quota
        if quota < 0:
            quota = -1
        return pstr, qstr, period, quota

    # BL comment:
    # This may not be the right API,
    # since it doesn't specify CPU bandwidth in "absolute"
    # units the way link bandwidth is specified.
    # We should use MIPS or SPECINT or something instead.
```

© University of South Africa November 2018

```
    # Alternatively, we should change from system fraction
    # to CPU seconds per second, essentially assuming that
    # all CPUs are the same.

    def setCPUFrac( self, f, sched=None ):
        """Set overall CPU fraction for this host
           f: CPU bandwidth limit (positive fraction, or -1 for cfs
unlimited)
           sched: 'rt' or 'cfs'
           Note 'cfs' requires CONFIG_CFS_BANDWIDTH,
           and 'rt' requires CONFIG_RT_GROUP_SCHED"""
        if not sched:
            sched = self.sched
        if sched == 'rt':
            if not f or f < 0:
                raise Exception( 'Please set a positive CPU fraction'
                                 ' for sched=rt\n' )
            pstr, qstr, period, quota = self.rtInfo( f )
        elif sched == 'cfs':
            pstr, qstr, period, quota = self.cfsInfo( f )
        else:
            return
        # Set cgroup's period and quota
        setPeriod = self.cgroupSet( pstr, period )
        setQuota = self.cgroupSet( qstr, quota )
        if sched == 'rt':
            # Set RT priority if necessary
            sched = self.chrt()
        info( '(%s %d/%dus) ' % ( sched, setQuota, setPeriod ) )

    def setCPUs( self, cores, mems=0 ):
        "Specify (real) cores that our cgroup can run on"
        if not cores:
            return
        if isinstance( cores, list ):
            cores = ','.join( [ str( c ) for c in cores ] )
        self.cgroupSet( resource='cpuset', param='cpus',
                        value=cores )
        # Memory placement is probably not relevant, but we
        # must specify it anyway
        self.cgroupSet( resource='cpuset', param='mems',
                        value=mems)
        # We have to do this here after we've specified
        # cpus and mems
        errFail( 'cgclassify -g cpuset:/%s %s' % (
                 self.name, self.pid ) )

    def config( self, cpu=-1, cores=None, **params ):
        """cpu: desired overall system CPU fraction
           cores: (real) core(s) this host can run on
           params: parameters for Node.config()"""
        r = Node.config( self, **params )
        # Was considering cpu={'cpu': cpu , 'sched': sched}, but
        # that seems redundant
        self.setParam( r, 'setCPUFrac', cpu=cpu )
        self.setParam( r, 'setCPUs', cores=cores )
```

© University of South Africa November 2018

```
            return r

        inited = False

        @classmethod
        def init( cls ):
            "Initialization for CPULimitedHost class"
            mountCgroups()
            cls.inited = True


# Some important things to note:
#
# The "IP" address which setIP() assigns to the switch is not
# an "IP address for the switch" in the sense of IP routing.
# Rather, it is the IP address for the control interface,
# on the control network, and it is only relevant to the
# controller. If you are running in the root namespace
# (which is the only way to run OVS at the moment), the
# control interface is the loopback interface, and you
# normally never want to change its IP address!
#
# In general, you NEVER want to attempt to use Linux's
# network stack (i.e. ifconfig) to "assign" an IP address or
# MAC address to a switch data port. Instead, you "assign"
# the IP and MAC addresses in the controller by specifying
# packets that you want to receive or send. The "MAC" address
# reported by ifconfig for a switch data port is essentially
# meaningless. It is important to understand this if you
# want to create a functional router using OpenFlow.

class Switch( Node ):
    """A Switch is a Node that is running (or has execed?)
       an OpenFlow switch."""

    portBase = 1  # Switches start with port 1 in OpenFlow
    dpidLen = 16  # digits in dpid passed to switch

    def __init__( self, name, dpid=None, opts='', listenPort=None, **params):
        """dpid: dpid hex string (or None to derive from name, e.g. s1 -> 1)
           opts: additional switch options
           listenPort: port to listen on for dpctl connections"""
        Node.__init__( self, name, **params )
        self.dpid = self.defaultDpid( dpid )
        self.opts = opts
        self.listenPort = listenPort
        if not self.inNamespace:
            self.controlIntf = Intf( 'lo', self, port=0 )

    def defaultDpid( self, dpid=None ):
        "Return correctly formatted dpid from dpid or switch name (s1 -> 1)"
        if dpid:
            # Remove any colons and make sure it's a good hex number
            dpid = dpid.translate( None, ':' )
            assert len( dpid ) <= self.dpidLen and int( dpid, 16 ) >= 0
        else:
```

© University of South Africa November 2018

