DEVELOPMENT OF AN IMPROVED EXTENDED DIJKSTRA ALGORITHM FOR SOFTWARE DEFINED NETWORKS

BY

Abdul-Hafiz ABDULAZIZ

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

AHMADU BELLO UNIVERSITY ZARIA, NIGERIA

SEPTEMBER, 2017

DEVELOPMENT OF AN IMPROVED EXTENDED DIJKSTRA ALGORITHM FOR SOFTWARE DEFINED NETWORKS

BY

Abdul-Hafiz ABDULAZIZ, B.Eng (FUTY, 2011)

P13EGCM8042

hafizadaji@outlook.com

A DISSERTATION SUBMITTED TO THE SCHOOL OF POST GRADUATE STUDIES, AHMADU BELLO UNIVERSITY, ZARIA
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF MASTER OF SCIENCE (M.Sc) DEGREE IN COMPUTER ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
FACULTY OF ENGINEERING

AHMADU BELLO UNIVERSITY, ZARIA
NIGERIA

SEPTEMBER, 2017

DECLARATION

I, Abdul-hafiz ABDULAZIZ declare that this dissertation entitled "Development of an Improved Extended Dijkstra Algorithm for Software Defined Networks" was carried out by me in the Department of Electrical and Computer Engineering, Ahmadu Bello University, Zaria as part of the requirements for the award of degree of Master of Science in Computer Engineering. The information derived from literature has been duly acknowledged in the text and a list of references provided. No part of this dissertation was previously presented for another degree or diploma at this or any other institution.

Abdul-hafiz Abdulaziz _____ _____

(Student) (Signature) Date

CERTIFICATION

This Dissertation entitled **DEVELOPMENT OF AN IMPROVED EXTENDED DIJKSTRA ALGORITHM FOR SOFTWARE DEFINED NETWORKS** by ABDUL-HAFIZ Abdulaziz meets the regulations governing the award of degree of Master of Science (MSc) in Computer Engineering of the Ahmadu Bello University, and is approved for its contribution to knowledge and literary presentation.

_____	_____	_____
Chairman, Supervisory Committee (Dr. E.A Adedokun)	(Signature)	Date
_____	_____	_____
Member, Supervisory Committee (Dr. S-Man Yahya)	(Signature)	Date
_____	_____	_____
Head of Department (Dr. Y. Jibril)	(Signature)	Date
_____	_____	_____
Dean. School of Postgraduate Studies (Prof. S.Z. Abubakar)	(Signature)	Date

DEDICATION

This research work is dedicated to the Almighty God, my beloved Family and Friends.

ACKNOWLEDGMENT

My appreciation goes to my Parents Mr. Ibrahim A. Adaji and Mrs. Amina A. Abdulaziz and my siblings for their prayers and encouragement throughout my studies.

My appreciation also goes to the Chairman of my supervisory committee, Dr. E.A. Adedokun for his immense contribution throughout this research work. I sincerely thank you for the creative criticisms and motivation. I am proud to have you as my supervisor, you helped me to shape the research problem and provided valuable insights on the solution to the problem. Many thanks to my co-supervisor Dr. Sani Man-Yahya for his encouragement and guidance during the course of my research.

I sincerely wish to express my immense gratitude and appreciation to Prof. M.B. Muazu for his invaluable contributions at the Computer and Control Research Group. I'm glad and privileged to be part of the group.

I am grateful to all my lecturers and colleagues namely: Dr. AMS Tekanyi, Mr. Abdulrahman Olaniyan, Mr. Saluwudeen A. Tijanni, Mr. Bashir Sadiq, Mr. Ajayi Ore-Ofe, Mr. Sale Gelta, Mr. Abubakar Umar, Mr. Abdulfatai Adekale, Mr. Salawu Idris for their suggestions, and creative criticisms they gave to me.

I am highly indebted to my siblings Karima, Ta'ariq and Amal, your kindness and understanding personalities will always be treasured. Thank you very much.

Finally, to all those who, in one way or another, contributed to this work, but whose names do not appear. May God reward all of you abundantly (Amin).

Abdul-hafiz Abdulaziz
September, 2017

ABSTRACT

This research presents the development of an improved Extended Dijkstra Algorithm (mED-SDN) for Software Defined Networks (SDN) using Representational State Transfer (REST). The original Extended Dijkstra Algorithm for SDN (ED-SDN) cannot handle shortcomings associated with the traditional shortest path routing used in SDN. The traditional shortest path routing does not take knowledge of the topology into consideration and may result in sub-optimal performance of applications and underutilization of network. The ED-SDN returns the shortest path from the single source node to every other node with the consideration of the edge weights and node weights. However, ED-SDN lacks congestion control strategies and the mechanism for computing the edge and node weights incurs significant overhead. This research presents a framework on how a load balancer can be designed in a truly RESTful manner. mED-SDN algorithm was developed using the python programming language and implemented as a component of the controller. Three different network sizes were used to evaluate the performance of the proposed approach using latency and throughput as performance metric. The advantage of the improved approach was verified in two folds. First, this research showed that mED-SDN had a 24.3% and 12.5% improvement in terms of latency and throughput respectively when compared with ED-SDN and then demonstrated that mED-SDN gave better results in terms of latency and efficiency when compared with the existing round robin load balancing approach with a 30.6% throughput improvement.

TABLE OF CONTENT

TITLE PAGE ……. i

DECLARATION …….. iii

CERTIFICATION ……... iv

DEDICATION …….. v

ACKNOWLEDGEMENT ……..... vi

ABSTRACT …… vii

LIST OF FIGURES ……..... viii

LIST OF TABLES ……... ix

LIST OF ABBREVIATIONS ……..... x

TABLE OF CONTENTS ……....xi

CHAPTER ONE: INTRODUCTION

1.1 Background 1
1.2 Significance of Research 3
1.3 Problem Statement 3
1.4 Aim and Objectives 4
1.5 Methodology 4
1.6 Organization of Dissertation 6

CHAPTER TWO: LITERATURE REVIEW

2.2 Review of Fundamental Concepts 7
2.2.1 Legacy Networks 7
2.2.2 Software Defined Networking (SDN) 8
2.2.3 The OpenFlow Protocol – A type of SDN mechanism 10
2.2.4 Controller 13
2.2.5 OpenFlow Switch 15
2.2.6 Abilene Network Topology 21
2.2.7 iPerf (Internet Performance) 22
2.2.8 Application Programmble Interface (API) 26
2.2.9 Representational State Transfer API (REST API) 25

2.2.10 The Extended Dijkstra Algorithm for SDN (ED-SDN) 27
2.2.11 Load Balancing 29
2.3 Review of Similar Works 30

CHAPTER THREE: METHODS AND MATERIALS

3.1 Introduction 40
3.2 Installation and Configuration 40
3.3 Replication of the ED-SDN 41
3.3.1 Initializing the ED-SDN parameters 41
3.2.2 Computing node weight of ED-SDN 41
3.2.3 Computing edge weight of ED-SDN 42
3.2.4 Importing ED-SDN module into the POX 2.0 controller 43
3.2.5 Modified ED-SDN 48
3.2.7Accessing the controller 48
3.2.8 Making REST HTTP Calls 49
3.4 Performance Evaluation 55

CHAPTER FOUR: RESULTS AND DISCUSSION

4.1 Introduction 57
4.2 Simulation Scenario 57
4.3 Throughput test on Abilene Network Topology 58
4.4 Throughput test on a larger Abilene network scenario 60
4.5 Latency test on the Abilene network 61
4.6 Comparison of mED-SDN and Round Robin approach 62

CHAPTER FIVE: SUMMARY, CONCLUSION AND RECOMMENDATION

5.1 Introduction 64
5.2 Summary 64
5.3 Conclusion 65
5.4 Significant Contributions 66
5.5 Limitations 66
5.6 Recommendations for Further Work 66
REFERENCES 67

APPENDIX A

MININET SCRIPT FOR ABILENE NETWORK TOPOLOGY …….70

APPENDIX B

PYTHON SOURCE CODE FOR EXTENDED DIJKSTRA ALGORITHM……..…….72

APPENDIX C

PYTHON SOURCE CODE FOR MODIFIED EXTENDED DIJKSTRA ALGORITHM…….81

APPENDIX D

THROUGHPUT AND LATENCY TEST RESULTS…….85

LIST OF FIGURES

Figure 2.1	Traditional and Software-defined Architecture	8
Figure 2.2	The Architecture of SDN	9
Figure 2.3	Main Components of an OpenFlow Switch	15
Figure 2.4	Flowchart of Packet Flow through an OpenFlow Switch	20
Figure 2.5	The Abilene Network Core Topology	21
Figure 2.6	Bandwidth Measurement using iPerf	22
Figure 2.7	REST Client Performing CRUD Process	23
Figure 2.8	REST Command to Get all Devices	25
Figure 2.9	The Extended Dijkstra Algorithm	26
Figure 3.1	Mininet Snippet for Abilene Network Topology	40
Figure 3.2	MiniEdit Preferences	43
Figure 3.3	Selecting the Remote Controller	44
Figure 3.4	Emulated Abilene Topology	44
Figure 3.5	Authenticating to the Controller	45
Figure 3.6	Pushing a Flow into OpenFlow Switch	45
Figure 3.7	JSON Payload used to Update an OpenFlow Switch	46
Figure 3.8	Connecting OpenDaylight to host-only adapter	47
Figure 3.9	Login Dialog of OpenDaylight controller	48

Figure 3.10	Command to Launch Abilene Topology	48
Figure 4.1	Throughput Test on Abilene Network Topology	57
Figure 4.2	Throughput Test on Larger Network Topology	58
Figure 4.3	Latency Test Result on Abilene Network	59
Figure 4.6	mED-SDN and Round Robin Throughput Test Comparison	60
Figure 4.7	mED-SDN and Round Robin Latency Test Comparison	61

LIST OF TABLES

Table 2.1	A flow entry of header fields, counters, and actions	15
Table 2.2	Fields from packets used to match against flow entries	15
Table 2.3	Field lengths and the way they must be applied to flow entries	16
Table 2.4	Required list of counters for use in statistics messages	17
Table 3.1	Assigned static IP addresses for network set up	39
Table 3.2	Simulation Parameters and Settings	41
Table 4.1	Simulation Parameters	55
Table 4.2	Simulation Parameters for larger Abilene network	56

LIST OF ABBREVIATIONS

Symbol	Description
API	Application Programmable Interface
CLI	Command Line Interface
DA	Dijkstra Algorithm
DLB	Dynamic Load Balancing
DPI	Deep Packet Inspection
ECMP	Equal Cost Multi-Path
HTTP	Hypertext Transfer Protocol
IGP	Interior Gateway Protocol
IP	Internet Protocol
IS-IS	Intermediate System
MAC	Media Access Control
MPLS	Multi-Label Packet Switching
ODL	OpenDaylight
OF	OpenFlow
OSGi	Open Services Gateway Interface
OSPF	Open Shortest Path First
REST	Representational State Transfer
SAL	Service Abstraction Layer

SDN	Software Defined Network
SIP	Session Initiation Protocol
SLB	Static Load Balancing
SPA	Shortest Path Algorithm
SPF	Shortest Path First
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	Universal Datagram Packet
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
MTU	Maximum Transmission Unit
VLAN	Virtual Local Area Network

CHAPTER ONE

INTRODUCTION

1.1 Background to the study

Software Defined Networking is a new paradigm that breaks the vertical integration between the network control plane and its data plane. OpenFlow switches implement data transmission function, so as to simplify the design of switches, and control functions are provided by controllers. SDN architecture is composed of three layers (Mckeown *et al.,* 2008), which are referred as data layer, control layer and application layer.

