DYNAMIC BANDWIDTH UTILIZATION IN SOFTWARE DEFINED-BASED CAMPUS NETWORKS

A CASE STUDY OF THE KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

By

Kobby Asare Obeng

(BSc. Telecommunication Engineering)

A thesis submitted to the Department of Telecommunications Engineering, Kwame Nkrumah University of Science and Technology, Kumasi in partial fulfilment of the requirements for the degree of,

MPhil. Telecommunication Engineering

NOVEMBER 2019

DECLARATION

I hereby declare that this study was carried out by me and that to the best of my knowledge and belief, it has not been presented anywhere for the award of a degree and that where use is made of other related work, due acknowledgment is made in the thesis.

Kobby Asare Obeng	…………………………	…………………
Student (20517989)	Signature	Date

Certified by:

Dr. James Dzisi Gadze	…………………………	…………………
Supervisor	Signature	Date

Certified by:

Ing. Dr. Abdul-Rahman Ahmed	…………………………	…………………
Head of Department	Signature	Date

ACKNOWLEDGEMENT

All thanks and praise be to God Almighty whose grace and mercy have brought me this far. Every step of the way in this journey has been his orchestration and I am grateful. I would like to express my profound gratitude to my research supervisor Dr. J.D Gadze for his critical input in shaping this work into what is has become. A big thank you goes to Mr. Justice Owusu Agyeman who contributed immensely to this work. I am highly indebted to Dr. Eliel Keelson and Mr. Courage Logah of the Department of Computer Engineering KNUST and UITS respectively. Their advice and consistent commentary at every phase of the work were very useful. I also appreciate the support of Dr. Omari Sasu of the Department of Mathematics and Professor Lydia Apori Nkansah Dean of the Faculty of Law and their families. I wish to thank all the members of staff in the Department of Telecommunications Engineering for their immense support. A special thank you goes to my parents Dr. Emmanuel Obeng Apori and Mrs. Elizabeth Apori for all the financial and spiritual support given to me from birth even until now. My sincere gratitude to my brothers Kweku and Emmanuel Obeng. God bless Mrs. Emelia Turkson for her encouragement, direction and help. Thank you to Rev. Dr. Joseph Baah Obeng and Pastor Mrs. Adwoa Afrakoma Baah Obeng and their family who have been a strong pillar of support for the past ten years. To Mrs. Roseline Brown who literally pushed me to pursue my admission for this program when I had given up, I say God bless you and the family. A big thank you also goes to all pastors and members of Harvest Chapel International KNUST and Danyame. You are the best family ever. To all the Sunday School Teachers in Fountain Gate Chapel this is the fruit of your tireless labors of love.

DEDICATION

This thesis is dedicated to three great mentors who have shaped my perspective in life. Mrs. Priscilla Yakung, Mrs. Akweley Laryea and Pastor Jeffery Amoasah.

ABSTRACT

The efficient utilization of bandwidth in campus networks is a major traffic engineering issue. It requires a complete knowledge of the underlying physical network architecture as well a means to automate or reactively and proactively program the network. The static nature of traditional network creates a hurdle that must be overcome to achieve the above. The Software Defined Network architecture proposes a novel way to automate, program and dynamically configure computer networks. This work uses the VMware virtualization software and the GNS3 network emulator to convert a traditional campus network into a Software Defined-based campus network. A data plane made up of software-based replicas of network devices is designed and configured to connect to a controller software. A network application scheme is implemented by leveraging the Hierarchical Token Bucket Queuing Discipline which automatically programs bandwidth allocation at the data plane through the controller based on traffic demands. The functionality of the architecture is tested by carrying out a number of parallel-connections to simulate changing traffic patterns. This is done using the Iperf Application. The results show the conversion of a traditional campus network into a Software Defined-based campus network. It also depicts the complete emulation of the entire Software Defined-based campus network. At the data plane of the emulated network, devices are able to forward packets to one another with the most active port forwarding about 9,000 packets. The controller obtains a global of all 11-network devices in the emulated network. The latency between the controller and the software defined switches at the data plane ranges between 50 and 62.5 milliseconds. The throughput between the controller and the software defined switches at the plane ranges between 2 and 9 Mbps. Application Plane to Control Plane communication in the emulated network is executed in an average of 30 milliseconds and bandwidth utilization occurs in a minimum of 11seconds and peaks at 27.5 seconds. It however becomes steady at 17 seconds as traffic patterns vary.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 BACKGROUND OF THE STUDY

Network Technology is currently going through a third major shift. The first was the shift from circuit switching to packet switching. This involved the use of the packet as the main means of transmitting a message from one device to another. The second was the shift from the hard-wired to the wireless means of switching which saw the introduction of Wi-Fi technology, 3G,4G and 5G technologies.

The third revolution in network technology has to do with a shift from the hardware-based mode of networking to a software-based mode of networking [1] This transition is taking place because of the limitations that exist in current networks.

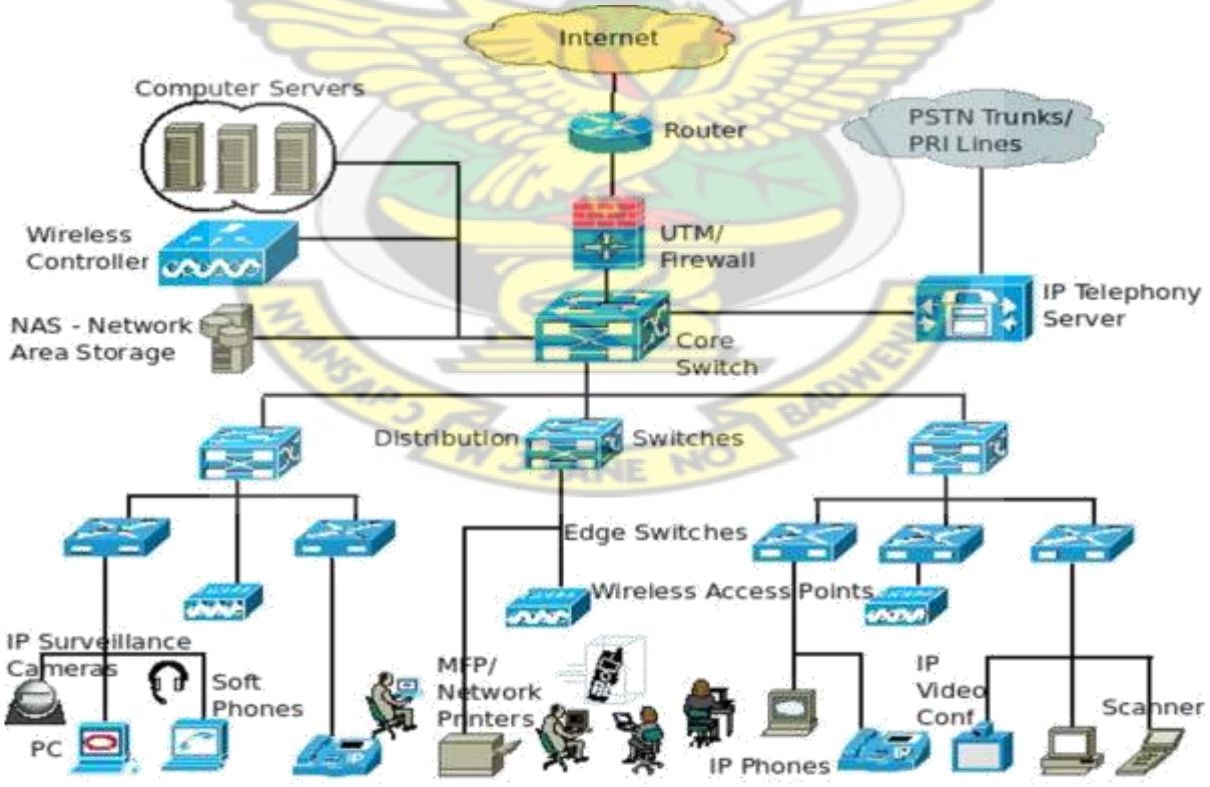


Figure 1.1 A typical enterprise network.

Figure 1.1 represents a typical enterprise network that is responsible for providing different technological services such as IP video conferencing, video surveillance, printing, scanning and internet browsing for various users. It provides these services through a set of network devices such as switches which facilitate communication between various users on different parts of the network.

Existing computer networks like Figure 1.1 have a number of limitations.

These include network provisioning complexity, tightly managed network functions, technology specific connections and purpose-built hardware which carry out each network function. Such an environment could also consist of multivendor equipment manufacturers with their own means of orchestrating and controlling specific equipment. Current networks are also limited by the fact that the applications which run on the devices in the network are programmed to suit current network needs [2].

Network Provisioning Complexity occurs mainly due to the difficulty involved in preparing and equipping existing networks when there is the need to add new devices or configure new services. Such tasks require a very good understanding of the state of the existing network and a proper anticipation of the effects of any change on the state of the network. This is a difficult process which requires days or weeks of planning to properly execute. As a result, network provisioning in current networks happens at off-peak hours to forestall any shutdown and unanticipated changes.

The control logic of most devices in existing networks consists of many protocols which are responsible for carrying out forwarding of packets. Also, with all of these protocols and algorithms running at the same time on the same devices, there is a propensity for them to freeze out and reboot causing frequent down times which affect communication. In order to change any network function, the existing protocol will have to be manually deleted and the

new change effected by manual configuration on every device in the network. This creates a situation where the management of these protocols becomes very difficult.
The connections between devices in current networks are technology specific. Two devices in a typical current network would have to run on software manufactured by the same vendor in order to work together effectively. This poses a problem especially where there is the need to integrate services from different providers who run different proprietary technologies. Also, network engineers would require knowledge of a vast range of technologies in order to manage a network if it consists of many different equipment running on different technologies. This problem leads to issues with service orchestration. This arises from the fact that multiple equipment vendors have their own predefined policies for coordinating the systems that run in their equipment. There are thus various means of coordinating the configuration of the services provided by different equipment vendors in one network environment which poses problems for network engineers.

Network functions such as switching, routing and intrusion detection in current networks are typically implemented in different equipment. Thus, it is not too strange to find routers, switches, firewalls and load balancers in different portions of the design of current networks.

As the network gets larger the topology becomes more complex due to the need for more planning and configuration. This creates problems with implementation, troubleshooting and fault isolation.

The applications that run on the back of current networks are designed in tandem with predefined network functions. This means that the applications and how they function are inflexible and cannot be altered if the underlying network function changes.

The above issues occur primarily because the decision-making and the forwarding functions of devices used in existing computer networks are implemented in a distributed fashioned. Each router, switch or firewall has independent control and forwarding functions implemented in them. In a network that has 50 routers, each of them will have to be configured to facilitate control and forwarding independently. All of these issues make existing computer networks static and inflexible.

1.2. RESEARCH PROBLEM

The Kwame Nkrumah University of Science and Technology plays hosts to a campus enterprise network [3]. The campus network is split up into two main parts; the faculty portion consisting of all the colleges and the residential portion consisting of halls of residence and the residences of the university's workers. Both portions of the University's network have a fixed bandwidth allocation to facilitate networkwide communication. This static assignment of bandwidth for communication creates a problem. Majority of students and lecturers find themselves in the faculty area in the day. During this period, traffic volumes at that portion of the network increases affecting network performance. This is seen in difficulty to browse the web, download content or stream resources for teaching and learning. The reverse is seen in the evening as the concentration of traffic shifts to the halls of residence. The devices that make up the Local Area Network of the school do not have the ability to dynamically prioritize and utilize the allocated bandwidth to facilitate communication in the network. In as much as there can be a manual reconfiguration of these devices it would require systematic planning, addition of new nodes, constant analysis of traffic and vast knowledge from technical experts. The reiteration of such processes every single day would be tiring and costly. Without recourse to increasing resources or overhauling existing infrastructure a way must be found to efficiently use the bandwidth already allocated in a dynamic need-based manner to ensure optimal performance. This situation provides an opportunity for the proposal of a new architecture that will utilize the already existing bandwidth at both portions of the network based on the numbers of students, lecturers and staff members who are accessing the network at any place at any point in time.

1.3 MOTIVATION

Software Defined Networking presents an approach to solve the major limitations that plague existing campus networks. It seeks to provide a programmable open scale approach to designing, building and managing networks.

Through the principle of plane separation, it provides a centralized view of distributed network states.

The Software Defined Networking architecture consists of two planes. These are the control and data planes. The control plane has a central intelligent agent called a controller that implements the logical, decision-making aspects of a network through programmed network policies. The Data Plane consists of programmable OpenFlow switches that facilitate communication between the controller and network devices.

A unified and global view of networks as envisaged by the SDN paradigm creates a powerful centralized platform for efficiently managing networks.

A centralized, self-provisioning network will have the ability to implement changes without the need for the tortuous planning and anticipation that is needed in carrying out provisioning in current networks. Automating of network protocols provides an easier way to configure, run and change the protocols that make the network devices function. It provides a way to dynamically configure network protocols on a needs-based approach.

Adopting a programmable approach to networking means that functions such as routing, switching and firewalling can be implemented in few devices eliminating the need to buy specific devices for specific network functions.

Traffic Engineering is one major aspect of networking in which Software Defined Networking architecture and principles can be applied. Traffic Engineering basically seeks to manage the flow of traffic within a network by taking stock of the topology and changing traffic patterns in order to prevent network resources from being constrained.

Increasing volumes of traffic in networks places a strain on bandwidth which is one of the most important resources in a network. Software Defined Networking proposes a way of efficiently utilizing bandwidth by using the controller to dynamically allocate capacity on a needs-based basis. The proposition seeks to use the controller's knowledge of the characteristics of the links in the network topology to respond to changes in traffic volumes in various portions of the network by prioritizing the bandwidth required for traffic flow in the network. [4]

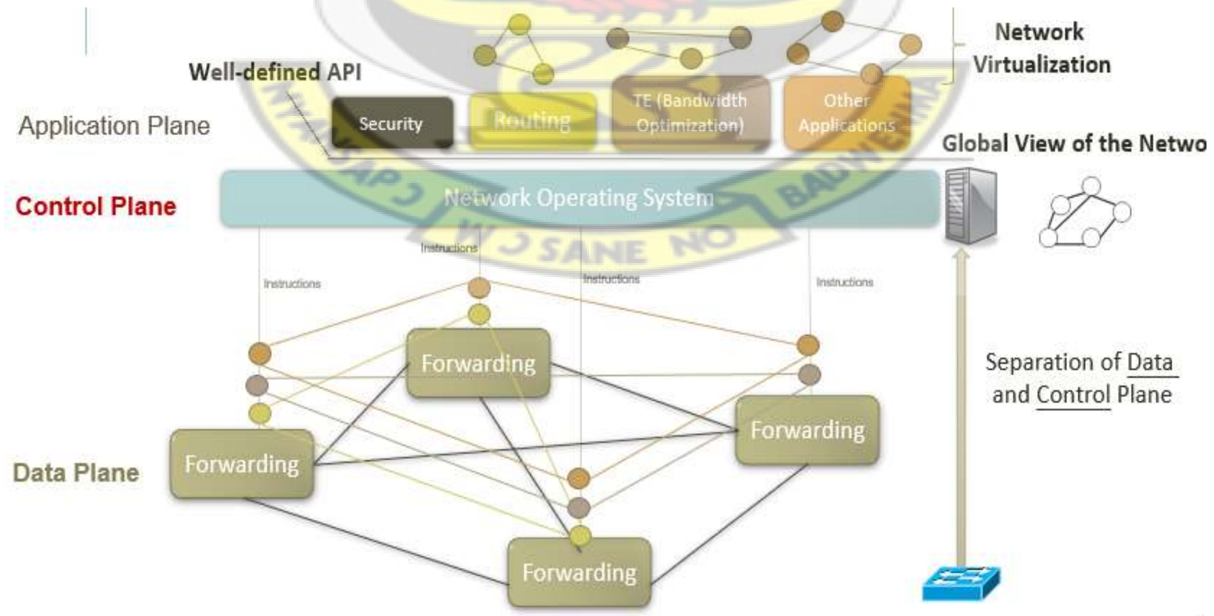


Figure 1.2 The Software Defined Networking Architecture

This work seeks to propose the Software Defined Networking approach to dynamically utilize the bandwidth allocated to campus network, by using the Local Area Network of the Kwame Nkrumah University of Science and Technology as a case study.

1.4 RESEARCH OBEJCTIVES

Goal

To develop a dynamic approach to bandwidth utilization in a campus network using the concepts of Software Defined Networking and the Hierarchical Token Bucket Queuing Discipline.

Specific Objectives

1. Evaluate the existing network infrastructure of a campus network (KNUST)
2. Convert the traditional network infrastructure of KNUST into a Software Defined-based campus network and emulate the separation of the control and data plane functions.
3. Demonstrate the forwarding of packets at the data plane of the Software Defined-based campus network.
4. Demonstrate that communication occurs between the control and data planes of the Software Defined-based campus network.
5. Develop a scheme that can dynamically utilize bandwidth within the Software Defined-based campus network

1.5 SIGNIFICANCE AND CONTRIBUTIONS

This works seeks to leverage the virtual emulation tools GSN3 and VMware as the basis for designing and testing functional parts of the Software Defined Networking Architecture.

This work would present a real-world working prototype that can be used as the foundation for evolving the existing Local Area Network Infrastructure of a campus network (KNUST) into a software based one.

This work would demonstrate how the OpenVswitch leverages the Hierarchical Token Bucket Queuing Discipline as a theoretical basis for dynamic bandwidth utilization in campus networks.

1.6 ORGANISATION OF THESIS

This work is organised into five main sections

- Chapter 1 provides a general introduction and motivation for this work. It also presents the research problem, goal and specific objectives as well as a proposal of the tools to be used to obtain results

- Chapter 2 is a review of relevant works with respect to the history of networks and the need for Software Defined Networking. It also takes a look at various aspect of the Software Defined Networking architecture and reviews the implementations of Software Defined Networking in enterprise and carrier networks.

- Chapter 3 discusses the methodology and theory used in dynamic bandwidth utilization in a Software Defined Network Architecture.

- Chapter 4 discusses the results obtained from the implementation of the dynamic approach to bandwidth utilization.

- Chapter 5 takes a look at the conclusions that can be drawn from the work. It also takes a look at further work which can be done within the context of Software Defined Network Architectures.

CHAPTER TWO

LITERATURE REVIEW

2.0 INTRODUCTION

This chapter presents a review of works related to Software Defined Networking. The Chapter is divided into three main sections. The first section reviews the need for the Software Defined Networking paradigm and introduces the components of the SDN architecture. The second section describes into detail the various parts of the SDN architecture by taking into consideration the devices and protocols which make up each layer of the architecture. It also delves into network virtualization which is the enabler for the design of Software Defined Networks and the concept of Traffic Engineering in SDNs. The third section of this work is a review of research into how the SDN architecture and design principles can be applied to enterprise and carrier networks.

2.1 GENERAL OVERVIEW OF SOFTWARE DEFINED NETWORKING

For decades, computer networks have been built on an interconnection of network devices such as routers, switches, firewalls and end-user equipment like the personal computers, printers and IP phones. These devices are the foundation of enterprise and carrier networks. A typical network device like a router device consists of a hardware portion and a software portion.

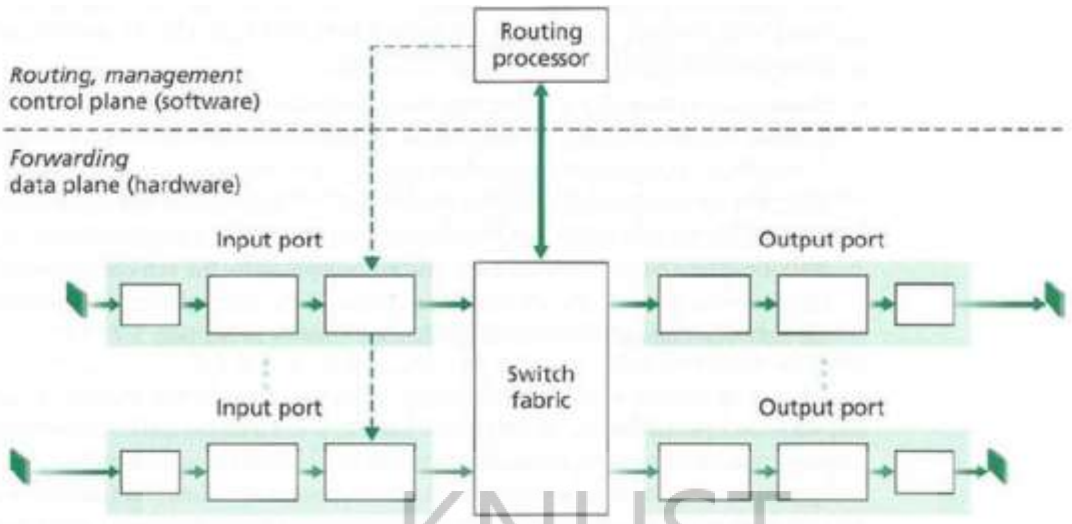

Routing processor
Routing, management control plane (software)
Forwarding data plane (hardware)
Input port
Input port
Switch fabric
Output port
Output port

Figure 2.1 Typical router architecture

The hardware portion known as the data plane, consists of input ports which are linked to output ports via a switching fabric. The switching fabric usually makes use of shared memory and data buffers [5] to move data from source to destination port.

The software portion of the router consists of a routing processor which implements routing, refreshes, routing tables and keep information about connected links. It also is responsible for populating the router's forwarding table.

Thus, both hardware and software functions are tightly baked into a typical network device. In a traditional computer network as seen in Figure 2.1[6], the network element is the one-stop site for all control, management and user data.

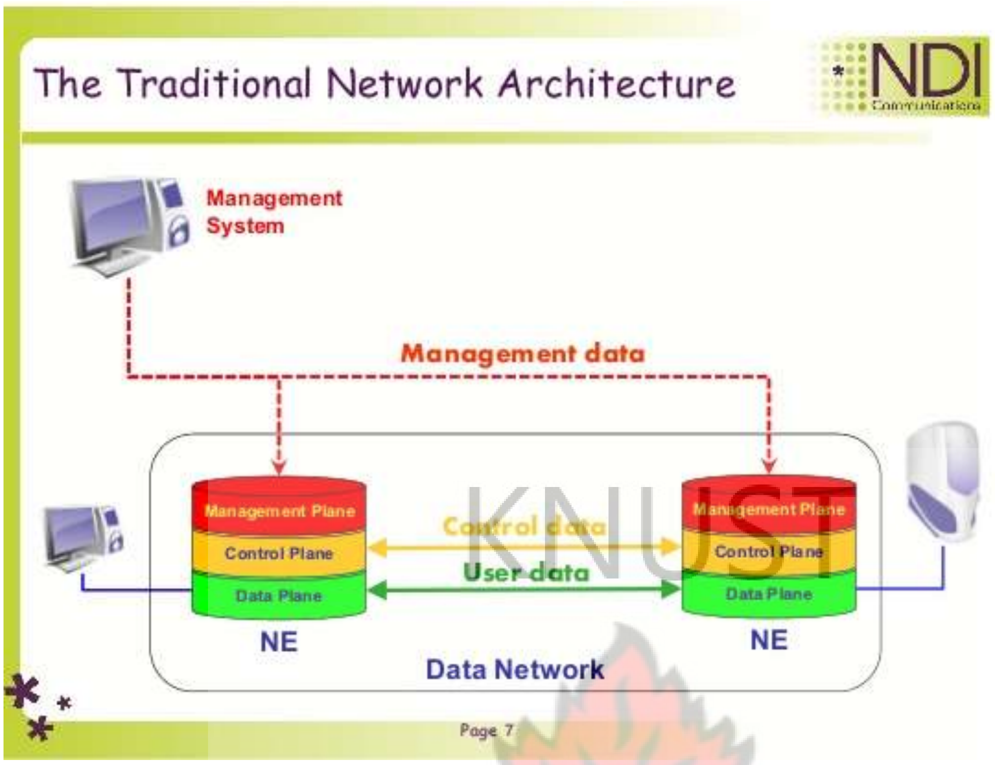


Figure 2.2 Traditional Network Architecture.

Even though the traditional network architecture has facilitated the rise of the information age, the fundamental basis for its design poses a number of issues which have drawn the attention of researchers and industry players.

In [7], protocol complexity, challenging and error-prone network management, performance tuning and internet ossification have been described as problems that exist in traditional networks. Internet Ossification refers to the extreme difficulty of the internet to evolve in terms of the hardware, rules and efficiency due to its relatively static nature.

Other problems faced by traditional networks include vendor dependence, the rise of cloud computing services and the advent of the big data movement which encompasses the Internet of Things, Artificial Intelligent and Machine Learning Technologies.

Software Defined Networking is a new paradigm in computer networks which seeks to make enterprise and carrier networks programmable, highly automated and easily controllable [8].

In [9], the authors describe a Software Defined Network as a network that carries out two main functions. Firstly, it separates the control plane from the data plane. Secondly, it allows multiple data plane devices to be controlled by a single software program.

The proponents of Software Defined Networking proposed an architecture based on the overall vision of a Software Defined Network. This new architecture consists of three interdependent planes. These are the data or infrastructure plane, the control plane and the application plane.

From Figure 2.3, the data and control plane interact via instructions while the control and application plane interact via a well-defined application programming interface. This creates an environment in which the underlying network infrastructure can be controlled, programmed and automated in response to alternating traffic demands placed on it. High-level functions such as routing, security and traffic engineering are written as computer programs and implemented on devices in the data plane through the controller.

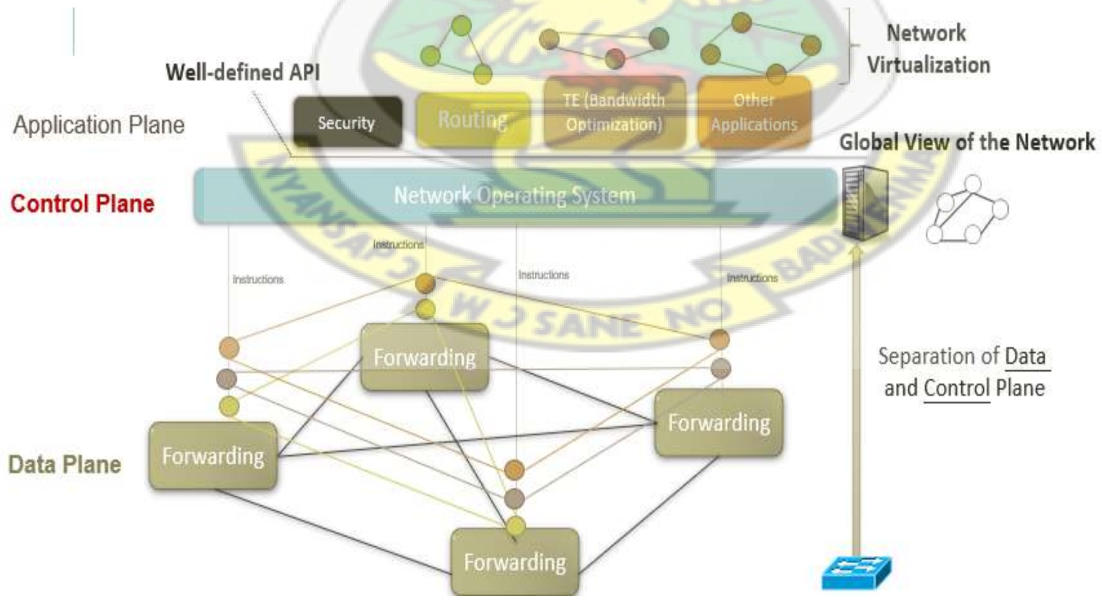


Figure 2.3 Software Defined Networking Architecture.

This new architecture is a clear departure from that of traditional networks. Software Defined Networking places a priority on centralized network control instead of the distributed approach used in traditional networks.

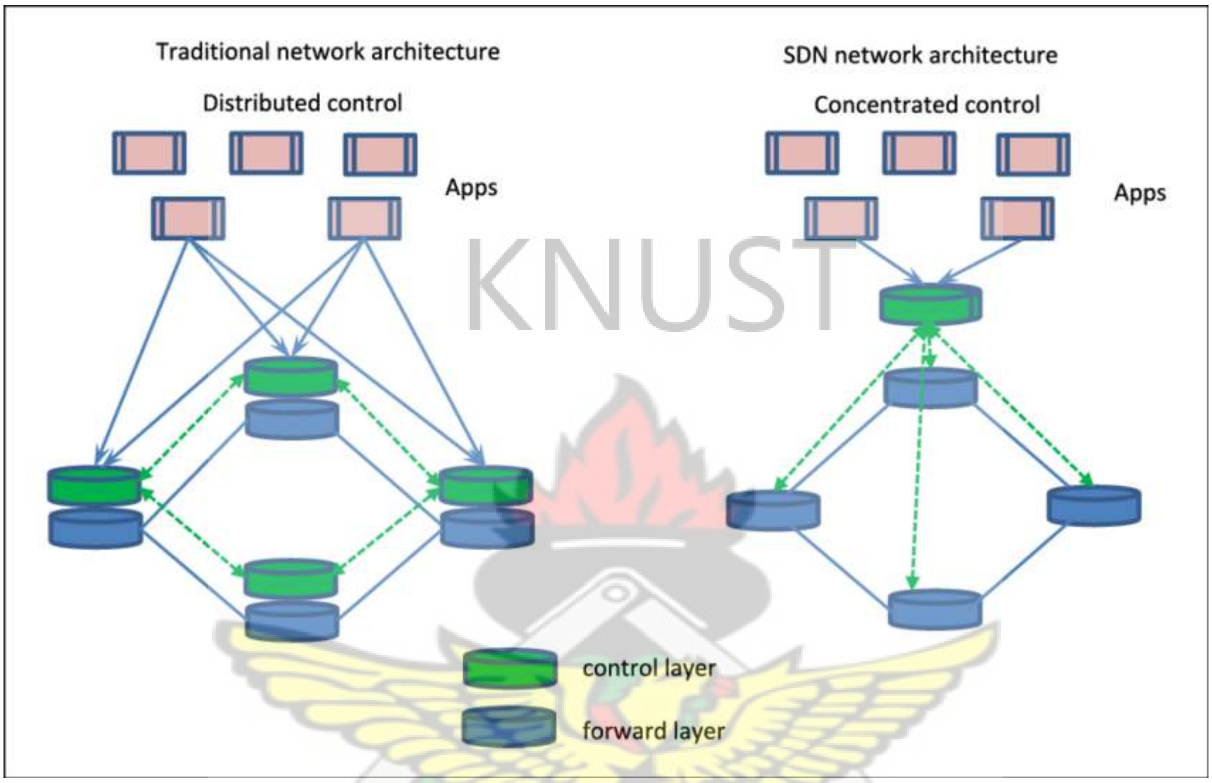


Figure 2.4 Traditional Network Architecture vs SDN network architecture

Centralized control is based on the idea that the various components of architecture work hand in hand to ensure that the network is fully functional.

The next section of this chapter takes a look at the redefined network architecture elements, their functions and how they coordinate to achieve network programmability, automation and control.

2.2 THE DATA PLANE

The data plane is the part of the Software Defined Networking architecture that consists of network devices. In describing the data plane, the proponents of the architecture envision a vendor neutral platform with network devices from different manufacturers.

The data plane of a Software Defined Network is primarily responsible for carrying out the transfer of a packet from the input interface to the output interface.

The key principle underlying the redefined data plane is device simplification [10].

A critical part of the data plane is a network device called a Software Defined Networking Device.

An SDN device is composed of an API for communication with the controller, an abstraction layer, and a packet-processing function [11]. It can be a virtual switch or a physical switch. In the case of a virtual switch, the packet-processing function is carried out by a packet processing software [11]. In the case of a physical switch, the packet-processing function is embodied in hardware 11].

Figures 2.5 and 2.6 show a virtual and physical Software Defined Networking Device.