```
            # Use hex of the first number in the switch name
            nums = re.findall( r'\d+', self.name )
            if nums:
                dpid = hex( int( nums[ 0 ] ) )[ 2: ]
            else:
                raise Exception( 'Unable to derive default datapath ID - '
                                 'please either specify a dpid or use a '
                                 'canonical switch name such as s23.' )
        return '0' * ( self.dpidLen - len( dpid ) ) + dpid

    def defaultIntf( self ):
        "Return control interface"
        if self.controlIntf:
            return self.controlIntf
        else:
            return Node.defaultIntf( self )

    def sendCmd( self, *cmd, **kwargs ):
        """Send command to Node.
           cmd: string"""
        kwargs.setdefault( 'printPid', False )
        if not self.execed:
            return Node.sendCmd( self, *cmd, **kwargs )
        else:
            error( '*** Error: %s has execed and cannot accept commands' %
                   self.name )

    def connected( self ):
        "Is the switch connected to a controller? (override this method)"
        # Assume that we are connected by default to whatever we need to
        # be connected to. This should be overridden by any OpenFlow
        # switch, but not by a standalone bridge.
        debug( 'Assuming', repr( self ), 'is connected to a controller\n' )
        return True

    def stop( self, deleteIntfs=True ):
        """Stop switch
           deleteIntfs: delete interfaces? (True)"""
        if deleteIntfs:
            self.deleteIntfs()

    def __repr__( self ):
        "More informative string representation"
        intfs = ( ','.join( [ '%s:%s' % ( i.name, i.IP() )
                              for i in self.intfList() ] ) )
        return '<%s %s: %s pid=%s> ' % (
            self.__class__.__name__, self.name, intfs, self.pid )


class UserSwitch( Switch ):
    "User-space switch."

    dpidLen = 12

    def __init__( self, name, dpopts='--no-slicing', **kwargs ):
        """Init.
```

© University of South Africa November 2018

```
           name: name for the switch
           dpopts: additional arguments to ofdatapath (--no-slicing)"""
        Switch.__init__( self, name, **kwargs )
        pathCheck( 'ofdatapath', 'ofprotocol',
                   moduleName='the OpenFlow reference user switch' +
                              '(openflow.org)' )
        if self.listenPort:
            self.opts += ' --listen=ptcp:%i ' % self.listenPort
        else:
            self.opts += ' --listen=punix:/tmp/%s.listen' % self.name
        self.dpopts = dpopts

    @classmethod
    def setup( cls ):
        "Ensure any dependencies are loaded; if not, try to load them."
        if not os.path.exists( '/dev/net/tun' ):
            moduleDeps( add=TUN )

    def dpctl( self, *args ):
        "Run dpctl command"
        listenAddr = None
        if not self.listenPort:
            listenAddr = 'unix:/tmp/%s.listen' % self.name
        else:
            listenAddr = 'tcp:127.0.0.1:%i' % self.listenPort
        return self.cmd( 'dpctl ' + ' '.join( args ) +
                         ' ' + listenAddr )

    def connected( self ):
        "Is the switch connected to a controller?"
        status = self.dpctl( 'status' )
        return ( 'remote.is-connected=true' in status and
                 'local.is-connected=true' in status )

    @staticmethod
    def TCReapply( intf ):
        """Unfortunately user switch and Mininet are fighting
           over tc queuing disciplines. To resolve the conflict,
           we re-create the user switch's configuration, but as a
           leaf of the TCIntf-created configuration."""
        if isinstance( intf, TCIntf ):
            ifspeed = 10000000000  # 10 Gbps
            minspeed = ifspeed * 0.001

            res = intf.config( **intf.params )

            if res is None:  # link may not have TC parameters
                return

            # Re-add qdisc, root, and default classes user switch created,
but
            # with new parent, as setup by Mininet's TCIntf
            parent = res['parent']
            intf.tc( "%s qdisc add dev %s " + parent +
                     " handle 1: htb default 0xfffe" )
```

© University of South Africa November 2018

```
            intf.tc( "%s class add dev %s classid 1:0xffff parent 1: htb rate
"
                     + str(ifspeed) )
            intf.tc( "%s class add dev %s classid 1:0xfffe parent 1:0xffff "
+
                     "htb rate " + str(minspeed) + " ceil " + str(ifspeed) )

    def start( self, controllers ):
        """Start OpenFlow reference user datapath.
           Log to /tmp/sN-{ofd,ofp}.log.
           controllers: list of controller objects"""
        # Add controllers
        clist = ','.join( [ 'tcp:%s:%d' % ( c.IP(), c.port )
                            for c in controllers ] )
        ofdlog = '/tmp/' + self.name + '-ofd.log'
        ofplog = '/tmp/' + self.name + '-ofp.log'
        intfs = [ str( i ) for i in self.intfList() if not i.IP() ]
        self.cmd( 'ofdatapath -i ' + ','.join( intfs ) +
                  ' punix:/tmp/' + self.name + ' -d %s ' % self.dpid +
                  self.dpopts +
                  ' 1> ' + ofdlog + ' 2> ' + ofdlog + ' &' )
        self.cmd( 'ofprotocol unix:/tmp/' + self.name +
                  ' ' + clist +
                  ' --fail=closed ' + self.opts +
                  ' 1> ' + ofplog + ' 2>' + ofplog + ' &' )
        if "no-slicing" not in self.dpopts:
            # Only TCReapply if slicing is enable
            sleep(1)  # Allow ofdatapath to start before re-arranging qdisc's
            for intf in self.intfList():
                if not intf.IP():
                    self.TCReapply( intf )

    def stop( self, deleteIntfs=True ):
        """Stop OpenFlow reference user datapath.
           deleteIntfs: delete interfaces? (True)"""
        self.cmd( 'kill %ofdatapath' )
        self.cmd( 'kill %ofprotocol' )
        super( UserSwitch, self ).stop( deleteIntfs )


class OVSSwitch( Switch ):
    "Open vSwitch switch. Depends on ovs-vsctl."