As the basic network device in data layer, OpenFlow switches implement data transmission function according to flow tables allocated from controller (Mckeown *et al.,* 2008). Being as the "brain" of SDN, the controller acquires application information from upper layer through the unified northbound interface. Flow tables are generated in controller and allocated to OpenFlow switches through OpenFlow protocol. By acquiring network topology information, SDN controller provides the global network view for OpenFlow switches and implements the flexible network configuration and network management (Cui *et al.,* 2016).

SDN has gained a lot of attention in recent years, because it addresses the lack of programmability in existing networking architectures and faster network innovation (Wolfgang and Michael 2014). The emergence of the SDN technology brings many new network applications realized by programming the SDN controller (Widhi *et al.,* 2015). Typical examples include load balancing, multimedia multicast and intrusion detection systems (IDS). Load balancing is an important concept in networking. The purpose of the load balancing application

is to distribute the loads into multiple servers in order to get the best performance (Hui *et al.,* 2013).

In recent years, various load balancing methods for Data Center Networks (DCNs) using the SDN paradigm have been introduced. A load balancing algorithm called LABERIO (LoAd-Balancing Routing wIth OpenFlow), to minimize latency and maximize the network throughput was proposed by Hui *et al.,* (2013). A Plug-n-Serve system implementing a load balancing algorithm called LOBUS (Load-Balancing over Unstructured networks), using OpenFlow for unstructured networks was proposed by Handigol *et al.,* (2013). LOBUS maintains the network topology and link status, and greedily chooses the client-server pair that yields the lowest total response time for each newly arriving request. Dijkstra algorithm, the classical shortest path algorithm used by many routing protocols for finding the shortest path two points in the networks was extended by Jiang *et al.,* (2014). The extended Dijkstra algorithm can be applied to derive a pair of shortest path in an SDN topology. By taking advantage of the extended Dijkstra shortest path algorithm for SDN (ED-SDN) conceptualized by Jiang *et al.,* (2014), a load balancer for SDN was proposed by Widhi *et al.,* (2015).

There are many approaches to designing a load balancer using SDN technology. However, most load balancing and routing algorithms in the context of SDN allocate resources based on statically configured routes and therefore may experience uneven load balancing. Today's controllers utilize the REST Application Programmable Interface (API) technology, an effective mechanism to communicate with various components in a network. By leveraging the REST API technology, this research aims to improve the work of Jiang *et al.,* (2014) by developing an efficient load balancer for OpenFlow based DCNs.

1.2 Significance of Research

The significance of this research is the development of an efficient load balance algorithm in the context of SDNs by modifying the existing extended Dijkstra algorithm for SDN (ED-SDN) to leverage the RESTful API of the controller for retrieving topology information and congestion control strategy for allocating routes to packets.

1.3 Problem Statement

The extended Dijkstra shortest path algorithm for SDN (ED-SDN) was an approach to address the shortcomings associated with shortest path algorithms used in controllers today for allocating resources and path finding. The traditional shortest path routing does not take knowledge of the topology into consideration and may result in sub-optimal performance of applications and underutilization of network. However, ED-SDN lacks congestion control strategies and the mechanism for computing the edge and node weights incurs significant overhead. This research is aimed at improving ED-SDN by leveraging the REST API of the controller and introduces a congestion control mechanism that handles network overhead.

1.4 Aim and Objectives

The aim of this research work is to develop a improved extended Dijkstra shortest path algorithm for SDN with a view of minimizing the latency and maximizing the network throughput.

The objectives of the research are as follows:

1. Replication and implementation of ED-SDN for Software Defined Networks.
2. Adoption and implementation of Abilene network topology.
3. Comparison of the performance of ED-SDN and mED-SDN for throughput and latency using iPerf network utility testing tool.

1.5 Methodology

The methodology adopted for this research are as follows:

1. Replication of the existing extended Dijkstra algorithm (Jiang *et al.,* 2014) for SDN. To develop the ED-SDN algorithm, the following steps were taken:
    a) Import the Dijkstra Shortest Path First (SPF) module from the Python base class library
    b) Create member python functions responsible for computing node weight ($nw[v]$) and edge weight ($ew[e]$)
    c) Import the ED-SDN module into the POX controller
2. Development of the modified Extended Dijkstra algorithm for SDN. To realize this, the following steps were carried out:
    a) Repeat step 1a.
    b) Perform HTTP requests to the controller to retrieve the topology information of

the SDN topology using REST API of the controller

c) Create a member python function on the controller responsible for congestion control on the SDN topology

d) Import the modified SDN module into the OpenDaylight controller

3. Development of the Abilene SDN topology

    a) Using the Mininet API to create the Abilene network topology

    b) Instantiate the network comprising of the controller and OpenFlow switches

4. Comparison of the performance of the developed approach with the existing Extended Dijkstra algorithm for SDN and the Round Robin approach using the SDN topology in 3.

1.6 Organization of Dissertation

The general introduction of this research has been presented in Chapter one. Chapter two presents preliminaries of the research, i.e., Legacy networks, Software Defined Networking, OpenFlow switches, REST API, and a comprehensive review of similar works carried out. Chapter three, covers the procedure to actualizing the objective of this research which comprises of the materials and methodology adopted. Chapter four presents and discusses the simulation settings and results obtained. Finally, Chapter five covers the conclusion and recommendation of areas for further research.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

The literature review comprises of an overview of fundamental concepts and the review of similar works. In the review of fundamental concepts, most of the pertinent works and the fundamental theories that have been used for the successful implementation of this research are reviewed.

2.2 Review of Fundamental Concepts

This chapter begins with a brief introduction to legacy network architecture, Software Defined Networking and a detailed study of the OpenFlow technology.

2.2.1 Legacy Networks

Ethernet switch is one of the basic network elements used in Local Area Networks (LANs). It uses hardware addresses to forward data packets at the data link layer of the (Open Systems Interconnection) OSI model (Idris *et al.,* 2016). The switch operates at data link layer of the OSI model to create a separate collision domain of every switch interface such that the interfaces can transmit and receive the data simultaneously.

The switch forwards data frames based on Media Access Control (MAC) table. When a frame arrives at a switch, the switch will put the source MAC address and correspond port number in the MAC table as the basis for forwarding new frames. Then the destination MAC address will be inspected. If the destination MAC address is multicast address it will forward the frame to all the interfaces except the incoming interface. Otherwise, the frame will be forwarded to specific interface according to the MAC table (Idris *et al.,* 2016).

When the switch floods a frame to the network, it may create the traffic loop in the network whose topology consists of loops. To solve this, legacy switches usually uses a spanning tree protocol that blocks some interfaces so that the resulting logical LAN topology is a tree. Through the spanning tree protocol, traffic loops can be prevented.

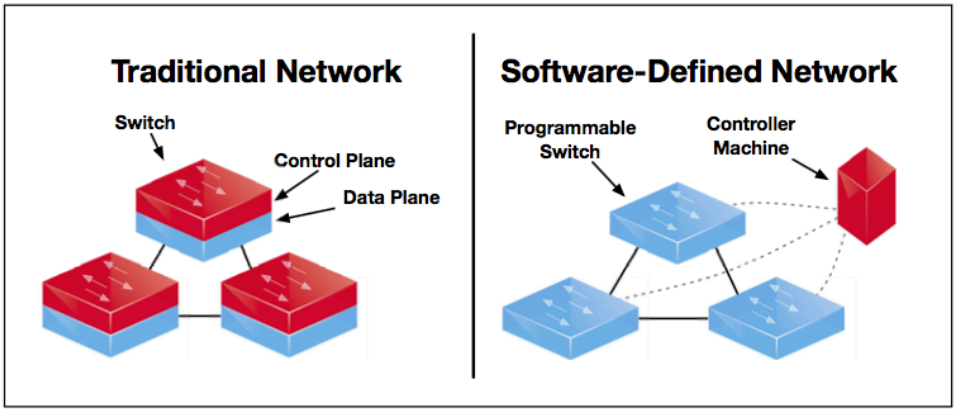


Figure 2.1: Traditional and software-defined architectures (Martin *et al*., 2014)

2.2.2 Software Defined Networking (SDN)

SDN is an approach to computer networking which evolved from work done at UC Berkeley and Stanford University around 2008 (Mckeown *et al.,* 2008). It is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the system that makes decisions about where traffic is sent (control plane) from the underlying systems that forwards traffic to the selected destination (data plane), thus enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for application and network services (Mckeown *et al.,* 2008). The OpenFlow protocol is a foundational element for building SDN solutions (Martin *et al.,* 2014). A typical SDN configuration can be thought of as shown in Figure 2.2.