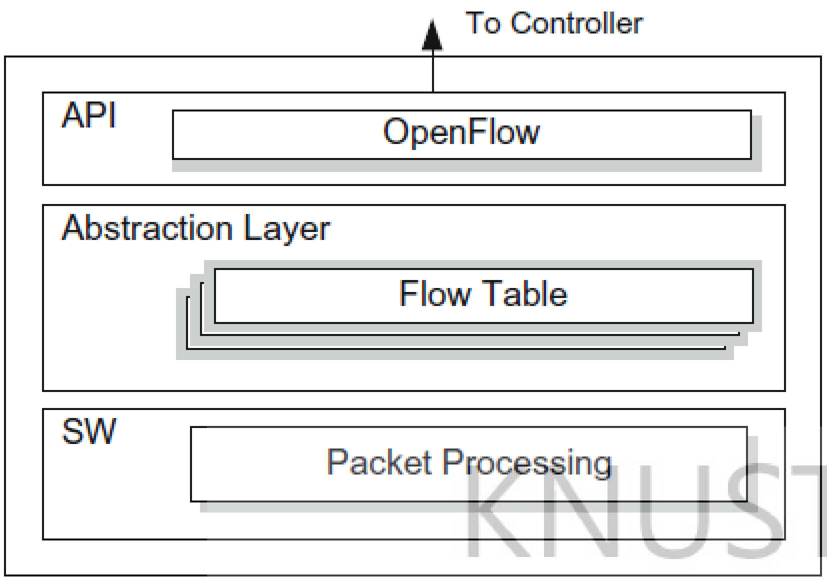


Figure 2.5 Virtual SDN switch anatomy

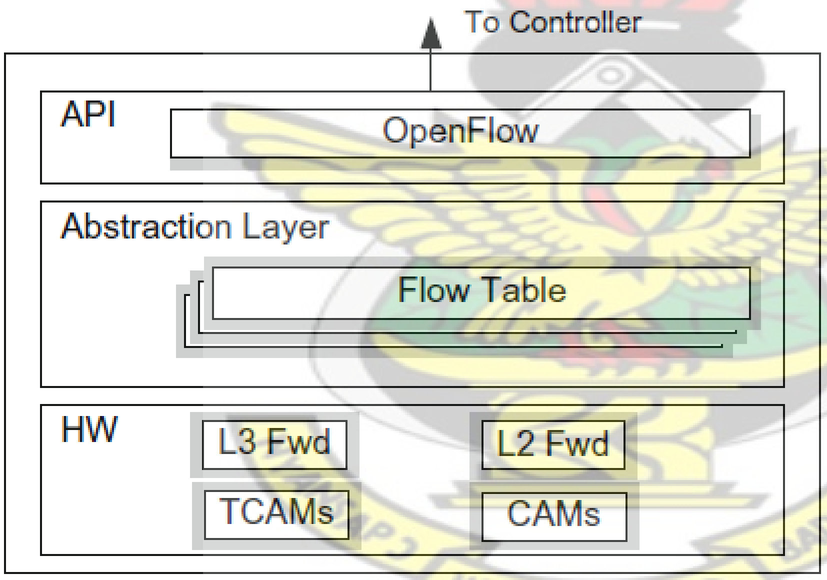


Figure 2.6 Physical SDN switch anatomy

In the hardware implementation, a software defined networking device includes forwarding table responsible for routing and switching. [11]

The advent of Software Defined Networking has seen a sharp rise in the production of a number of Software Defined Networking switches or devices. The Open vSwitch [12] manufactured by Nicira and Big Switch [13] are common examples of such software defined networking switches. Other Software Defined Networking switches include the Arista 7500R Series [14] from Arista Networks and the NFX Series [15] from Juniper. From [16], exposing Open vSwitch control abstractions allows both bare-metal and virtualized hosting environments to be managed using the same mechanism for automated network control. This clearly shows that the software platform for the Open vSwitch can be implemented on any physical or virtual switch once its control abstractions are obtained. Figure 2.7 below shows the anatomy of an Open vSwitch. It consists of a server and a switching daemon which are interconnected and linked to an operating system that provides the platform for control. The above components form the hypervisor platform. A number of virtual machines can run on top of the hypervisor.

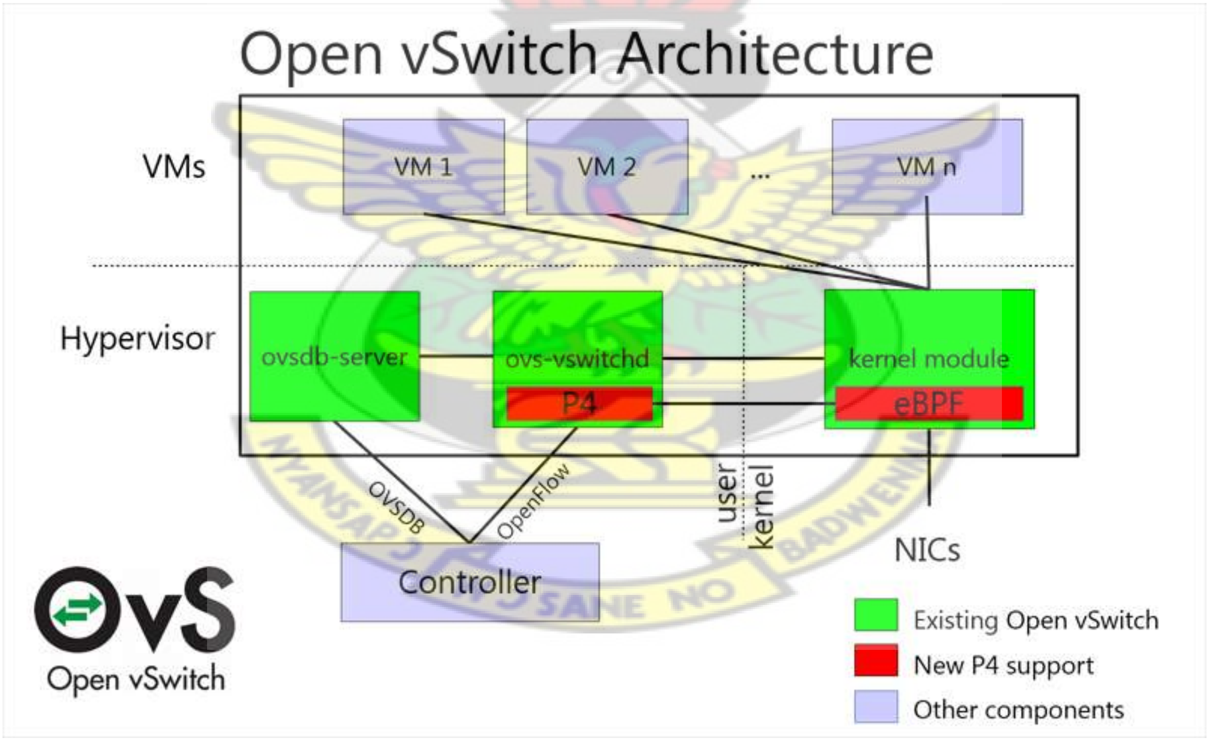


Figure 2.7 Open vSwitch Anatomy

2.3 THE CONTROL PLANE

At the heart of the Software Defined Network is the control plane. The control plane is made up of a controller. The controller keeps track of all devices in the network. It carries out policy decisions and controls all the devices in the data plane.

Also, the controller makes all traffic forwarding decisions and updates SDN-capable network switches in the data plane according to a defined network policy. It is responsible for changing the network rules into actual packet forwarding rules. The network controller establishes a connection to each OpenFlow-capable switch through the OpenFlow protocol [17].

A typical controller has two interfaces. There is an interface between the controller and a software defined networking device and an interface between the controller and the application plane.

The interface between the Controller and a software defined networking device is known as the Southbound Interface or the Southbound API. The interface between the Controller and the application plane is known as the Northbound Interface or the Northbound API.

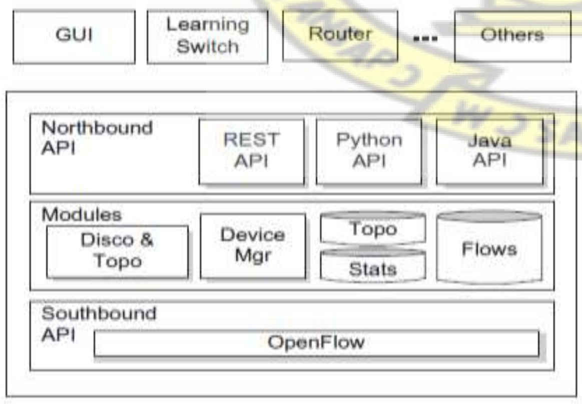


Figure 2.8 SDN Controller Anatomy

The modules portion of the controller anatomy consists of blocks that implement the controller's core functionality. These include the discovery and topology module, the device manager module, the topology and statistics module and the flow module. Common Software Defined Network controllers include the OpenDaylight Controller [18], the ONOS Controller [19], the POX Controller [20], the Ryu Controller [21] and the Floodlight Controller [22].

The OpenDaylight Controller is an Open source platform that provides centralized, programmatic control as well as network device monitoring using open protocols [23]. From [24], OpenDaylight is a Java Virtual Machine software and can be run from any operating system and hardware as long as it supports Java. It uses tools such as maven, a backend framework called OSGI, java interfaces and REST APIs to implement the Software Defined Networking Concept of a Controller.

In OpenDaylight, there are some dynamically pluggable modules, responsible for performing network tasks, which are contained in the controller itself but it is also possible to insert other services and extensions for enhanced SDN functionality. [25]

The OpenDaylight Carbon release shows its place within the overall Software Defined Networking Architecture. These include the data plane element layer, the Southbound interface and protocol plugin layer, the Controller Platform Services/ Applications layer and the Northbound API (Orchestrators and Applications).

The Southbound Interfaces and Protocol plugins layer consists of protocols such as OVSDB, PCEP, IoThttp/CoAP and LACP. The Service Abstraction Layer supports multiple protocols on the Southbound Interface while providing consistent services for modules and the application plane.[23]. The Controller platform itself consists of three key blocks including Base Network Function, Enhanced Network Services and Network Abstractions. Each of these blocks contains distinct protocols that enable the controller to effectively manage the underlying network infrastructure. The OpenDaylight Controller uses the REST, RESTCONF, NETCONF/AMQP Application interfaces to communicate with any set of independent Network Applications and its Graphical User Interface. The Graphical User Interface of the OpenDaylight is called the DLUX [25] or the NEXTUI [25].

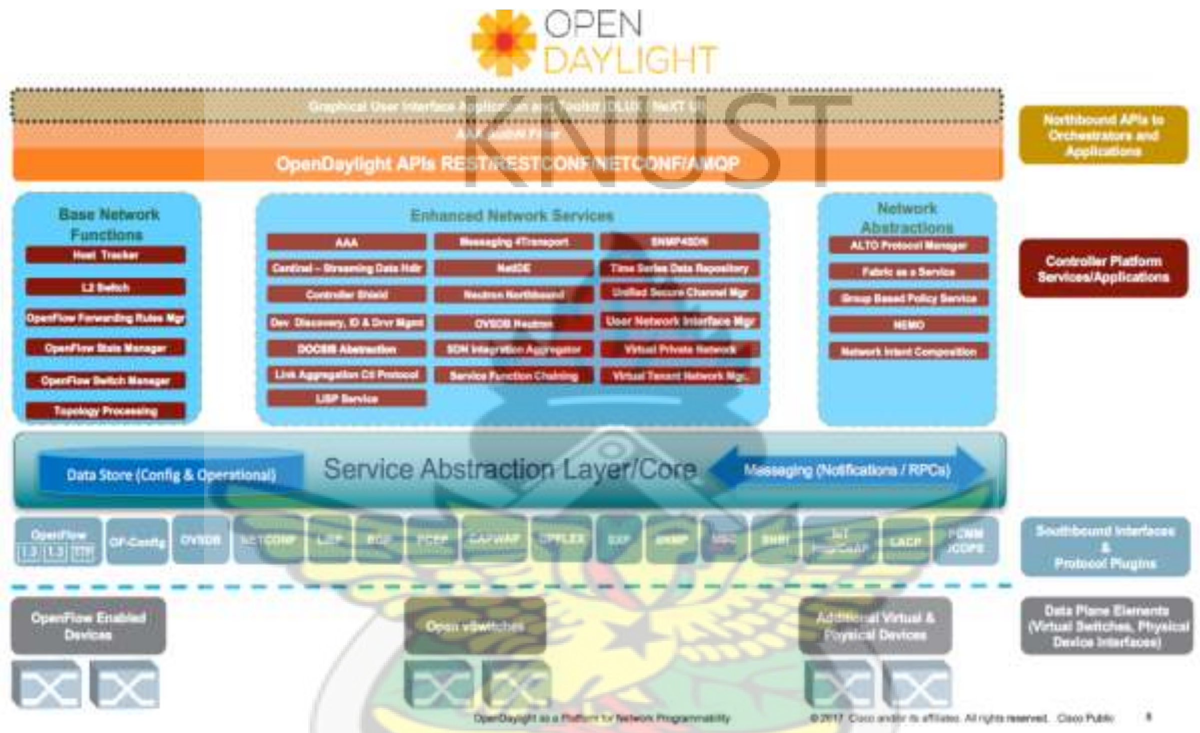

OPEN DAYLIGHT
Graphical User Interface Application and Toolkit (DLUX / NeXT UI)
AAA AuthN Filter
OpenDaylight APIs REST/RESTCONF/NETCONF/AMQP
Northbound APIs to Orchestrators and Applications
Base Network Functions
Host Tracker
L2 Switch
OpenFlow Forwarding Rules Mgr
OpenFlow Stats Manager
OpenFlow Switch Manager
Topology Processing
Enhanced Network Services
AAA
Messaging 4Transport
SNMP4SDN
Cardinal – Streaming Data Hdlr
NetIDE
Time Series Data Repository
Controller Shield
Neutron Northbound
Unified Secure Channel Mgr
Dev. Discovery, ID & Drvr Mgmt
OVSDB Neutron
User Network Interface Mgr
DOCSIS Abstraction
SDN Integration Aggregator
Virtual Private Network
Link Aggregation Ctl Protocol
Service Function Chaining
Virtual Tenant Network Mgr.
LISP Service
Network Abstractions
ALTO Protocol Manager
Fabric as a Service
Group Based Policy Service
NEMO
Network Intent Composition
Controller Platform Services/Applications
Data Store (Config & Operational)
Service Abstraction Layer/Core
Messaging (Notifications / RPCs)
OpenFlow
1.0
1.3
TTP
OF-Config
OVSDB
NETCONF
LISP
BGP
PCEP
CAPWAP
OPFLEX
SXP
SNMP
USC
SNBI
IoT HTTP/CoAP
LACP
PCMM /COPS
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)
OpenDaylight as a Platform for Network Programmability
© 2017 Cisco and/or its affiliates. All rights reserved. Cisco Public
8

Figure 2.9 The OpenDaylight Architecture

2.4 THE APPLICATION PLANE

A computer network application is a software application that uses the Internet or other network hardware to perform useful functions [26].

There are two types of computer network applications. These are Pure Network Applications and Standalone Network Applications. A Pure Network Application is an application created to be used in a computer network to transfer data from one end user to another. A Standalone Network Application is an application that runs on a single end user computer. These include applications such as word processors, database management systems, presentation graphics and spreadsheets. A standalone network application can function even when the computer is offline.

A Software Defined Network application is an application that manages network policies that are programmed on the network devices By utilizing an Application Programming Interface(API),the application is able to configure the network policies as flows to route packets through the best path between two endpoints, balance traffic loads across multiple paths or destined to a set of endpoints, react to changes in the network topology such as link failures and the addition of new devices and paths, and to redirect traffic for purposes of inspection, authentication, segregation, and similar security-related tasks [11].

SDN applications are free, open source flexible, responsive and agile centralised set of control logic that take decisions such as what to do with packets in a Software Defined Network [27]. Based on an abstraction of the network state, management applications can be written to satisfy all the control requirements that may exist in a network while new applications can also be written in the face of new network requirements [28].

There two types of applications that can be developed in the Application Plane of a Software Defined Network Architecture. These are reactive applications and proactive applications.

Reactive SDN applications are programs which modify the devices at the data plane of the Software Defined Network based on the incoming packets that have been forwarded to it from the switches in the network.

Proactive SDN applications are programs which modify the Software Defined Network at the data plane of the Software Defined Network by setting flows proactively on the switches in the data plane. They only respond to changes in the network that require some type of reconfiguration.

2.5 THE SOUTHBOUND INTERFACE

The Southbound Interface is the communication interface between a software defined network device and the controller in an SDN architecture.

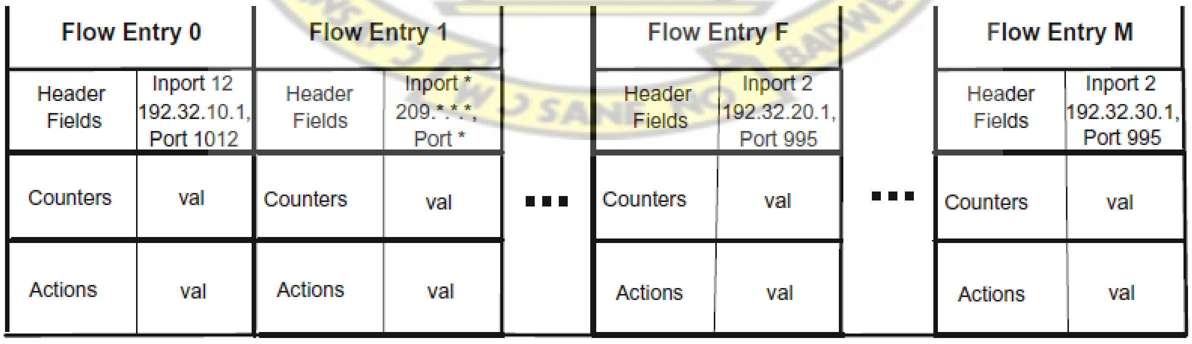
The OpenFlow protocol is the main communication protocol that runs on the Southbound Interface and is described in [8] as the first standard communications interface between the data and control planes.

Prior to Open Network Foundation's creation in 2011, the OpenFlow specification was defined and managed by a group of individuals meeting at Stanford University [29].

There have been four variants of the OpenFlow protocol till date. These are OpenFlow versions 1.1, 1.2, 1.3 and 1.4. A Software Defined network device that is capable of running the OpenFlow protocol is called an OpenFlow switch.

An OpenFlow switch has an OpenFlow channel to the external controller. Using the OpenFlow protocol, the controller can add, update, and delete flow entries in flow tables, both reactively and proactively [30].

An OpenFlow switch has a flow table. A typical flow table is shown below.



Flow Entry 0		Flow Entry 1			Flow Entry F			Flow Entry M	
Header Fields	Inport 12 192.32.10.1, Port 1012	Header Fields	Inport * 209.*.*.*, Port *	...	Header Fields	Inport 2 192.32.20.1, Port 995	...	Header Fields	Inport 2 192.32.30.1, Port 995
Counters	val	Counters	val		Counters	val		Counters	val
Actions	val	Actions	val		Actions	val		Actions	val

Figure 2.10 A typical flow table

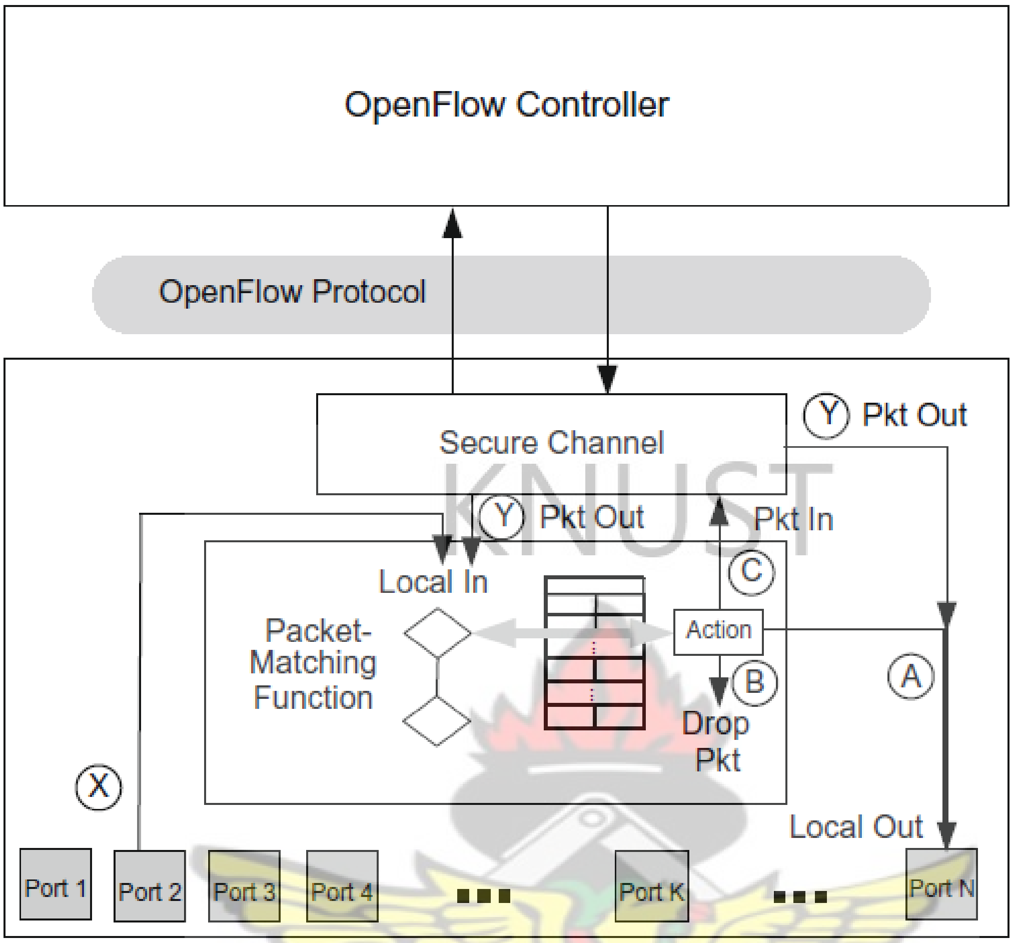


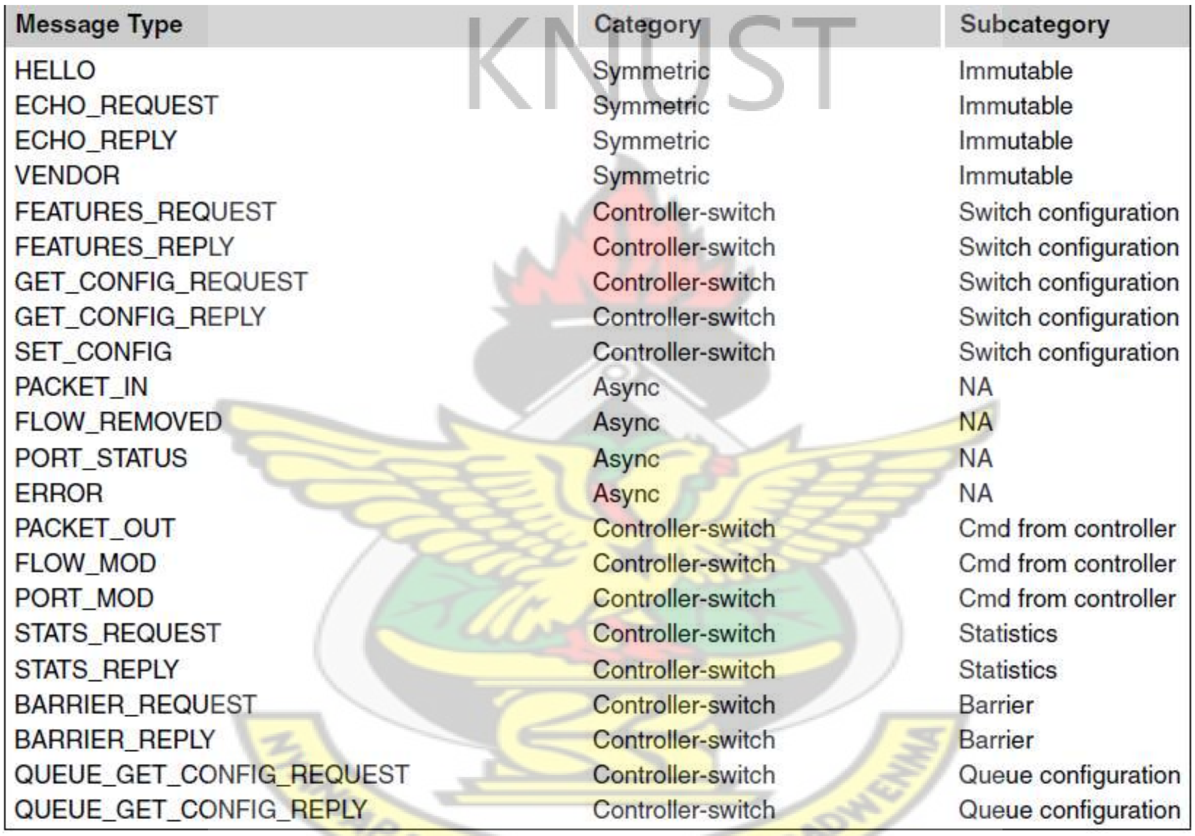
Figure 2.11 OpenFlow Capable Switch

There are three fundamental paths associated with the packets that arrive on the input port of OpenFlow capable switch as shown in Figure 2.11. In Path A, the arriving packet is destined for the output of a local port, in Path B, the packet is destined to be dropped, while in Path C, it is destined to be passed to the Controller. These paths are taken based on the instructions implemented by the packet matching function which is a unique feature of the OpenFlow protocol.

The OpenFlow protocols uses an OpenFlow port. These ports generally make use of scheduling algorithms that allow different quality of service (QoS) levels to be defined for different types of packets. OpenFlow embraces this concept and permits a flow to be mapped to an already defined queue at an output port [11].

The messaging between the controller and switch is transmitted over a secure channel. This secure channel is implemented via an initial TLS connection over TCP [7]. Each message between controller and switch starts with the OpenFlow header which specifies the OpenFlow version number, the message type, the length of the message, and the transaction ID of the message [11].

OpenFlow messages fall into three general categories as shown below.



Message Type	Category	Subcategory
HELLO	Symmetric	Immutable
ECHO_REQUEST	Symmetric	Immutable
ECHO_REPLY	Symmetric	Immutable
VENDOR	Symmetric	Immutable
FEATURES_REQUEST	Controller-switch	Switch configuration
FEATURES_REPLY	Controller-switch	Switch configuration
GET_CONFIG_REQUEST	Controller-switch	Switch configuration
GET_CONFIG_REPLY	Controller-switch	Switch configuration
SET_CONFIG	Controller-switch	Switch configuration
PACKET_IN	Async	NA
FLOW_REMOVED	Async	NA
PORT_STATUS	Async	NA
ERROR	Async	NA
PACKET_OUT	Controller-switch	Cmd from controller
FLOW_MOD	Controller-switch	Cmd from controller
PORT_MOD	Controller-switch	Cmd from controller
STATS_REQUEST	Controller-switch	Statistics
STATS_REPLY	Controller-switch	Statistics
BARRIER_REQUEST	Controller-switch	Barrier
BARRIER_REPLY	Controller-switch	Barrier
QUEUE_GET_CONFIG_REQUEST	Controller-switch	Queue configuration
QUEUE_GET_CONFIG_REPLY	Controller-switch	Queue configuration

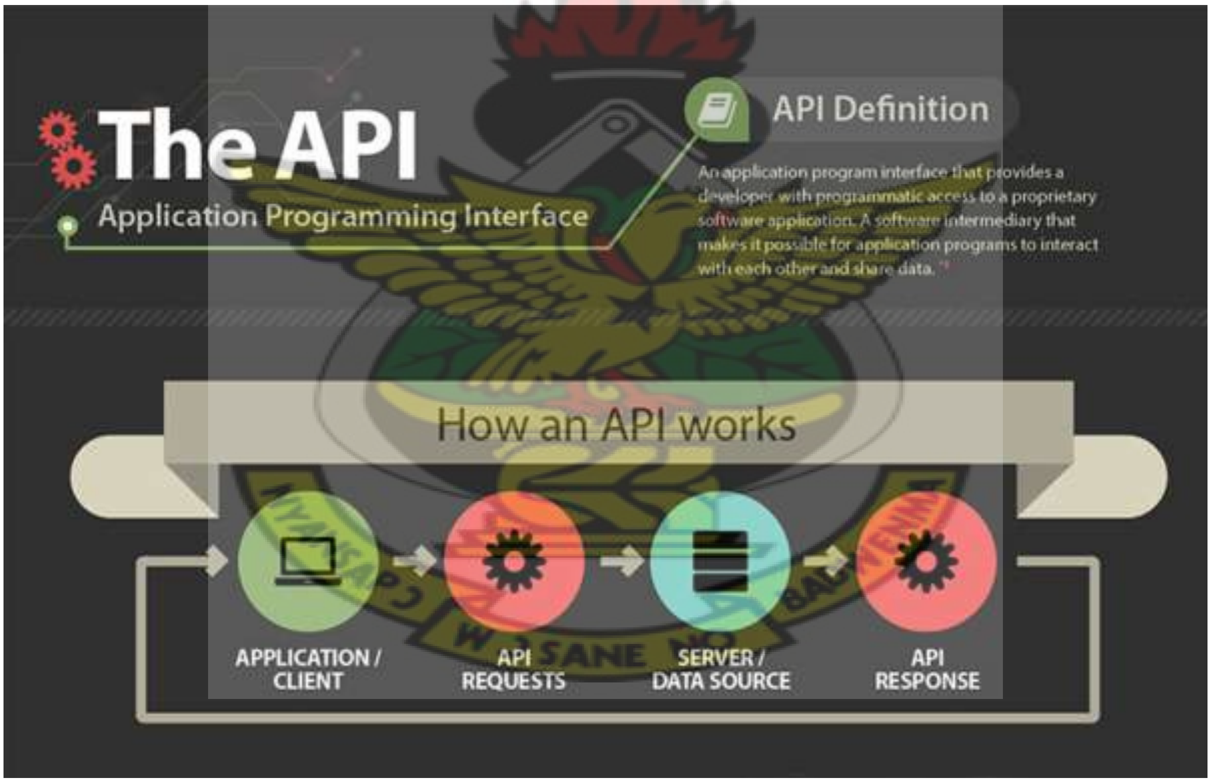
Figure 2.12 Types of Messages in OpenFlow

2.6 THE NORTHBOUND INTERFACE

The Northbound Interface is the communications interface between the Controller the Software Defined Network Application. This communication is facilitated by a variety of Application Programming Interfaces.

An Application Programming Interface is a set of routines, protocols, and tools for building software applications which specifies how software components should interact [31].

A typical application programming interface uses a client-server model to send and receive requests to a software application residing on the client.



An application program interface that provides a developer with programmatic access to a proprietary software application. A software intermediary that makes it possible for application programs to interact with each other and share data.

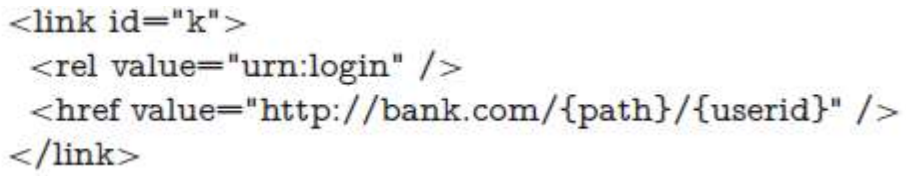
Figure 2. 15 Application Programming Interface.

In the OpenDaylight controller, the typical application programming interfaces used are REST, RESTCONF, NETCONF and AMQP.

2.6.1 The Rest Architecture

REST, or REpresentational State Transfer, is an architectural style for providing standards between computer systems that make it easier for systems to communicate with each other. REST-compliant systems, often called RESTful systems, are characterized by statelessness and their ability to separate client and server concerns [32]. Adopting REST for the SDN northbound API offers decentralized management of dynamic resources by relying on connections between resources to discover and manage them as a whole [33]. It also allows network elements to be dynamically configured and reconfigured in a distributed fashion. Additionally, REST can provide service-based compositions using various programming languages on different platforms [33]. A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data based on the REST architecture [34].

In case of SDN, the REST API can be used to program network devices like switches, routers and NAT devices [35]. A typical REST API has five main layers. These are Connections, Communications, Identification, Representation and Description. The Connections layer consists of links between the resources implemented in any programming languages and running on any devices. The communications layer uses methods or protocols such as HTTP to communicate with resources. Uniform Resource Identifiers are used by the API to identify resources at the Identification layer. The Representation layer uses hypertext language like XML to represent information from the COMMUNICATIONs. Figure 2.16 is an XML element showing a URI in the line 2 and the http protocol in line 3.



```
<link id="k">
  <rel value="urn:login" />
  <href value="http://bank.com/{path}/{userid}" />
</link>
```

Figure 2.16 XML element in a REST API

The Description layer uses descriptions such as REST charts to describe possible representations.