    def __init__( self, name, failMode='secure', datapath='kernel',
                  inband=False, protocols=None,
                  reconnectms=1000, stp=False, batch=False, **params ):
        """name: name for switch
           failMode: controller loss behavior (secure|standalone)
           datapath: userspace or kernel mode (kernel|user)
           inband: use in-band control (False)
           protocols: use specific OpenFlow version(s) (e.g. OpenFlow13)
                      Unspecified (or old OVS version) uses OVS default
           reconnectms: max reconnect timeout in ms (0/None for default)
           stp: enable STP (False, requires failMode=standalone)
           batch: enable batch startup (False)"""
        Switch.__init__( self, name, **params )
```

© University of South Africa November 2018

```
        self.failMode = failMode
        self.datapath = datapath
        self.inband = inband
        self.protocols = protocols
        self.reconnectms = reconnectms
        self.stp = stp
        self._uuids = []  # controller UUIDs
        self.batch = batch
        self.commands = []  # saved commands for batch startup

    @classmethod
    def setup( cls ):
        "Make sure Open vSwitch is installed and working"
        pathCheck( 'ovs-vsctl',
                   moduleName='Open vSwitch (openvswitch.org)')
        # This should no longer be needed, and it breaks
        # with OVS 1.7 which has renamed the kernel module:
        #  moduleDeps( subtract=OF_KMOD, add=OVS_KMOD )
        out, err, exitcode = errRun( 'ovs-vsctl -t 1 show' )
        if exitcode:
            error( out + err +
                   'ovs-vsctl exited with code %d\n' % exitcode +
                   '*** Error connecting to ovs-db with ovs-vsctl\n'
                   'Make sure that Open vSwitch is installed, '
                   'that ovsdb-server is running, and that\n'
                   '"ovs-vsctl show" works correctly.\n'
                   'You may wish to try '
                   '"service openvswitch-switch start".\n' )
            exit( 1 )
        version = quietRun( 'ovs-vsctl --version' )
        cls.OVSVersion = findall( r'\d+\.\d+', version )[ 0 ]

    @classmethod
    def isOldOVS( cls ):
        "Is OVS ersion < 1.10?"
        return ( StrictVersion( cls.OVSVersion ) <
                 StrictVersion( '1.10' ) )

    def dpctl( self, *args ):
        "Run ovs-ofctl command"
        return self.cmd( 'ovs-ofctl', args[ 0 ], self, *args[ 1: ] )

    def vsctl( self, *args, **kwargs ):
        "Run ovs-vsctl command (or queue for later execution)"
        if self.batch:
            cmd = ' '.join( str( arg ).strip() for arg in args )
            self.commands.append( cmd )
        else:
            return self.cmd( 'ovs-vsctl', *args, **kwargs )

    @staticmethod
    def TCReapply( intf ):
        """Unfortunately OVS and Mininet are fighting
           over tc queuing disciplines. As a quick hack/
           workaround, we clear OVS's and reapply our own."""
        if isinstance( intf, TCIntf ):
```

© University of South Africa November 2018

```
            intf.config( **intf.params )

    def attach( self, intf ):
        "Connect a data port"
        self.vsctl( 'add-port', self, intf )
        self.cmd( 'ifconfig', intf, 'up' )
        self.TCReapply( intf )

    def detach( self, intf ):
        "Disconnect a data port"
        self.vsctl( 'del-port', self, intf )

    def controllerUUIDs( self, update=False ):
        """Return ovsdb UUIDs for our controllers
           update: update cached value"""
        if not self._uuids or update:
            controllers = self.cmd( 'ovs-vsctl -- get Bridge', self,
                                    'Controller' ).strip()
            if controllers.startswith( '[' ) and controllers.endswith( ']' ):
                controllers = controllers[ 1 : -1 ]
                if controllers:
                    self._uuids = [ c.strip()
                                    for c in controllers.split( ',' ) ]
        return self._uuids

    def connected( self ):
        "Are we connected to at least one of our controllers?"
        for uuid in self.controllerUUIDs():
            if 'true' in self.vsctl( '-- get Controller',
                                     uuid, 'is_connected' ):
                return True
        return self.failMode == 'standalone'

    def intfOpts( self, intf ):
        "Return OVS interface options for intf"
        opts = ''
        if not self.isOldOVS():
            # ofport_request is not supported on old OVS
            opts += ' ofport_request=%s' % self.ports[ intf ]
            # Patch ports don't work well with old OVS
            if isinstance( intf, OVSIntf ):
                intf1, intf2 = intf.link.intf1, intf.link.intf2
                peer = intf1 if intf1 != intf else intf2
                opts += ' type=patch options:peer=%s' % peer
        return '' if not opts else ' -- set Interface %s' % intf + opts

    def bridgeOpts( self ):
        "Return OVS bridge options"
        opts = ( ' other_config:datapath-id=%s' % self.dpid +
                 ' fail_mode=%s' % self.failMode )
        if not self.inband:
            opts += ' other-config:disable-in-band=true'
        if self.datapath == 'user':
            opts += ' datapath_type=netdev'
        if self.protocols and not self.isOldOVS():
            opts += ' protocols=%s' % self.protocols
```

© University of South Africa November 2018