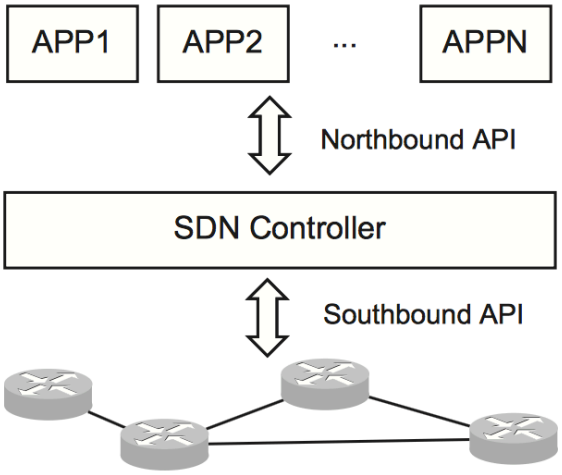


Figure 2.2: The Architecture of Software-Defined Network (Wei *et al.,* 2014).

In SDN, the data plane and the control plane are typically connected by a closed control loop:

1. The control plane receives network events from the data plane
2. The control plane (the SDN controller and applications) computes some network operations based on the events for the data plane; and
3. The data plane executes the operations which can change the network states, e.g. data path. The role of SDN northbound API is to provide a high-level API between the controller and the applications to facilitate step 2 in the control loop (Wei *et al.,* 2014).

2.2.3 The OpenFlow Protocol Overview– A type of SDN mechanism

The OpenFlow specification defines a standard collection of features that switches must provide, as well as an interface that controllers can use to communicate with switches including instructions for installing and deleting forwarding rules, and notifications about flows, topology, and traffic statistics (Mckeown *et al.,* 2008).

The OpenFlow protocol is the most commonly used protocol for the southbound interface of SDN, which separates the data plane from the control plane. The white paper about OpenFlow (ONF, 2013) points out the advantages of a flexibly configurable forwarding plane. OpenFlow was initially proposed by Stanford University, and it is now standardized by the Open Networking Foundation (ONF, 2013).

The OpenFlow protocol supports three message types, *controller-to-switch*, *asynchronous*, and *symmetric*. Controller-to-switch messages are initiated by the controller and used by the switch to update the controller of network events and changes to the switch state. Symmetric messages are initiated by the switch or the controller and sent without solicitation (Brandon *et al.,* 2008). The message types used by OpenFlow are described below.

2.2.3.1 Controller-to-switch

Controller-to-switch messages are initiated by the controller and may or may not require response from the switch. These include:

1. **Features:** Upon SSL session establishment, the controller sends a features request message to the switch. The switch must reply with a features reply that specifies the capabilities supported by the switch.
2. **Configuration:** The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.
3. **Modify-State:** Modify-State messages are sent by the controller to manage state of the switches. Their primary purpose is to add/delete and modify flows in the flow tables and to set switch port properties.

4. **Read-State:** Read-State messages are used by the controller to collect statistics from the switches flow-tables, ports and the individual flow entries.
5. **Send-Packet:** These are used by the controller to send packets out of a specified port on the switch.

2.2.3.2 Asynchronous

Asynchronous messages are sent without solicitation from the switch to the controller and denote a change in switch or network state. The four (4) main asynchronous messages are: Packet-in, Flow Expiration, Port Status and Error (Brandon *et al.,* 2008).

2.2.3.3 Symmetric

Symmetric messages are sent without solicitation, in either direction. These include:

1. **Hello:** Hello messages are exchanged between the switch and controller upon connection startup.
2. **Echo:** Echo request/reply messages can be sent from either the switch or the controller, and must return an echo reply. They can be used to indicate the latency, bandwidth, and/or liveness of a controller-switch connection.
3. **Vendor:** Vendor messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions.

2.2.3.1 How Openflow works

In classical router or switch, the fast packet forwarding (data path) and the high level routing decisions (control path) occur on the same device. An OpenFlow switch separates these two

functions. The data path portion still resides on the switch, while the high-level routing decisions are moved to a separate controller, typically a standard server. The OpenFlow switch and the controller communicate via the OpenFlow protocol, which defines the messages such as *packet-received*, *send-packet-out*, *modify-forwarding-table*, and *get-stats* (Mckeown *et al.,* 2008).

The data path of an OpenFlow switch presents a clean flow table abstraction, each flow table entry contains a set of packet fields to match, and an action such as send-out-port, modify-field, or drop. When an OpenFlow switch receives a packet it has never seen before, for which it has no matching flow entry, it sends this packet to the controller. The controller then makes a decision on how to handle this packet. It can drop the packet, or it can add a flow entry directing the switch on how to forward similar packets in the future (Astuto *et al.,* 2014).

2.2.4 Controller

The controller is the basic element in the control layer. The control layer can consist of one or several physical controllers. The controller or network operating system is the heart of the SDN, which is responsible for controlling and managing all the OF switches (Manar *et al.,* 2014). Nowadays most controllers provide Representational State Transfer (REST) API to make the controller be accessible readily by the application layer. REST APIs use the HyperText Transfer

Protocol (HTTP) to execute common operational on resources represented by Uniform Resource Identifier (URI) strings.

Some of the SDN controllers used widely in academia and industry (Doan and Minh, 2014) are summarized as follows:

1. OpenDaylight Controller

   OpenDaylight is an open source project with a modular, pluggable, and flexible controller platform at its core. This OpenFlow controller is implemented strictly in software and is contained within its own Java Virtual Machine (JVM). As such, it can be deployed on any hardware and operating system platform that supports Java (Gunjan and Adithi, 2013).

   The controller platform itself contains a collection of dynamically pluggable modules to perform needed network tasks. There are a series of base network services for tasks such as understanding what devices are contained within the network and the capabilities of each, statistics gathering, etc. In addition, platform oriented services and other extensions can also be inserted into the controller platform for enhanced SDN functionality. The southbound interface is capable of supporting multiple protocols (as separate plugins), e.g. OpenFlow 1.0, OpenFlow 1.3, BGP-LS, etc. These modules are dynamically linked into a Service Abstraction Layer (SAL). The SAL exposes device services to which the modules north of it are written. The SAL determines how to fulfill the requested service irrespective of the underlying protocol used between the controller and the network devi§ces (Gunjan and Adithi, 2013). OpenDaylight has good documentation on its API and also provides a web-based Graphical User Interface (GUI). So, using this controller

greatly helps in making our proposed load balancer easier to configure and flexible enough (Gunjan and Adithi, 2013).

2. Floodlight Controller

   Floodlight Controller, an Apache licensed, Java-based OpenFlow controller, is one of the significant contributions from Big Switch Networks to the open source community. When floodlight is run, both northbound and southbound operations of the controller become active. Floodlight, currently supports OpenFlow 1.0. Versions of 1.3 and 1.4 are in the pipeline. With an extensible Java development environment, and enterprise-grade core engine, Floodlight is both an easy to use and robust SDN controller.

3. Network Operating System (NOX)

   NOX is developed by Nicira Networks and is among first SDN controllers used in DCNs. NOX provides a C++ API to OpenFlow and an asynchronous, event-based model. NOX is both a primordial controller and a component-based framework for developing SDN applications.

4. POX (Python based controller)

   POX is the successor to NOX, a newer Python-based version of NOX, POX has a high-level SDN API including a queryable topology graph and support for virtualization. This controller comes pre-packaged with the Mininet network emulator.

2.2.5 Switch Components

An OpenFlow switch consists of a flow table, which performs packet lookup and forwarding, and secure channel to an external controller as depicted in Figure 2.3. The controller manages the switch over the secure channel using the OpenFlow protocol (Brandon *et al.,* 2008).

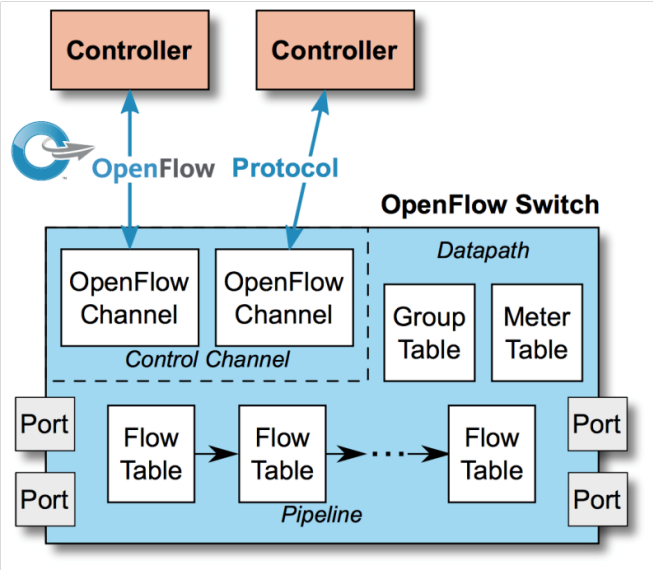


Figure 2.3: Components of an OpenFlow Switch (Tim, 2011)

The flow table contains a set of flow entries (header values to match packets against), activity counters, and a set of zero or more actions to apply to matching packets. All packets processed by the switch are compared against the flow table. If a matching entry is found, any actions for that entry are performed on the packet (e.g., the action might be to forward a packet out a specified port). If no match is found, the packet is forwarded to the controller over the secure channel. The controller is responsible for determining how to handle packets without valid flow entries, and it manages the switch flow table by adding and removing flow entries (Brandon *et al.,* 2008).

2.2.5.1 Flow Table

This section describes the components of flow table entries and the process by which incoming packets are matched against flow table entries.

Each flow table entry (see Table 2.1) contains:

a. **header fields** to match against packets
b. **counters** to update for matching packet
c. **actions** to apply to matching packets

Table 2.1: A Flow Entry of Header Fields, Counters, and Actions (Brandon *et al.,* 2008)

Header Fields	Counters	Actions

1. **Header Fields**

   Table 2.2 shows the header fields an incoming packet is compared against.

Table 2.2. Fields from packets used to match against Flow Entries (Brandon *et al.,* 2008)

Ingress Port	Ether source	Ether dst	Ether type	VLAN id	IP src	IP dst	IP proto	TCP/ UDP src port	TCP/ UDP dst port

Table 2.3 illustrates the fields of a typical flow entry, their corresponding bit sizes and where they are applied.