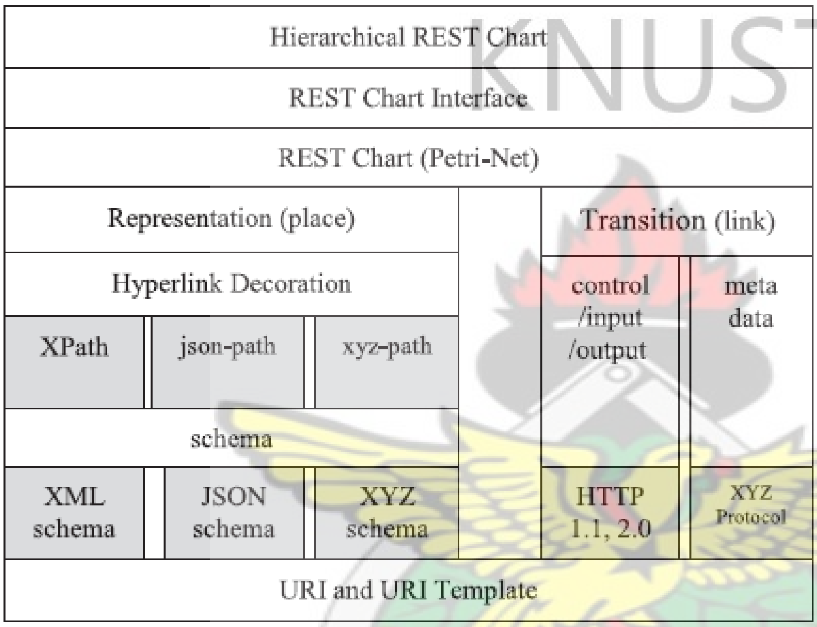


Figure 2.17 The Rest Chart Figure 2.16 XML element in a REST API

2.6.2 Netconf

NETCONF defines a network device management mechanism by setting or changing the current state of a network equipment using a technology called Remote Procedure Call that allows programs to make procedure calls or functions from a different address space. These commands are encoded in XML and sent to the network device using Secure Shell protocol [36]. The NETCONF protocol has four layers namely content, operations, messages and secure transport layers as shown in Figure 2.18 below.

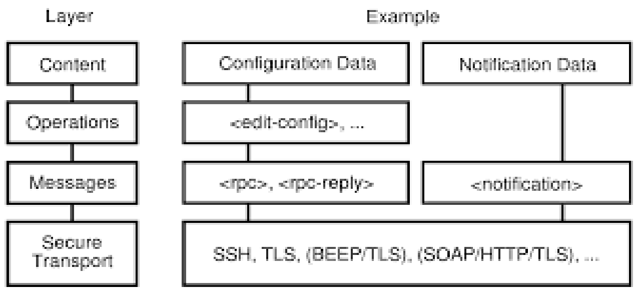


Figure 2.18 NETCONF protocol layers

2.6.3 Restconf

RESTCONF is an IETF management protocol that uses an HTTP API to provide an additional simplified interface for the NETCONF protocol.[37]

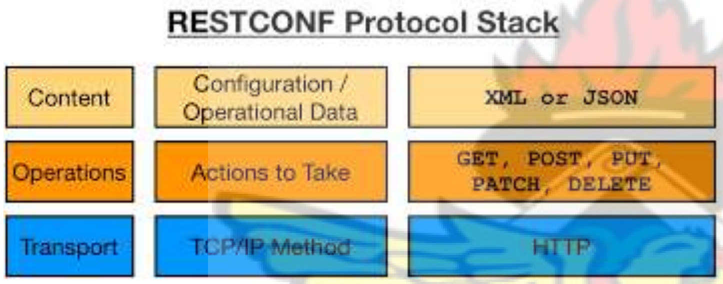


Figure 2.19 RESTCONF protocol stack.

2.7 NETWORK VIRTUALIZATION

Network Virtualization is the process of combining hardware and software network resources and network functionality into a single, software-based administrative entity. This software-based administrative entity is called a virtual network. Network virtualization helps network operators divide physical computing resources to ensure efficient use of computer resources. In the networking field, physical equipment can be abstracted as a resource pool from which virtual entities are created and redundancy assured. In the computing field, virtual machines (VMs) are created on the resource pool and their backups are set to physically distributed locations; these correspond to slices and protection in the networking field [38]. Network virtualization provides an effective means of creating software-based replicas of hardware devices by using physical network infrastructure. Network virtualization be used to create multiple virtual networks above a shared physical network, each of which can be deployed and managed independently [39]. Network virtualization is viewed as a solution to the rigidity and inflexibility of the current internet architecture. There are two main forms network virtualization. These are external virtualization and internal virtualization.

External virtualization refers to the means of putting together many networks, or various parts of networks, into a virtual unit. Internal virtualization refers to the provision of network-like functionality to the software containers on a single system.

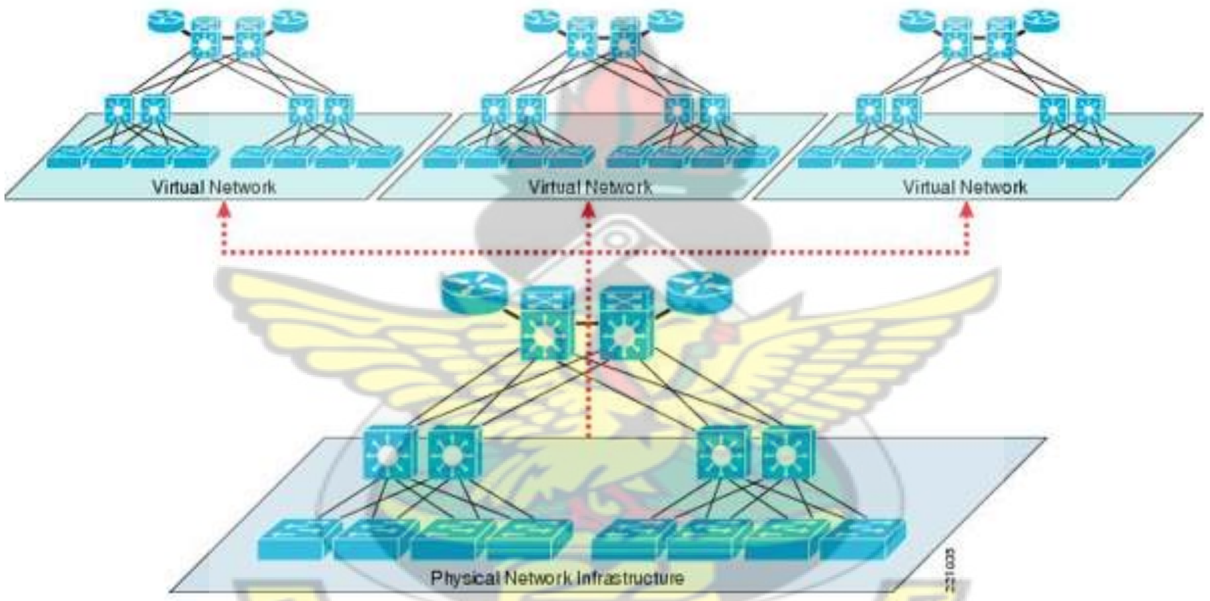


Figure 2.20 Network Virtualization

In the field of Software Defined Networking, researchers have to design virtual networks in order to test the functional parts of the Software Defined Networking architecture. Such virtual networks usually depict the data plane devices, connections to the controller and an application utilizing the REST API. One common tool used in creating SDN networks is Mininet. Mininet is a network emulator which is used for research in SDN. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking [40]. Figure 2.21 shows a typical network topology created using Mininet and visualized in the MiniEdit graphical user interface.

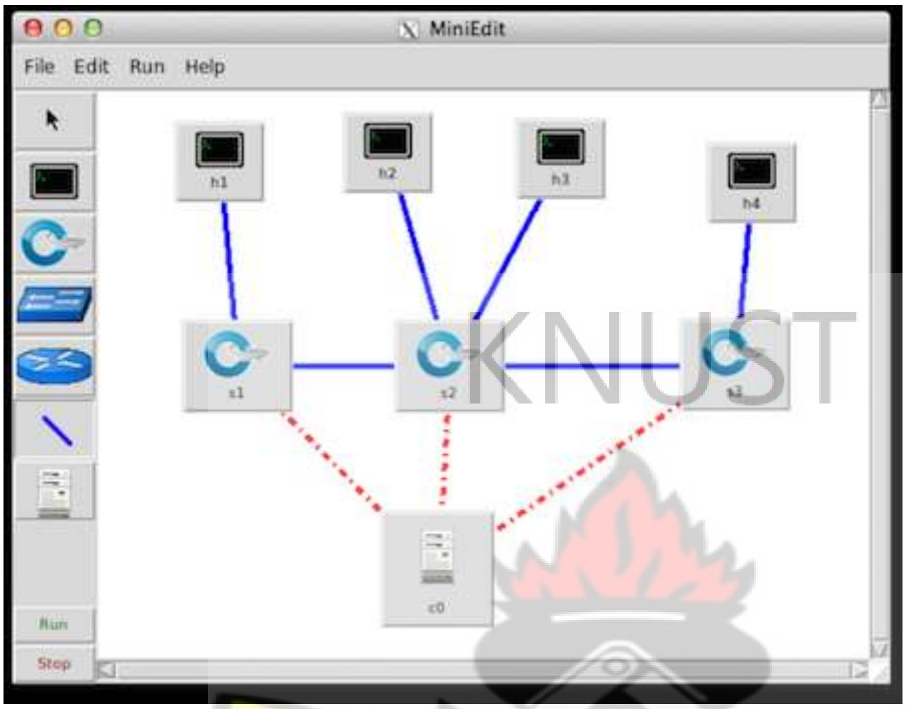


Figure 2.21 A Typical Mininet Topology

The most significant limitation of Mininet is that it cannot work efficiently at high loads because it has one default scheduler that multiplexes CPU resources [41]. Also Mininet is not capable of prototyping large-scale networks having many nodes. Other Software Defined Networking emulators include IMUNES [42], ESTINET [43] and EMULAB [44]. In this work, the GNS3 Virtual Machine [45] and VMware Workstation 14[46] are used to carry out the emulation of the entire Software Defined Networking Architecture. The choice of these two tools stem from the fact that they can be used to carry out nested virtualization. Nested virtualization is the ability of a virtualization tool to replicate different operating systems with the same tool. Nested virtualization provides the ability to test Software Defined Networking's proposition of vendor neutrality, openness and device simplification. In addition to the above, the GNS3 Virtual Machine makes use of a graphical interface that facilitates the design, configuration and testing of a virtual network.

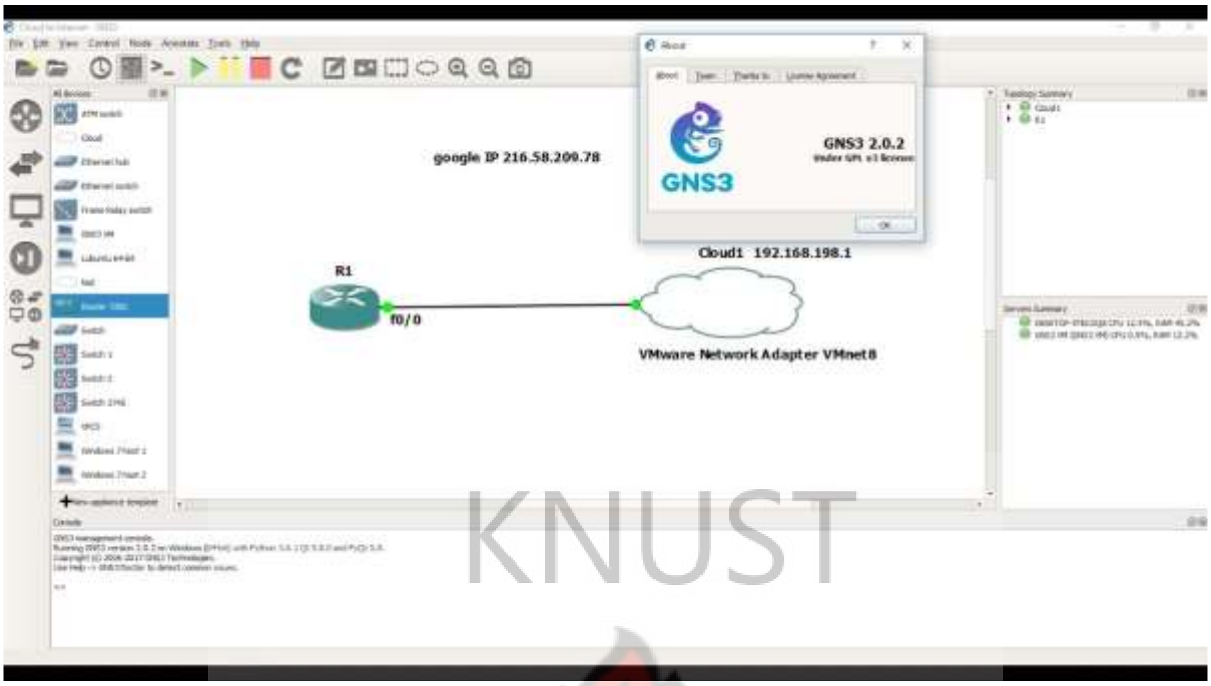


Figure 2.22 GNS3 topology created using the GNS3 GUI running in VMware

2.8 TRAFFIC ENGINEERING IN SOFTWARE DEFINED NETWORKS

Software Defined Networking proposes an easier way to carry out traffic engineering in enterprise and carrier networks. Traffic Engineering aims to provide efficient use of network resources based on the traffic in a network usually involving measurement, modelling, characterization and control of IP traffic [47]. The framework for traffic engineering in SDN includes two parts: traffic measurement and traffic management. Traffic measurement mainly studies how to monitor, measure and acquire information about the current state of a Software Defined Network [48]. In the SDN-paradigm, an SDN controller can be used for Traffic Engineering to improve network utilization, reduce packet loss and delay when the entire network consists of OpenFlow and traditional network devices [49]. Bandwidth Utilization in Software Defined Networks would require the implementation of a Traffic Engineering network application. Based on SDN, dynamic bandwidth adjustment can be efficiently implemented for improving the flexibility of resource allocation and resource utilization by monitoring the bandwidth value an end-to-end (E2E) service actually uses, monitoring how much bandwidth of a transport channel is actually used by end-to-end services and knowing which transport links the end-to-end service is carried across [50].

Based on the above statements, various queuing disciplines can be implemented as network applications to carry out bandwidth adjustment in a Software Defined Network. One such queuing discipline is the Hierarchical Token Bucket Queuing Discipline.

2.9 THE HIERARCHICAL TOKEN BUCKET QUEUING DISCIPLINE

A queuing discipline is a resource sharing mechanism that governs how packets are buffered while waiting to be transmitted. The Hierarchical Token Bucket Queuing Discipline (HTB) is a class-based queue discipline that controls the use of bandwidth on a given output link and implements efficient resource allocation. It uses the concept of multilevel token buckets to allow for efficient dynamic control of the egress bandwidth on a given link. HTB is made up of three class types known as root, inner and leaf classes. The HTB traffic shaper has the ability to carry out bandwidth sharing in a network.

The Hierarchical Token Bucket (HTB) classification is used for traffic control [51]. It is used to guarantee bandwidth to classes, allows engineers to define upper limits to inter-class sharing and allows the prioritization of classes[52]. The root class represents the minimum and maximum amount of bandwidth (guaranteed bandwidth) that is set for communication between network devices. Any form of service request that occurs in a network is allocated bandwidth borrowed from the root class. The bandwidth for such communication is represented by an interior class. Each amount of bandwidth that is shared from an interior class is termed as a leaf class.

HTB allows cross-device bandwidth sharing and control-borrowing [53]. This approach is well suited in scenarios where a user has a fixed amount of bandwidth and each application is allocated a guaranteed bandwidth, with the possibility of specifying how much bandwidth can be borrowed [54]. Every class has an associated Ceiling Rate (CR) and Rate (R). The CR specifies the highest amount of bandwidth that a class can use while R represents the lowest amount of bandwidth the class can use. When one class requires bandwidth greater than R, it borrows bandwidth from its parent class until CR is reached, when this class reaches CR, the packets are queued until new tokens are available in the token-bucket [54].

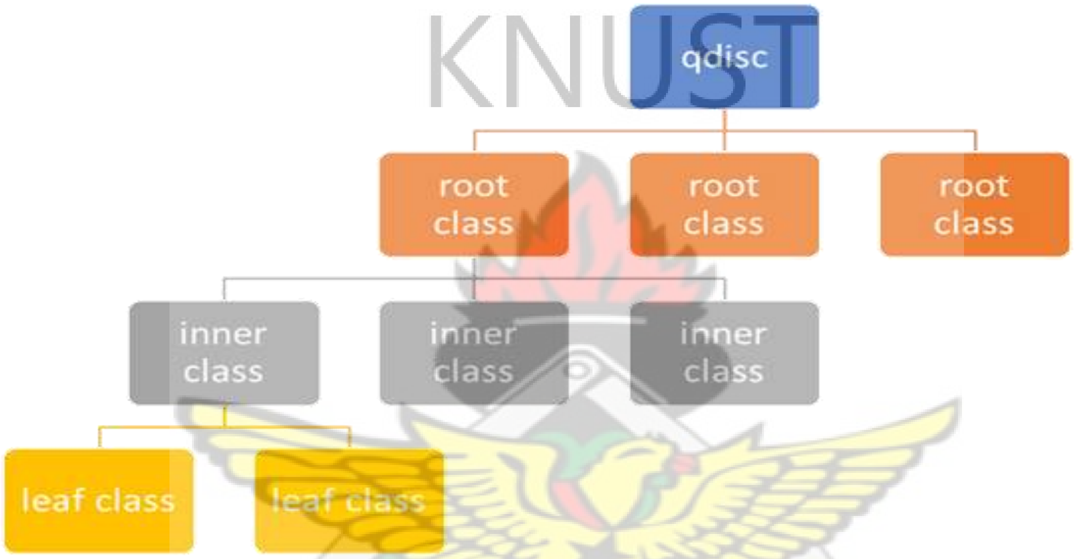


Figure 2.23 HTB class structure and borrowing

The Hierarchical Token Bucket Queuing Discipline uses tokens and buckets to dynamically share bandwidth. Traffic shaping is performed by the Token Bucket Filter. The Token Bucket Filter implementation in Linux has two filters, each with its own buckets. In order to be transmitted to the receiver, a packet must be able to pass both the filters. A packet being sent from a transmitter to a receiver passes through both filters. The second filter typically allows packets to flow out of it faster in order to limit the speed of burst traffic. When implemented in a Software Defined Network device, the HTB goes through four phases namely classifying, policing, scheduling and borrowing. The figure below shows the algorithm for the operation of HTB.

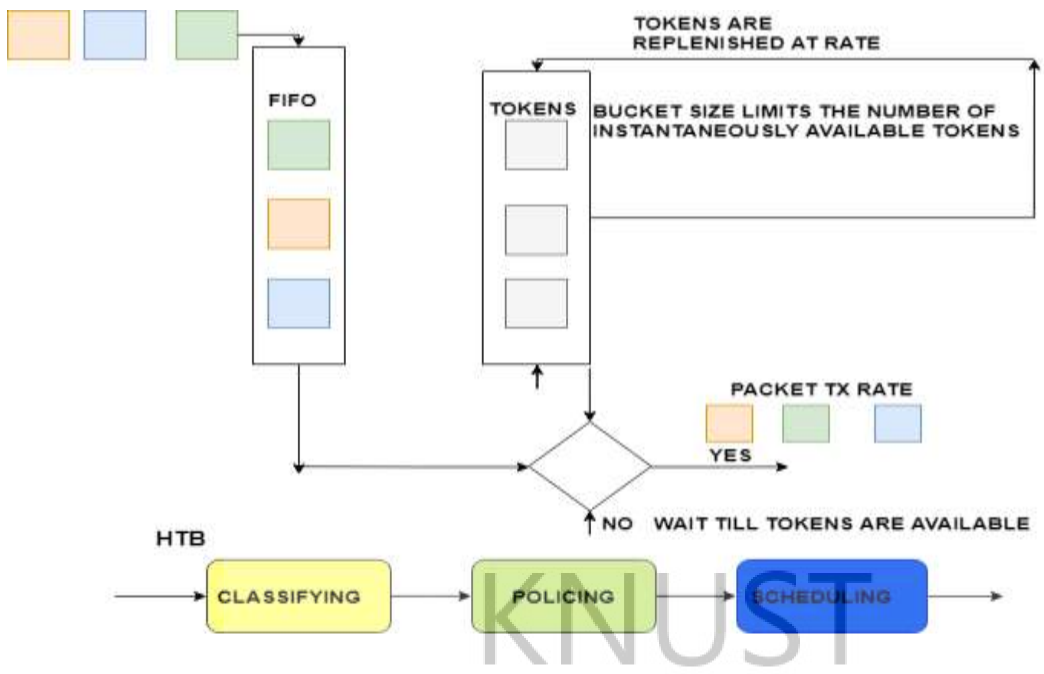

TOKENS ARE REPLENISHED AT RATE
FIFO
TOKENS
BUCKET SIZE LIMITS THE NUMBER OF INSTANTANEOUSLY AVAILABLE TOKENS
PACKET TX RATE
YES
NO
WAIT TILL TOKENS ARE AVAILABLE
HTB
CLASSIFYING
POLICING
SCHEDULING

Figure 2.24 Hierarchical Token Bucket Queuing Discipline Operation

2.10 RELATED RESEARCH IN SOFTWARE DEFINED NETWORKING

There has been a vast amount of research into the implementation of Software Defined Networking concepts in the Enterprise and Carrier Networks. The main issues in enterprise networks are security, load balancing, Quality of Service configurations, bandwidth management and traffic control. Some work has also gone into the application of Software Defined Networks in optical and satellite networks. This section takes a look at related research in the above fields with respect to the tools used for emulation and the metrics used when carrying out research into Software Defined Networking concepts.

2.11 SOFTWARE DEFINED NETWORKING IN THE ENTERPRISE NETWORK

The reviewed works in this section represent research carried out on the application of Software Defined Networking in Enterprise and Data Centre Networks.

2.11.1 Load Balancing And Firewall Implementations In Software Defined Networks

In [55], *S. Bhelekar et al* from the Sardar Patel Institute of Technology in India present a dynamic load balancing strategy in Software Defined Networking. They took into account the number of active connections to a set of individualized servers and the shortest path taken from a specific client to a specific server in order to avoid congestion. The authors used the Mininet Emulation Tool and VirtualBox Software to simulate a fat tree topology of a modern data centre network at the data plane. The data plane topology consisted of ten OpenFlow capable switches and eight host devices. At the control plane, the OpenDaylight controller used RESTCONF APIs to communicate with an application plane implementation of Dijkstra's algorithm and least connections. The dynamic nature of the Software Defined Network is seen in the load balancer's ability to determine the least path to a server at certain specific points in time. At time t=60 seconds, host 2 has the least number of connections and is selected as the server of choice hence avoiding congestion.

One key network functionality that can be implemented in the application plane of a Software Defined Network is security. *N. Zope et al* in [56] of Usha Mittal Institute of Technology, Mumbai, India took a look at replacing a physical switch in a network by virtual switches and the development of a firewall and load balancing application in a Software Defined Network. The work seeks to demonstrate that most of the firewalls in traditional computer networks can be replaced by software firewalls. Using the Floodlight controller and the REST API, a set of firewall rules were constructed and pushed to an independent physical network consisting of an Open vSwitch and two hosts. The firewall module is capable of pushing flows containing the firewall rules to the hosts to either block or allow traffic.

D. Satasiya and Raviya Rupal D [57] from the University of Pune, India carried out an analysis of a Software Defined Network implementation of a firewall designed by *Karamjeet Kaur et al* [58] who created topology including the POX controller, 6 OpenFlow switches and 5 hosts. Karamjeet Kaur et al built a learning switch application and firewall application which restricted or allowed the traffic by proactively placing rules into the network based on key network parameters such as the IP address. into an OpenFlow switch-based source MAC address, destination MAC address (Layer 2), source IP address, destination IP address (Layer 3), network protocol, destination port (Layer 4). Results show the firewall was able to block access to the whole network. It also allowed web access to one host in a scenario where all other communications were on going. In addition to the above, after firewall implantation latency got increased which implied that the firewall introduced overheads, throughput got reduced which means unwanted traffic was reduced. *D. Satasiya and Raviya Rupal D* [57] are of the view that lack of authentication and authorization, fraudulent rule insertion and lack of access control and availability are drawbacks to the implementation in [58]. They propose an SDN architecture which implements a stateful firewall capable of analysing security threats at all levels of the SDN architecture.

A stateful firewall implementation was carried out by [59] *P. Krongbaramee and Y. Somchit* from Chiang Mai University in Thailand. The research focused on the use of the Open vSwitch to configure a stateful firewall using a TCP three-way handshake. The Mininet emulation tool was used with virtual server in the Digital Ocean Cloud. An analysis of the results shows that the average connection time of host in the stateful firewall is 20.06 milliseconds while the average connection time of host in the stateless firewall is 18.05 milliseconds. The stateful firewall increases the time for each connection for only about 2.01 milliseconds or about 11.14% longer. This is attributed to the SDN switch having to update rules when denying or permitting the connection of an external host to the network. The authors believe that performance would be faster when using with a hardware based SDN switch.

2.11.2 Bandwidth On Demand And Quality Of Service Implementations In Software Defined Networks

To exploit the capabilities of Software Defined Networking when implementing a Bandwidth on Demand service, *A. Mendiola et al* [60] proposed a framework called DynPaC, which they believe is able to provide efficient switching services based on bandwidth and vlan utilization. They used two Software Defined Networks in two different research labs located in Cambridge and Belgrade to create a multidomain network architecture. Each of the networks run based on the ONOS controller. In the emulation, a client based at the Cambridge campus tries to request two Bandwidth on Demand services from the server located in Belgrade using a portal developed on the DynPaC framework which resides on an intermediary network in Spain. DynPaC calculates the optimal intra-domain path taking into account the amount of bandwidth needed, the VLAN of the service and the service which has already been requested. The authors were able to prove that Bandwidth-on-Demand service provisioning is possible. This includes intra-domain bandwidth provisioning, limiting of traffic rate based on QoS requirements, link failure reactivity and automatic installation of backup paths.

Quality of Service in Software Defined Networks has gained attention over the past few years, *A. O. Adedayo and B. Twala*,[61] from University of Johannesburg carried out research into the use of Quality of Service configurations to control network bandwidth, latency and throughput. They leveraged the Mininet emulation tool to create a two-host topology at the data plane connected to an Open vSwitch. The Ryu controller was used as the centralized controller and Hierarchical Token Bucket Queuing discipline was used to ensure each queue in the QoS settings had a number of resources allocated to them. The authors used three scenarios to check for the use of traffic policing on the ports on the switch. They used the IntServ classification to assign bandwidth to certain services DiffServ classification to assign bandwidth to certain services. The results show that the use of DiffServ classification facilitates the scalability of QoS in the network. Also, the IntServ classification is capable of assigning bandwidth to different queues representing different classes of traffic with different priorities. The traffic policing scenario is also able to limit traffic to 10Mbps, while dropping traffic that exceeds 10Mbps.

F. Volpato et al [62] proposed a network application (Autonomic QoS Broker) and a controller module that implements the OpenVswitch Database Management Protocol (OVSDB). These two components used to provide QoS management based on the prioritization of queues in an SDN environment. The Autonomic QoS Broker is a resource and QoS provisioning application that was designed based on the MAPE-K control loop functionality. It was implemented to carry out analysis, planning, execution and monitoring of network resources and works in conjunction with a QoS configuration module. The module performs the configuration of switch's QoS resources and the management of forwarding rules. The Mininet emulator, Open vSwitch (OVS) (version 2.5.0) and the Floodlight controller were used to carry out the experiment. The Broker improved flows throughput and packet loss rates. Flows in the same network path had the same latency values. A change in path caused an increase in latency values.

2.11.3 Vlan Configuration In Software Defined Networks

The concept of Virtual Local Area Network configuration in Software Defined Networks was explored in [63] by *Van-Giang Nguyen and Young-Han Kim* in Seoul, Korea. The authors designed and implemented an application for easily managing and flexibly troubleshooting the VLANs in an SDN architecture. They used an all OpenFlow data plane connected to the Floodlight Controller on the control plane. They developed a REST API-based module in the Floodlight controller to create static vlans in the underlying data plane. Two hosts in the vlan

10 network were able to reach each other. The authors also went further to set up a hybrid testbed consisting of an OpenFlow switch connected to an HP switch. Two hosts were connected to the HP switch and one host is connected to the OpenFlow switch. The REST API-based module was able to create vlans in the hybrid testbed as well. They also carried out an implementation of a dynamic vlan application based on the Mininet in out-of-band control mode. They used Floodlight controller in the same Mininet host. The results of the above showed that the time for sending packets and installing the flow modifications were independent of the type of topology. The latency on the switch and the controller were also very similar.

2.11.4 Traffic Classification In SDN-Based Wireless LANS

An investigation into the operability of Software Defined Networking in wireless local area networks was carried out by *A. Amelyanovich et al*[64]. The work proposed a solution to the control traffic in wireless local area networks using a simulated version model of the St. Petersburg State University of Telecommunications network. The work sought to prove end-to-end quality of service support using traffic classification and priority-based queuing. The emulation was carried out using a two-switch-two wireless access point topology in Mininet. The OpenDaylight controller was used to facilitate the flow of different streams configured from the two Open vSwitches to the two hosts. The results showed that classification of traffic based on Differentiated Services Code Point (DSCP) values in the IP-headers of packets is possible in wireless networks. It reduces the number of simultaneously operating wireless devices. As A result, the effect of interference is also reduced. Another significant observation is the change in traffic priority with respect to applications which are sensitive to delay. Also, the work showed that the most frequent interval between messages was in the range from 100 μs to 1 ms. This shows that the controller responded to changes in the network.

2.11.5 Traffic Management And Measurement In Software Defined Networks

Z. Shu et al[48] from China and South Korea, propose a framework responsible for monitoring and analysing real-time network traffic as a prerequisite for traffic management. The authors designed the framework in a hybrid IP/SDN network consisting of two SDN capable switches and a single controller. The proposed framework utilizes the Link Layer Discovery Protocol (LLDP) implemented in an SDN capable switch to gain a complete view of the network while measuring the number of flows generated as the network elements communicate.

2.11.6 Management Of Network Resources And Data Flow In Software Defined Networks

In [65], *M. S. Olimjonovich* from the Tashkent University of Information Technologies used the Mininet simulator, a topology consisting of 5 switches and 10 host in the Python language to carry out a research on management of network resources in SDNs. Specialized language modules were developed for the hardware switches using Python programming. These software modules were remotely managed through an encrypted SSH-channel. He designed a REST API to increase the amount of unused resources of the networks allowing approximately a 30% increase in network management efficiency.

2.11.7 Routing In Software Defined Networks

A key factor in any enterprise network is routing. *S. Kaur et al* [27] designed a static router application in the Mininet emulator consisting of three hosts, an Open vSwitch and the POX controller. The router application was built in the POX controller. Using ping utility and elinks web browser the functionality of the application was tested. The authors used round trip time as metric. They created 3 different file sizes and checked how long it took to make http requests to them. Results showed that as the size of files increases, the round-trip time of the application also increases. A Layer 3 learning application which was also designed provided the same result but the round-trip time of the router application was less when compared to Layer 3 learning application.

2.11.8 Utilization Of Qos And Routing In Software Defined Networks

S.-C. Lin et al[66] from the Georgia Institute of Technology proposed a framework for optimizing QoS and Routing in Software Defined Networks. They used tenant isolation, prioritization and flow allocation in a multitenant network for utilization. They designed network and switch hypervisors to isolate and prioritize tenants to create fine-grained isolation in the network. A dynamic flow allocation was also proposed in their work to enable optimal flow route selection. They also designed an adaptive feedback management tool to combine virtualization and flow allocation. These three implementations were carried out using algorithms. In analysing the results, it was seen that the network and switch hypervisors were able to isolate three tenant networks in three subnets. The results show that feedback tool was able to maintain the number of shared links at a constant value. It was able to optimize route selection and wisely utilize link capacity for future flows. It was also capable of providing bandwidth for future flows.

2.11.9 Queue Scheduling In Software Defined Networks

Umadevi et al.[67] proposed a scheduling algorithm for controlling the incoming data traffic in a Software Defined Network in an effective manner. The simulation was carried out using the OpenFlow package in OMNeT++. The authors constructed a multi-level switching queue. Multiple queues were maintained with varying priority levels. Analysis of the results shows that in case of a normal First Come, First Serve (FCFS) queue, bits are received by the queue only after the 1350 breakpoint. In the multilevel queue, packets were serviced even as the queue size decreases below 1350. Also, the packet drop count value was as high as 1600 in case of a normal FCFS queue. In the multilevel priority queue, packets enter different levels of queues thereby minimizing traffic in a single queue. The number of packets dropped in the case of multilevel queues was as low as 0.