```
        if self.stp and self.failMode == 'standalone':
            opts += ' stp_enable=true'
        opts += ' other-config:dp-desc=%s' % self.name
        return opts

    def start( self, controllers ):
        "Start up a new OVS OpenFlow switch using ovs-vsctl"
        if self.inNamespace:
            raise Exception(
                'OVS kernel switch does not work in a namespace' )
        int( self.dpid, 16 )  # DPID must be a hex string
        # Command to add interfaces
        intfs = ''.join( ' -- add-port %s %s' % ( self, intf ) +
                         self.intfOpts( intf )
                         for intf in self.intfList()
                         if self.ports[ intf ] and not intf.IP() )
        # Command to create controller entries
        clist = [ ( self.name + c.name, '%s:%s:%d' %
                  ( c.protocol, c.IP(), c.port ) )
                  for c in controllers ]
        if self.listenPort:
            clist.append( ( self.name + '-listen',
                            'ptcp:%s' % self.listenPort ) )
        ccmd = '-- --id=@%s create Controller target=\\"%s\\"'
        if self.reconnectms:
            ccmd += ' max_backoff=%d' % self.reconnectms
        cargs = ' '.join( ccmd % ( name, target )
                          for name, target in clist )
        # Controller ID list
        cids = ','.join( '@%s' % name for name, _target in clist )
        # Try to delete any existing bridges with the same name
        if not self.isOldOVS():
            cargs += ' -- --if-exists del-br %s' % self
        # One ovs-vsctl command to rule them all!
        self.vsctl( cargs +
                    ' -- add-br %s' % self +
                    ' -- set bridge %s controller=[%s]' % ( self, cids  ) +
                    self.bridgeOpts() +
                    intfs )
        # If necessary, restore TC config overwritten by OVS
        if not self.batch:
            for intf in self.intfList():
                self.TCReapply( intf )

    # This should be ~ int( quietRun( 'getconf ARG_MAX' ) ),
    # but the real limit seems to be much lower
    argmax = 128000

    @classmethod
    def batchStartup( cls, switches, run=errRun ):
        """Batch startup for OVS
           switches: switches to start up
           run: function to run commands (errRun)"""
        info( '...' )
        cmds = 'ovs-vsctl'
        for switch in switches:
```

© University of South Africa November 2018

```
            if switch.isOldOVS():
                # Ideally we'd optimize this also
                run( 'ovs-vsctl del-br %s' % switch )
            for cmd in switch.commands:
                cmd = cmd.strip()
                # Don't exceed ARG_MAX
                if len( cmds ) + len( cmd ) >= cls.argmax:
                    run( cmds, shell=True )
                    cmds = 'ovs-vsctl'
                cmds += ' ' + cmd
                switch.cmds = []
                switch.batch = False
        if cmds:
            run( cmds, shell=True )
        # Reapply link config if necessary...
        for switch in switches:
            for intf in switch.intfs.itervalues():
                if isinstance( intf, TCIntf ):
                    intf.config( **intf.params )
        return switches

    def stop( self, deleteIntfs=True ):
        """Terminate OVS switch.
           deleteIntfs: delete interfaces? (True)"""
        self.cmd( 'ovs-vsctl del-br', self )
        if self.datapath == 'user':
            self.cmd( 'ip link del', self )
        super( OVSSwitch, self ).stop( deleteIntfs )

    @classmethod
    def batchShutdown( cls, switches, run=errRun ):
        "Shut down a list of OVS switches"
        delcmd = 'del-br %s'
        if switches and not switches[ 0 ].isOldOVS():
            delcmd = '--if-exists ' + delcmd
        # First, delete them all from ovsdb
        run( 'ovs-vsctl ' +
             ' -- '.join( delcmd % s for s in switches ) )
        # Next, shut down all of the processes
        pids = ' '.join( str( switch.pid ) for switch in switches )
        run( 'kill -HUP ' + pids )
        for switch in switches:
            switch.shell = None
        return switches


OVSKernelSwitch = OVSSwitch


class OVSBridge( OVSSwitch ):
    "OVSBridge is an OVSSwitch in standalone/bridge mode"

    def __init__( self, *args, **kwargs ):
        """stp: enable Spanning Tree Protocol (False)
           see OVSSwitch for other options"""
        kwargs.update( failMode='standalone' )
```

© University of South Africa November 2018

```
        OVSSwitch.__init__( self, *args, **kwargs )

    def start( self, controllers ):
        "Start bridge, ignoring controllers argument"
        OVSSwitch.start( self, controllers=[] )

    def connected( self ):
        "Are we forwarding yet?"
        if self.stp:
            status = self.dpctl( 'show' )
            return 'STP_FORWARD' in status and not 'STP_LEARN' in status
        else:
            return True


class IVSSwitch( Switch ):
    "Indigo Virtual Switch"

    def __init__( self, name, verbose=False, **kwargs ):
        Switch.__init__( self, name, **kwargs )
        self.verbose = verbose