Table 2.3. Field Lengths and Flow Entries (Brandon *et al*., 2008)

Field	Bits	When applicable	Notes
Ingress Port	(Implementation dependent)	All packets	Numerical representation of incoming port.
Ethernet source address	48	All packets on enabled ports	
Ethernet destination address	48	All packets on enabled ports	
Ethernet type	16	All packets on enabled ports	A Type 0 switch is required to match the type in both standard Ethernet and 802.2 with a SNAP header and OUI of 0x000000. The special value of 0x05FF is used to match all 802.3 packets without SNAP headers.
VLAN id	12	All packets of Ethernet type 0x8100	
IP source address	32	All IP packets	Can be subnet masked
IP destination address	32	All IP packets	Can be subnet masked
IP protocol	8	All IP packets	
Transport source port / ICMP Type	16	All TCP, UDP, and ICMP packets	Only lower 8 bits used for ICMP Type
Transport destination port / ICMP Code	16	All TCP, UDP, and ICMP packets	Only lower 8 bits used for ICMP Code

2. Counters

Counters are maintained per-table, per-flow, and per-port. Table 2.4 contains the required set of counters for OpenFlow switches. Duration refers to the number of seconds a flow

has been active. The Receive Errors field includes all explicitly specified errors, including frame, overrun, and Cyclic Redundancy Check (CRC) errors, plus any others (Brandon *et al.,* 2008).

Table 2.4. Required List of Counters for use in Statistics Messages (Brandon *et al*., 2008).

Counter	Bits
Per Table	
Active Entries	32
Packet Lookups	64
Packet Matches	64
Per Flow	
Received Packets	64
Received Bytes	64
Duration	32
Per Port	
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64

3. Actions

Each flow entry is associated with zero or more actions that dictates how the switch handles matching packets (Mckeown *et al.,* 2008). Actions need not be executed in the order in which they are specified in the flow entry. If no actions are present, the packet is dropped. OpenFlow-complaint switches come in two types: OpenFlow-only, and OpenFlow-enabled (Brandon *et al.,* 2008).

OpenFlow-only switches support only the required actions below, while OpenFlow-enabled hggswitches, router, and access points may also support the NORMAL action. Either type of switch can also support the FLOOD action.

Required Action:

a. ALL: Send the packet out all interfaces, not including the incoming interface.

b. CONTROLLER: Encapsulate and send the packet to the controller.

c. LOCAL: Send the packet to the switches local networking stack.

d. TABLE: Perform actions in flow table. Only for packet-out messages.

e. IN_PORT: Send the packet out of the input port.

Optional Action: The switch may optionally support the following virtual ports:

a. NORMAL: Process the packet using the traditional forwarding path supported by the switch (i.e., traditional Layer 2, VLAN (Virtual LAN), and Layer 3 processing).

b. FLOOD: Flood the packet along the minimum spanning tree, not including the incoming interface.

2.2.5.2 Secure Channel

The secure channel is the interface that connects each OpenFlow switch to a controller. Through this interface the controller configures and manages the switch, receives events from the switch, and sends packets out of the switch.

The workflow of an OpenFlow switch processing a packet is presented in Figure 2.4.

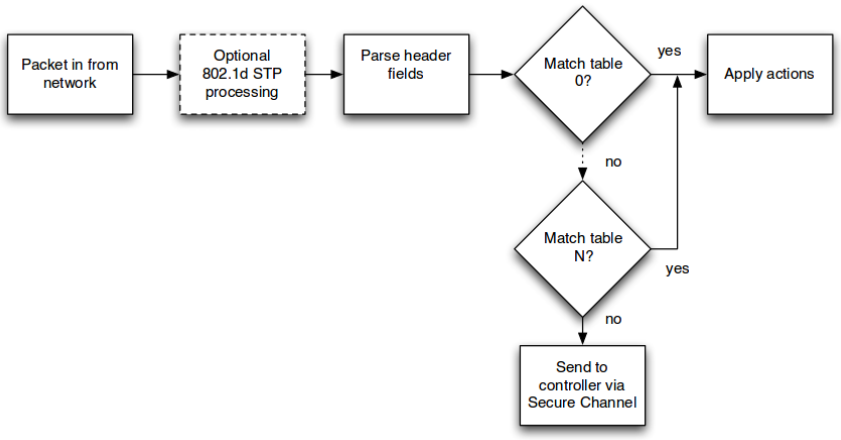


Figure 2.4: Flowchart of Packet Flow through an OpenFlow Switch (Tim, 2011)

2.2.6 Abilene Network Topology

This research uses Mininet and adopts the topology of the Abilene network to perform simulation. The Abilene network is a high-performance backbone network suggested by the Internet2 project. Figure 2.5 shows the historical Abilene (network) core topology, connecting 11 regional sites or nodes across the United States. The Abilene network has 10 Gbps connectivity between nodes and 100 Mbps connectivity between a host and a node.

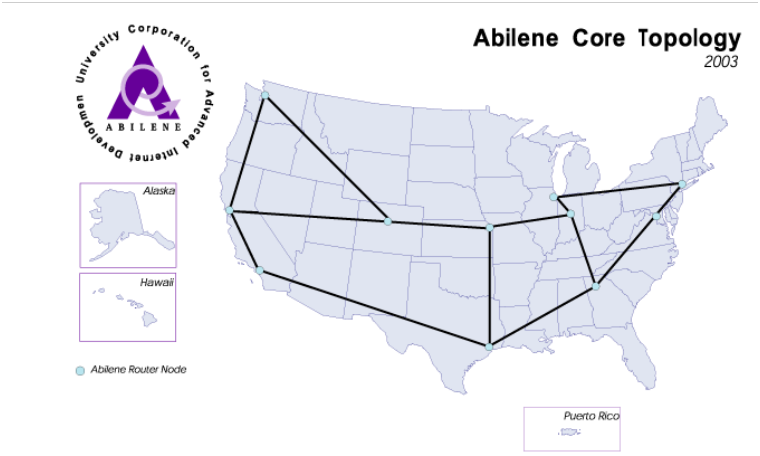


Figure 2.5: The Abilene Network Core Topology (Aron and Assane, 2015)

2.2.7 iPerf (Internet Performance)

iPerf is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP with IPv4 and IPv6). For each test it reports the bandwidth, loss, and other parameters (Diyar *et al.,* 2014).

iPerf has a client and server functionality, and can measure the throughput between the two ends, either unidirectional or bidirectional. It is open source and runs on various platforms including Linux, UNIX and Windows (Diyar *et al.,* 2014).

2.2.7.1 IPerf features

- TCP (Transmission Control Protocol)
    - Measure bandwidth
    - Report Maximum Transmission Unit (MTU) size and observed read sizes.
    - Support for TCP window size via socket buffers.
- UDP (User Datagram Packet)
    - Client can create UDP streams of specified bandwidth.
    - Measure packet loss

- Measure delay jitter

- Cross-platform: Windows, Linux, Android, Macintosh Open System and other Linux distributions
- Client and server can have multiple simultaneous connections (-P option).
- Server handles multiple connections, rather than quitting after a single test.
- Print periodic, intermediate bandwidth, jitter, and loss reports at specified intervals (-i option).

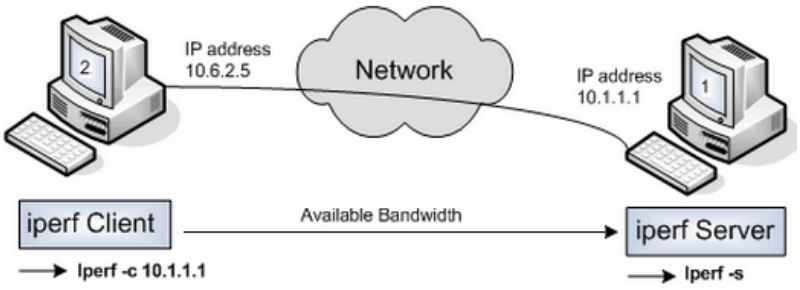


Figure 2.6: Bandwidth Measurement by iPerf (Gunjan and Adithi, 2013).

2.2.8 Application Programmable Interface (API)

Simply put, an API is an interface presented by software (such as a network operating system) that provides the capability to collect information from or make a change to an underlying set of resources (Michael, 2011).

2.2.8.1 REST APIs in the context of SDN

In an open SDN model, a common interface discussed is the northbound interface (NBI). The NBI is the interface between software applications, such as operational support systems, and a centralized SDN controller (Wei *et al.,* 2014).

One of the common API technologies used at the northbound interface is the Representational State Transfer (REST) API. REST APIs use the Hypertext Transfer Protocol (HTTP/HTTPS)

protocol to execute common operations on resources represented by Uniform Resource Identifier (URI) strings (Wei *et al.,* 2014). An application may use REST APIs to send an HTTP/HTTPS GET message via an SDN controller's IP address. That message would contain a URI string referencing the relevant network device and comprising an HTTP payload with a JSON (JavaScript Object Notation), header that has the proper parameters for a particular interface and statistic.

2.2.9 Representational State Transfer API (REST API)

This section provides an overview of the REST API and how it works. REST is an API that allows clients to perform read/write operations on data stored on the server. REST uses HTTP to perform a set of actions commonly known as : Create, Read Update, Delete (CRUD). Most commonly the payload will get encoded as JSON, however the structure of JSON is not defined and is different for each application (Wei *et al.,* 2014).

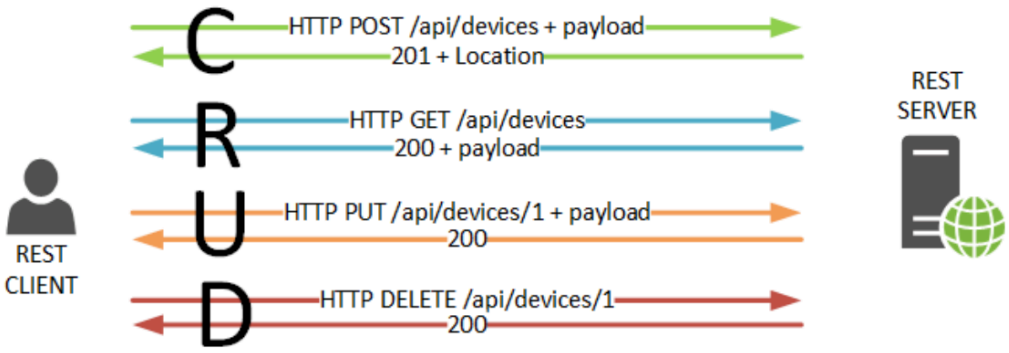


Figure 2.7: REST Client Performing CRUD Process on a REST Server (Michael, 2011)

For API to be considered RESTful it needs to comply with a number of formal constraints (Umma *et al*., 2015). One of the most important constraint of all is the stateless nature of the interface. That means that no cookie or session information should be stored on the server, and every request is treated independently from every other request. The implication here is that all

state needs to be maintained on the client and information needs to be cacheable in order to improve interface responsiveness (Michael, 2011).