2.11.10 Network Management And Performance Monitoring In Software Defined Networks

Based on four objectives, *Veena et* al[68] from PES University in India proposed a way to manage and monitor a network using Software Defined Networking. The work was centred around optimizing the Mininet emulation tool in order to create custom topologies, collecting and preserving historical data from a controller for analysis, the reduction of layer 2 broadcast traffic in data centres and the introduction of Cross Layer Utilization algorithms for better resource utilization in Data Centres. They designed an Abstraction layer between the infrastructure and application planes. Also, Pseudo MAC addresses were used to reduce the ARP broadcast traffic in the emulated data centre. The enhancement of the Mininet Emulation tool, enabled the researchers specify the different link parameters such as bandwidth and latency using a simple text script. They were also able to create varied network topologies to test their ideas and new protocols.

2.12 SOFTWARE DEFINED NETWORKING IN CARRIER NETWORKS.

Although relatively new, research is being carried out into the application of Software Defined Networking to carrier networks. The section below discusses the application of Software Defined Networking architecture to switched backhaul, IoT, satellite and 5G networks.

2.12.1 On Demand Bandwidth-Based Pricing In Software Defined Networking

Gu et al [69] from Japan designed an application for on-demand bandwidth pricing using the Software Defined Network Architecture. The application was based on a Stackelberg game constructed to analyse the competitive communication between an ISP and a home network. A pricing strategy was determined using the Nash equilibrium solution of the Stackelberg game. Using the pricing strategy, network subscribers can decide the bandwidth to be reserved in an on-demand basis. The research was carried out using an SDN enabled carrier network consisting of an SDN-enabled backhaul, a Controller, OpenFlow switches, a Controller API and a data plane consisting of wireless access points. The simulation was carried out using MATLAB R2016a. All applications were modelled using a Mathematical Network Model Simulation. Results show that during off-peak, mid-peak, and peak time, the payoff of network subscribers is improved by 388.9%, 134.6%, and 19.8% respectively. The payoff of ISP is improved by 98.5%, 47%, and 7%, respectively. Results show that the optimal price increases with the increase of traffic load. Also, with the increase of traffic load, a large portion of surplus goes to ISP.

2.12.2 Software Defined Networking For Satellite Networks

A Chinese group *Fei et al* [70] in 2017 led research into the implementation of Software Defined Networking in satellite networks. They propose OpenSatNet, a platform for software defined satellite networking research. OpenSatNet uses lightweight OSlevel virtualization, including network namespaces and virtual network devices to emulate a realistic satellite network. It also implemented a user-friendly graphical user interface (GUI). The component of the OpenSatNet are an osndaemon, osn-gui and Scenario Designer. Authors adopted Open vSwitch (OVS) and Floodlight as switch namespace and controller namespace respectively. OpenSatNet uses the Scenario Designer to configure the emulated networks. It manages the location of satellite nodes, configures link parameters and manages the coverage of satellites.

The Link Configuration module configured topology and link parameters dynamically during the emulation. The Scenario Designer was developed by using reports exported from popular satellite emulators. The OSN-GUI controls the daemon through an API based on sockets.

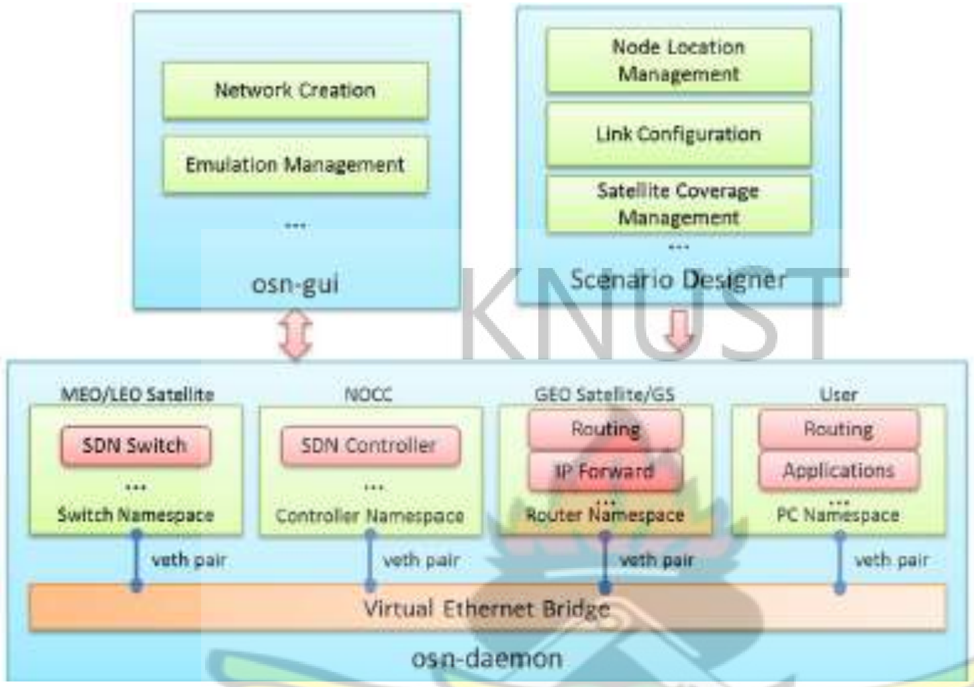


Figure 2.25 OpenSatNet architecture by **Fei et al**

2.12.3 Software Defined Networking For IoT Networks

In [71], *T. Theodorou and L. Mamatas,* from the University of Macedonia Greece stake a claim for Software Defined Networking in Internet of Things Networks. Using a Wireless Senor Node Network (WSN) at the data plane, they demonstrate CORAL-SDN, an SDN solution which uses intelligent centralized control mechanisms to dynamically change the protocol functionalities of Wireless Sensor Nodes. It supports flexibility to the challenging requirements of the Wireless Sensor Nodes while allowing for architectural scalability. The architecture was tested using two tests beds implemented in Ghent and Macedonia. Researchers were able to utilize a configuration manager to choose a type of topology control through algorithms for node advertisement and flow establishment. The centralized intelligent network manager was able to configure routes, setup wireless operating channels and antenna channel check rate. Different data sizes and data transmission frequencies can also be applied using the right protocols. The results show that software defined networking principles can improve control in IoT networks while providing efficient solutions.

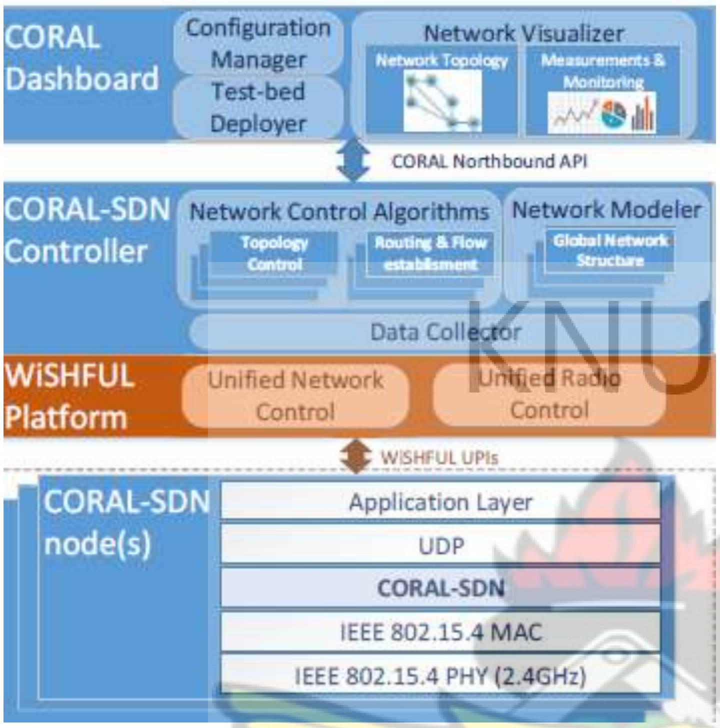


Figure 2.26 CORAL-SDN for IoT

2.12.4 Software Defined Networking For Fifth Generation Networks

Researchers from Georgia Institute of Technology in [72] *I. F. Akyildiz et al* proposed SoftAir, an SDN architecture for 5G cellular systems. The SoftAir architecture is a consists of three planes. Its data plane is made up of a Software Defined Network Radio Access Network, Software Defined Baseband Servers and A Software Defined Core Network. The control plane consists of a network controller. Management applications make up the Application plane. The authors employ Network Functions Virtualization to expand the SDN architecture for the 5G network. By using three network management tools namely mobility-aware control traffic balancing, resource-efficient network virtualization, and a distributed/ collaborative traffic classifier, authors were able to provide for scalability of the network architecture. In conclusion the researchers state that Wireless SDNs provide cellular networks with the needed flexibility to evolve and adapt according to the ever-changing network context for 5G cellular systems.

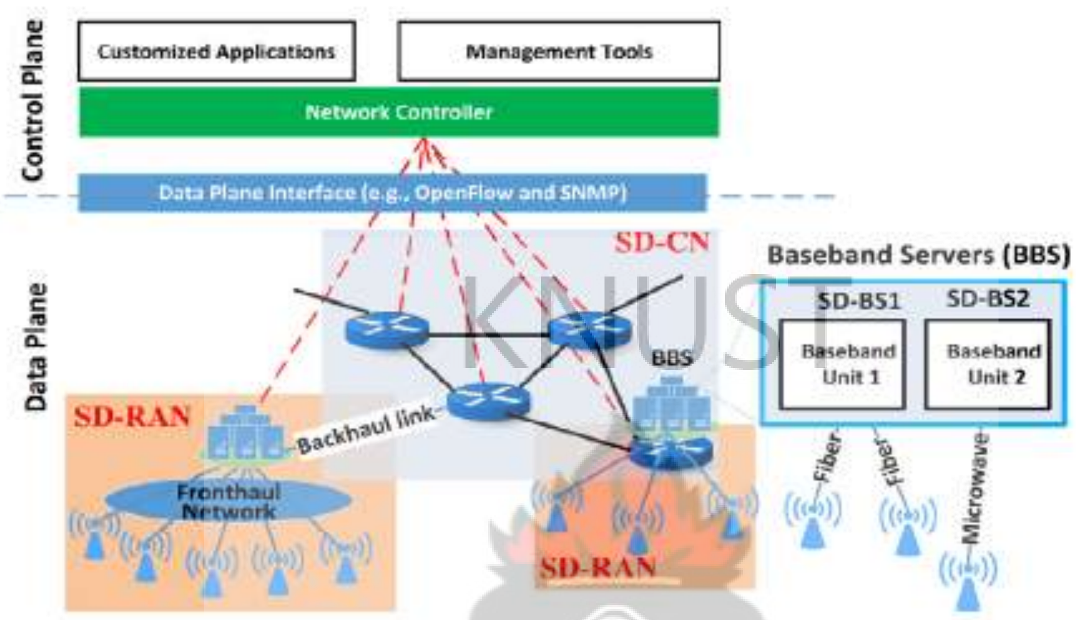


Figure 2.27 SoftAir Architecture for 5G Networks

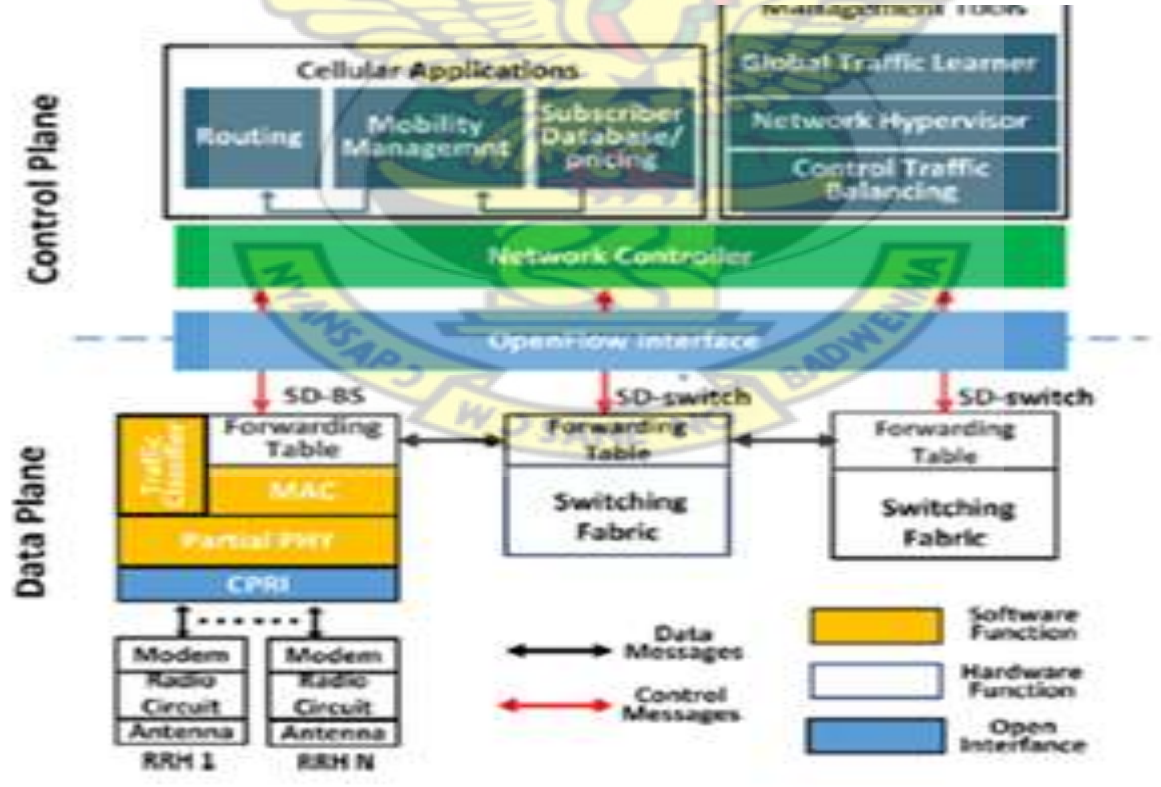


Figure 2.28 NFV implementation of SoftAir Architecture.

2.13 CONCLUSION

This chapter reviewed the concept of Software Defined Networking by introducing and describing its architecture in detail. It also took a look at the protocols and the devices that make up the SDN architecture. Network Virtualization and Traffic Engineering were also reviewed. The last part of this chapter was a review of scholarly work carried out in the field of Software Defined Networking

CHAPTER THREE

METHODOLOGY AND THEORECTICAL BACKGROUND

3.0 INTRODUCTION

In this chapter, the methodology used in converting a typical traditional Campus Network into a Software Defined Campus network is discussed. The chapter also takes a look at the methodology and theory employed in the development of a dynamic bandwidth utilization scheme in a Software-Defined-based Campus network.

The chapter is divided into three main sections. The first section describes the process of converting a traditional campus network into a software-defined campus network. This conversion involves mapping of network elements and the reproduction of the mapped elements using an appropriate emulation tool. The second section presents the theoretical background used in the development of an optimal dynamic bandwidth shaping scheme using the Hierarchical Token Bucket Queuing Discipline.

The third section discusses the development of a Software testbed that implements optimal dynamic bandwidth sharing in an SDN campus network. It also takes a look at the theoretical basis for the communications that occur within a Software Defined-based network.

3.1 SOFTWARE-DEFINED BASED CAMPUS NETWORK

In this section the method used in converting a traditional campus network into a software-defined-based campus network is discussed. A typical campus network is made up of three levels consisting of a core switching level, a distribution switching level and an edge switching level as shown in Figure 3.1. The core switch is the direct point of connection of the entire network to the internet, the traditional telephone network and other external networks. The distribution switches create redundant paths from the core switch to the edge switches. The edge switches represent the access portion of the network to which end users and end user equipment are connected. These switches have both the data plane and control plane functionalities baked into them as shown in Figure 3.2

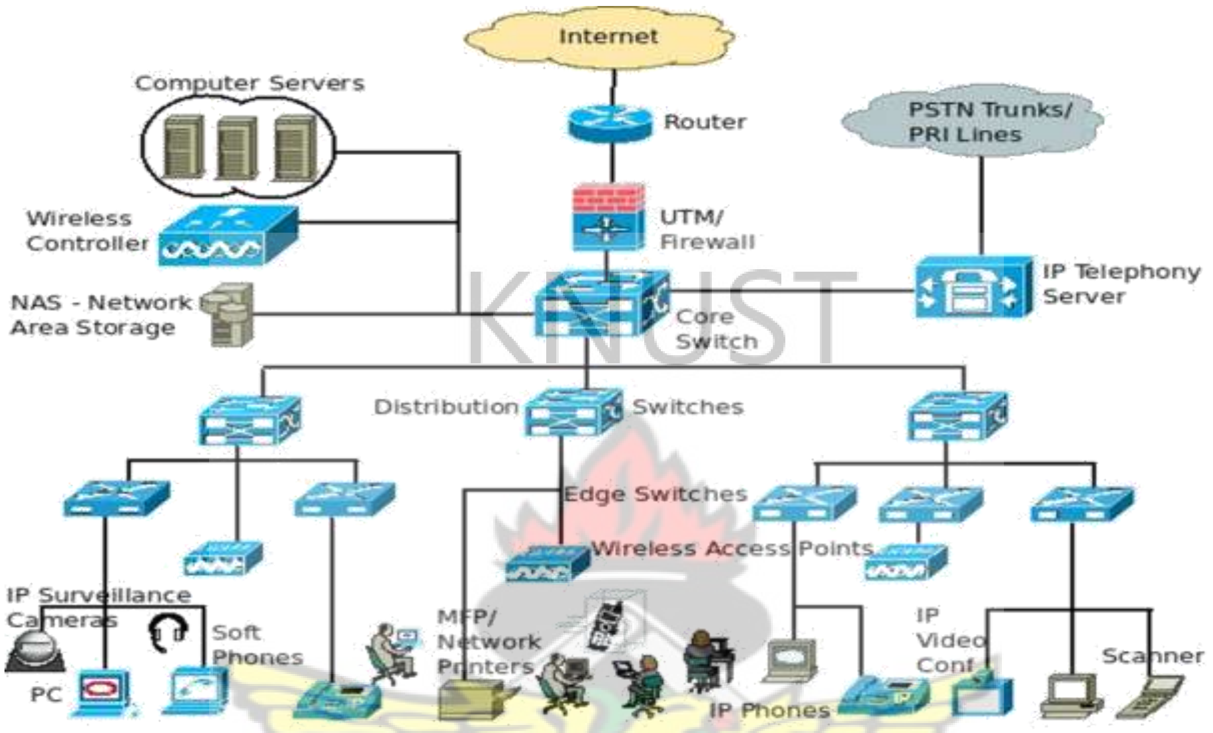


Figure 3.1 A Typical Campus Network

In an SDN-based campus network, switches are made programmable and are of three levels. The three groups of switches in the traditional network are combined to form the data plane. The main function of the devices in the data plane is to forward packets. The control plane functions are moved into a centralized Controller. The controller uses the OpenFlow protocol to obtain the individual characteristics of the devices and links that make up the infrastructure layer and leverages this knowledge to manage the network centrally as shown in Figure 3.2 The interface between the Control and Data planes is the Southbound Interface (SBI).

The Application Plane which consists of applications that are used to implement routing, security and traffic engineering policies within the network via an Application Programming

Interface. This interface which exists between the Application Plane and the Control Plane is called the Northbound Interface (NBI).

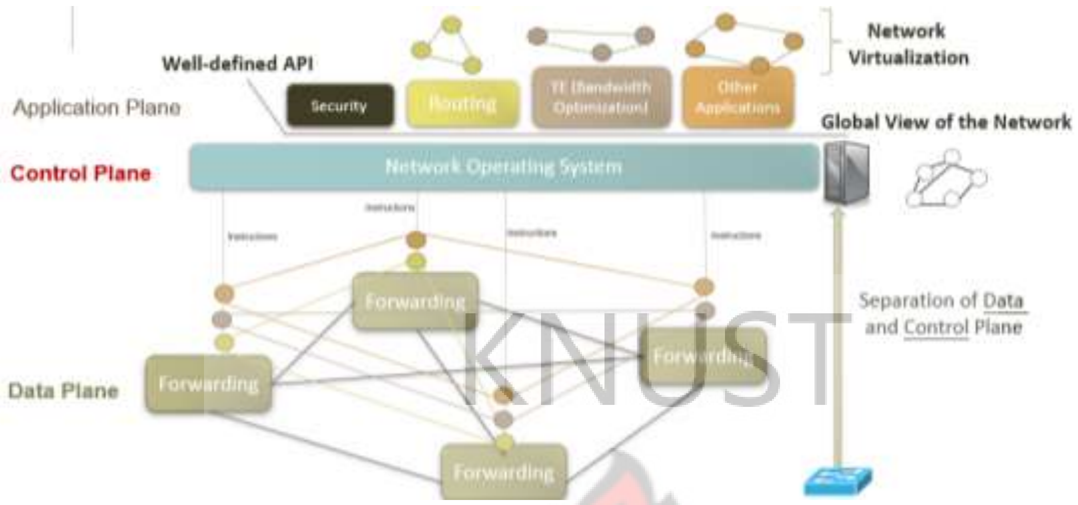


Figure 3.2 Software Defined Network Architecture.

3.1.1 Conversion Of A Traditional Campus Network Into An Sdn-Based Campus Network

The first task in this thesis is to map the current KNUST Campus network into an SDN-based campus network based on the architecture in Figure 3.2 as shown in Figure 3.3 below

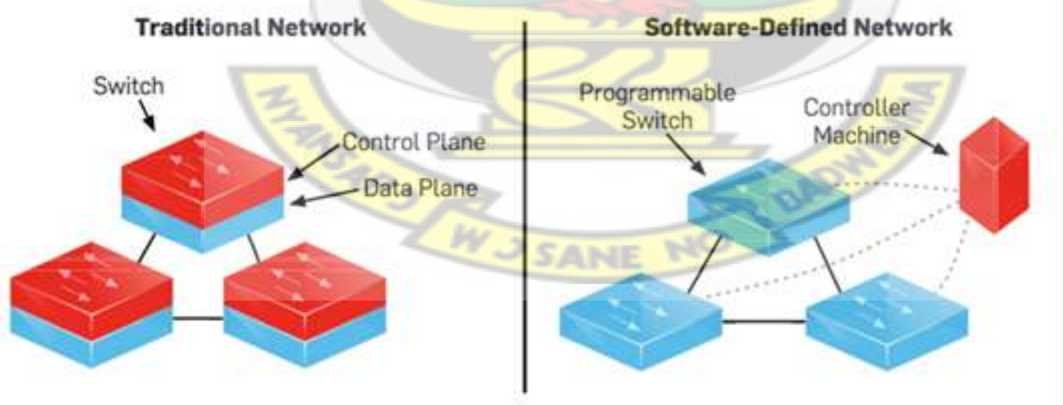


Figure 3.3 Traditional Network Architecture vs SDN Architecture

First the Traditional Network in Figure3.1 is mapped into the SDN architecture in Figure 3.2

The core switch is replaced by an OpenVswitch which is a layer 3 routing switch which is capable of carrying out both switching and routing functions. The three distribution switches are replaced by three OpenVswitches. The five edge switches were replaced by six network devices (either routers or switches) which are connected to end users and the other equipment that form the access portion of the network. The OpenVswitches and network devices were connected to a Control Software to form the data plane or infrastructure layer. The Control Software was connected to a set of network applications which would facilitate configuration of the infrastructure layer thus creating the Northbound Interface.The resulting network architecture form the conversion is as shown in Figure 3.4

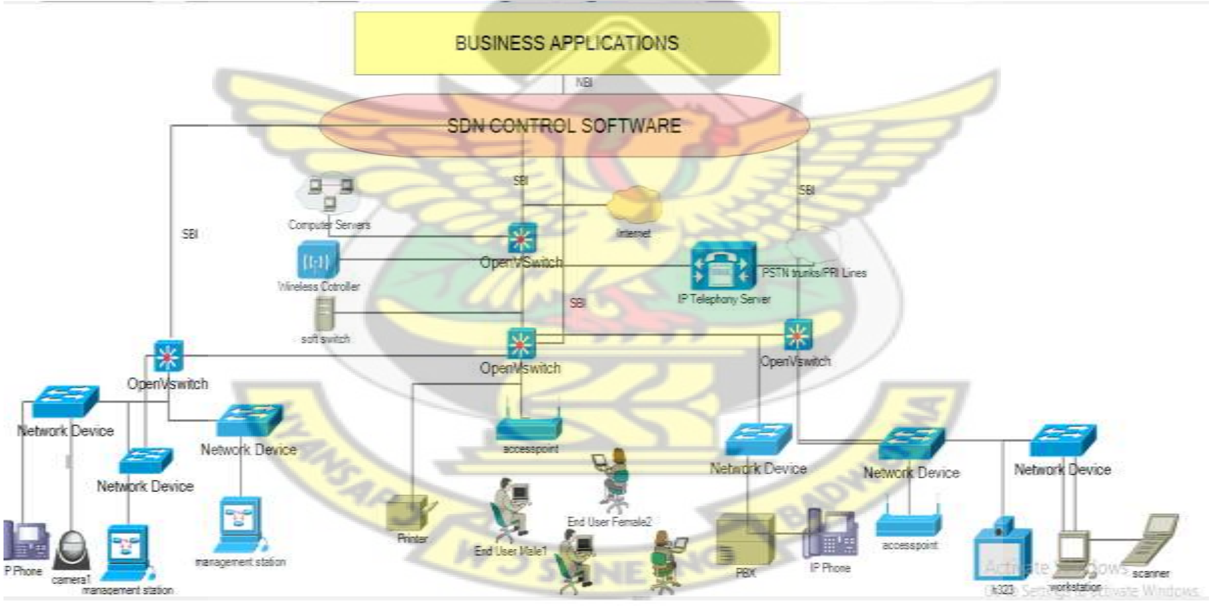


Figure 3.4 Generic SDN-based Campus Network Source: Author's Construct 2019

The KNUST Campus network is divided into two portions. Portion 1 services the six colleges and the Institute of Distance Learning (IDL). Portion 2 serves the residential area which is divided into Residential 1 and Residential 2.

Based on the conversion described above, a similar Software Defined Networking Model was developed for the conversion of the KNUST network into a Software Defined Network.

The following process was used:

Five OpenVswitches were used to represent a collapsed core and distribution switches since an OpenVswitch is a layer 3 switch, it is capable of carrying out the functions of both a core switch and distribution switch. The five OpenVswitches are proposed based on the locations of the six colleges, the Institute of Distance Learning Centre (IDL) and the Residential part of the campus. The six colleges of the university were represented by six network devices each given an IP address. The residential portion of the network was also represented using two network devices. The KNUST campus network has access to the internet via two Internet Service Providers (ISPs). Two network devices were used in representing these.

The five OpenVswitches were connected to the OpenDaylight Controller to form the Southbound Interface (SBI). The OpenDaylight Controller was connected to an application that would dynamically configure and optimize bandwidth in the underlying network through an Application Programming Interface (API). This forms the Northbound Interface. (NBI)

The schematic diagram of the proposed KNUST SDN campus network is shown in Figure 3.5 following the above mapping process.

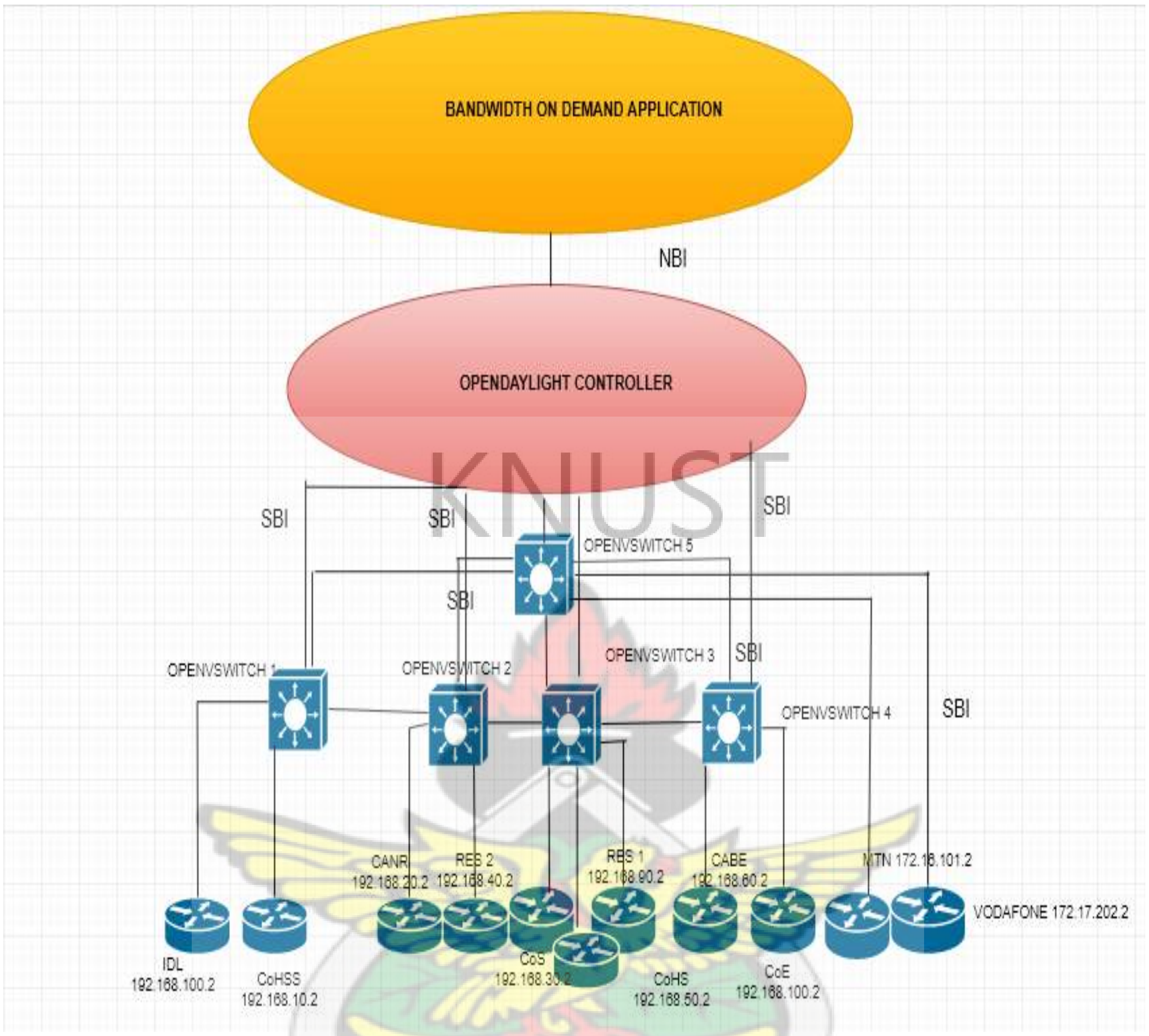


Figure 3.6 Proposed KNUST Campus Based Software Defined Network Source: Author's Construct 2019

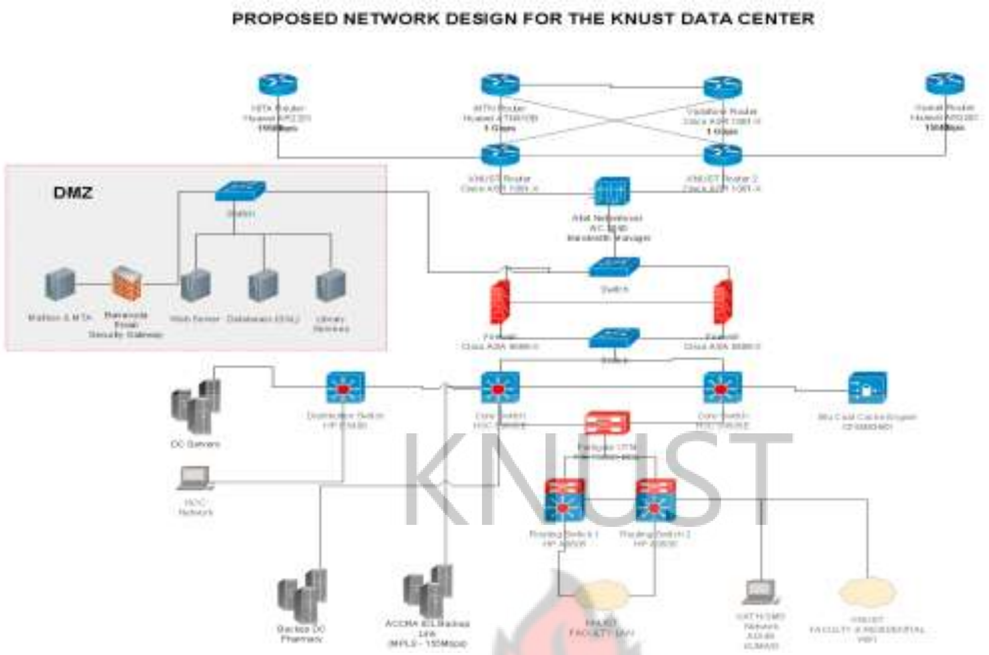

PROPOSED NETWORK DESIGN FOR THE KNUST DATA CENTER
DMZ
Switch
Mailbox & MTA
Barracuda Email Security Gateway
Web Server
Database (SQL)
Library Services
Switch
Switch
DC Servers
HOC Network
Barclays DC Pharmacy
Accra KNUST Backup Link (MPLS - 150Mbps)
KNUST Faculty LAN
KNUST Faculty & Residential WiFi

Figure 3.6 Traditional KNUST Campus Network Source: UITS, KNUST

3.1.2 Emulation of Knust SDN-Based Campus Network

In order to test the workability of the proposed KNUST SDN-based Campus Network, the devices in Figure 3.5 and their functionalities are reproduced in a software for analysis. This software reproduction was done using the VMware Workstation 14 Software and the GNS3 Virtual Machine.