    @classmethod
    def setup( cls ):
        "Make sure IVS is installed"
        pathCheck( 'ivs-ctl', 'ivs',
                   moduleName="Indigo Virtual Switch (projectfloodlight.org)"
)
        out, err, exitcode = errRun( 'ivs-ctl show' )
        if exitcode:
            error( out + err +
                   'ivs-ctl exited with code %d\n' % exitcode +
                   '*** The openvswitch kernel module might '
                   'not be loaded. Try modprobe openvswitch.\n' )
            exit( 1 )

    @classmethod
    def batchShutdown( cls, switches ):
        "Kill each IVS switch, to be waited on later in stop()"
        for switch in switches:
            switch.cmd( 'kill %ivs' )
        return switches

    def start( self, controllers ):
        "Start up a new IVS switch"
        args = ['ivs']
        args.extend( ['--name', self.name] )
        args.extend( ['--dpid', self.dpid] )
        if self.verbose:
            args.extend( ['--verbose'] )
        for intf in self.intfs.values():
            if not intf.IP():
                args.extend( ['-i', intf.name] )
        for c in controllers:
            args.extend( ['-c', '%s:%d' % (c.IP(), c.port)] )
        if self.listenPort:
```

© University of South Africa November 2018

```
            args.extend( ['--listen', '127.0.0.1:%i' % self.listenPort] )
        args.append( self.opts )

        logfile = '/tmp/ivs.%s.log' % self.name

        self.cmd( ' '.join(args) + ' >' + logfile + ' 2>&1 </dev/null &' )

    def stop( self, deleteIntfs=True ):
        """Terminate IVS switch.
           deleteIntfs: delete interfaces? (True)"""
        self.cmd( 'kill %ivs' )
        self.cmd( 'wait' )
        super( IVSSwitch, self ).stop( deleteIntfs )

    def attach( self, intf ):
        "Connect a data port"
        self.cmd( 'ivs-ctl', 'add-port', '--datapath', self.name, intf )

    def detach( self, intf ):
        "Disconnect a data port"
        self.cmd( 'ivs-ctl', 'del-port', '--datapath', self.name, intf )

    def dpctl( self, *args ):
        "Run dpctl command"
        if not self.listenPort:
            return "can't run dpctl without passive listening port"
        return self.cmd( 'ovs-ofctl ' + ' '.join( args ) +
                         ' tcp:127.0.0.1:%i' % self.listenPort )


class Controller( Node ):
    """A Controller is a Node that is running (or has execed?) an
       OpenFlow controller."""

    def __init__( self, name, inNamespace=False, command='controller',
                  cargs='-v ptcp:%d', cdir=None, ip="127.0.0.1",
                  port=6653, protocol='tcp', **params ):
        self.command = command
        self.cargs = cargs
        self.cdir = cdir
        # Accept 'ip:port' syntax as shorthand
        if ':' in ip:
            ip, port = ip.split( ':' )
            port = int( port )
        self.ip = ip
        self.port = port
        self.protocol = protocol
        Node.__init__( self, name, inNamespace=inNamespace,
                       ip=ip, **params  )
        self.checkListening()

    def checkListening( self ):
        "Make sure no controllers are running on our port"
        # Verify that Telnet is installed first:
        out, _err, returnCode = errRun( "which telnet" )
        if 'telnet' not in out or returnCode != 0:
```

© University of South Africa November 2018

```
            raise Exception( "Error running telnet to check for listening "
                             "controllers; please check that it is "
                             "installed." )
        listening = self.cmd( "echo A | telnet -e A %s %d" %
                              ( self.ip, self.port ) )
        if 'Connected' in listening:
            servers = self.cmd( 'netstat -natp' ).split( '\n' )
            pstr = ':%d ' % self.port
            clist = servers[ 0:1 ] + [ s for s in servers if pstr in s ]
            raise Exception( "Please shut down the controller which is"
                             " running on port %d:\n" % self.port +
                             '\n'.join( clist ) )

    def start( self ):
        """Start <controller> <args> on controller.
           Log to /tmp/cN.log"""
        pathCheck( self.command )
        cout = '/tmp/' + self.name + '.log'
        if self.cdir is not None:
            self.cmd( 'cd ' + self.cdir )
        self.cmd( self.command + ' ' + self.cargs % self.port +
                  ' 1>' + cout + ' 2>' + cout + ' &' )
        self.execed = False

    def stop( self, *args, **kwargs ):
        "Stop controller."
        self.cmd( 'kill %' + self.command )
        self.cmd( 'wait %' + self.command )
        super( Controller, self ).stop( *args, **kwargs )

    def IP( self, intf=None ):
        "Return IP address of the Controller"
        if self.intfs:
            ip = Node.IP( self, intf )
        else:
            ip = self.ip
        return ip

    def __repr__( self ):
        "More informative string representation"
        return '<%s %s: %s:%s pid=%s> ' % (
            self.__class__.__name__, self.name,
            self.IP(), self.port, self.pid )

    @classmethod
    def isAvailable( cls ):
        "Is controller available?"
        return quietRun( 'which controller' )


class OVSController( Controller ):
    "Open vSwitch controller"
    def __init__( self, name, **kwargs ):
        kwargs.setdefault( 'command', self.isAvailable() or
                           'ovs-controller' )
        Controller.__init__( self, name, **kwargs )
```

© University of South Africa November 2018