REST is an architecture style for designing networked applications. As REST architectural style has gained more popularity in implementing loosely-coupled systems, RESTful services are becoming the style of choice for northbound API and gaining increasing importance in SDN architecture. Adopting REST for the SDN northbound API within this control architecture has the following benefits (Michael, 2011):

1. Decentralized management of dynamic resources: REST does not use any centralized resource registry but relies on connections between resources to discover and manage them as a whole. REST allows network elements, such as routers and switches to be dynamically deployed and changed in a distributed fashion.
2. Heterogeneous clients: because REST separates resource representations, identification, and interaction, it can adjust resource representations and network protocols based on SDN client capabilities and network conditions to optimize API performance.
3. Scalability: REST achieves server scalability by keeping the server stateless and improves server performance through layered caches. This feature will become useful, when an SDN controller needs to support a large number of concurrent host-based applications and to use network resources in an efficient way.

2.2.9.1 Using REST

cURL is a light-weight Linux command-line tool for transferring data. It's very simple to get started and very frequently used to make HTTP requests. For example, the Figure 2.8 below

illustrates how you would issue a Read request to get a list of devices configured on a hypothetical SDN controller:

```
$ curl -X GET http://sdn-controller.org:8181/api/devices
```

Figure 2.8: REST command to get all devices using cURL (Michael, 2011)

2.2.9.2 Writing an application using RESTful API

Restful APIs can be provided by a plugin integrated into the controller platform as part of their northbound interfaces. These plugins provide various core network services. The following are some of the RESTful APIs exposed (Wei *et al.,* 2016):

1. Topology: Contains RESTful APIs to access the network graph represented by edges and nodes and their properties. A predefined XML schema defines the data elements for this schema.
2. Flow programmer: APIs exposed for configuring flows on the network elements, controlled by their respective protocol plugins on the southbound side.
3. Simple APIs for creating IP L3 static routing entries on the networking fabric.
4. Statistics: APIs providing the statistics of flow entries, and the node elements.
5. Subnets: APIs for configuring network nodes and the hosts attached to them into predefined IP Subnets.
6. Switch Manager: APIs which focus on exposing various nodes in underlying network as a switch profile, listing their ports and properties.

Using the various APIs listed above, and others available through the OpenDaylight documentation, northbound applications can be created over the OpenDaylight platform.

Though almost all the APIs provide the RESTful XML Schema for data interaction, some of the APIs may as well support the JSON structure (Wei *et al.,* 2016).

2.2.10 The Extended Dijkstra Algorithm for SDN (ED-SDN)

Given a weighted, directed graph G = (V, E) and a single source node *s*, the classical Dijkstra's algorithm can return a shortest path from the source node s to every other node, where *V* is the set of nodes and *E* is the set of edges, each of which is associated with no weight. In the original Dijkstra algorithm nodes are associated with no weight. The extension of Dijkstra algorithm by Jiang *et al.,* (2014) is best illustrated with the Figure 2.9 below. This shows the extended Dijkstra algorithm, whose input is a given graph G = (V, E), the edge weight setting (ew), the node weight setting (nw), and the single source node s. The extended Dijkstra algorithm uses d[u] to store the distance of the current shortest path, from the shortest node s to the destination node u, and uses p[u] to store the previous node preceding u on the current shortest path. Initially, d[s]=0, d[u]= ∞ for u $\in V$, u$\neq$s, p[u]=null for u $\in V$.

Extended Dijkstra's Algorithm
Input: G=(V, E), ew, nw, s **Output:** d[|V|], p[|V|]
1: $d[s]\leftarrow 0$; $d[u]\leftarrow\infty$, for each $u\neq s$, $u\in V$
2: **insert** u with key $d[u]$ into the priority queue Q, for each $u\in V$
3: **while** ($Q\neq\varnothing$)
4: $u\leftarrow$Extract-Min(Q)
5: **for** each v adjacent to u
6: **if** $d[v] > d[u]+ew[u,v]+nw[u]$ **then**
7: $d[v]\leftarrow d[u]+ew[u,v]+nw[u]$
8: $p[v]\leftarrow d[u]$

Figure 2.9: The Extended Dijkstra Algorithm (Jiang *et al.,* 2014)

The extended algorithm returns the shortest path from the single source node to every other node with the consideration of the edge weights and node weights. Assume from the SDN topology, a graph G= (V, E) can be derived, which is weighted, directed and connected. For a node v ∈ V and an edge e ∈ E, let Flow (v) and Flow (e) denote the set of all the flows passing through v and e, respectively, let Capacity (v) be the capacity of v (i.e., the number of bits that v can process per second), and let Bandwidth (e) be the bandwidth of e (i.e., the number of bits that e can transmit per second). The node weight $nw[v]$ of v is defined according to Equation (2.1), and the edge weight $ew[e]$ of e is defined according to Equation (2.2).

$$nw[v] = \frac{\sum_{f \in Flow(v)} Bits(f)}{Capability\ (v)} \tag{2.1}$$

where $Bits(f)$ stands for the number of $f's$ bits passing through node v per second.

$$ew[e] = \frac{\sum_{f \in Flow(e)} Bits(f)}{Bandwidth\ (e)} \tag{2.2}$$

where $Bits(f)$ stands for the number of $f's$ bits passing through an edge e per second.

The number of flow bits processed by a node or passing through an edge is obtained with the "counters field" of the OpenFlow switches' flow tables. The numerators are of units "bits" and the denominators are of unit "bit per second". Therefore, the node weight and the edge weight are of unit "seconds". When the node weight and the edge weights along a path are accumulated. The end-to-end latency from one end to the other is obtained.

2.2.11 Load Balancing

Load balancing is the process of improving the performance of a parallel and distributed system through a redistribution of load among the processors (Senthil and Rajani, 2015). The purpose of load balancing is to ensure that every processor in the system does approximately the same amount of work at any point of time. Processes may migrate from one node to another even in the middle of execution to ensure equal workload (Sandeep *et al.,* 2008). Load balancing algorithms are broadly classified into:

a. Static Load Balance Algorithm

In this method, the performance of the processor is determined at the beginning of execution. A general disadvantage of all static schemes is that the final selection of a host for process allocation is made when the process is created and cannot be changed during the process execution to make changes in the system load. The most common static load balancing algorithm is the Round Robin algorithm.

b. Dynamic Load Balancing

Unlike static algorithms, dynamic load balancers allocate processes dynamically when one of the processors becomes under loaded, they are buffered in the queue on the main host and allocated dynamically upon request from remote hosts (Payal *et al.,* 2014).

2.3 Review of Similar Works

This section presents existing literature of load balancing and routing in the context of SDN.

Handigol *et al.*, (2013) showcased *Plug-n-Serve*, an OpenFlow based server load-balancing system that effectively reduces response time of web services in unstructured networks. Using OpenFlow to keep track of state to control the routes allows the system to be easily re-configured. This approach implemented a load-balancing solution for un-structured networks that takes into account both the congestion of the network and the load on the servers. The *Plug-n-Serve* mechanism tries to minimize response time by controlling the load on the network and the servers using customized flow routing. However, the controller (NOX) used in the research, is now obsolete and antiquated. It also had significant performance and security issues.

Yu and Deng (2013) introduced an OpenFlow based dynamic load balancer for fat-tree networks with multi-path support in order to achieve high performance and low latency. They implemented a dynamic load balancing algorithm in the load balancer. The task of the algorithm is to distribute traffic of upcoming network flows and make each alternative path receive equal amounts of traffic load. The load balancer works as a module of the OpenFlow controller program called Beacon, which runs on the controller host. The load balancer has two important functionalities: monitoring port statistics and scheduling new flows. To evaluate the scalability of the algorithm, they utilized the Mininet network emulator to model a data center network (hierarchical k=8 fat-tree topology) with 80 switches and 128 hosts and used two other load balancing algorithms as comparisons. However, selecting the maximum available bandwidth for every hop may not be the optimal path for delay sensitive applications.

Hui *et al.*, (2013) considered the fact that existing load balancing approaches based on OpenFlow are static, which means they try to find a static routing path during the initialization step. Static routing path suffers from poor performance since the network configuration may change during data transmission. To address this problem, they leveraged a novel integrated novel optimization algorithm, called LABERIO (Load-BalancEd Routing with OpenFlow). It is designed to improve the overall file transmission performance, to minimize latency and response time as well as to maximize the network throughput by better utilizing the available resources. However, this approach used the OpenFlow 1.0 protocol for communication between the controller and the switches. OpenFlow 1.0 has been reported to experience performance challenges and were not compatible with most SDN controllers.

Jiang *et al.*, (2014) considered the fact that the existing Dijkstra shortest path algorithm is not scalable in routing and load balancing in the context of OpenFlow networks. On this basis, they extended the well-known Dijkstra's shortest path algorithm to consider not only the edge weights but also the node weights for a graph derived from the underlying SDN topology. In other words, the extended Dijkstra's algorithm returns the shortest path from the single source node to every other node considering the edge weights $ew[e]$ and node weights $nw[v]$. The extended Dijkstra algorithm is very useful in deriving the best routing path to send a packet from a specific source node to another node (i.e., the destination node) for the SDN environment in which significant latency occurs when the packet goes through intermediate nodes and edges (or links). $nw[v]$ of v is defined according to time in seconds it takes to process a packet , $ew[e]$ is the time it takes for a packet to transverse a link. They leveraged the "counters field" of the OpenFlow switches in deriving the number of a flow's bits processed by a node or passing through an edge with the

help of the "counters field" of the OpenFlow switches' flow tables. Implementation of this algorithm was studied with Mininet emulation tool under the Abilene network topology. From the simulation results, the extended Dijkstra algorithm outperforms the Dijkstra algorithm and the non-weighted Dijkstra algorithm under the Abilene network in terms of end-to-end latency. However, the mechanism for retrieving $nw[v]$ and $ew[e]$ incurs significant overhead from querying the counters' field of the OF switches repeatedly and this approach lacked any congestion control strategy from traffic congestion which could be created by data spikes or link breakdown.