VMware Workstation 14 is a software that is used to design virtual networks by creating software-based replicas of real network devices such as routers and switches and the links that connect them. This tool provides an environment for designing virtual networks. Thus, the Infrastructure and Control and Application layers of Figure 3.5 are implemented using the VMware Workstation 14 Software.

The KNUST Campus network is divided into two portions. Portion 1 services the six colleges and the Institute of Distance Learning (IDL). Portion 2 serves the residential area which is divided into Residential 1 and Residential 2.

Software-based replicas of the Cisco c3600 switching platform was used to reproduce the six Colleges, the IDL and two Residential portions of the network in the GNS3 Virtual Machine. This was done by importing the software platform of the switches into the GNS3 Virtual Machine and configuring them to operate within the VMware Workstation 14 Software.

Each of the Cisco c3600 switches was given an IP address from the 192.168.x.x/24 block.

The OpenVswitches that make up the combined distribution and core network were added to the network by importing a Docker Container version of the OpenVswitch switching platform. The management interfaces of each of the OpenVswitches were configured with an IP address obtained from a DHCP pool created in VMware Workstation using the 192.168.198.x/24 addressing block.

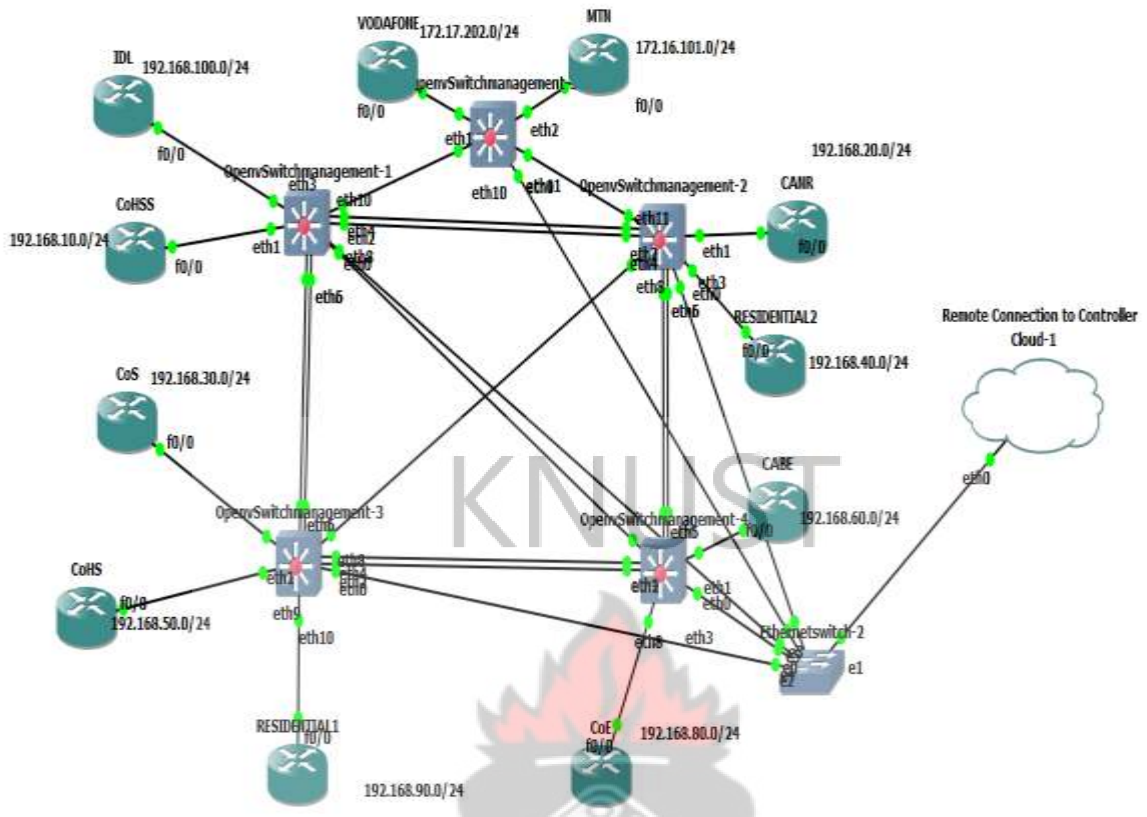
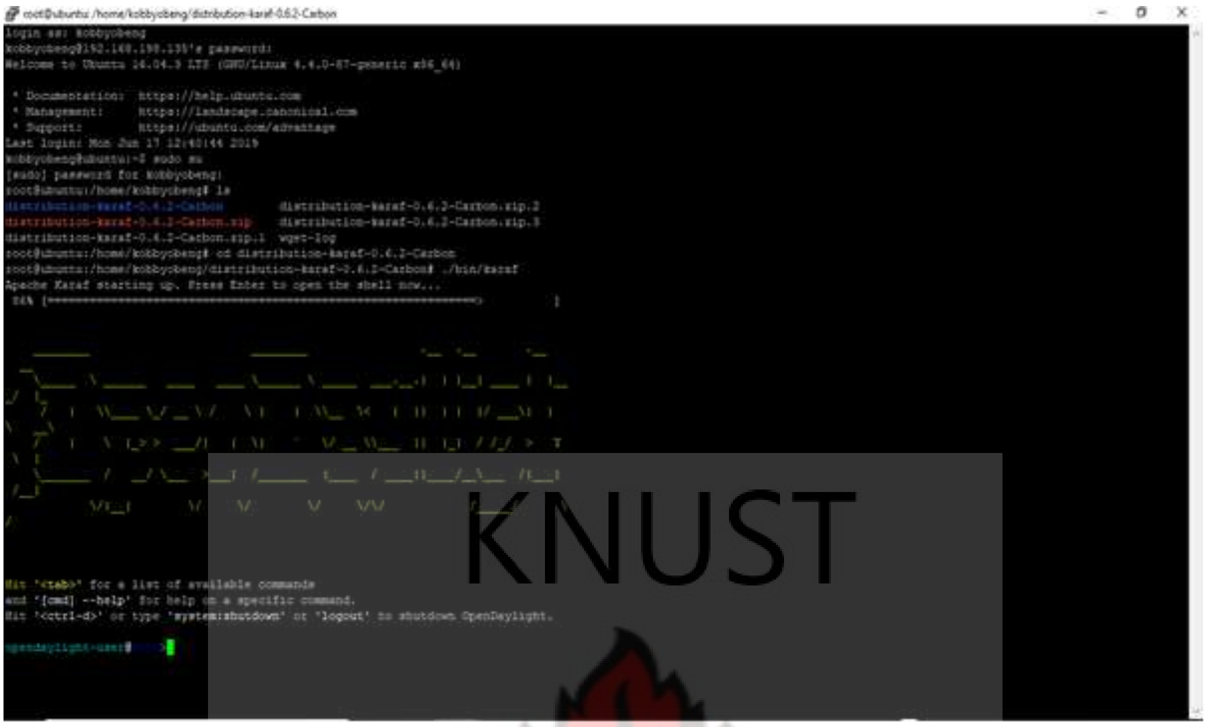


Figure 3. 7 Graphical User Interface Implementation of Infrastructure and Control layers in GNS3 and VMware.

The OpenDaylight Controller Software is downloaded from the OpenDaylight website, imported into VMware Workstation 14 as an Open Virtual Appliance file and configured with an IP address from the 192.168.198.x/24 pool similar to those of the OpenVswitches. The Open Virtual Appliance file runs an Ubuntu Operating System which contains the carbon edition of the OpenDaylight controller in a zip file. The zip file was extracted and the controller started by running the karaf file stored in the .bin directory of the Ubuntu Operating System

The Figure 3.7 below shows the fully installed controller which is described as Connection to Remote Controller in Figure 3.6.



```
root@ubuntu: /home/kobbyobeng/distribution-karaf-0.6.2-Carbon
login as: kobbyobeng
kobbyobeng@192.168.198.135's password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-87-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
Last login: Mon Jun 17 12:40:44 2019
kobbyobeng@ubuntu:~$ sudo su
[sudo] password for kobbyobeng:
root@ubuntu:/home/kobbyobeng# ls
distribution-karaf-0.6.2-Carbon          distribution-karaf-0.6.2-Carbon.zip.2
distribution-karaf-0.6.2-Carbon.zip      distribution-karaf-0.6.2-Carbon.zip.3
distribution-karaf-0.6.2-Carbon.zip.1  wget-log
root@ubuntu:/home/kobbyobeng# cd distribution-karaf-0.6.2-Carbon
root@ubuntu:/home/kobbyobeng/distribution-karaf-0.6.2-Carbon# ./bin/karaf
Apache Karaf starting up. Press Enter to open the shell now...
 86% [=================================================================>          ]

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

Figure 3.8 Controller Interface Installation

3.2 OPTIMAL DYNAMIC BANDWIDTH SHARING SCHEME

The faculty and residential portions of the traditional KNUST network have a fixed bandwidth allocation to facilitate internal and external communication. This static assignment of bandwidth for communication creates a problem. Majority of students and lecturers find themselves in the faculty area during the day. During this period, traffic volumes at the faculty portion of the network increases. This affects network performance. This is seen in difficulty to browse the web, download content or stream resources for teaching and learning. The reverse is seen in the evening as the concentration of traffic shifts to the halls of residence.

A Software Defined Network architecture promises flexibility and programmability in managing computer networks. The proposed SDN-based KNUST network can thus be programmed to dynamically borrow bandwidth from portions of the network where there is unused bandwidth the traffic demands or requirements in the network. The proposed dynamic bandwidth sharing technique is based on the theory of the Hierarchical Token Bucket Queuing Discipline

3.2.1 The Hierarchical Token Bucket Queuing Discipline

The Hierarchical Token Bucket Queuing Discipline (HTB) is a class-based queue discipline that controls the use of bandwidth on a given output link and implements efficient resource allocation. It uses the concept of multilevel token buckets to allow for efficient dynamic control of the egress bandwidth on a given link. HTB is based on hierarchical classes and is made up of three class types known as root, inner and leaf classes. The root class represents the minimum and maximum amount of bandwidth (guaranteed bandwidth) that is set for communication between network devices. Any form of service request that occurs between a client and server connected to a particular network device is allocated bandwidth borrowed from the root class. The bandwidth for such communication is represented by an interior class. Each amount of bandwidth that is shared from an interior class is termed as a leaf class. Placing traffic into classes is termed as Classification.

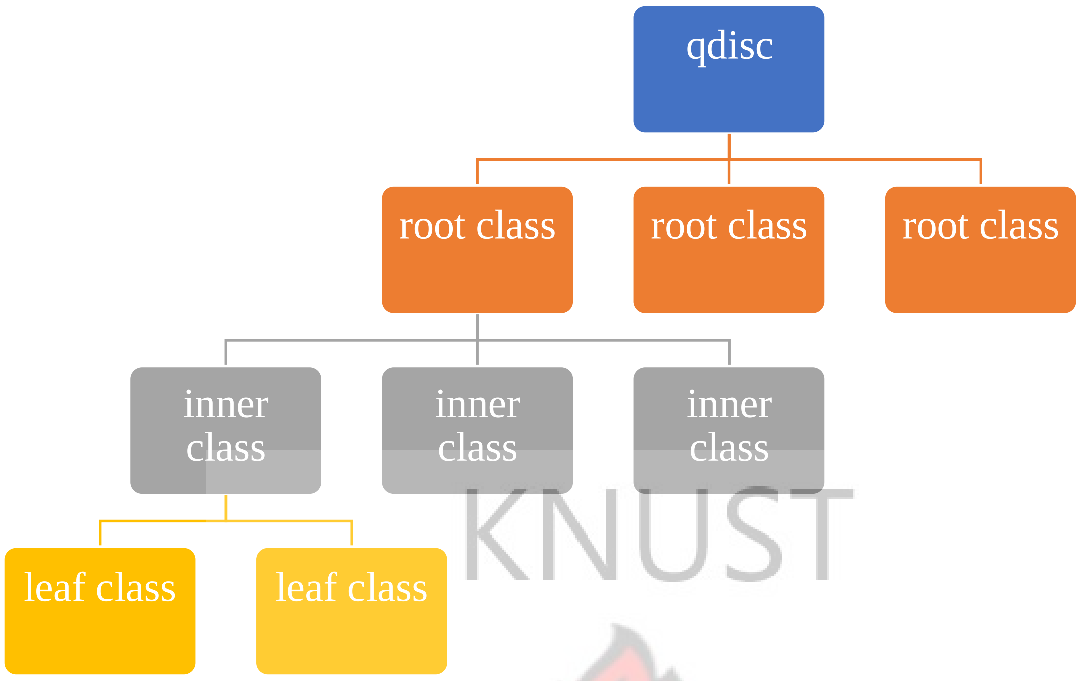


Figure 3.9 HTB class structure and borrowing

Once traffic has been classified, the Hierarchical Token Bucket Queuing Discipline uses the concept of tokens and buckets to schedule and shape traffic by utilizing a classless queuing discipline called the Token Bucket Filter. The Token Bucket filter uses two filters. A packet being sent from a transmitter to a receiver passes through both filters. The second filter typically allows packets to flow out of it faster in order to limit the speed of burst traffic.

From Figure 3.9, specific requests made to an OpenVswitch running the HTB queuing discipline will go through four phases. These phases are classifying, policing, scheduling and borrowing.

The borrowing of bandwidth is a function of Traffic Policing. In traffic policing the bandwidth is limited to applications based on the class that they belong to.

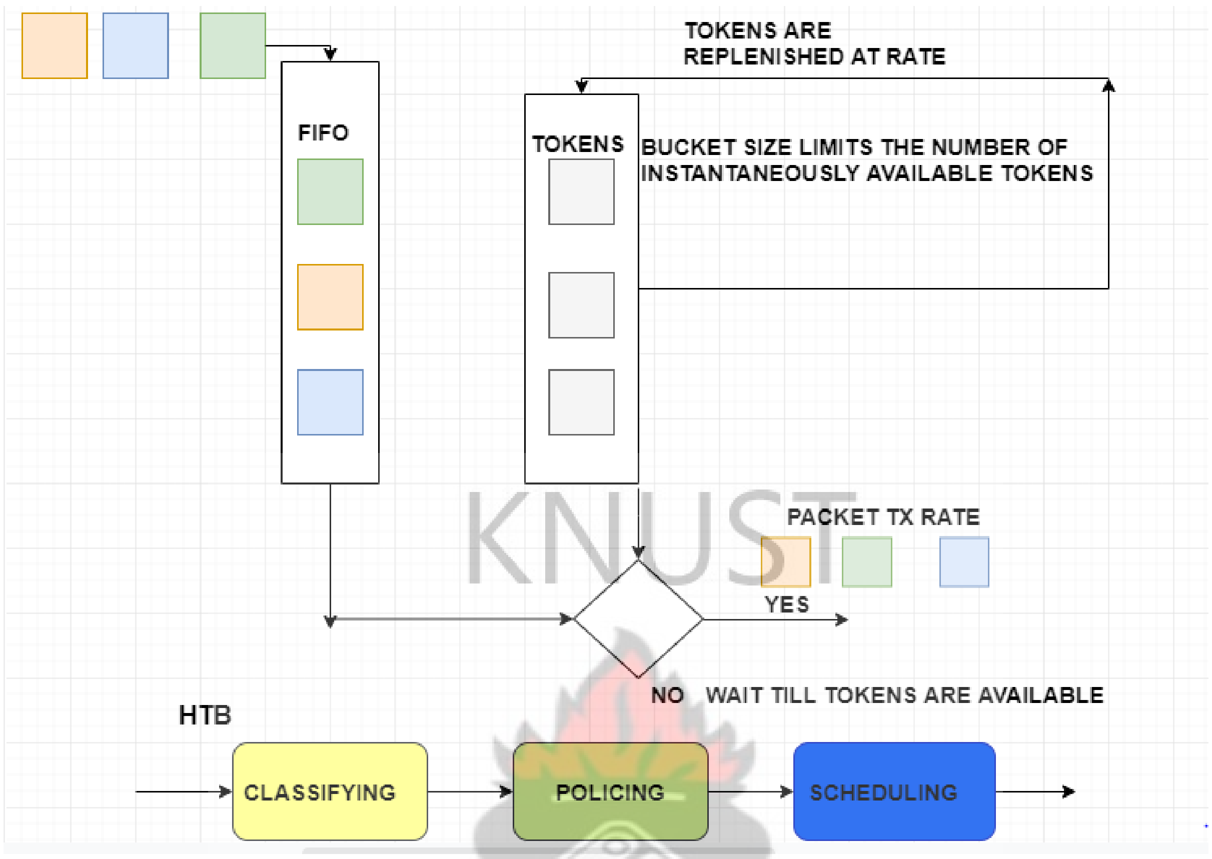

TOKENS ARE REPLENISHED AT RATE
FIFO
TOKENS
BUCKET SIZE LIMITS THE NUMBER OF INSTANTANEOUSLY AVAILABLE TOKENS
PACKET TX RATE
YES
NO
WAIT TILL TOKENS ARE AVAILABLE
HTB
CLASSIFYING
POLICING
SCHEDULING

Figure 3.10 Hierarchical Token Bucket Queuing Discipline Operation Source: Author's Construct 2019

3.2.2 Bandwidth Borrowing

Figure 3.10 below demonstrates an adaption of HTB to the Software Defined-based campus network**.**

The KNUST Campus link which connects the entire university to the internet is divided into two portions. One portion of the link goes to the faculty and the other portion of the link goes to the residential area. In adapting the HTB for the KNUST Campus an instance is considered where there is the need to allocate 500 Mbps to the College of Engineering (CoE) during the day due to high traffic demand and 200 Mbps to the residential portion during the day due to lower traffic demand.

The 500 Mbps allocated to the College of Engineering (CoE) needs to be subdivided into 100Mbps for the wired access to the office of lecturers and 400 Mbps for the wireless connections used by students. Any unused bandwidth from the 100 Mbps allocated to the lecturers' offices should be given to the wireless connection for students and vice-versa. If the total traffic requested at the College of Engineering (CoE) does not exceed 500 Mbps, the excess will be given to the residential portion. Using the theoretical explanation of HTB given earlier, the 500Mbps allocated for the College of Engineering CoE is the root class. It represents the Ceil Rate for all communication in the College of Engineering. The 400 Mbps and the 100 Mbps represent the inner classes for the College of Engineering (CoE). These values represent the Assured Data Rates for both the wired and wireless connections to the College of Engineering. This policy of quantifying bandwidth in such a hierarchy is implemented in a configuration policy using variable sized arrays to accommodate any changes in the preconfigured bandwidth guarantees.

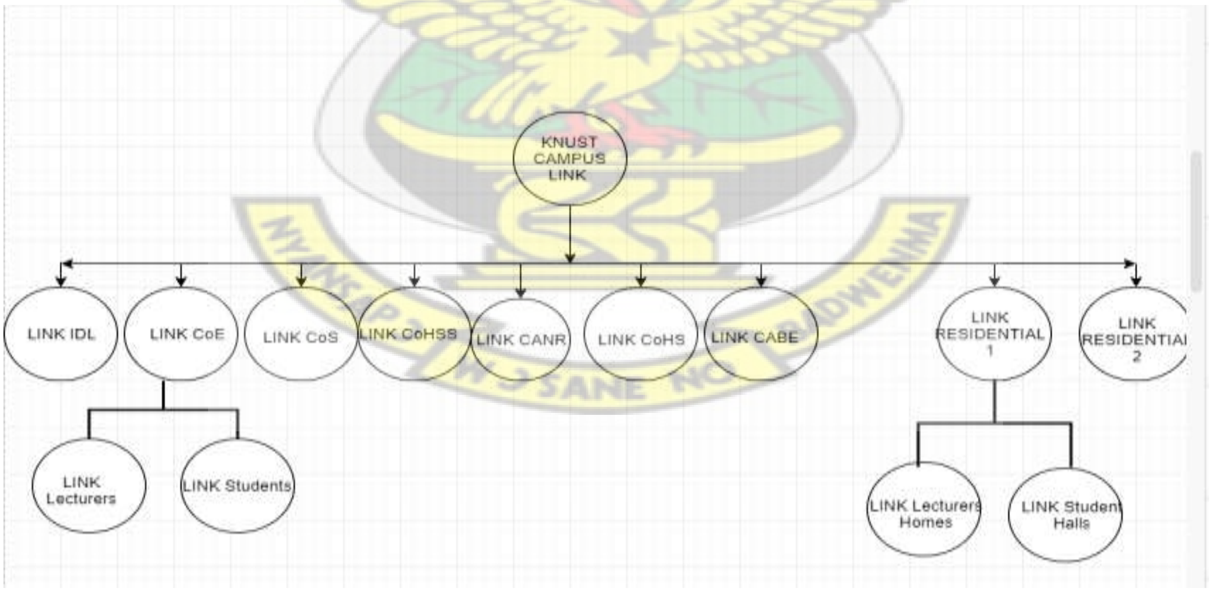


Figure 3.11 Sample HTB class hierarchy for KNUST SDN-based campus network

From Deterministic Network Calculus borrowing of bandwidth as described can be modelled using arrival and service curves.

The Borrowing Phase

In the HTB model the number of bits in a flow in a period (0, t) is

$$R(t) = \min(R_c(t), R_a(t) + B(t) \tag{3.1}$$

where Ra(t) is assured data rate (minimum threshold) and Rc(t) is ceil data rate (maximum threshold). In a leaf class, if data rate R(t) exceeds assured data rate Ra(t) and less than ceil data rate Rc(t), then this leaf class would borrow bandwidth from its parent class. Parent class could also borrow bandwidth from its parent too. If a parent class has more than one child class and all of them run out of bandwidth, parent would distribute its resource based on child class's priority(P), quantum(Q) and list of backlogged or queued packets in the FIFO queue(D). The equation below represents the model used for bandwidth borrowing.

$$B(t) = \frac{QR_{Pexcess}(t)}{\sum i \in D(p) Q_i} \text{ if } \min_{i \in D(p)} P_i \leq P \text{ and } B(t) = 0 \text{ otherwise} \tag{3.2}$$

3.2.3 Classification, Policing and Scheduling Of Packets In HTB Queuing Discipline

From Deterministic Network Calculus, the classification, policing and scheduling of packets can be modelled using arrival and service curves

The Classifying Phase

Based on root, leaf and interior classes scenario used earlier, incoming requests from the SDN-based campus network will have a service curve given by

$$\beta_{classify}(t) = [\delta_{\tau c} - l_{max}]max \tag{3.3}$$

where $\delta_{\tau c}$ is the delay and $l_{\max}$ is the maximum length of the arriving packet.

If the incoming request to the network has an arrival curve of $\alpha(t)$, then after classifying the arrival curve will be given by

$$\alpha(t)+l_{\max}1\{t>0\}. \qquad (3.4)$$

The Policing Phase

The classified incoming request is then policed based on the HTB QoS setting. Policing is carried out based on the root class minimum value.

Policing function makes sure that a flow does not exceed guaranteed service. Excess traffic may be dropped or sent to best effort path. Policing devices always buffer flows and leaks in the guaranteed rate. A packetized shaper is a shaper that forms its output packets has a data rate *r*. Output flow of a policing device implemented in the HTB Queuing Discipline is

$$\alpha_{policing}(t)=kv_{T,\tau}=k\left\lfloor t-\tau\_d\big/T\right\rfloor^{+} \qquad (3.5)$$

where k is the rate of data flow, $v_{T,\tau}$ is a stair function and

is defined by $\left\lceil \frac{t+T}{T} \right\rceil$ T is the interval and $\tau-d$ is the packet delay and $\lfloor x\rfloor^{+}$ is the floor of x.

For example, if the amount of traffic at College of Engineering exceeds the minimum value assigned to the root class, the policing criteria changes. For a packetized model, the output flow in such a case is

$$\alpha_{policing}(t)=k\left\lfloor \frac{t-\tau\_d}{T}\right\rfloor^{+} if\ \frac{k}{T}t\le\frac{R}{t}t+B$$

otherwise

$$\alpha_{policing}\ (t)=k\ \left[\frac{R}{k}\right]\max(t-\tau_d)\ \ if\ \frac{k}{T}\ t>\ \frac{R}{t}t+B \qquad (3.6)$$

Also

$$\alpha_{policing}\ (t) = \alpha(t) \otimes\ \beta_{policing}\ (t) \tag{3.7}$$

where R is the number of bits seen in a flow of packets and *B* is queue length which buffers traffic burst.

The Scheduling Phase

In the HTB Queuing Discipline, the Scheduling Phase only occurs with packets that are in the leaf class. The leaky bucket model has a Service Curve of

$$\beta_{leaky\_bucket}(t) = \gamma(t) + b\ if\ t > 0\ otherwise\ \beta_{leaky\_bucket}(t) = 0 \tag{3.8}$$

In the HTB model the number of bits in a flow in a period (0, t) is

$$R(t) = \min(R_c(t), R_a(t) + B(t) \tag{3.9}$$

where *Ra*(*t*) is assured data rate (minimum threshold) and *Rc*(*t*) is ceil data rate (maximum threshold) The HTB FIFO Queue that uses a time varying leaky bucket model has Service Curve of

$$\beta_{shaping}(t) = R(t) + b\ if\ t >)\ otherwise\ \beta_{shaping}(t) = 0 \tag{3.10}$$

3.3 DEVELOPMENT OF THE BANDWIDTH ON DEMAND TESTBED

The Bandwidth on Demand Testbed was designed to carry out bandwidth utilization within the Software Defined-based campus network. It was designed and implemented using a Python Program that coordinates with the QoS Configuration for optimizing Bandwidth allocated to links, the OpenDaylight Controller and the OpenVswitch Database.

The Bandwidth on Demand Application uses a python program that runs a Main Process. This Main Process uses web sockets to facilitate TCP connections between three sub-process. It coordinates connections to the script containing the QoS definition for the Bandwidth on Demand, the OpenDaylight Controller and the OpenVswitch Database.

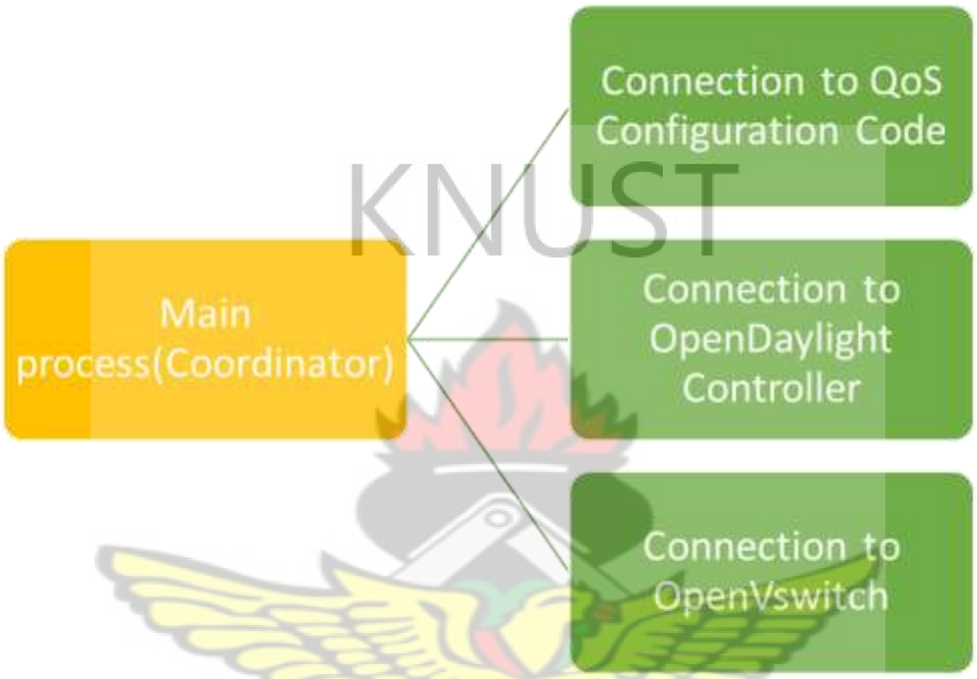

Main process(Coordinator)
Connection to QoS Configuration Code
Connection to OpenDaylight Controller
Connection to OpenVswitch

Figure 3.12 Implementation of Bandwidth on Demand Application

The QoS configuration code that was used is displayed below

QoS CONFIGURATION CODE

```
def createQ (interface, minB, maxB):
return "ovs-vsctl set port {0} qos=@newqos -- --id=@newqos create qos type=linux-htb other-config:max-rate={2} queues:1=@q1 queues:2=@q2 -- --id=@q1 create queue other-config:min-rate={1} other-config:max-rate={2} -- --id=@q2 create queue other-config:min-rate={1} other-config:max-rate={2}".format(interface, minB, maxB)
```

The API that was used by the main process to coordinate all three sub-processes is displayed below

API CALL URL:

```
def url(url):
return 'http://{0}:8181/restconf{1}'.format(ip, url)


def creds():
return ('admin', 'admin')


def nodes ():
try:
a = requests.get (url('/operational/opendaylight-inventory: nodes/'), auth=creds())
return json.dumps(a.json())
except:
return json.dumps({})


def uploadFlow(s, d):
url = "http://{0}:8181/nodes/node/openflow:1/table/0/flow/iperf".format(ip)
```

The term highlighted opendaylight-inventory: nodes represents the five OpenVswitches that are connected to the OpenDaylight Controller.

3.4 COMMUNICATION BETWEEN THE PLANES OF THE PROPOSED SDN ARCHITECTURE

In the Software Defined-based campus network, the OpenDaylight Controller acts as a network operating system and controls the Software Defined Networking capable network devices (OpenVswitches) in a central way. It uses the Southbound Interface Protocol called

OpenFlow version 1.3 and Northbound Application Interface called REST to facilitate communication between the Infrastructure, Control and Data Planes.

Figure 3.12 below shows the communication that occurs in a Generic Software Defined Network architecture

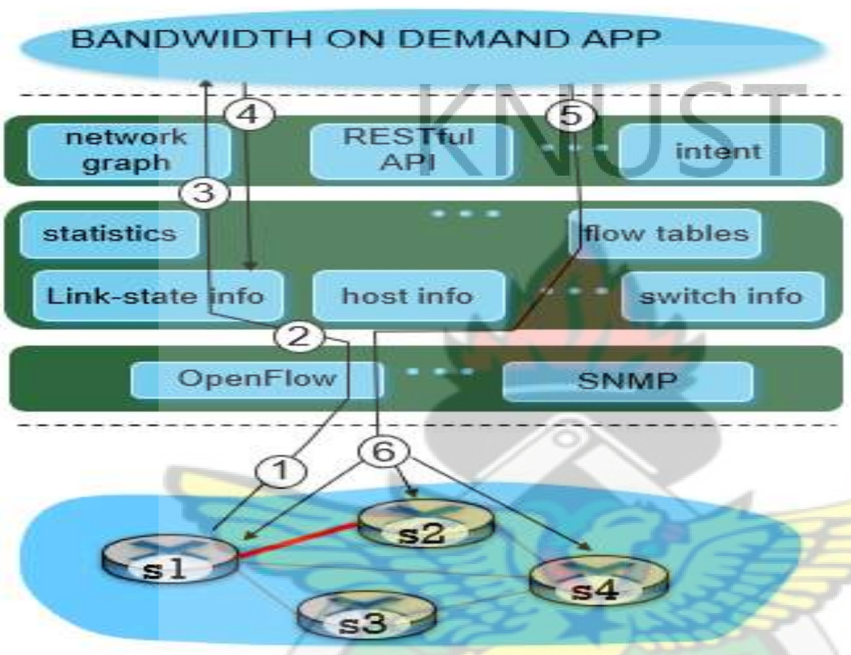


Figure 3.13 Communication between the planes of an SDN architecture

The scenario below is used to further explain the communication shown in Figure 3.13. Link between S1(College of Engineering) and S2(IDL) is slow. Users connecting to both switches cannot browse or download content. OpenFlow Port Status message notifies the OpenDaylight Controller via the management interface of S1 The OpenDaylight controller receives OpenFlow message and updates link capacity info. The QoS configuration code has previously registered in the Bandwidth on the Demand Application to be called whenever link capacity has to be changed. It is called.

The OpenFlow Protocol which has already accessed network graph info, link state info in the controller and communicates it to the app running HTB.

The testbed running HTB interacts with flow-table-computation component in the OpenDaylight SDN controller, which computes the instructions needed to dynamically configure bandwidth are sent to S1 and S2. Controller uses the OpenFlow Protocol to send the instructions to switches S1 and S2 to update their link-capacity information.

After reproducing the functionalities of components found in the various planes of the Software Defined Network based campus network architecture an analysis was carried out to figure out if the following could take place based on the communication stated above. These are Control Plane-Application Plane Communication (NBI Communication), Control Plane-Data Plane Communication (SBI Communication) and Data Plane Device communication.

The results of the analyses are presented in Chapter 4.

3.6 THEORECTICAL BASIS FOR THE COMMUNICATION BETWEEN THE PLANES OF A SOFTWARE DEFINED NETWORK ARCHITECTURE

The scenario used to explain the communication that occur between the various planes of the Software Defined-based campus network Architecture in Figure 3.13, can be modelled based on Deterministic Network Calculus by taking into account the control cycle that occurs between the Data Plane, the OpenDaylight Controller and the Bandwidth on Demand Testbed.

The QoS setting which is triggered at the application layer of the architecture as shown in Figure 3.12, is used determine a minimum and maximum threshold of bandwidth for communication between a particular OpenVswitch and a client at the faculty or residential portion of the campus network (KNUST LAN). A control cycle is evoked based on the trigger.

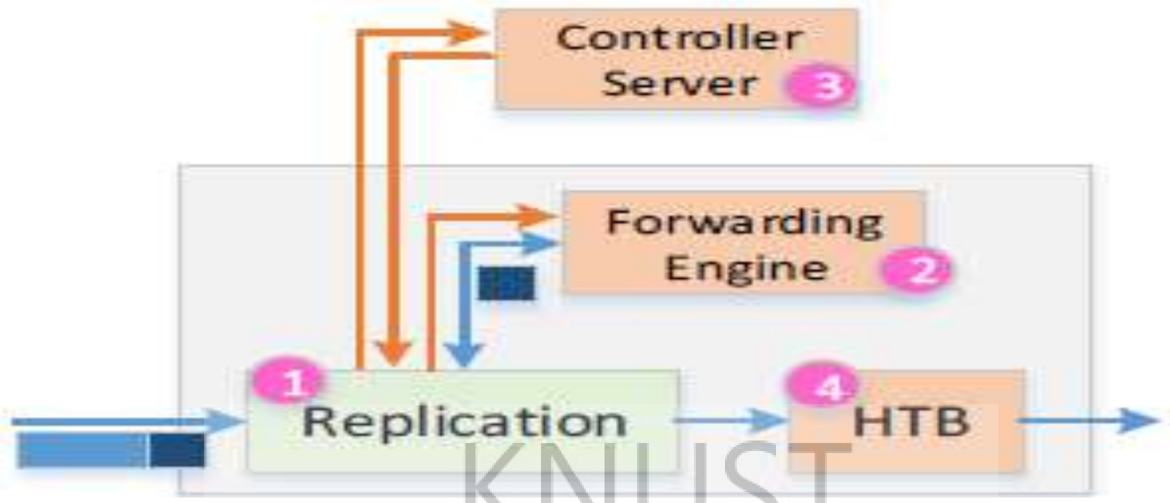


Figure 3.14 A Typical OpenVswitch and Controller Cycle in a Software Defined Network

The OpenVswitch is a switching platform that serves as the interface between the Data plane and the Control plane. It consists of a forwarding engine that is used to forward packets after the HTB discipline has been implemented

Below is a theoretical analysis of the maximum and minimum resource demand by an event(packet) stream on the OpenVswitch connected to S1(College of Engineering) using Nodes Concatenation Theorem.

For a guaranteed service flow (root class definition),

$$\beta_{switch}(t) = \beta_{classifing} \otimes \beta_{policing} \otimes \beta_{shaping} \qquad (3.11)$$

where β_{switch} is the service curve for the switch, $\beta_{classifying}$ is the service curve from classification of packets, $\beta_{policing}$is the service curve model for policing of classified packets and $\beta_{shaping}$ is the service curve model for scheduling of packets.

3.6.1 Analysis of The Control Cycle for A Software Defined Network

Steps 1 to 6 in Figure 3.13, show the communication between the Infrastructure, Controller and Network Layer. Such a communication is termed as a control cycle.

Nodes Concatenation Theorem can be used to analyze the maximum and minimum resource demand by the various components in a Software Defined Network Architecture.

Software Defined Networking facilitates the abstraction of the control portion of the network devices in the Infrastructure layer to the Open Daylight Controller. The behavior of the Controller is given by its Control Service Cycle which is given by

$$\beta_{control\_cycle} = \beta_{2\tau link} \otimes \beta_{ctlopendaylight} \quad (3.12)$$

where is service $\beta_{ctlopendaylight}$ curve of controller, $\beta 2\tau\_link$ is service curve of link between controller and switch while its propagation and transmission delay given by τ_link.

When a QoS Queue defined by the Bandwidth on Demand App with a specific priority is traverses the network in a packet stream, the packet stream(flow) is given a numbered priority. The arriving curve of such a flow is given by $\alpha_{higher\_priority}$ if it is placed in a high priority queue. A lower priority flow with arriving curve as $\alpha_{lower\_priority}$ goes through lower priority queue.

The service curve for a high priority queue is given by $\beta_{higher\_priority}$. The service curve of a lower priority queue is $\beta_{lower\_prority} = [\beta_{higher\_priority} - \alpha_{higher\_priority}\ ]\, max$

(3.13)

When a high priority flow destined for S1(College of Engineering) is created after the HTB QoS is triggered without a flow table, the service curve is given by

$$\beta = \beta_{\tau\_link} \otimes \beta_{ctl\_opendaylight} \otimes \beta_{openVswitch} \otimes \beta_{\tau\_link} \quad (3.14)$$

Service curve for a lower priority queue under the same condition is given by

$$\beta = \beta_{\tau\_link} \otimes \beta_{ctl\_opendaylight} \otimes \left[\beta_{openvswitch} - \alpha_{higher\_priority}\right]^{+} \otimes \beta_{\tau\_link} \tag{3.15}$$

After a flow table is created for the high priority flow in the above scenario, the service curve is given by

$$\beta = \beta_{\tau\_link} \otimes \beta_{openvswitch} \otimes \beta_{\tau\_link} \tag{3.16}$$

Service curve for a lower priority queue after a flow table is created is given by

$$\beta = \beta_{\tau\_link} \otimes \left[\beta_{openvswitch} - \alpha_{higher\_priority}\right]^{+} \otimes \beta_{\tau\_link} \tag{3.17}$$

3.7 CONCLUSION

This chapter has explained the theory and methodology used in converting a typical traditional Campus Network into a Software-Defined Campus network are discussed. The chapter has also taken a look at theory and methodology employed in the development of an application that dynamically optimizes link bandwidth with a Software-Defined Campus Network.

Deterministic Network Calculus was used to define arrival and service curve models for the communication between the various planes of the Software Defined-based campus network.

Network Concatenation Theorem was also used in defining the service curve models for the control cycle that takes place in the various planes of the Software Defined-based campus network.

CHAPTER FOUR

RESULTS AND DISCUSSIONS

4.0 INTRODUCTION

This chapter presents the results obtained from the implementation of a Software Defined - based campus network.

The chapter is divided into four main sections. The first section is a presentation of results of the conversion and of a traditional campus network into a Software Defined-based campus network. It also states the results of the emulation of the Software Defined-based campus network using the Kwame Nkrumah University LAN as a case study. The second section is a presentation and an analysis of the results obtained from communication between devices at the Data Plane Device of the Software Defined-based campus network. The third section takes a look at the results and analysis of the communication that occurs between the Control Plane and Data Plane of the proposed network. The final section is a presentation of the results and analysis of the communication that occurs between the Control Plane and the Application Plane based on a scheme that dynamically utilizes bandwidth within the Software Defined-based campus network

In each of these sections the experimental setup that was used is briefly described followed by a presentation and discussion of the results obtained.

4.1 RESULTS FOR THE CONVERSION OF A TRADITIONAL CAMPUS NETWORK INTO A SOFTWARE DEFINED-BASED CAMPUS NETWORK

Figure 4.1 below shows a traditional campus network.

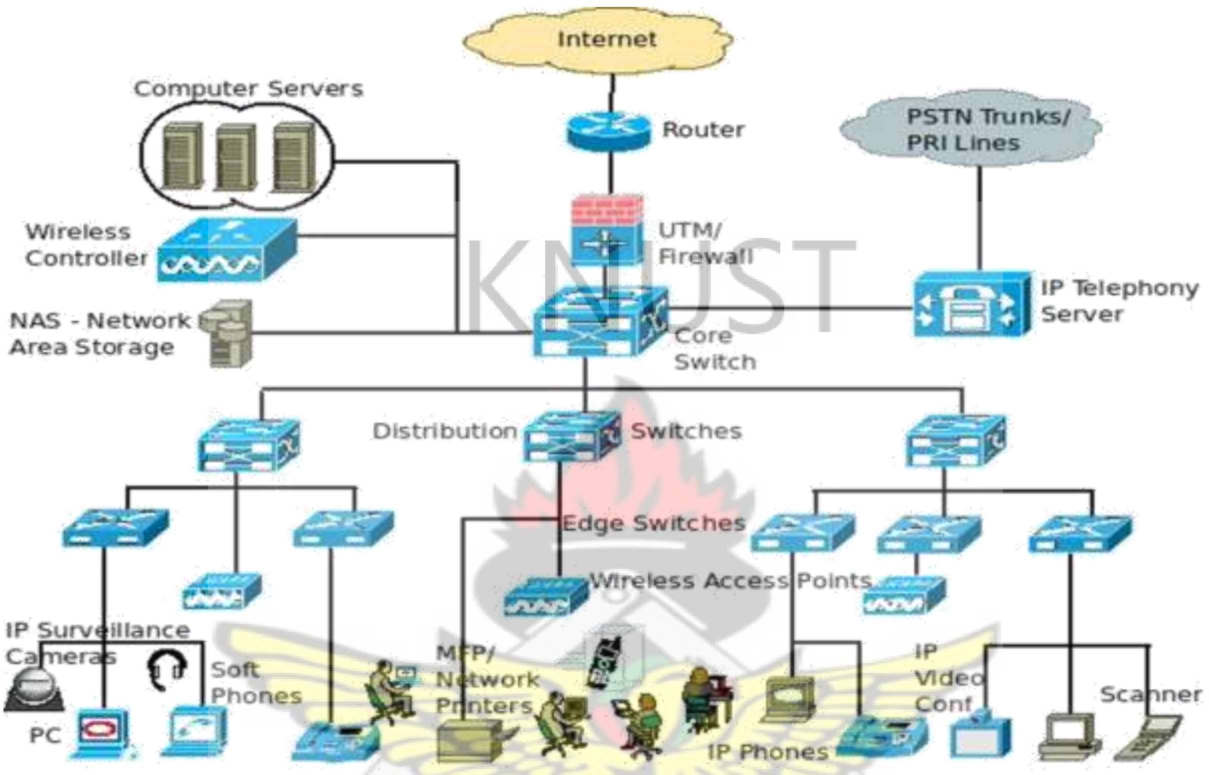


Figure 4.1 Traditional Campus Network

The Figure below shows the result of the conversion of Figure 4.1 into a generic Software Defined-based campus network. Comparing Figure 4. 1 and Figure 4. 2, the core switch is replaced by an OpenVswitch which is a layer 3 routing switch. The three distribution switches are replaced by three OpenVswitches. The five edge switches are replaced by six network devices(either routers or switches) which are connected to end users and the other equipment that form the access portion of the network. The OpenVswitches and network devices are connected to a Control Software to form the data plane or infrastructure layer. The Control Software is connected to a set of network applications which would facilitate configuration of the infrastructure layer thus creating the Northbound Interface.

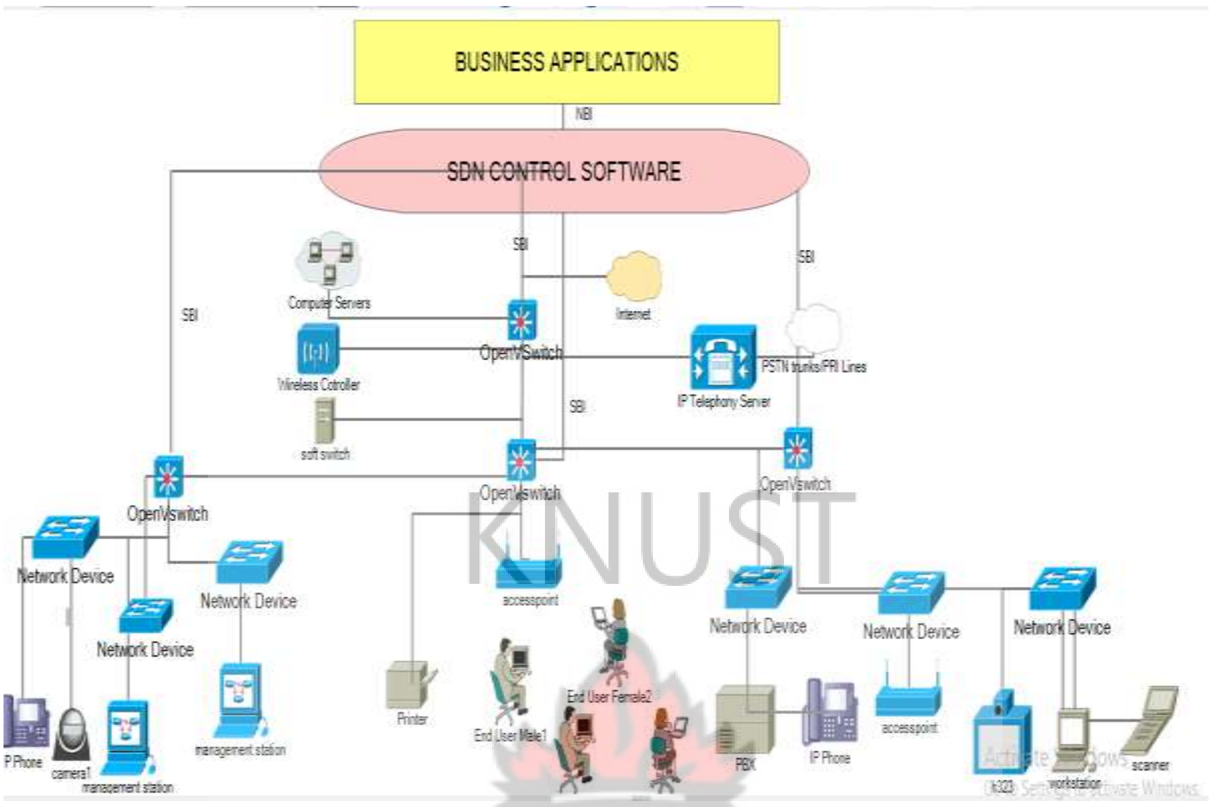


Figure 4.2 Generic Software Defined-based campus network

The Local Area Network of the Kwame Nkrumah University of Science and Technology is subdivided into a faculty portion and a residential portion. The faculty portion is made up of six colleges and the Institute of Distance Learning. The residential portion is made up of students' residential halls and the homes of university staff.

The Kwame Nkrumah University of Science and Technology LAN is mapped into a Software Defined-based campus network. The result from the mapping shows five OpenVswitches which form the collapsed distribution and core network, the six colleges of the university and the Institute of Distance Learning and the residential portion of the network and the two Internet Service Providers (ISPs) which provide access to the internet.

It also shows the five OpenVswitches connected to the OpenDaylight Controller to form the Southbound Interface (SBI) as well as the connection between the OpenDaylight Controller and an application that would dynamically configure and optimize bandwidth in the underlying network through an Application Programming Interface (API). This forms the Northbound Interface. (NBI)

The schematic diagram below shows the Software Defined-based campus network of the Kwame Nkrumah University of Science and Technology LAN.

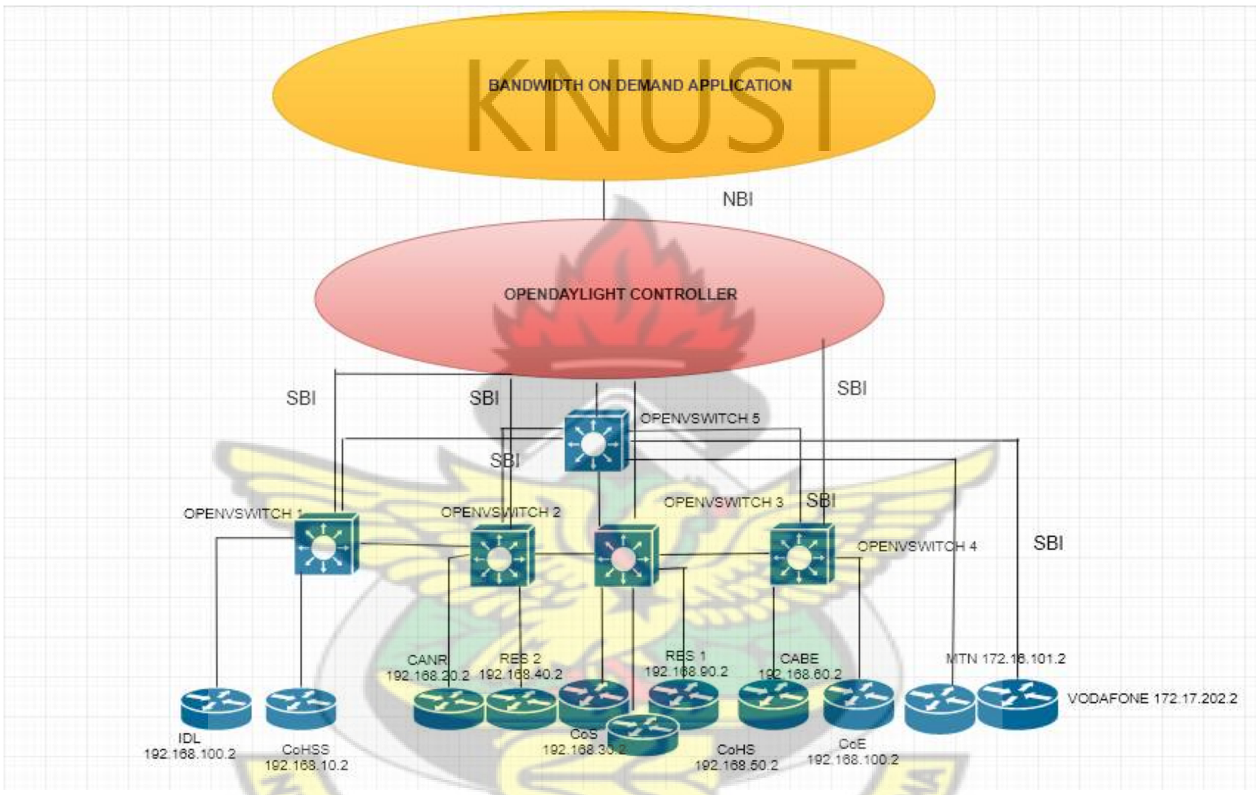


Figure 4.3 Schematic of Software Defined-based campus network for KNUST

Figure 4.4 shows the emulation of the schematic in Figure 4. 3 using the GNS3 VM and the VMware Workstation 14 software. It depicts the network devices in the data plane and a connection to a remote controller proving that data plane and control plane functionality have been separated.

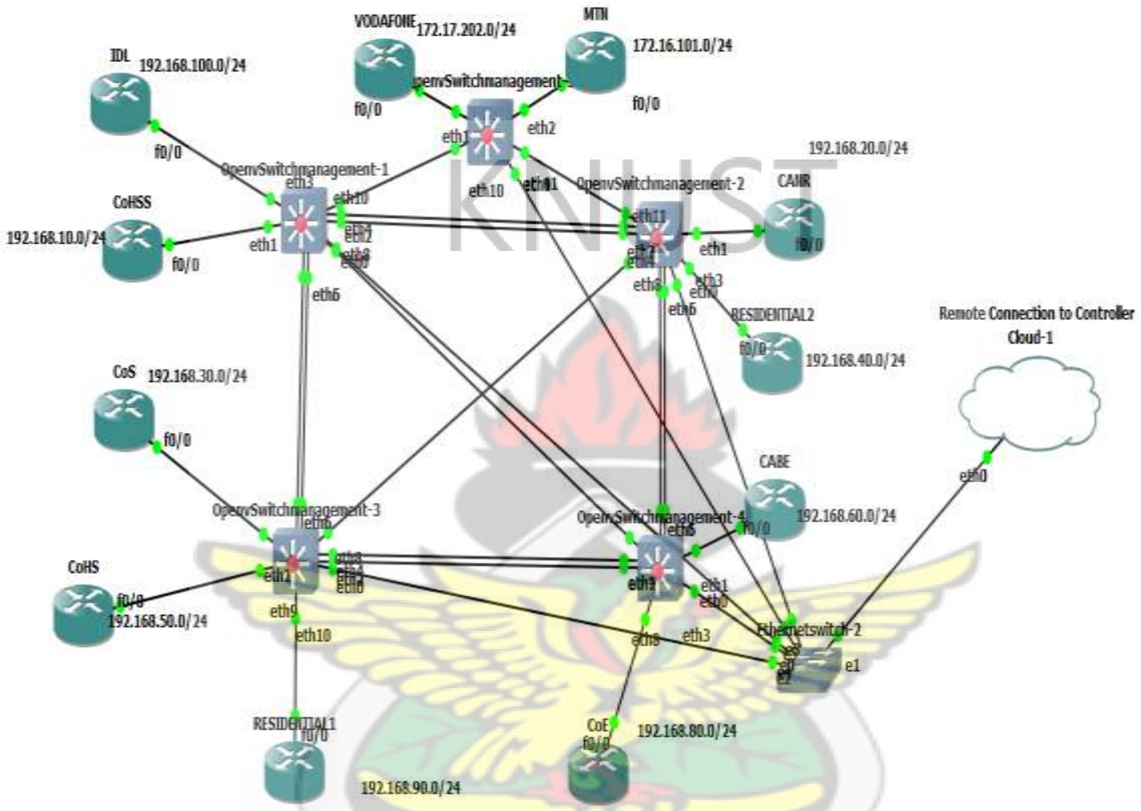


Figure 4.4 Graphical User Interface Implementation of Software Defined-based LAN for KNUST

4.2 EXPERIMENTAL SET UP FOR DATA PLANE DEVICE COMMUNICATION

In order to validate forwarding between the network devices at the data plane of the emulated campus network, ICMP echoes are sent from the network device labelled IDL to the network device labelled CoE as highlighted using the red arrow in Figure 4. 5

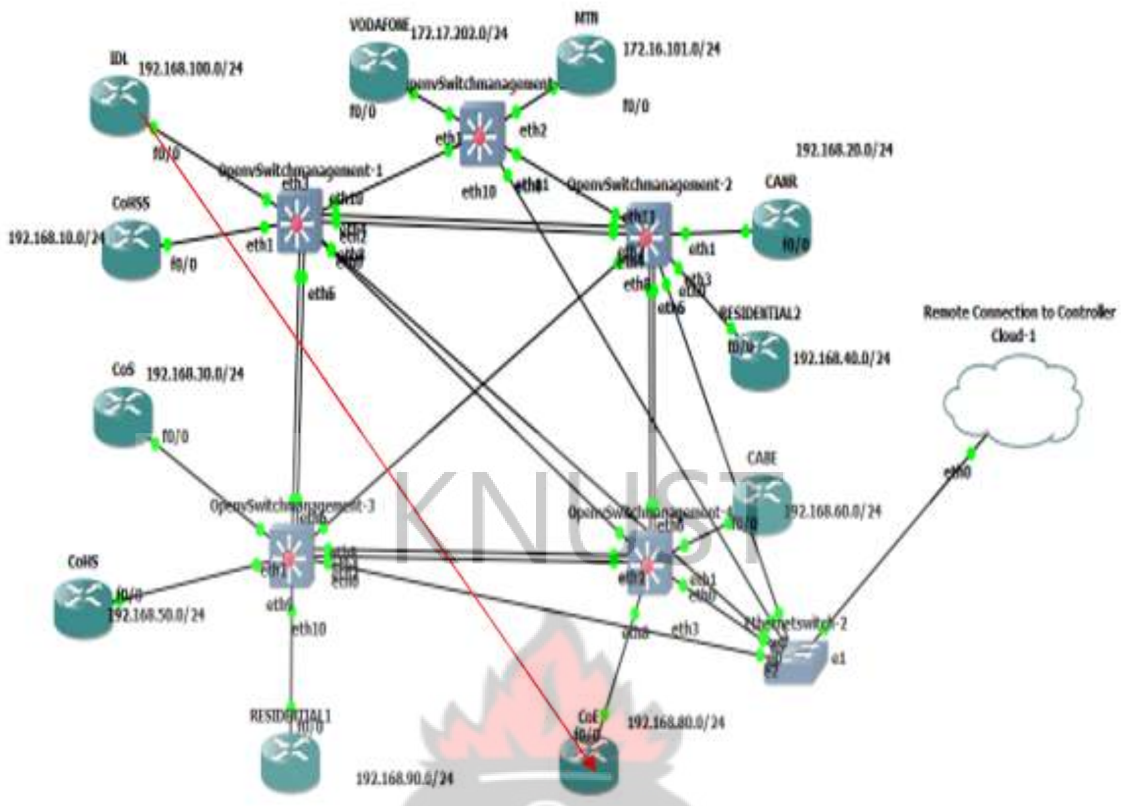


Figure 4.5 Ping Trace from IDL to CoE

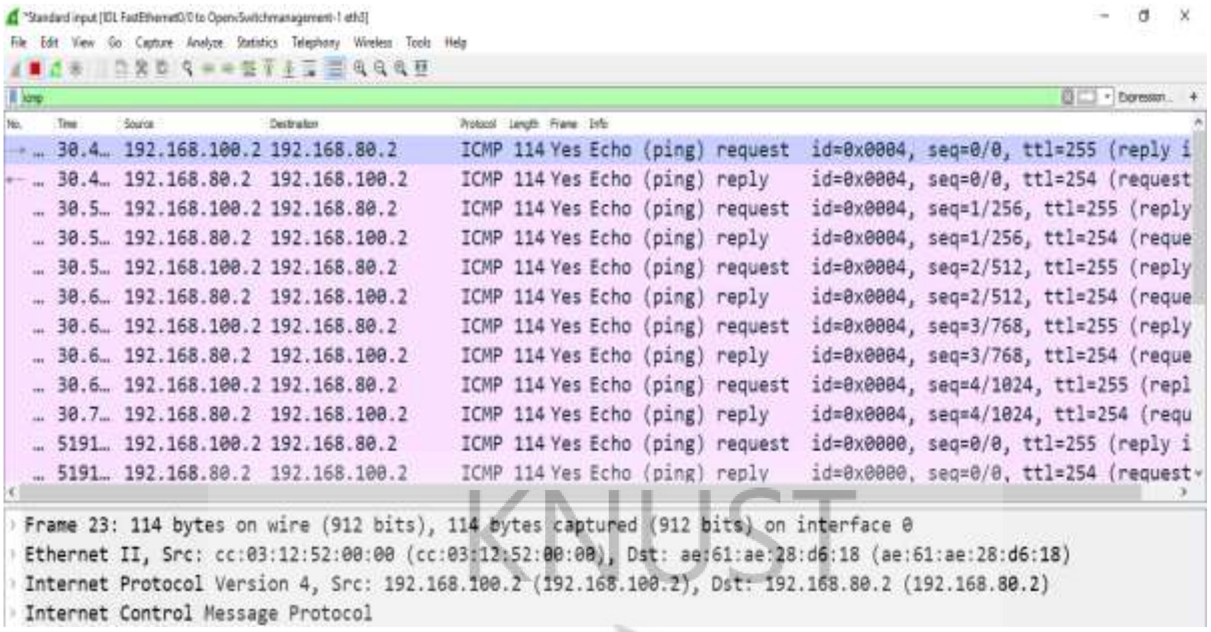
4.2.1 Results And Analysis Of Data Plane Communication

The results from the ping trace carried out from IDL to CoE devices are discussed below.

4.2.1.1 Ping Trace Statisics

In order to check the forwarding of packets from one host to another, the Internet Control Message Protocol was used to carry out a ping trace from one host (192.168.100.2 representing IDL) to another host (192.168.80.2 representing the College of Engineering).

From the figure below, the ping trace request shows a reply from the College Engineering host. This proves that forwarding of packets has been achieved.


*Standard input [IOL FastEthernet0/0 to OpenvSwitchmanagement-1 eth3]
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
icmp
Expression...