```
    @classmethod
    def isAvailable( cls ):
        return ( quietRun( 'which ovs-controller' ) or
                 quietRun( 'which test-controller' ) or
                 quietRun( 'which ovs-testcontroller' ) ).strip()

class NOX( Controller ):
    "Controller to run a NOX application."

    def __init__( self, name, *noxArgs, **kwargs ):
        """Init.
           name: name to give controller
           noxArgs: arguments (strings) to pass to NOX"""
        if not noxArgs:
            warn( 'warning: no NOX modules specified; '
                  'running packetdump only\n' )
            noxArgs = [ 'packetdump' ]
        elif type( noxArgs ) not in ( list, tuple ):
            noxArgs = [ noxArgs ]

        if 'NOX_CORE_DIR' not in os.environ:
            exit( 'exiting; please set missing NOX_CORE_DIR env var' )
        noxCoreDir = os.environ[ 'NOX_CORE_DIR' ]

        Controller.__init__( self, name,
                             command=noxCoreDir + '/nox_core',
                             cargs='--libdir=/usr/local/lib -v -i ptcp:%s ' +
                             ' '.join( noxArgs ),
                             cdir=noxCoreDir,
                             **kwargs )

class Ryu( Controller ):
    "Controller to run Ryu application"
    def __init__( self, name, *ryuArgs, **kwargs ):
        """Init.
        name: name to give controller.
        ryuArgs: arguments and modules to pass to Ryu"""
        homeDir = quietRun( 'printenv HOME' ).strip( '\r\n' )
        ryuCoreDir = '%s/ryu/ryu/app/' % homeDir
        if not ryuArgs:
            warn( 'warning: no Ryu modules specified; '
                  'running simple_switch only\n' )
            ryuArgs = [ ryuCoreDir + 'simple_switch.py' ]
        elif type( ryuArgs ) not in ( list, tuple ):
            ryuArgs = [ ryuArgs ]

        Controller.__init__( self, name,
                             command='ryu-manager',
                             cargs='--ofp-tcp-listen-port %s ' +
                             ' '.join( ryuArgs ),
                             cdir=ryuCoreDir,
                             **kwargs )


class RemoteController( Controller ):
```

© University of South Africa November 2018

```
    "Controller running outside of Mininet's control."

    def __init__( self, name, ip='127.0.0.1',
                  port=None, **kwargs):
        """Init.
           name: name to give controller
           ip: the IP address where the remote controller is
           listening
           port: the port where the remote controller is listening"""
        Controller.__init__( self, name, ip=ip, port=port, **kwargs )

    def start( self ):
        "Overridden to do nothing."
        return

    def stop( self ):
        "Overridden to do nothing."
        return

    def checkListening( self ):
        "Warn if remote controller is not accessible"
        if self.port is not None:
            self.isListening( self.ip, self.port )
        else:
            for port in 6653, 6633:
                if self.isListening( self.ip, port ):
                    self.port = port
                    info( "Connecting to remote controller"
                          " at %s:%d\n" % ( self.ip, self.port ))
                    break

        if self.port is None:
            self.port = 6653
            warn( "Setting remote controller"
                  " to %s:%d\n" % ( self.ip, self.port ))

    def isListening( self, ip, port ):
        "Check if a remote controller is listening at a specific ip and port"
        listening = self.cmd( "echo A | telnet -e A %s %d" % ( ip, port ) )
        if 'Connected' not in listening:
            warn( "Unable to contact the remote controller"
                  " at %s:%d\n" % ( ip, port ) )
            return False
        else:
            return True

DefaultControllers = ( Controller, OVSController )

def findController( controllers=DefaultControllers ):
    "Return first available controller from list, if any"
    for controller in controllers:
        if controller.isAvailable():
            return controller

def DefaultController( name, controllers=DefaultControllers, **kwargs ):
    "Find a controller that is available and instantiate it"
```

© University of South Africa November 2018

```
    controller = findController( controllers )
    if not controller:
        raise Exception( 'Could not find a default OpenFlow controller' )
    return controller( name, **kwargs )

def NullController( *_args, **_kwargs ):
    "Nonexistent controller - simply returns None"
    return None
```

© University of South Africa November 2018

Appendix D: MININET TOPO CLASS

```
#!/usr/bin/env python
"""@package topo

Network topology creation.

@author Brandon Heller (brandonh@stanford.edu)

This package includes code to represent network topologies.

A Topo object can be a topology database for NOX, can represent a physical
setup for testing, and can even be emulated with the Mininet package.
"""

from mininet.util import irange, natural, naturalSeq

class MultiGraph( object ):
    "Utility class to track nodes and edges - replaces networkx.MultiGraph"

    def __init__( self ):
        self.node = {}
        self.edge = {}

    def add_node( self, node, attr_dict=None, **attrs):
        """Add node to graph
           attr_dict: attribute dict (optional)
           attrs: more attributes (optional)
           warning: updates attr_dict with attrs"""
        attr_dict = {} if attr_dict is None else attr_dict
        attr_dict.update( attrs )
        self.node[ node ] = attr_dict