Marconett *et al.,* (2014) presented a new OpenFlow control architecture, referred to as Optical FlowBroker, for multi-domain software-defined optical networks. They also considered that the centralized control plane (e.g. the OpenFlow architecture and its associated protocol) within a single administrative domain faces difficulties in scalability in extending it to multiple administrative domains. The hierarchical brokers improve scalability and inter-domain global coordination, while allowing domain controllers to manage intra-domain forwarding decisions. A preliminary study of FlowBroker performance was conducted using the Mininet network emulation tool. The Floodlight Openflow controller was chosen, due to the ease of extensibility. However, this approach used a single controller for their multi-domain approach. This can lead to disruption of service when the controller is not accessible.

Bindhu *et al.,* (2015) proposed Load Balancing and Congestion Control in Software Defined Networking using the Extended Johnson Algorithm for Data Centre Networks. The algorithm solves all pair's shortest paths, in a sparse weighted graph with negative numbers. The extended

Johnson algorithm is used in analyzing parameters like response time, throughput, latency and server load variation. The idea of the Extended Johnson algorithm is to re-weight all the edges and make them all positive, then apply Dijkstra for every vertex. The network congestion is prevented by request to the nearest server with threshold Ø value which normally is lower than specified one.

In addition to the OpenFlow nodes, other utility modules are installed which provide further required functionality like a spanning tree module and a controller placement module. The Abilene core topology had an OpenFlow Controller with five (5) OpenFlow switches as SDN nodes, each of which was linked to the controller logically, each domain consists of an OpenFlow switch, several hosts and optionally an OpenFlow controller. The network modelling software OMNeT++ was used to build the project. However, this approach experienced performance degradation when the number of OpenFlow switches increased. It could only handle five (5) OpenFlow switches effectively.

Widhi *et al.,* (2015) proposed an extended Dijkstra based load balancing for OpenFlow network. The Extended Dijkstra algorithm considers not only the edge weights, but also the node weights to find the nearest server for a requesting client. The proposed algorithm also considers the link load in order to avoid congestion. They used Pyretic to implement the proposed load balancer and compared it with related ones under the Abilene network topology with the Mininet emulation tool. The load balancer was placed in the front-end of the online services systems to map the request from the clients to the servers. According to the Abilene network topology, they set up an OpenFlow controller and 11 OpenFlow switches as nodes, each of which was linked to the controller logically and assumed that there were two web servers that will be accessed from clients. The python-based controller (POX) was used as the OpenFlow controller. The load

balancing algorithm was implemented using Pyretic and the VIP address was used as an IP public address that will be accessed from the client. For simulation testing they generated Transmission Control Protocol (TCP) data streams from clients to the servers using iPerf. To evaluate the capability of their proposed algorithm, throughput and end-to-end latency tests were conducted with LABERIO and the round robin load balancing approach. From the output of the results, the proposed approach showed improvement in terms of the network end-to-end latency and server load variation, when compared with the other two. However, the mechanism packet transversal lacks any traffic congestion control, which may result to significant latency when congestion is experienced.

Senthil and Rajani (2015) proposed a dynamic load balancer using Software Defined Networks. Here Domain Name Server (DNS) maps the host name to an IP address in a network. Round-robin works by responding to DNS requests not only with a single IP address, but with a list of IP addresses of several servers that host identical services. Using the Mininet network emulator they created a virtual network topology with six hosts nodes, where two nodes are taken as http servers (web server) and the other four (4) as http clients. The POX controller was used for implementing the load balancer. The load balancer rewrites the destination IP address of each incoming client packet to the address of the assigned replica. The round robin policy uses a circular queue to decide where to send a request. Using the curl command traffic is sent to the server. The curl command is used from all the four *HTTPClient* nodes to send traffic to the *HTTPServers* on port number 80. iPerf network tool was used to measure TCP and UDP bandwidth performance. Round robin DNS load balancing works best for services with a large number of uniformly distributed connections to servers of equivalent capacity. However, the

round robin approach may deflect a client's request to a farther server even when the link is congested. In a real network, high performance IP routers and switches add approximately 200 milliseconds of latency to the link due to packet processing. It means, if the requests are deflected to the farthest server, the latency will increase significantly.

Smriti and Kotla (2015) proposed a load balancer in the SDN architecture based on traffic volume. In this implementation the dynamic load balancer is a module integrated with the floodlight controller to set different custom costs on each link dynamically cached by the topology module. The load balancer is comprised of four components namely: Topology module, DBalancer, IDListenser and JSON Serializer. The topology module is the most essential component whose main function is to get dependencies of modules and for loading modules. Using Mininet, a topology consisting of 150 OpenFlow switches and 400 hosts attached per switch was created. The number of flows per second and number of responses given by the controller per second was calculated using Cbench (a network testing utility tool) in both modes i.e. throughput mode and latency mode. They further computed the throughput and latency and proved it can handle more packets and have greater efficiency than round robin load balancer. Software-defined networks is developed to manage large networks like WAN, cloud computing technologies like data center, big data. This algorithm would not scale well on such networks.

Pongsakorn *et al.,* (2015) designed an SDN-Assisted Bandwidth and Latency Aware Route Allocation. The goal of this research, was to improve the overall performance of the network by separating applications into bandwidth-oriented and latency-oriented applications and allocate different routes for each type of application accordingly. Routes were calculated from the monitored information using Dijkstra algorithm and its variation. To realize bandwidth and latency aware network, they took advantage of several key technologies including OpenFlow and

Overlord. Their proposed bandwidth and latency aware network consists of four (4) primary components: OpenFlow Network, Bandwidth, Latency Monitor and Bandwidth/Latency (BW/LAT) Controller Supported Application. They presented a use case in which their proposed method achieved better performance than traditional shortest-path routing mechanism. However, the bandwidth and latency aware routing dynamically controls routes based on application's requirements, it doesn't address any congestion on the link.

Seonhyeok *et al.*, (2015) presented a congestion prevention mechanism based on Q-learning for Efficient Routing in SDN. They considered the fact that the existing routing algorithms used in controllers use the Dijkstra algorithm, which does not consider the changes in bandwidth of each link when selecting the routing path and therefore lead to congestion. To solve this problem, they introduced the Q-learning routing algorithm. The Q-learning algorithm monitors and measures the bandwidth of the network edges, they defined the threshold for when the congestion occurs and calculated a new optimal path to prevent network congestion. To prove the efficiency of the proposed algorithm under a virtual network environment, they compared their approach with the classical Dijkstra algorithm and the extended Dijkstra algorithm for bandwidth and average data transmission time. The bandwidth test proved that their approach has slight improvement with respect to the other two. However, this approach is only limited to a certain environment that has fixed traffic generation rate and the size of bandwidth.

Inderpreet *et al.*, (2016) presented a dynamic load balance packet switching using OpenFlow protocol. This algorithm utilizes the hierarchical feature of fat-tree networks to recursively hunt down a way and makes decisions based on real-time traffic statistics obtained through the OF

protocol. The algorithm was implemented as module of the OpenDaylight controller with two functional capabilities: observing traffic insights and planning flows. This research utilized the Mininet network emulator to assess the dynamic load balance algorithm, by comparing it with the static load balancing algorithm. Results from the test proved that, at low communication overheads, the dynamic load balancer showed superior performance over the static strategy. But as the network overhead increases, this dynamic strategy experienced a significant drop in performance compared to that of the static strategy, due to the recursive mechanism used to query the controller for obtaining traffic statistics.

Cui and Xu (2016) proposed a load balance solution scheme by taking advantage of the global network view of SDN in order to improve the performance of data transmission in SDN. They collected four load features from each transmission path. These features are bandwidth utilization ratio, packet loss rate, transmission latency and transmission hops. Using these features, Artificial Neural Network model is trained to predict the integrated load for different path and to choose one with least load as the data-flow transmission path. In order to lighten the load of SDN controller, the proposed architecture employs a dedicated load balancer server to balance the load in each path. For the purpose of choosing real-time least loaded path, load balancer immediately calculates the integrated load condition of multiple path when receiving the path information transmitted from SDN controller. Hence, in the proposed design, the controller periodically transmits the load information of each path to load balancer and when the SDN controller need to process the load balance function, the load balancer returns a least loaded path back to controller according to the calculated load condition of each path. The load balancing strategy proposed in this research selects more rational transmission path for data-flow

and achieve 19.3% network latency decrease at most. However, the Artificial Neutral Network model used for this research incurs significant overhead from mechanism for obtaining load information.

Deep *et al.,* (2016) proposed Round Robin Load Balancer using Software Defined Networking. Round Robin works by responding to DNS requests not only with a single IP address, but with a list of IP addresses of several servers that host identical services. This approach works best for services with a large number of uniformly distributed connections to servers of equivalent capacity. The round robin approach may be simple and effective for load distribution, but there are certain limitations such as uneven load balancing. Just like other first-come, first-served methods, it doesn't give priority to applications that are sensitive to bandwidth or latency. This means an urgent request doesn't get handled any faster than other requests in the queue. Furthermore, the Round Robin approach may deflect a client's request to a farther server even if the link is congested. Therefore, causing a significant drop in throughput and latency.

From the literatures reviewed, it is apparent that load balancing and routing in the context of SDN is an active research field. However, most load balancing approaches are static, in other words they allocate resources based on statically configured routes and therefore may experience uneven load balancing and congestion in an SDN topology. Using the REST API technology, which is a component of controllers for communicating with various elements in a network this research work aims to refine the work of Jiang *et al.,* (2014).

CHAPTER THREE

MATERIALS AND METHODS

3.1 Introduction

In this chapter, the methods, materials and procedures used for the successful completion of this research are explained and these involves creating the Abilene network topology, obtaining network topology information using REST API, modifying the mechanism for computing node weight $nw[v]$ and edge weight $ew[e]$ and including congestion control strategies into the load balancer.

The steps of the methodology adopted for this research, towards developing an improved extended Dijkstra algorithm for OpenFlow networks using REST API are as highlighted in the sections below.