```
No.    Time    Source          Destination       Protocol Length Frame Info
→ …    30.4…   192.168.100.2   192.168.80.2      ICMP 114 Yes Echo (ping) request  id=0x0004, seq=0/0, ttl=255 (reply i
←  …   30.4…   192.168.80.2    192.168.100.2     ICMP 114 Yes Echo (ping) reply    id=0x0004, seq=0/0, ttl=254 (request
   …   30.5…   192.168.100.2   192.168.80.2      ICMP 114 Yes Echo (ping) request  id=0x0004, seq=1/256, ttl=255 (reply
   …   30.5…   192.168.80.2    192.168.100.2     ICMP 114 Yes Echo (ping) reply    id=0x0004, seq=1/256, ttl=254 (reque
   …   30.5…   192.168.100.2   192.168.80.2      ICMP 114 Yes Echo (ping) request  id=0x0004, seq=2/512, ttl=255 (reply
   …   30.6…   192.168.80.2    192.168.100.2     ICMP 114 Yes Echo (ping) reply    id=0x0004, seq=2/512, ttl=254 (reque
   …   30.6…   192.168.100.2   192.168.80.2      ICMP 114 Yes Echo (ping) request  id=0x0004, seq=3/768, ttl=255 (reply
   …   30.6…   192.168.80.2    192.168.100.2     ICMP 114 Yes Echo (ping) reply    id=0x0004, seq=3/768, ttl=254 (reque
   …   30.6…   192.168.100.2   192.168.80.2      ICMP 114 Yes Echo (ping) request  id=0x0004, seq=4/1024, ttl=255 (repl
   …   30.7…   192.168.80.2    192.168.100.2     ICMP 114 Yes Echo (ping) reply    id=0x0004, seq=4/1024, ttl=254 (requ
   …   5191…   192.168.100.2   192.168.80.2      ICMP 114 Yes Echo (ping) request  id=0x0000, seq=0/0, ttl=255 (reply i
   …   5191…   192.168.80.2    192.168.100.2     ICMP 114 Yes Echo (ping) reply    id=0x0000, seq=0/0, ttl=254 (request

› Frame 23: 114 bytes on wire (912 bits), 114 bytes captured (912 bits) on interface 0
› Ethernet II, Src: cc:03:12:52:00:00 (cc:03:12:52:00:00), Dst: ae:61:ae:28:d6:18 (ae:61:ae:28:d6:18)
› Internet Protocol Version 4, Src: 192.168.100.2 (192.168.100.2), Dst: 192.168.80.2 (192.168.80.2)
› Internet Control Message Protocol
```

Figure 4.6 Ping Trace Statistics

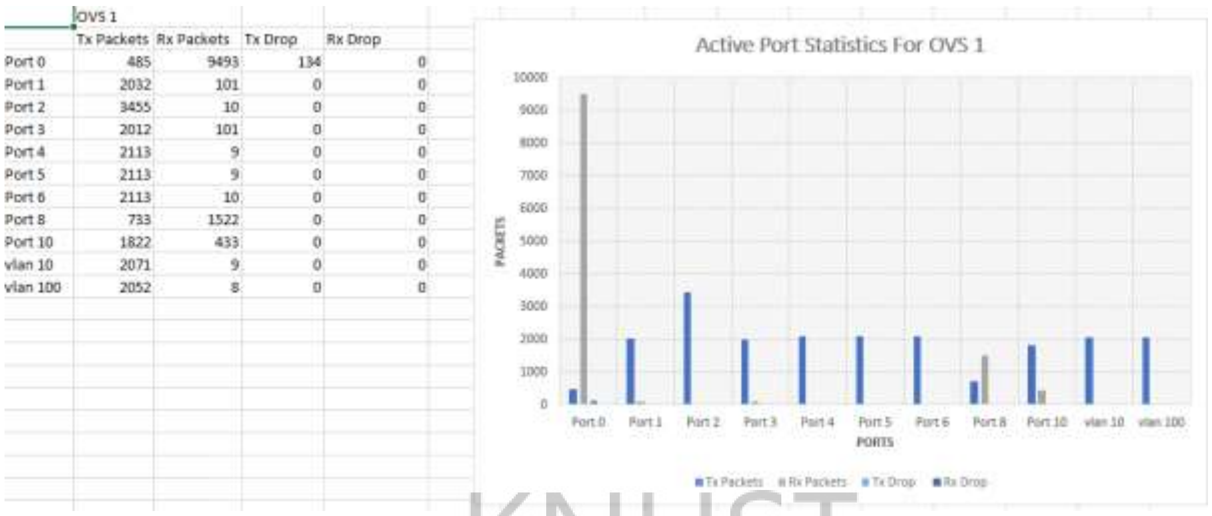
4.2.1.2 Openvswitch Port Statistics

In Figure 4.5, the green coloured portion on the OpenVswitches shows the active ports. These ports are capable of forwarding and receiving packets. The active ports of OpenVswitch 1 were queried to validate the above proposition.

Figure 4.7 shows the port statistics of the active ports of OpenVswitch 1. There is 0 transmission packet drop on all ports of the switch except Port 0.

Port 2 transmits the highest number of 3,455 packets and Port 0 has the highest number of 9,493 packets. This represents the number of packets transmitted from the Controller to the OpenVswitch.

Port 8 which is the port that connects OpenVswitch 1 to OpenVswitch 4 has 733 transmitted packets and 1522 received packets from the ping trace carried out and displayed in Figure 3.



	OVS 1			
	Tx Packets	Rx Packets	Tx Drop	Rx Drop
Port 0	485	9493	134	0
Port 1	2032	101	0	0
Port 2	3455	10	0	0
Port 3	2012	101	0	0
Port 4	2113	9	0	0
Port 5	2113	9	0	0
Port 6	2113	10	0	0
Port 8	733	1522	0	0
Port 10	1822	433	0	0
vlan 10	2071	9	0	0
vlan 100	2052	8	0	0

Figure 4.7 Open vSwitch Port Statistics

4.3 EXPERIMENTAL SETUP FOR RESULT DATA PLANE-CONTROL PLANE COMMUNICATION

In order to obtain the metrices from the communication between the data plane and the control plane, the Iperf tool was used. It has the ability to create data streams to measure the throughput and round-trip time between the two ends in one or both directions.

Below is the Graphical User Implementation used in GNS3 for obtaining the metrices for the data plane-control plane communication.

Two desktop devices running the Ubuntu Operating System were connected to OpenVswitch 1 representing IDL and OpenVswitch 4 representing the College of Engineering.

The desktop representing the College of Engineering is the client and is used in generating streams of packets in the form of a set of parallel threads to the desktop representing IDL which is the server.

The client generates an iterated number of parallel threads starting from 2 threads to 18 threads to the server with the root QoS class set at a minimum of 1 Mbps and maximum of 9 Mbps at the Application Layer.

The Wireshark Packet Analyzer was used in obtaining the results for this section.

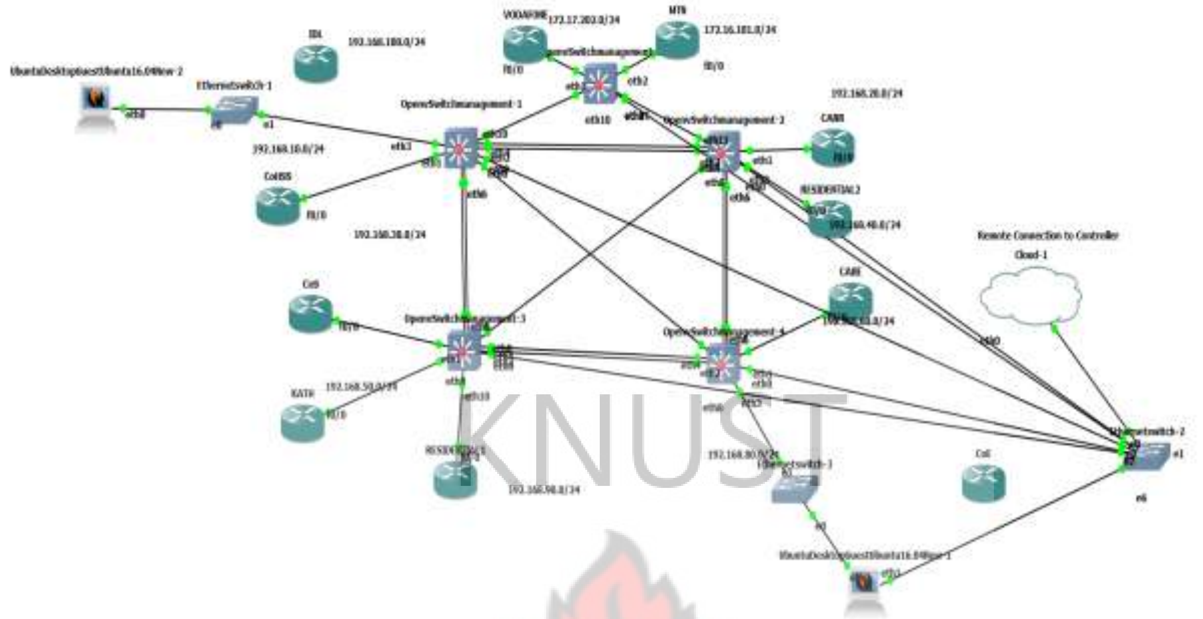


Figure 4.8 Graphical User Implementation used in GNS3 for obtaining metrices for data plane-control plane communication.

4.3.1 Results and Analysis of Data Plane-Control Plane Communication

The results and analysis of the communication between the devices at the data plane and the OpenDaylight Controller are presented in this section.

4.3.1.1 Control Plane Global View

The Controller has to have a full knowledge of all the devices and links in the infrastructure layer or data plane in order to automate and orchestrate network configurations. This full knowledge is called the Global View.

There is a direct correlation between Figure 4.9 and Figure 4.8 There are 5 OpenVswitches and 11 network devices in both.

Also, a node with IP address 192.168.90.2 representing Residential 1 portion of the network is clearly seen. The correlation between both figures shows that the Controller has obtained a global view of the underlying infrastructure layer

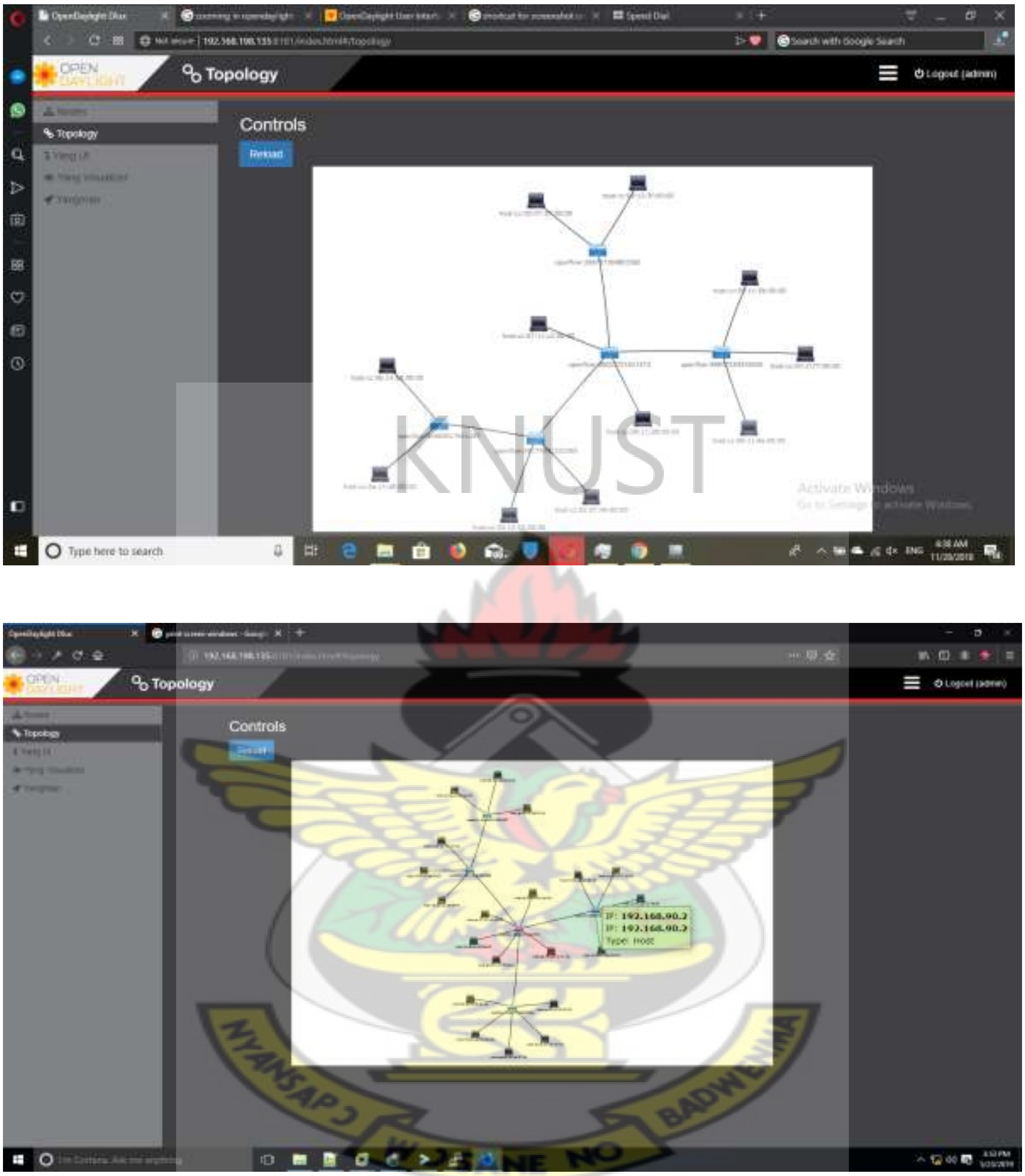


Figure 4.9 Global View of Implemented Campus Network

4.3.1.2 Southbound Interface Statistics

The results in the next set of figures show the communication between the Controller and OpenVswitch1 and the Controller and OpenVswitch 4 during the Iperf test using an implementation of the QoS Configuration in the Bandwidth on Demand Application

These include OpenVswitch to Controller Latency, OpenVswitch to Controller Throughput and OpenVswitch Flow Statistics.

4.3.1.3 Openvswitch To Controller Latency (Openvswitch 1)

Round trip time (RTT) is the propagation time for sending and receiving a packet in a communication network.[73]. Round trip time is the latency from the OpenVswitch to the Controller and vice-versa plus the processing time

The latency between the switch and the controller becomes steady at 62.5 milliseconds as the traffic generation simulation is carried out. This is the result of the service curve emanating from the control cycle that occurs between the controller and the switch. This was discussed in Chapter 3. It peaks at 1.5 seconds after 400 seconds of the simulation.

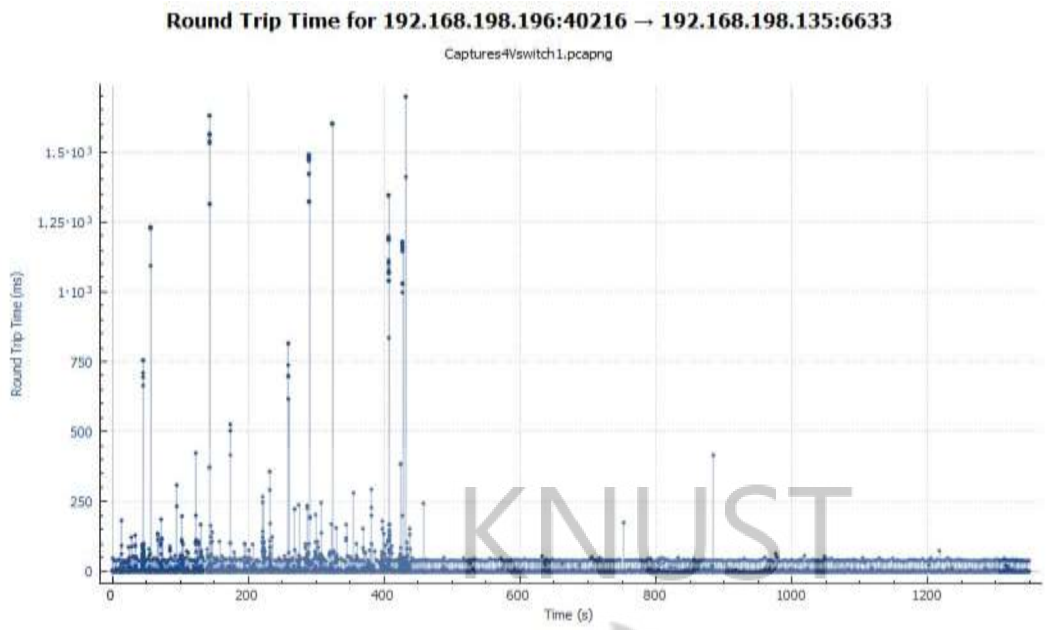


Figure 4.10 Latency between OpenVswitch 1 and Controller

4.3.1.4 OpenVswitch to Controller Latency (OpenVswitch 4)

The latency between OpenVswitch 4 and the Controller also becomes steady at 50 milliseconds during the traffic simulation. It peaks at 2.1 seconds during the 80th second of the simulation as per the service cycle between the switch and the controller.

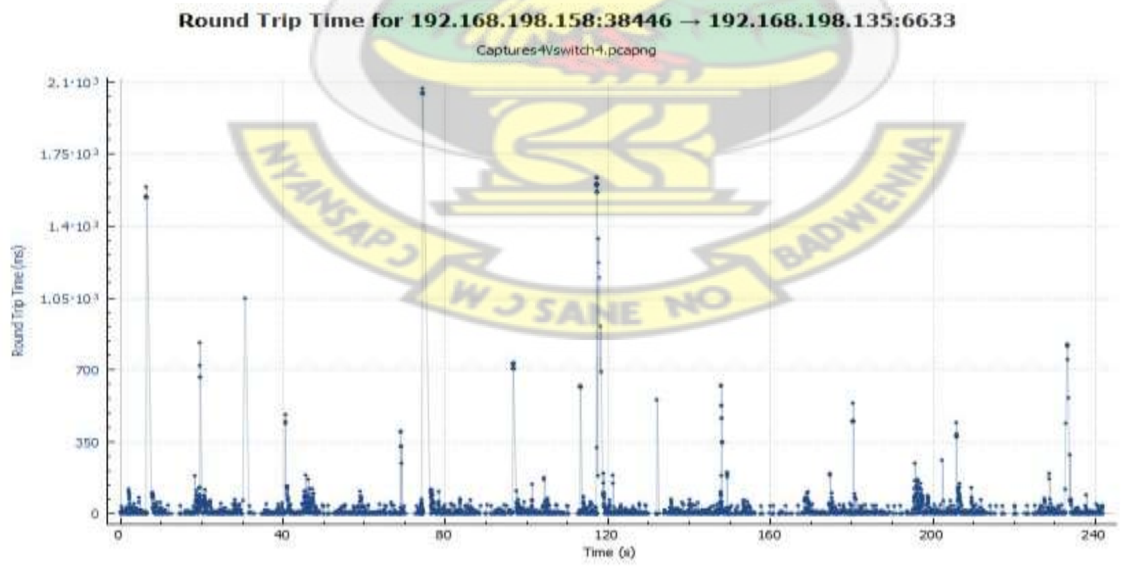


Figure 4.11 Latency between OpenVswitch 4 and Controller

There is a clear correlation in the average and peak values of the roundtrip measured from both Open vSwitches.

4.3.1.5 OpenVswitch To Controller Throughput (Openvswitch 1)

This metric measures the amount of data moved successfully from OpenVswitch 1 to the Controller in a given time period, and typically measured in megabits per second (Mbps) [74].

There is a steady flow of 2Mbps of data from the OpenVswitch to the Controller. The peak throughput is 8Mbps

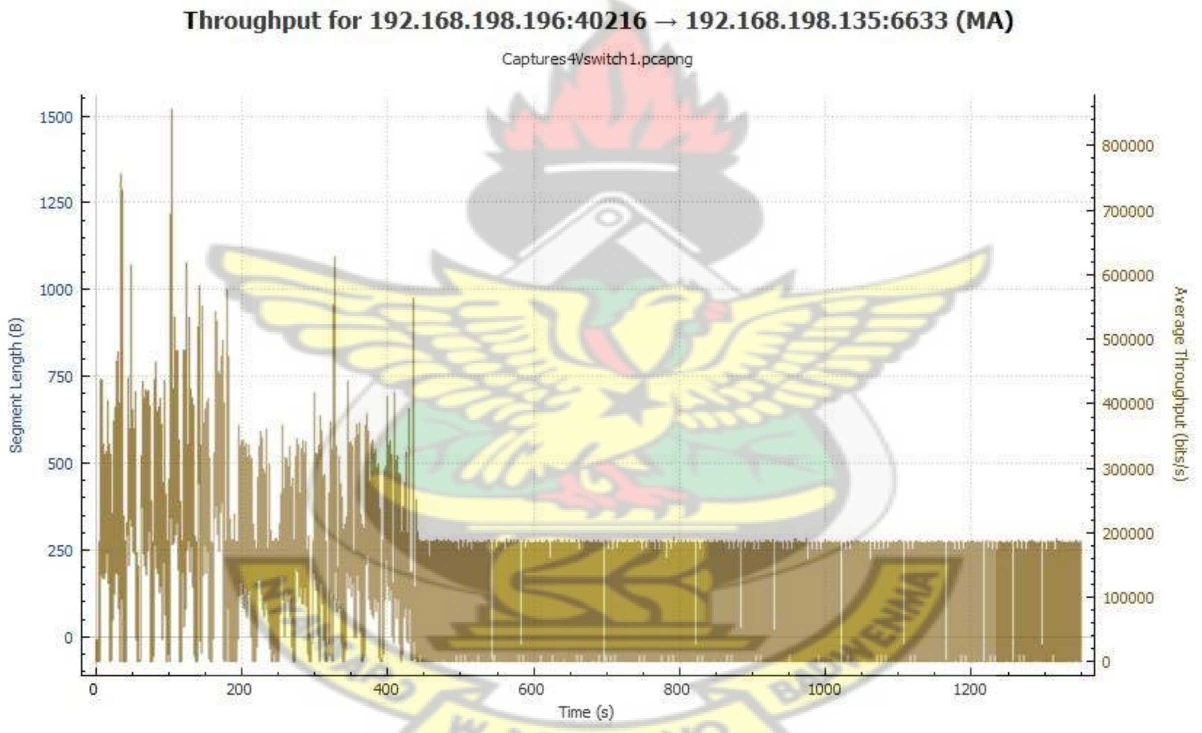


Figure 4.12 Throughput between OpenVswitch 1 and Controller

4.3.1.6 Openvswitch To Controller Throughput (Openvswitch 4)

This metric measures the amount of data moved successfully from OpenVswitch 4 to the Controller in a given time period, and typically measured in megabits per second (Mbps)[75]

For OpenVswitch 4, the throughput peaks at 18Mbps in the 120th second of the traffic simulation due to the increasing number of parallel requests being made by the host directly attached it. It has an average of 9Mbps during the simulation period.

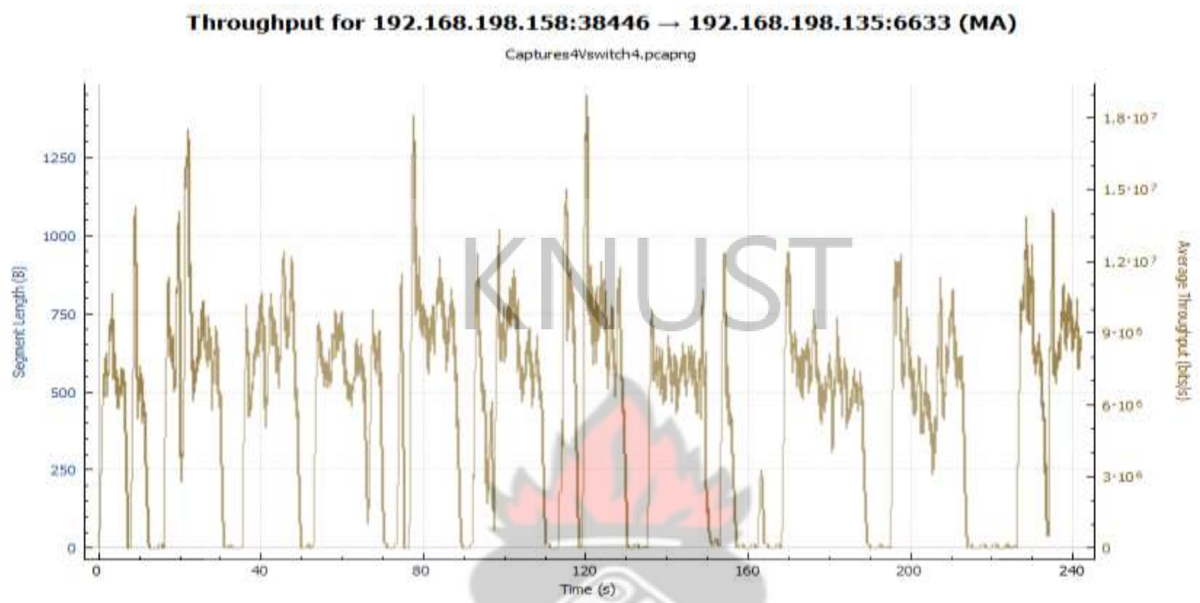


Figure 4.13 Throughput between OpenVswitch 4 and Controller

Results in Figure 4.12 and Figure 4.13 show that the peak throughput is a two-fold increment of the average throughput.

4.4 EXPERIMENTAL SETUP FOR CONTROL PLANE-APPLICATION PLANE COMMUNICATION

In order to obtain the metrices from the communication between the application plane and the control plane, the Iperf tool was used. An Iperf test produces a report of the bandwidth, packet loss and other parameters using timestamps.

Figure 4.8 is used to carry out the experimental set up for the control plane-application plane communication

The desktop representing the College of Engineering is the client and is used in generating streams of packets in the form of a set of parallel threads to the desktop representing IDL which is the server.

The client generates an iterated number of parallel threads starting from 2 threads to 18 threads to the server with the root QoS class set at a minimum of 1 Mbps and maximum of 9 Mbps.

The following metrics are measured and graphed as the iterations go on. These are OpenVswitch Flow Statistics, Latency per Thread or Packet Group, Bandwidth Change per Thread or Packet Group and Bandwidth Change per Thread or Packet Group per Time

4.4.1 RESULTS AND ANALYSIS OF CONTROL PLANE-APPLICATION PLANE COMMUNICATION

The results and analysis of the communication between the devices at the control plane and the OpenDaylight Controller are presented in this section.

4.4.1.1 Flow Statistics for Openvswitch 1

Figure 4.14 details 7 axis points showing the characteristics of 7 flows picked from OpenVswitch 1.

On axis point 4, the flow has a priority of 2 with 98,480 packets in the flow. The flow stays in the network for 5739 seconds.

On axis 7 for example the flow has a priority of 100 with 2,304 packets in the flow stays in the network for 2304 seconds. These results stem from data transferred between the application and the switch after the bandwidth change has been triggered for the duration of the simulation.

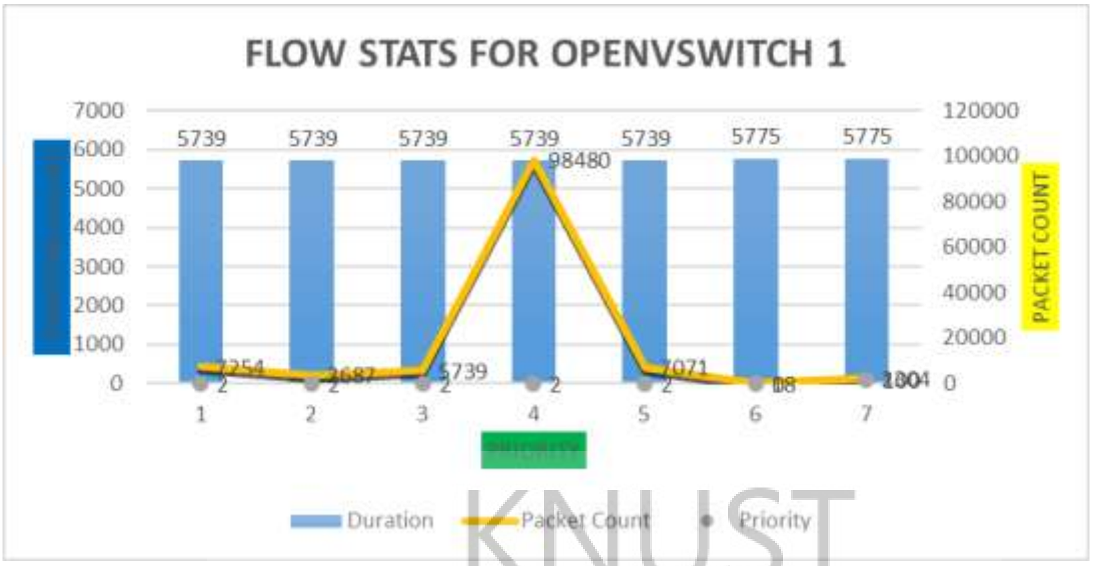


Figure 4.14 Flow Statistics for OpenVswitch 1

4.4.1.2 Flow Statistics For Openvswitch 4

The figure below details 12 axis points showing the characteristics of 12 flows picked from OpenVswitch 4.

On axis point 11, the flow has a priority of 2 with 62,375 packets in the flow. The flow stays in the network for 5739 seconds.

On axis 2 for example the flow has a priority of 100 with 3,623 packets in the flow stays in the network for 6168 seconds.

There is a correlation between the durations of the flows for priority 2 in both OpenVswitch 1 and OpenVswitch 4 at 5739 seconds in both cases.

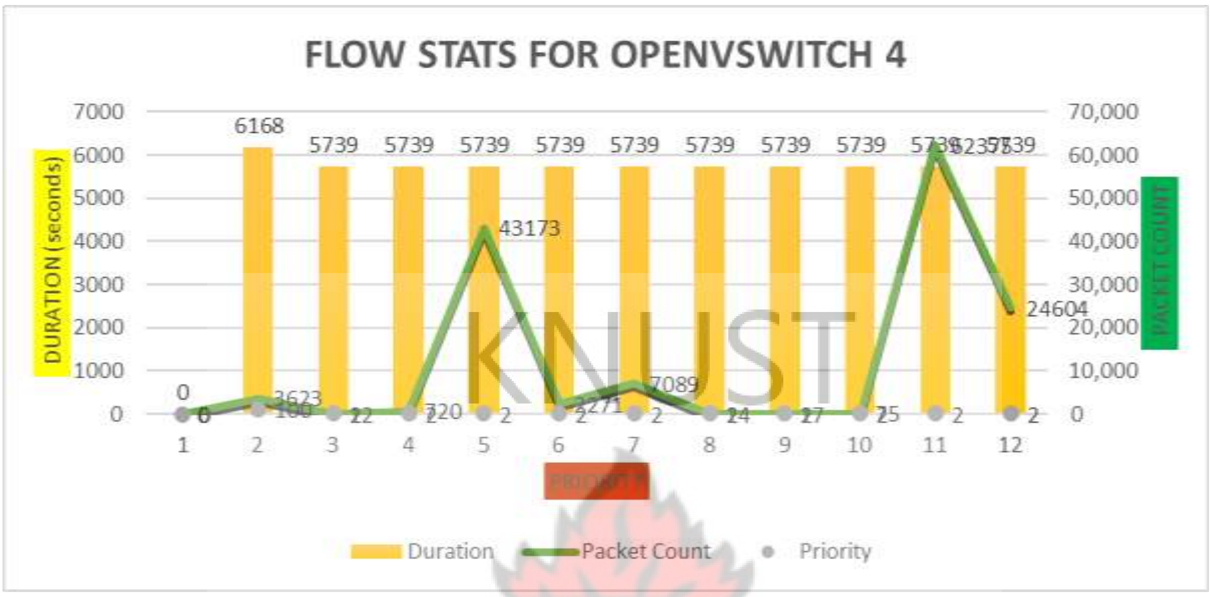


Figure 4.15 Flow Statistics for OpenVswitch 4

4.4.2 Results and Analysis For The Implementation Of The Bandwidth Utilization Scheme

The results show a record of the Iperf tool running on the client device representing the College of Engineering connected to OpenVswitch 4 after the triggering of the QoS Configuration setting in the Bandwidth on Demand Application via an API call. These results are based on the service curve model for a Software Defined Network derived from the Node Concatenation Theorem.

An iteration of parallel connections was used to model a number of requests that are made from the iperf client to the iperf server.

These include Latency per Thread Group, Bandwidth Change per Thread Group and Bandwidth Change per Thread Group per Time

4.4.2.1 Latency Per Thread Group

For every parallel thread, a number of ICMP echo messages are sent from the client at the College of Engineering and the server at IDL.

The round-trip time value shows the time for the movement of requests from the Infrastructure layer through the Control layer and to the Bandwidth on Demand Application layer and back.

From the graph in Figure 4.16 the 5 ping trace attempts represent 5 parallel requests from the client to the server, responses are received in both directions in a total of 39 milliseconds. The highest round-trip time of 141 milliseconds is received during 7 ping trace attempts. On the average the round-trip time lies in the range of 20 to 40 milliseconds for a vast majority of the attempts. This value shows a good control cycle time of the Campus Based Software Defined Network model for the Kwame Nkrumah University of Science and Technology as compared to 1000ms in the work done by F. Volpato et al [62]

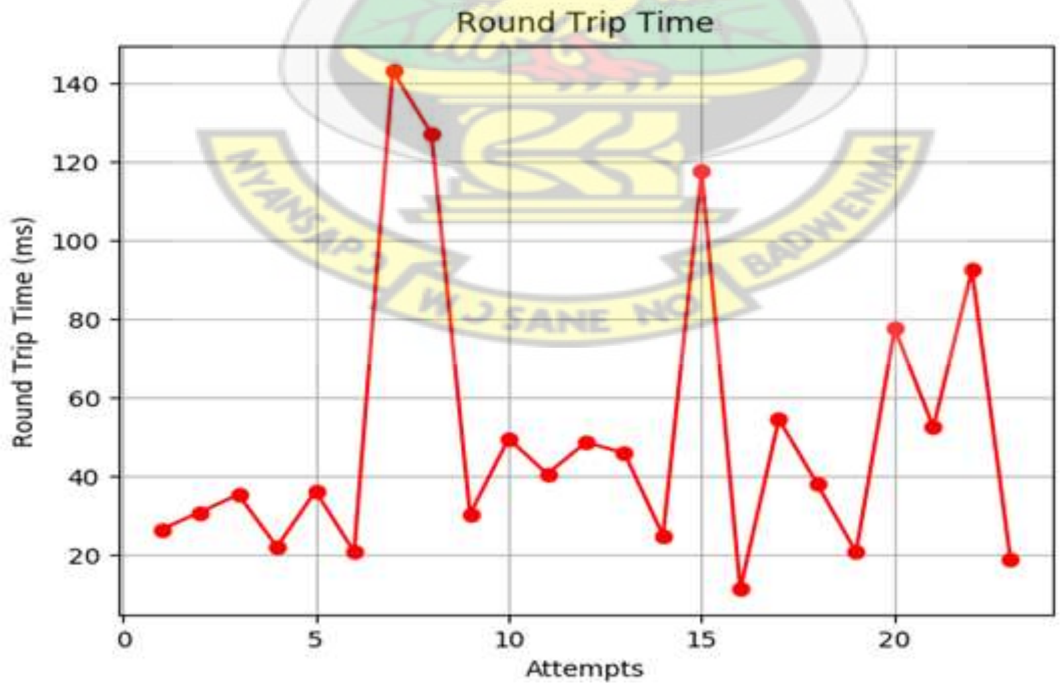


Figure 4.16 Round Trip per Thread Group

4.4.2.2 Bandwidth Change Per Thread Group

This metric shows the dynamic change of bandwidth based on the QoS Configuration setting which has the root class set to 1Mbps minimum and 9Mbps maximum.

Based on classifying, policing, scheduling and borrowing mechanisms, bandwidth is changed dynamically per the parallel thread iteration.

In the figure below, a total of 4Mbps is allocated for the 10-parallel thread iperf request. There is a steady rise in allocated bandwidth from the 1thread iteration to the 18-thread iteration peaking at about 7.8 Mbps for the 18th thread. This clearly shows that bandwidth is borrowed from the root class to an inner class and is shared equally in a leaf class for every single thread iteration satisfying the QoS Configuration definition.

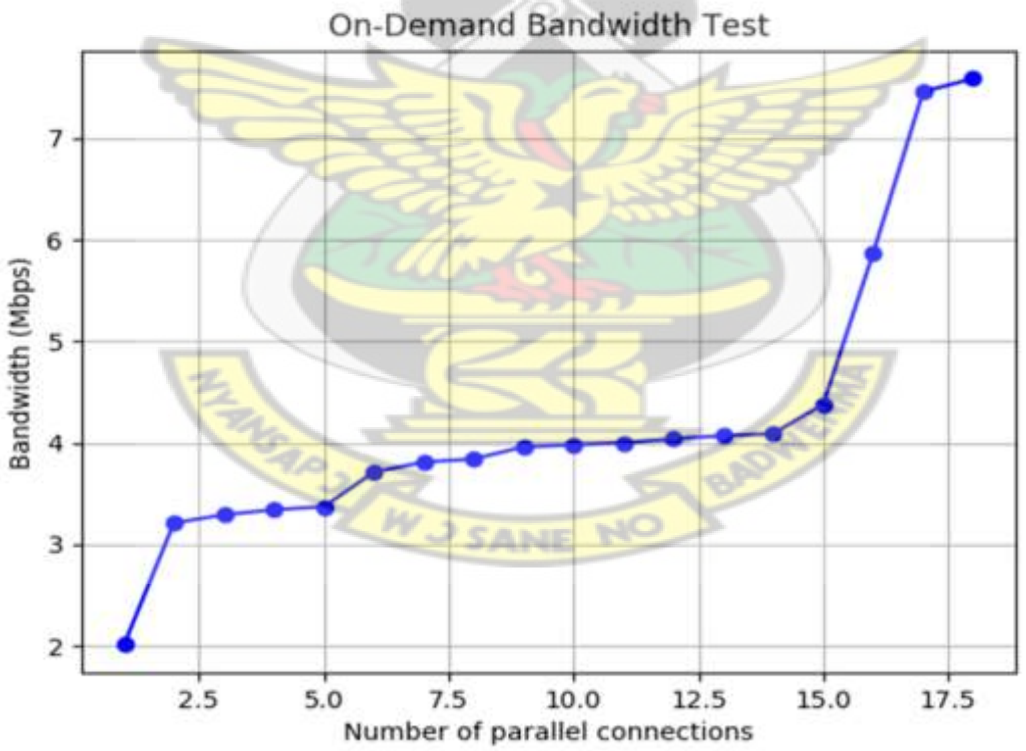


Figure 4 17 Bandwidth Change per Thread Group

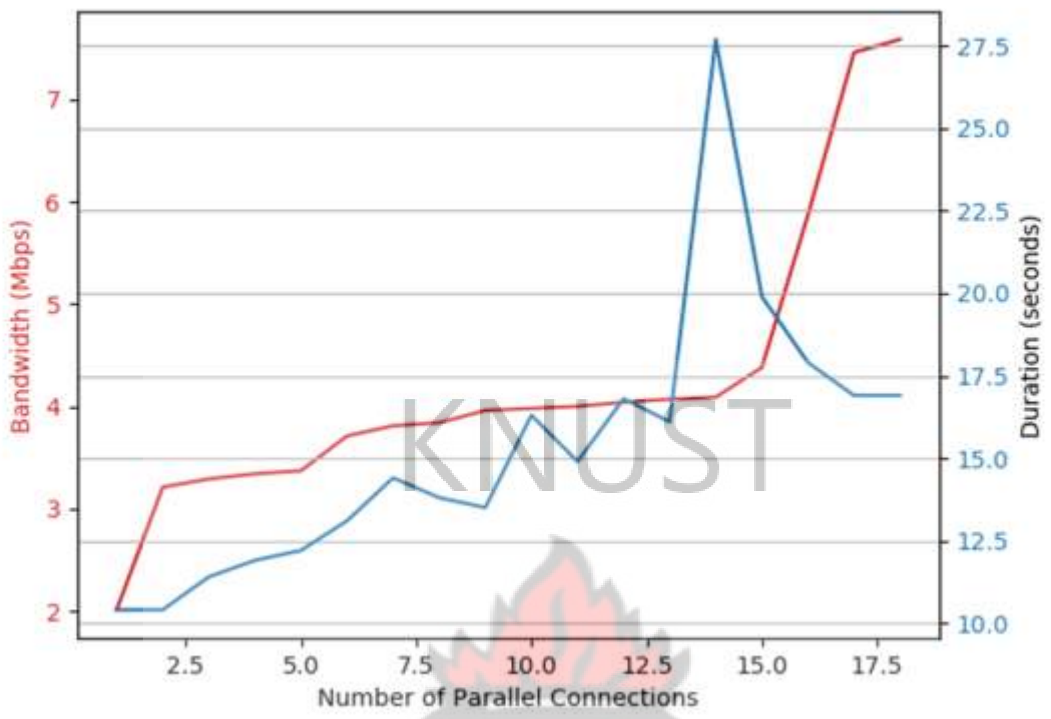
4.3.2.3 Bandwidth Change Per Thread Group Per Time

This metric shows the amount of time taken for the QoS setting to dynamically change the bandwidth for the thread-based iteration.

The graph above shows that for 10 parallel connections 4 Mbps is allocated in 17 seconds. It takes 17 seconds to allocate a leaf class for each of the 10 parallel connections.

This allocation peaks at a value of 4.3 Mbps in 27.5 seconds for 13 parallel connections. The graph however takes a downward trend from this peak value of 27.5 seconds. The amount of time for the dynamic allocation to take place reduces steady and tapers to 17 seconds for the 18-thread iteration.

This implies that after 27.5 seconds, the Software Defined-based campus network dynamically adapts to the QoS configuration changes as the thread iterations increase. The controller dynamically adapts to the changing conditions in the network and carries out the bandwidth changes in a shorter space of time.



🟥 **Bandwidth (Mbps)**

🟦 **Duration(seconds)**

Figure 4. 18 Bandwidth change per thread group per time.

4.5 CONCLUSION

In this chapter, a presentation and analysis of the results obtained from the conversion and emulation of a traditional campus network into a Software Defined campus-based one.

The results show the behaviour of packets based on Data Plane Device COMMUNICATION, Control Plane-Data Plane Communication (SBI Communication) and Control Plane-Application Plane Communication (NBI Communication).

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

This thesis focused on the design and implementation of a dynamic bandwidth scheme in a Software Defined-based campus network as means a of resolving the issue of static bandwidth configuration. The Local Area Network of the Kwame Nkrumah University of Science and Technology was used as a case study.

A traditional network was converted into a Software Defined-based campus network. Virtualization technology was used to emulate a three-tier architecture made up of network devices, a controller and a network application. The entire architecture was tested based on a three-stage process.

In the first phase, the lowest tier called the data plane was tested to check for the forwarding of packets using ICMP echo pings. The test proved successful as the ping request from the network device named Institute of Distance Learning (IDL) received a reply from the network device named College of Engineering. To further prove the forwarding of packets, the ports of an SDN-capable switch called the Open vSwitch were queried. All active ports of the switch were seen to be passing packets with the highest transmitting port being port 2 which passed 3,455 packets and the highest receiving port being port 0 which carried 9,493 packets.

In the second phase, a simulation was carried to test communication between the control plane and the data plane using the iperf application running on two Ubuntu hosts that were connected to two SDN-capable Open vSwitches. The latency and throughput between the controller and two SDN-capable Open Vswitches was investigated. The latency for the SDN-capable Open vSwitch labelled Open vSwitch 1 was an average of 62.5 milliseconds with a peak of 1.5 seconds. The latency for the SDN-capable Open vSwitch labelled Open vSwitch 4 was an average of 50 milliseconds with a peak of 2.1 seconds.

The throughput of the SDN-capable Open vSwitch labelled Open vSwitch 1 was an average of 2Mbps and a peak of 8Mbps while that of Open vSwitch 4 was 9Mbps averagely with a peak of 18Mbps.

The third phase was a test of communication between the control plane and the application plane using the iperf application running on two Ubuntu hosts that were connected to two SDN-capable Open vSwitches. The number of flows on both switches were queried. Open vSwitch 1 recorded 98,480 packets moving from the application plane. Open vSwitch 4 recorded 62,375 packets.

In this phase an iteration of parallel connections was carried out to test the concept of bandwidth utilization based on the Hierarchical Token Bucket Theorem. These parallel connections were called thread groups. The thread group was used to model a number of requests that are made from the iperf client to the iperf server.

The average latency for a thread group was 20 to 40 milliseconds. A maximum threshold of 9Mbps was set for all connections. The bandwidth variation per thread group occurs from as low as 2Mbps for 1 connection to 8Mbps for 18 connections. This variation shows borrowing from 9Mbps cap based on the amount of traffic requests being made. Also, this bandwidth variations based on the parallel iterations occur in 27.5 seconds during the 15th iteration. As the number of iterations increases the amount of time for the variations reduces and becomes steady at about 17 seconds. The results of the work show that Software Defined Networking and the Hierarchical Token Bucket Queuing Discipline can be used to facilitate dynamic utilization of bandwidth in campus networks.

5.2 RECOMMENDATIONS

In this work a Software Defined-based campus network was designed and tested within a virtual environment which emulates the software running on a set of network devices. Dynamic bandwidth utilization was carried out using a network application. However, a Software Defined Network has the capability of carrying out other network functions including routing, security and the creation of virtual local area networks. Based on the above, future work will be carried out to test whether all of these functions can be implemented within one architecture.

REFERENCES

[1] G. Pujolle, “Software Networks, Wiley ISTE 2015” p. 262

[2] Software Defined Networking for the Utilities and Energy Sector: Fujitsu Network Communications Inc 2014.

[3] Omollo, Kathleen Ludewig, “Information and Communication Technology Infrastructure Analysis of Kwame Nkrumah University of Science and Technology and University of Ghana 2011”

[4] https://www.networkcomputing.com/networking/new-network-management-tactic-bandwidth-demand/592428015, [Accessed:4-Dec-2018]

[5] “What is Switching Fabric? - Definition from Techopedia,” *Techopedia.com*. [Online]. Available: https://www.techopedia.com/definition/16015/switching-fabric. [Accessed: 24-Jun-2019].

[6] Yoram Orzach, “Ch 01 --- introduction to sdn-nfv,” 15:25:52 UTC.

[7] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, “A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks,” *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[8] “Software-Defined Networking The New Norm for Networks.pdf, ONF White Paper April 13, 2012.”

[9] N. Feamster, J. Rexford, and E. Zegura, “The road to SDN: an intellectual history of programmable networks,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[10] Scott Shenker, “The Future of Networking, and the Past of Protocols,” p. 30.

[11] P. Göransson and C. Black, *Software defined networks: a comprehensive approach*. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014.

[12] “OpenVswitch https://www.openvswitch.org” [Accessed:Oct-13-2018]

[13] “Introduction of Indigo Virtual Switch and Switch Light BETA,” *Big Switch Networks, Inc.* [Online]. Available: https://www.bigswitch.com/topics/introduction-of-indigo-virtual-switch-and-switch-light-beta. [Accessed: 25-Jun-2019].

[14] A. Networks, “Arista Platforms 400GbE - 100GbE - 40GbE - 25GbE - 10GbE - Arista - Arista,” *Arista Networks*. [Online]. Available: https://www.arista.com/en/ products/ platforms. [Accessed: 25-Jun-2019].

[15] “NFX Series Product Comparison - Juniper Networks.” [Online]. Available: https://www.juniper.net/us/en/products-services/sdn/nfx-series/compare?p=NFX150,NFX250. [Accessed: 25-Jun-2019].

[16] Dean Pemberton, Andy Linton, and Sam Russell, “OpenVSwitch.” University of Oregon.

[17] N. McKeown *et al.*, “OpenFlow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, p. 69, Mar. 2008.

[18 "Home - OpenDaylight." [Online]. Available: https://www.opendaylight.org/. [Accessed: 25-Jun-2019].

[19] "ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out." [Online]. Available: https://onosproject.org/. [Accessed: 25-Jun-2019].

[20] N. O. X. Repo, *The POX network software platform. Contribute to noxrepo/pox development by creating an account on GitHub*. 2019.

[21] "Ryu SDN Framework." [Online]. Available: https://osrg.github.io/ryu/. [Accessed: 25-Jun-2019].

[22] "Floodlight OpenFlow Controller -," *Project Floodlight*. [Online]. Available: http://www.projectfloodlight.org/floodlight/. [Accessed: 25-Jun-2019].

[23] S. Badotra and J. Singh, "Open Daylight as a Controller for Software Defined Networking," *International Journal of Advanced Research in Computer Science*, p. 8, 2017.

[24] "OpenDaylight User Guide," p. 117 [Accessed:25-Jun-2019]

[25] Charles Eckel, "OpenDaylight-Network-Programmability," p. 58, 2017.

[26] "Applications of Computer Networks," *TurboFuture*. [Online]. Available: https://turbofuture.com/computers/Network-Application. [Accessed: 25-Jun-2019].

[27] S. Kaur, K. Kaur, and V. Gupta, "Implementing Static Router based on Software Defined Networking," in *2016 International Conference on Computational*

Techniques in Information and Communication Technologies (ICCTICT), New Delhi, India, 2016, pp. 358–360.

[28] V. Gupta, K. Kaur, and S. Kaur, "Network programmability using software defined networking," p. 4, 2016.

[1] G. Pujolle, "Software Networks," p. 262.

[29] J. Tourrilhes, P. Sharma, S. Banerjee, and J. Pettit, "SDN and OpenFlow Evolution: A Standards Perspective," *Computer*, vol. 47, no. 11, pp. 22–29, Nov. 2014.

[30] Open Network Foundation, "OpenFlow Switch Specification.pdf." 25-Jun-2012.

[31] "application programming interface - Google Search." [Online]. Available: https://www.google.com/search?q=application+programming+interface [Accessed: 26-Jun-2019].

[32] "What is REST?," *Codecademy*. [Online]. Available: https://www.codecademy.com/articles/what-is-rest. [Accessed: 26-Jun-2019].

[33] W. Zhou, L. Li, M. Luo, and W. Chou, "REST API Design Patterns for SDN Northbound API," in *2014 28th International Conference on Advanced Information Networking and Applications Workshops*, BC, Canada, 2014, pp. 358–365.

[34] "What is RESTful API? - Definition from WhatIs.com." [Online]. Available: https://searchmicroservices.techtarget.com/definition/RESTful-API. [Accessed: 26-Jun-2019].

[35] L. Li, W. Chou, W. Zhou, and 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, Mar. 2016.

[36] A. E. Maslov, S. L. Katuntsev, and A. A. Maliavko, “Study and implementation of authentication mechanism by RADIUS-server in switches and routers using NETCONF protocol,” in *2017 18th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM)*, Erlagol (Altai Republic), Russia, 2017, pp. 111–114.

[37] “YANG, NETCONF, RESTCONF What is this all about and how is it used for multi-layer networks.pdf.” .

[38] T. Tanaka, “Flexible and robust optical network technologies for SDN and network virtualization,” in *2014 12th International Conference on Optical Internet 2014 (COIN)*, Jeju, 2014, pp. 1–2.

[39] L. Xingtao, G. Yantao, W. Wei, Z. Sanyou, and L. Jiliang, “Network virtualization by using software-defined networking controller based Docker,” in *2016 IEEE Information Technology, Networking, Electronic and Automation Control Conference*, Chongqing, China, 2016, pp. 1112–1115.

[40] “Mininet http://mininet.org/overview” .

[41] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and Ligia Rodrigues Prete, “Using Mininet for emulation and prototyping Software-Defined Networks,” in *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, Bogota, Colombia, 2014, pp. 1–6.

[42] "IMUNES - IP network emulator / simulator." [Online]. Available: http://imunes.net/. [Accessed: 27-Jun-2019].

[43] "Home, https://www.estinet.com/ns/" *EstiNet - Simulator*. [Accessed: 27-Jun-2019]

[44] "Emulab - Emulab." [Online]. Available: https://www.emulab.net/portal/frontpage.php. [Accessed: 27-Jun-2019].

[45] "GNS3 Documentation.pdf https://docs.gns3.com"

[46] "Workstation Player : Run a Second, Isolated Operating System on a Single PC with VMware Workstation Player," *VMware*. [Online]. Available: https://www.vmware.com/products/workstation-player.html. [Accessed: 27-Jun-2019].

[47] P. Trimintzios, G. Pavlou, and I. Andrikopoulos, "Providing Traffic Engineering Capabilities in IP Networks Using Logical Paths p. 14.

[48] Z. Shu *et al.*, "Traffic engineering in software-defined networking: Measurement and management," *IEEE Access*, vol. 4, pp. 3246–3256, 2016.

[49] S. Jeong, D. Lee, J. Hyun, J. Li, and J. W.-K. Hong, "Application-aware traffic engineering in software-defined network," in *2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, Seoul, 2017, pp. 315–318.

[50] Y. Zhou, B. Ramamurthy, B. Guo, and S. Huang, "Supporting Dynamic Bandwidth Adjustment Based on Virtual Transport Link in Software-Defined IP Over Optical Networks," *Journal of Optical Communications and Networking*, vol. 10, no. 3, p. 125, Mar. 2018.

[51] S. Ren, Q. Feng, Y. Wang, and W. Dou, “A Service Curve of Hierarchical Token Bucket Queue Discipline on Soft-Ware Defined Networks Based on Deterministic Network Calculus: An Analysis and Simulation,” *J. Adv. Comput. Netw*, vol. 5, no. 1, 2017.

[52] D. G. Balan and D. A. Potorac, “Linux HTB queuing discipline implementations,” in *2009 First International Conference on Networked Digital Technologies*, Ostrava, 2009, pp. 122–126.

[53] “HTB manual - user guide.” [Online]. Available: http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm. [Accessed: 01-Jul-2019].

[54] T. Bhattacharjee, V. Gopal, L. N. Ngangoua, and C. Raghunath, “Traffic Light: Network Traffic Monitoring and Allocation,” p. 9.

[55] S. Bhelekar, M. Iyer, G. Mehta, and S. Chaudhari, “Dynamic load balancing strategy in software-defined networking,” in *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, Tirunelveli, 2017, pp. 875–878.

[56] N. Zope, S. Pawar, and Z. Saquib, “Firewall and load balancing as an application of SDN,” in *2016 Conference on Advances in Signal Processing (CASP)*, Pune, India, 2016, pp. 354–359.

[57] D. Satasiya and Raviya Rupal D., “Analysis of Software Defined Network firewall (SDF),” in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, Chennai, India, 2016, pp. 228–231.

[58] K. Kaur, K. Kumar, J. Singh, and N. S. Ghumman, “Programmable firewall using Software Defined Networking,” p. 5, 2015.

[59] P. Krongbaramee and Y. Somchit, “Implementation of SDN Stateful Firewall on Data Plane using Open vSwitch,” in *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, Nakhonpathom, 2018, pp. 1–5.

[60] A. Mendiola *et al.*, “Multi-domain bandwidth on demand service provisioning using SDN,” in *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, Seoul, South Korea, 2016, pp. 353–354.

[61] A. O. Adedayo and B. Twala, “QoS functionality in software defined network,” in *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju, 2017, pp. 693–699.

[62] F. Volpato, M. P. Da Silva, A. L. Goncalves, and M. A. R. Dantas, “An Autonomic QoS management architecture for Software-Defined Networking environments,” in *2017 IEEE Symposium on Computers and Communications (ISCC)*, Heraklion, Greece, 2017, pp. 418–423.

[63] Van-Giang Nguyen and Young-Han Kim “SDN-Based Enterprise and Campus Networks: A Case of VLAN Management,” *Journal of Information Processing Systems*, 2015.

[64] A. Amelyanovich, M. Shpakov, A. Muthanna, M. Buinevich, and A. Vladyko, “Centralized control of traffic flows in wireless LANs based on the SDN concept,” in *2017 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SINKHROINFO)*, Kazan, Russia, 2017, pp. 1–5.

[65] M. S. Olimjonovich, “Software Defined Networking: Management of network resources and data flow,” in *2016 International Conference on Information Science and Communications Technologies (ICISCT)*, Tashkent, Uzbekistan, 2016, pp. 1–3.

[66] S.-C. Lin, P. Wang, and M. Luo, "Jointly optimized QoS-aware virtualization and routing in software defined networks," *Computer Networks*, vol. 96, pp. 69–78, Feb. 2016.

[67] "Umadevi et al. - 2017 - Multilevel queue scheduling in software defined networks.pdf." .

[68] Veena S, R. P. Rustagi, and K. N. B. Murthy, "Network management and performance monitoring using Software Defined Networks," in *20th Annual International Conference on Advanced Computing and Communications (ADCOM)*, Bangalore, India, 2014, pp. 29–31.

[69] B. Gu, M. Dong, C. Zhang, Z. Liu, and Y. Tanaka, "Real-time pricing for on-demand bandwidth reservation in SDN-enabled networks," in *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA, 2017, pp. 696–699.

[70] C. Fei, B. Zhao, W. Yu, C. Wu, and J. Bao, "A research platform for software defined satellite networks," in *2017 16th International Conference on Optical Communications and Networks (ICOCN)*, Wuzhen, 2017, pp. 1–2.

[71] T. Theodorou and L. Mamatas, "CORAL-SDN: A software-defined networking solution for the Internet of Things," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Berlin, 2017, pp. 1–2.

[72] I. F. Akyildiz, S.-C. Lin, and P. Wang, "Wireless software-defined networks (W-SDNs) and network function virtualization (NFV) for 5G cellular systems: An overview and qualitative evaluation," *Computer Networks*, vol. 93, pp. 66–79, Dec. 2015.

[73] https://en.wikipedia.org/wiki/Round-trip_delay_time [Accessed: 28-Jun-2019]

[74] https://searchnetworking.techtarget.com/definition/throughput [Accessed: 28-Jun-2019]

[75] https://en.wikipedia.org/wiki/Traffic_flow_(computer_networking) [Accessed: 28-Jun-2019]

REFERENCES FOR FIGURES

Figure 1.1 http://www.excitingip.net/27/a-basic-enterprise-lan-network-architecture-block-diagram-and-components/

Figure 1.2 The Future of Networking and the Past of Protocols by Scott Shenker et al

Figure 2.1 What is Switching Fabric? - Definition from Techopedia," Techopedia.com. [Online]. Available: https://www.techopedia.com/definition/16015/switching-fabric. [Accessed: 24-Jun-2019

Figure 2.2 Source: NDI Communications-Training and Education

Figure 2.3 The Future of Networking and the Past of Protocols by Scott Shenker et al

Figure 2.4 https://www.researchgate.net/A novel industrial control architecture based on Software-Defined Network

Figure 2.5 P. Göransson and C. Black, Software defined networks: a comprehensive approach. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014

Figure 2.6 P. Göransson and C. Black, Software defined networks: a comprehensive approach. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014

Figure 2.7 online.fliphtml5.com

Figure 2.8 P. Göransson and C. Black, Software defined networks: a comprehensive approach. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014

Figure 2.9 Charles Eckel, OpenDaylight-Network-Programmability

Figure 2.10 P. Göransson and C. Black, Software defined networks: a comprehensive approach. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014

Figure 2.11 P. Göransson and C. Black, Software defined networks: a comprehensive approach. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014

Figure 2.12 P. Göransson and C. Black, Software defined networks: a comprehensive approach. Amsterdam Boston Heidelberg London: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2014

Figure 2.16 Source: W. Zhou, L. Li, M. Luo, and W. Chou, "REST API Design Patterns for SDN Northbound AP

Figure 2.17 W. Zhou, L. Li, M. Luo, and W. Chou, "REST API Design Patterns for SDN Northbound

Figure 2.18 A. E. Maslov, S. L. Katuntsev, and A. A. Maliavko, "Study and implementation of authentication mechanism by RADIUS-server in switches and routers using NETCONF protocol

Figure 2.19 YANG, NETCONF, RESTCONF What is this all about and how is it used for multi-layer networks.pdf

Figure 2.20 https://www.cisco.com

Figure 3.1 http://www.excitingip.net/27/a-basic-enterprise-lan-network-architecture-block-diagram-and-components/

Figure 3.2 The Future of Networking and the Past of Protocols by Scott Shenker et al

Figure 3.3 The Future of Networking and the Past of Protocols by Scott Shenker et al

Figure 3.14 A Service Curve of Hierarchical Token Bucket Queue Discipline on Soft-Ware Defined Networks Based on Deterministic Network Calculus: An Analysis and Simulation by Shuangyin Ren, Quanyou Feng, Yu Wang, and Wenhua Dou

APPENDICES

APPENDIX A

Code Listings

All the source code used in the implementation of the Bandwidth Utilization Scheme at the application plane of the SDN-based campus network are listed below

A.1 Code For Queue Creation