    def add_edge( self, src, dst, key=None, attr_dict=None, **attrs ):
        """Add edge to graph
           key: optional key
           attr_dict: optional attribute dict
           attrs: more attributes
           warning: udpates attr_dict with attrs"""
        attr_dict = {} if attr_dict is None else attr_dict
        attr_dict.update( attrs )
        self.node.setdefault( src, {} )
        self.node.setdefault( dst, {} )
        self.edge.setdefault( src, {} )
        self.edge.setdefault( dst, {} )
        self.edge[ src ].setdefault( dst, {} )
        entry = self.edge[ dst ][ src ] = self.edge[ src ][ dst ]
        # If no key, pick next ordinal number
        if key is None:
            keys = [ k for k in entry.keys() if isinstance( k, int ) ]
            key = max( [ 0 ] + keys ) + 1
        entry[ key ] = attr_dict
        return key

    def nodes( self, data=False):
```

© University of South Africa November 2018

```
        """Return list of graph nodes
           data: return list of ( node, attrs)"""
        return self.node.items() if data else self.node.keys()

    def edges_iter( self, data=False, keys=False ):
        "Iterator: return graph edges, optionally with data and keys"
        for src, entry in self.edge.iteritems():
            for dst, entrykeys in entry.iteritems():
                if src > dst:
                    # Skip duplicate edges
                    continue
                for k, attrs in entrykeys.iteritems():
                    if data:
                        if keys:
                            yield( src, dst, k, attrs )
                        else:
                            yield( src, dst, attrs )
                    else:
                        if keys:
                            yield( src, dst, k )
                        else:
                            yield( src, dst )

    def edges( self, data=False, keys=False ):
        "Return list of graph edges"
        return list( self.edges_iter( data=data, keys=keys ) )

    def __getitem__( self, node ):
        "Return link dict for given src node"
        return self.edge[ node ]

    def __len__( self ):
        "Return the number of nodes"
        return len( self.node )

    def convertTo( self, cls, data=False, keys=False ):
        """Convert to a new object of networkx.MultiGraph-like class cls
           data: include node and edge data
           keys: include edge keys as well as edge data"""
        g = cls()
        g.add_nodes_from( self.nodes( data=data ) )
        g.add_edges_from( self.edges( data=( data or keys ), keys=keys ) )
        return g


class Topo( object ):
    "Data center network representation for structured multi-trees."

    def __init__( self, *args, **params ):
        """Topo object.
           Optional named parameters:
           hinfo: default host options
           sopts: default switch options
           lopts: default link options
           calls build()"""
        self.g = MultiGraph()
```

© University of South Africa November 2018

```
        self.hopts = params.pop( 'hopts', {} )
        self.sopts = params.pop( 'sopts', {} )
        self.lopts = params.pop( 'lopts', {} )
        # ports[src][dst][sport] is port on dst that connects to src
        self.ports = {}
        self.build( *args, **params )

    def build( self, *args, **params ):
        "Override this method to build your topology."
        pass

    def addNode( self, name, **opts ):
        """Add Node to graph.
           name: name
           opts: node options
           returns: node name"""
        self.g.add_node( name, **opts )
        return name

    def addHost( self, name, **opts ):
        """Convenience method: Add host to graph.
           name: host name
           opts: host options
           returns: host name"""
        if not opts and self.hopts:
            opts = self.hopts
        return self.addNode( name, **opts )

    def addSwitch( self, name, **opts ):
        """Convenience method: Add switch to graph.
           name: switch name
           opts: switch options
           returns: switch name"""
        if not opts and self.sopts:
            opts = self.sopts
        result = self.addNode( name, isSwitch=True, **opts )
        return result

    def addLink( self, node1, node2, port1=None, port2=None,
                 key=None, **opts ):
        """node1, node2: nodes to link together
           port1, port2: ports (optional)
           opts: link options (optional)
           returns: link info key"""
        if not opts and self.lopts:
            opts = self.lopts
        port1, port2 = self.addPort( node1, node2, port1, port2 )
        opts = dict( opts )
        opts.update( node1=node1, node2=node2, port1=port1, port2=port2 )
        self.g.add_edge(node1, node2, key, opts )
        return key

    def nodes( self, sort=True ):
        "Return nodes in graph"
        if sort:
            return self.sorted( self.g.nodes() )
```

© University of South Africa November 2018

```
        else:
            return self.g.nodes()

    def isSwitch( self, n ):
        "Returns true if node is a switch."
        return self.g.node[ n ].get( 'isSwitch', False )

    def switches( self, sort=True ):
        """Return switches.
           sort: sort switches alphabetically
           returns: dpids list of dpids"""
        return [ n for n in self.nodes( sort ) if self.isSwitch( n ) ]

    def hosts( self, sort=True ):
        """Return hosts.
           sort: sort hosts alphabetically
           returns: list of hosts"""
        return [ n for n in self.nodes( sort ) if not self.isSwitch( n ) ]

    def iterLinks( self, withKeys=False, withInfo=False ):
        """Return links (iterator)
           withKeys: return link keys
           withInfo: return link info
           returns: list of ( src, dst [,key, info ] )"""
        for _src, _dst, key, info in self.g.edges_iter( data=True, keys=True
):
            node1, node2 = info[ 'node1' ], info[ 'node2' ]
            if withKeys:
                if withInfo:
                    yield( node1, node2, key, info )
                else:
                    yield( node1, node2, key )
            else:
                if withInfo:
                    yield( node1, node2, info )
                else:
                    yield( node1, node2 )

    def links( self, sort=False, withKeys=False, withInfo=False ):
        """Return links
           sort: sort links alphabetically, preserving (src, dst) order
           withKeys: return link keys
           withInfo: return link info
           returns: list of ( src, dst [,key, info ] )"""
        links = list( self.iterLinks( withKeys, withInfo ) )
        if not sort:
            return links
        # Ignore info when sorting
        tupleSize = 3 if withKeys else 2
        return sorted( links, key=( lambda l: naturalSeq( l[ :tupleSize ] ) )
)

    # This legacy port management mechanism is clunky and will probably
    # be removed at some point.

    def addPort( self, src, dst, sport=None, dport=None ):
```

© University of South Africa November 2018

```
        """Generate port mapping for new edge.
            src: source switch name
            dst: destination switch name"""
        # Initialize if necessary
        ports = self.ports
        ports.setdefault( src, {} )
        ports.setdefault( dst, {} )
        # New port: number of outlinks + base
        if sport is None:
            src_base = 1 if self.isSwitch( src ) else 0
            sport = len( ports[ src ] ) + src_base
        if dport is None:
            dst_base = 1 if self.isSwitch( dst ) else 0
            dport = len( ports[ dst ] ) + dst_base
        ports[ src ][ sport ] = ( dst, dport )
        ports[ dst ][ dport ] = ( src, sport )
        return sport, dport

    def port( self, src, dst ):
        """Get port numbers.
            src: source switch name
            dst: destination switch name
            sport: optional source port (otherwise use lowest src port)
            returns: tuple (sport, dport), where
                sport = port on source switch leading to the destination
switch
                dport = port on destination switch leading to the source
switch
            Note that you can also look up ports using linkInfo()"""
        # A bit ugly and slow vs. single-link implementation ;-(
        ports = [ ( sport, entry[ 1 ] )
                  for sport, entry in self.ports[ src ].items()
                  if entry[ 0 ] == dst ]
        return ports if len( ports ) != 1 else ports[ 0 ]

    def _linkEntry( self, src, dst, key=None ):
        "Helper function: return link entry and key"
        entry = self.g[ src ][ dst ]
        if key is None:
            key = min( entry )
        return entry, key

    def linkInfo( self, src, dst, key=None ):
        "Return link metadata dict"
        entry, key = self._linkEntry( src, dst, key )
        return entry[ key ]

    def setlinkInfo( self, src, dst, info, key=None ):
        "Set link metadata dict"
        entry, key = self._linkEntry( src, dst, key )
        entry[ key ] = info

    def nodeInfo( self, name ):
        "Return metadata (dict) for node"
        return self.g.node[ name ]
```

© University of South Africa November 2018

```
    def setNodeInfo( self, name, info ):
        "Set metadata (dict) for node"
        self.g.node[ name ] = info

    def convertTo( self, cls, data=True, keys=True ):
        """Convert to a new object of networkx.MultiGraph-like class cls
           data: include node and edge data (default True)
           keys: include edge keys as well as edge data (default True)"""
        return self.g.convertTo( cls, data=data, keys=keys )

    @staticmethod
    def sorted( items ):
        "Items sorted in natural (i.e. alphabetical) order"
        return sorted( items, key=natural )


# Our idiom defines additional parameters in build(param...)
# pylint: disable=arguments-differ

class SingleSwitchTopo( Topo ):
    "Single switch connected to k hosts."

    def build( self, k=2, **_opts ):
        "k: number of hosts"
        self.k = k
        switch = self.addSwitch( 's1' )
        for h in irange( 1, k ):
            host = self.addHost( 'h%s' % h )
            self.addLink( host, switch )


class SingleSwitchReversedTopo( Topo ):
    """Single switch connected to k hosts, with reversed ports.
       The lowest-numbered host is connected to the highest-numbered port.
       Useful to verify that Mininet properly handles custom port
       numberings."""

    def build( self, k=2 ):
        "k: number of hosts"
        self.k = k
        switch = self.addSwitch( 's1' )
        for h in irange( 1, k ):
            host = self.addHost( 'h%s' % h )
            self.addLink( host, switch,
                          port1=0, port2=( k - h + 1 ) )


class MinimalTopo( SingleSwitchTopo ):
    "Minimal topology with two hosts and one switch"
    def build( self ):
        return SingleSwitchTopo.build( self, k=2 )


class LinearTopo( Topo ):
    "Linear topology of k switches, with n hosts per switch."
```

© University of South Africa November 2018

```
    def build( self, k=2, n=1, **_opts):
        """k: number of switches
           n: number of hosts per switch"""
        self.k = k
        self.n = n

        if n == 1:
            genHostName = lambda i, j: 'h%s' % i
        else:
            genHostName = lambda i, j: 'h%ss%d' % ( j, i )

        lastSwitch = None
        for i in irange( 1, k ):
            # Add switch
            switch = self.addSwitch( 's%s' % i )
            # Add hosts to switch
            for j in irange( 1, n ):
                host = self.addHost( genHostName( i, j ) )
                self.addLink( host, switch )
            # Connect switch to previous
            if lastSwitch:
                self.addLink( switch, lastSwitch )
            lastSwitch = switch

# pylint: enable=arguments-differ
```

© University of South Africa November 2018