```
#!/usr/bin/python
                """
                Interfaces Detector
                """



                import sys
                # adding helper function
                sys.path.append('../SDN_Python')
                import helper
                import logger as d
                import db
                import socketio
                import telnetlib
                import requests
                import time
                # defaults
                sio = socketio.Client()
```

```
b5 = 0

b4 = 0

# default url

webUrl = "http://localhost:8000/"




try:

    sio.connect('http://127.0.0.1:5000')




except KeyboardInterrupt:

        sys.exit()




except:

        sys.exit()




def sendData(dat):

    d.warning(str(dat))

    res = requests.post(webUrl + 'obs/', data=dat)

    d.success(str(res.text))



    # add packets rx and tx

    res = requests.post(webUrl + 'packets/', data=dat)
    d.success(str(res.text))



# done connecting to websocket; connect to telnet
@sio.on('5005')
def op5(data):
    info = data['data']
    ip = info['ip']
    console = info['console']
    src = '192.168.80.9'
    dst = '192.168.100.8'
    interface = info['interface']
    tx = info['tx']
    rx = info['tx']


    tx = int(tx) + int(rx)
    minB = 0
    maxB = tx
    ip = '192.168.198.128'




    global b5
    # create queue
    if b5 != maxB:
        d.default('Creating queue: ' + str(ip) + ':' + str(console))
        queue = helper.createQ(interface, minB, maxB)
        tn = telnetlib.Telnet(ip, console)
        tn.write("ovs-vsctl -- --all destroy QoS -- --all destroy
Queue\n".encode('ascii'))
        tn.write((queue + "\n").encode('ascii'))
        d.success('Queue created...: ' + str(console))


        # push to odl
        helper.uploadFlow(src, dst)


        # sending data to web app
        webData = {'start': str(info['start']), 'stop': str(time.time()), 'dura-
tion': str(time.time() - info['start']), 'old': str(b5), 'new': str(maxB), 'tx':
str(tx), 'rx': str(tx)}
        sendData(webData)




        b5 = maxB
    else:
        d.default('Not setting bandwidth: no changes detected')



@sio.on('5014')
def op4(data):
    info = data['data']
    ip = info['ip']
    console = info['console']
    dst = '192.168.80.9'
    src = '192.168.100.8'
    interface = info['interface']
    tx = info['tx']
    rx = info['tx']


    tx = int(tx) + int(rx)
    minB = 0
    maxB = tx


    ip = '192.168.198.128'


    global b4
    if b4 != maxB:
        d.default('Creating queue: ' + str(ip) + ':' + str(console))
        # create queue
        queue = helper.createQ(interface, minB, maxB)
        tn = telnetlib.Telnet(ip, console)
        # delete existing queue
        tn.write("ovs-vsctl -- --all destroy QoS -- --all destroy
Queue\n".encode('ascii'))
        tn.write((queue + "\n").encode('ascii'))
        d.success('Queue created: ' + str(console))


        # push to odl
        helper.uploadFlow(src, dst)
    else:
        d.default('Not setting bandwidth: no changes detected')
```

A.2 Code For Connection To Opendaylight Controller

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

```
"""
IPC
"""


import socketio
import eventlet
import pprint
import sys
import logger as d
import db


### DEFAULTS #####
sio = socketio.Server()
app = socketio.WSGIApp(sio, static_files={
    '/': {'content_type': 'text/html', 'filename': 'index.html'}
})


## connection defaults ###
@sio.on('connect')
def connect(sid, environ):
    d.success('Client socket opened => ' + sid)
```

```
@sio.on('disconnect')
def disconnect(sid):
    d.error('Client socket closed => ' + sid)


##### Event Handlers
@sio.on('nodes')
def nodes(sid, data):
    d.success('Data received: ' + str(data))
    sio.emit('nodes', data)


@sio.on('5005')
def op5(sid, data):
    d.warning('Sending data to obs: 5005')
    sio.emit('5005', data)


@sio.on('5014')
def op4(sid, data):
    d.warning('Sending data to obs: 5014')
    sio.emit('5014', data)
```

```
try:
    eventlet.wsgi.server(eventlet.listen(('127.0.0.1', 5000)), app)
except KeyboardInterrupt:
    sys.exit()
```

```
#!/usr/bin/python
"""
Interfaces Detector
"""


import sys
import json
# adding helper function
sys.path.append('../SDN_Python')


import helper
import logger as d
import db
import socketio
import pprint
```

```
from time import sleep

import time



# defaults

sio = socketio.Client()




try:

   sio.connect('http://127.0.0.1:5000')



except KeyboardInterrupt:

      sys.exit()

except:

   print("No connection to server")

   sys.exit()



## getting all interfaces on openvswitches



def getNodes():
```

```
try:
    data = json.loads(helper.nodes())
    data = data['nodes']['node']

    switches = []

    for x in data:

        # checking through all nodes
        switchData = []
        for n in x['node-connector']:
            branch  =   n['opendaylight-port-statistics:flow-capable-node-
connector-statistics']

            if branch['packets']['received'] > 0:
                port = n['flow-node-inventory:port-number']
                link = n['flow-node-inventory:current-speed']
                interface = n['flow-node-inventory:name']
                tx = branch['bytes']['transmitted']
                rx = branch['bytes']['received']
                ip = x['flow-node-inventory:ip-address']
                if interface.find('eth3') != -1:
                    if  x['id']  ==  "openflow:38023701621572"  or  x['id']  ==
"openflow:60174091252288":
                        if x['id'] == "openflow:38023701621572":
```

```
                        console = 5014
                    else:
                        console = 5005
                    switchData.append({  'port':  port,  'interface':  interface,
'rx': rx, 'tx': tx, 'link': link , 'ip': ip , 'console': console, 'start': time.time()
})
            if len(switchData) > 0:
                switches.append({ 'id': x['id'], 'data': switchData })

        # send data to be analyzed
        d.default('Sending captured nodes to IPC')
        sio.emit('nodes', {'data': switches})


    except Exception as e:
        print(e)
        sio.disconnect()
        sys.exit()


if __name__ == "__main__":
    while True:
        getNodes()
        sleep(1)
```

A.3 Code For Realtime Analysis Of Bandwidth Utilization Application

```
#!/usr/bin/python
"""
Interfaces Detector
"""

import sys
from prettytable import PrettyTable
# adding helper function
sys.path.append('../SDN_Python')
import helper
import logger as d
import db
import socketio
import subprocess

# defaults
sio = socketio.Client()

def clearScreen():
    subprocess.run('clear')


############# Event Handlers ####################
```

```
@sio.on('nodes')
def nodes(data):
    clearScreen()
    #d.warning('Data received: ' + str(data))
    info = data['data']

    p = PrettyTable()
    p.field_names = ["Switches", "Interface", "Tx", "Rx", "Port Number", "IP Address",
"Console Port"]

    for x in info:
        p.add_row([str(x['id']), x['data'][0]['interface'], str(x['data'][0]['tx']), str(x['data'][0]['rx']),
str(x['data'][0]['port']), str(x['data'][0]['ip']), str(x['data'][0]['console'])])
        sio.emit(str(x['data'][0]['console']), { 'data': x['data'][0] })
        """
        if x['id'] == "openflow:60174091252288":
            p.add_row([str(x['id']),            x['data'][0]['interface'],            str(x['data'][0]['tx']),
str(x['data'][0]['rx']), str(x['data'][0]['port']), str(x['data'][0]['ip']), str(x['data'][0]['console'])])
        else:
            p.add_row([str(x['id']),            x['data'][3]['interface'],            str(x['data'][3]['tx']),
str(x['data'][3]['rx']), str(x['data'][3]['port']), str(x['data'][3]['ip']), str(x['data'][3]['console'])])

        """
    # display table
    print(p)
```

```
################ ... MAIN ... ###################################


if __name__  == "__main__":

   try:

      sio.connect('http://127.0.0.1:5000')


   except KeyboardInterrupt:

      sys.exit()


   except:

      sys.exit()





#!/usr/bin/python



                    """

                    Database module

                    """
```

```
import sqlite3

import os



prefix="dashboard_"



def init():

    conn = sqlite3.connect(os.path.dirname(os.path.realpath(__file__))
+'/../db.sqlite3')

    conn.row_factory = sqlite3.Row

    return conn



def getData(table):

    conn = init()

    cursor = conn.cursor()

    cursor.execute("select * from {0} order by id asc".format(prefix + ta-
ble))

    return cursor.fetchall()
```