3.2 Installation and Configuration

Development environment for the controller essentially includes the Java compiler and the Maven build tools. Integrated Development Environment (IDE) is not mandatory, but certainly recommended to ease the process of building and integrating the module with the controller framework.

For successful implementation of this research work. The following tools and simulators were used:

1. **OpenDaylight controller:** The OpenDaylight controller was selected as the choice for SDN controller due to its novel, intuitive Graphical User Interface (GUI), larger developer community and support. The version of the controller used for this research work is the Hydrogen release.

2. **Mininet Network Emulator:** For emulating the network topology, Mininet was used. It is one of the most popular tools used by the SDN research community

3. **Ubuntu Operating System:** The OpenDaylight controller requires a host Operating System (OS) to operate on. The choice of OS was the Ubuntu distribution of the Linux OS version 14.04.
4. **OpenFlow Switch:** The OpenVswitch (OVS), an open source implementation of a switch was implemented for this research. It can be used as an OpenFlow of legacy switch.

Table 3.1: Assigned Static IP Addresses for Network Set Up

SN	Network host	IP address	Subnet Mask
1	OpenDaylight controller	10.0.0.1	255.255.255.0
2	Mininet Network Emulator	10.0.0.2	255.255.255.0
3	Ubuntu host	10.0.0.3	255.255.255.0
4	OpenFlow switches	10.0.0.4-15	255.255.255.0

All the hosts in the network setup were assigned static IP addresses. As shown in Table 3.1 the Class C subnet mask was used for the network.

3.2.1 Network Topology

The network topology adopted for this research is the Abilene Network topology of the Internet2 project. This topology was created using the python API of the Mininet network emulator. A snippet of the Mininet script used to create this topology is displayed in Figure 3.1. The complete Mininet script for the Abilene topology is located at Appendix A.

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.node import Node

class GeneratedTopo( Topo ):
"Internet Abilene Topology."

def __init__( self, **opts ):
"Create a topology."

# Initialize Topology
Topo.__init__( self, **opts )

# add nodes, switches first...
NewYork = self.addSwitch( 's0' )
Chicago = self.addSwitch( 's1' )
WashingtonDC = self.addSwitch( 's2' )
Seattle = self.addSwitch( 's3' )...

# hosts
NewYork_host = self.addHost( 'h0' )
Chicago_host = self.addHost( 'h1' )
WashingtonDC_host = self.addHost( 'h2' )
Seattle_host = self.addHost( 'h3' )
Sunnyvale_host = self.addHost( 'h4' )
LosAngeles_host = self.addHost( 'h5' )...
```

Figure 3.1: Mininet Script for creating Abilene Network Topology

The Mininet script for the complete topology can be found at Appendix A.

3.3 Replication of the ED-SDN

The process involved in the replication the Extended Dijkstra algorithm (ED-SDN) for SDN are discussed in details in the following sub-sections:

3.3.1 Initializing the ED-SDN parameters

For the purpose of replication and implementation of ED-SDN, some of the parameters such as the number of vertices and number of edges implemented in the work of Jiang *et al.,* (2014) were used. The values selected for these parameters are presented in Table 3.2.

Table 3.2: Simulation Parameters and Settings (Jiang *et al.,* 2014)

Parameter	Setting
Bandwidth on Edges	100Mbps ~ 1Gps
Capacity of nodes	3Gbps ~ 7Gbps
Number of switches	11
Number of edges	25
Controller	POX (Python-based controller)
Testing tool	iPerf
Testing time	30 secs

iPerf is used as a testing tool to generate TCP data streams in the simulation. The complete Python file script for the algorithm can be found in Appendix B1.

The following model assumptions were made (Jiang *et al.,* 2014):

A. The network is represented by a weighted, directed graph G = (V, E), where V is the set of nodes and E is the set of edges, each of which is associated with a non-negative weight.
B. The Abilene network has 10 Gbps connectivity between neighboring nodes and 100 Mbps connectivity between a host and a node.

C. There are 12 iPerf nodes on the Abilene network, among which two nodes are assigned the iPerf client and a server respectively.

D. Two arbitrary iPerf hosts are selected for measuring the simulation results.

3.2.2 Computing node weight of ED-SDN

The node weight and the edge weight are of unit "seconds". When all the node weight and the edge weights along a path are accumulated. The end-to-end latency from one end to the other is obtained. The number of flow bits processed by a node (node weight) is obtained with the help of the "counters field" of the OpenFlow switches' flow table. The mathematical expression as discussed in Section 2.2.10 show how it is derived. The Python function responsible for obtaining the node weight by is at Appendix III.

3.2.3 Computing edge weight of ED-SDN

The number of flow bits passing through an edge is obtained using the "counters field" of the OpenFlow switches' flow tables, the edge weight is obtained using Equation (2.2) as discussed in Section 2.2.10.

3.2.4 Importing ED-SDN module into the POX 2.0 controller

Before the ED-SDN can be used on a network, it must be imported into the controller. The controller is usually hosted on a dedicated computer server. This section explains the process of importing ED-SDN written entirely in python into the controller. The POX comes already installed on the Mininet 2.2 VM image. The POX controller is bundled in a Linux kernel, hence it can be administered via the common Linux command prompt.

3.2.4.1 Starting MiniEdit

To start MiniEdit, the following command was run on a terminal window connected to the Mininet VM:

```
mininet@mininet-vm:~$ sudo ~/mininet/examples/miniedit.py
```

MiniEdit, was used to set up the emulated Abilene network made up of OpenFlow switches. The Figure 3.2 shows the MiniEdit preferences used to start the simulation.

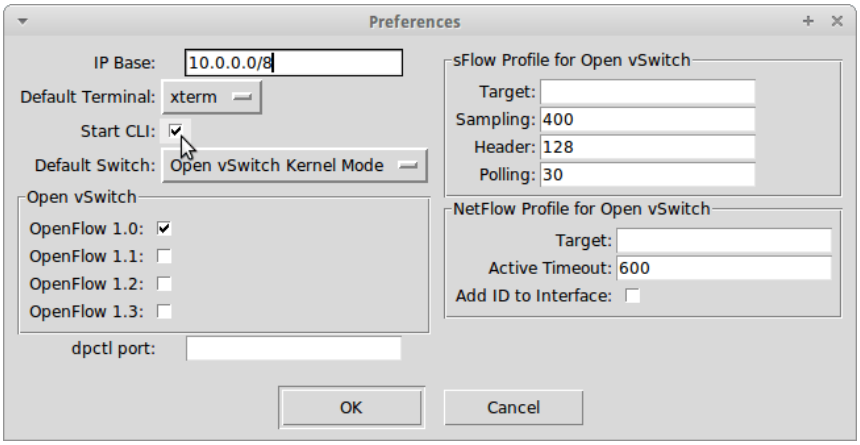


Figure 3.2: MiniEdit preferences

3.2.5 Modified ED-SDN

To make REST API calls to the controller, the application has to be authenticated against the controller. Therefore, the login information including the *username* and *password* is sent to the controller, upon successful authentication, the controller sends back a token. The token values are extracted and saved in a variable called *token*.

```
token=$(curl –sk –H 'Content-Type:application/json'
–d '{"login":{"user":sdn",
"$password":"opendaylight"}}'
https://192.168.56.1:6653/sdn/v2.0/auth
```

Figure 3.5: Authenticating to the Controller

All subsequent API calls require a token. As shown on Figure 3.6 notice the token is presented in the API call. An application called cURL (a command line tool for transferring data) is used to make the API call via the REST API.

```
curl –sk –H "X-Auth-Token:$token" –X POST –H
'Content-Type:application/json
$token –d
https://192.168.56.1/sdn/v2.0/of/datapats/00:00:00:0
0:00:0e/flows
echo "Finished"
printf "\n\n"
```

Figure 3.6: Pushing a Flow into OpenFlow Switch

Figure 3.7 illustrates the JSON payload used to update the flow entry in a switch.

```
payload0b='{
     "flow": {
          "priority": 3000,
          "idle_timeout": 60,
          "match": [
               {"eth_type": "ipv4"},
               {"ipv4_src": "10.0.0.1"},
               {"ipv4_dst": "10.0.0.12"},
               {"ip_proto": "tcp"},
               {"tcp_dst": "80"},
     ],
     "actions": [("output":1)]
       }
}
```

Figure 3.7: JSON Payload used to update an OpenFlow Switch

The HTTP traffic from host with an IP address *10.0.0.1* to destination of *10.0.0.12* will be sent out of port 1 on the switch. The flow priority is set to 3200, however the default priority is 2990 on the controller. The script puts the flow entry higher than the default, this is to ensure that the traffic will use the path as set by this script rather than the default from the controller. The idle timeout for flow entries were set to sixty (60) seconds. This can be set to a higher value however. The idle timeout signifies how long the flow entry will stay in the switch when there is no traffic matching the flow entry.

3.2.6 Setting up the Controller

Two Virtual Machines (VM) were used for the network setup. One ran the Mininet emulated network and the other ran the OpenDaylight controller. Both VMs were connected to a host-only network so they could both communicate with each other. Figure 3.8 shows how the OpenDaylight controller was set up to the host-only network adapter. After which the Mininet emulator and the OpenDaylight controller were assigned static IP addresses of *192.168.56.101* and *192.168.56.102* respectively. The controller communicates using port 6633.

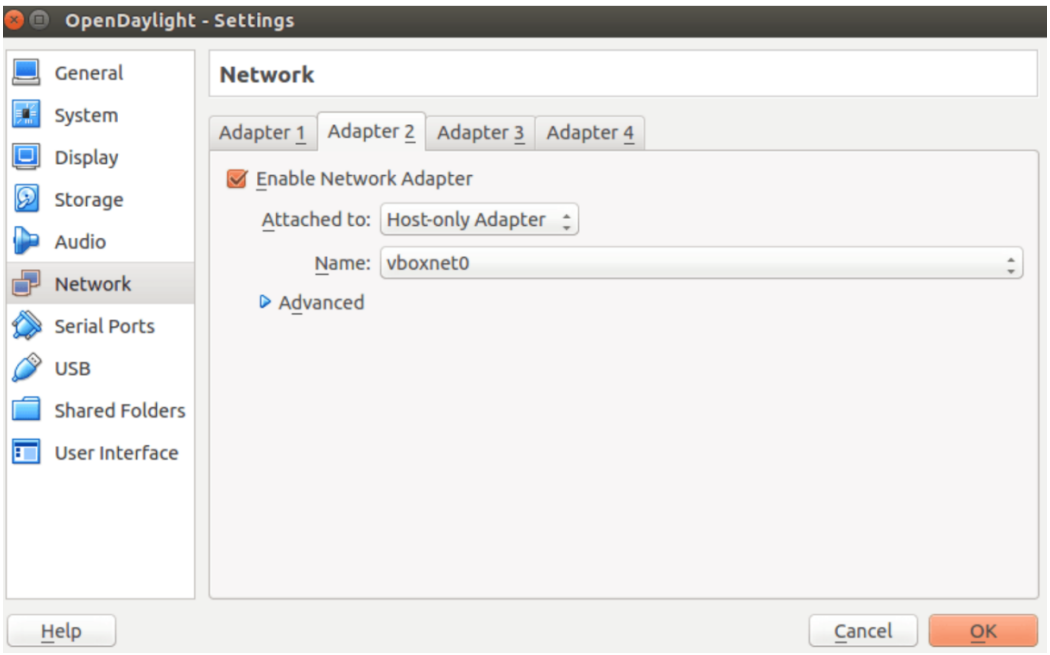


Figure 3.8: Connecting OpenDaylight to host-only adapter

3.2.6.1 Accessing the controller

The controller interface was accessed via a web interface, by directing the web browser to the URL: http://192.168.56.102:6653, Figure 3.9 below presents the login interface of the OpenDaylight controller.

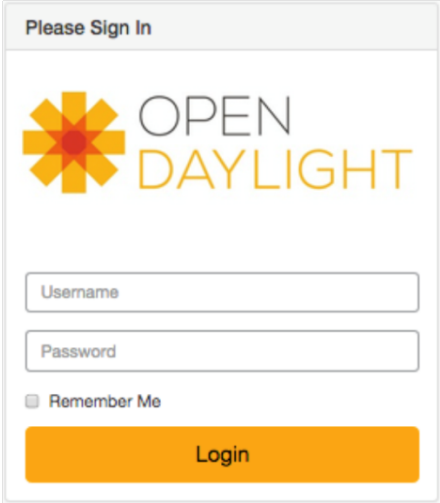


Figure 3.9: Login Dialog of OpenDaylight Controller

3.2.6.2 Launching the Abilene Topology

From the Mininet console, in the *custom/mininet* directory, the following command was run to launch the topology:

```
sudo mn --custom abilene.py --topo mytopo
controller=remote,ip=192.168.56.102,port=6633
```

Figure 3.10: Command to Launch Abilene Topology

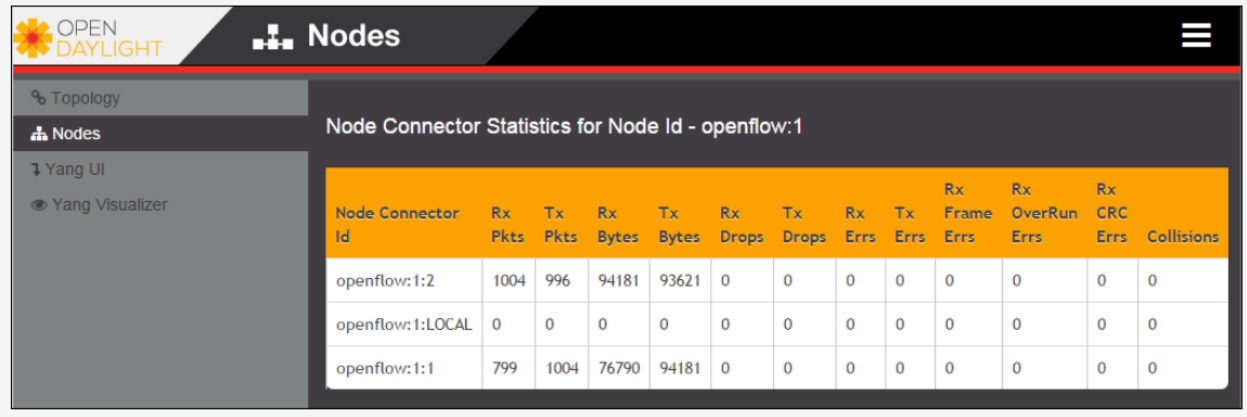
After launching the topology, a *pingall* command was performed a couple of times, until no packet loss occurred. Subsequently, the load balance module (*loadbalancer.py)* was run from the terminal monitor of the host OS, using the command:

```
opendaylight-sdn$: python loadbalancer.py
```

Figure 3.11: Command to Launch Load Balancer Script

3.2.6.3 Displaying node information

Nodes tab displays information about each switch in the network as depicted in figure 3.12.



Node Connector Id	Rx Pkts	Tx Pkts	Rx Bytes	Tx Bytes	Rx Drops	Tx Drops	Rx Errs	Tx Errs	Rx Frame Errs	Rx OverRun Errs	Rx CRC Errs	Collisions
openflow:1:2	1004	996	94181	93621	0	0	0	0	0	0	0	0
openflow:1:LOCAL	0	0	0	0	0	0	0	0	0	0	0	0
openflow:1:1	799	1004	76790	94181	0	0	0	0	0	0	0	0

Figure 3.12: Node information on OpenDaylight controller

3.2.7 Congestion Prevention Mechanism

The congestion control component of this algorithm uses the bandwidth utilization as its evaluation criterion for improving congestion in an SDN topology. When the bandwidth utilization exceeds the threshold value set by mED-SDN, it reverts to the controller to search for a new path. In other to obtain the link bandwidth utilization, the congestion congestion component proactively measures the bandwidth in the topology and utilizes the REST API on the controller to collect cumulative transmitted bytes B_t at corresponding OpenFlow switches port.

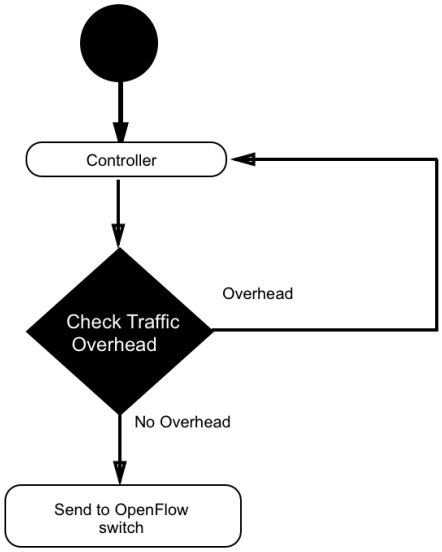


Figure 3.13: Congestion control component

If one path contains several links, L_1, L_2, L_3 ... L_n, with the corresponding bandwidth as BW_1, BW_2, BW_3 ... BW_n, the bandwidth utilization ratio of this path can be calculated by Equation (3.1).

$$BW = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{BW\ i} \tag{3.1}$$

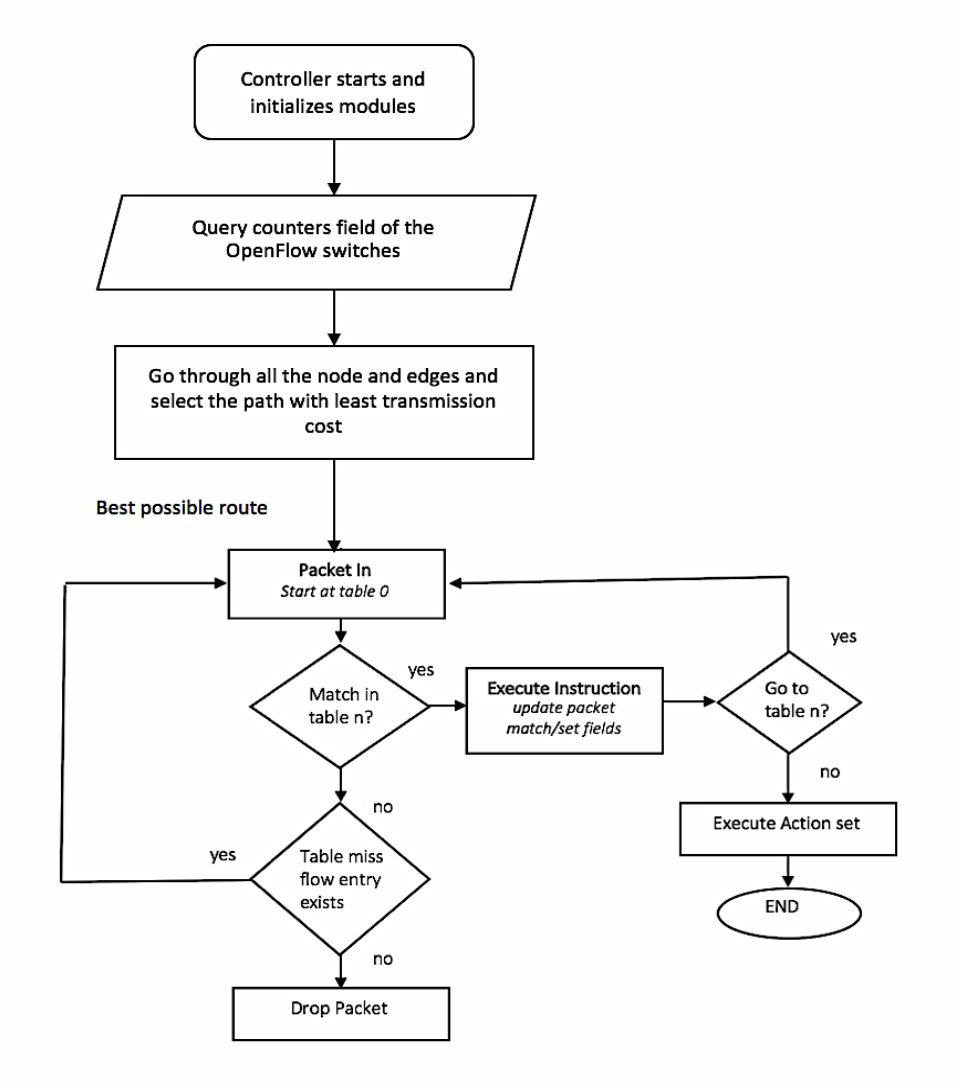


Figure 3.14: Flowchart of Extended Dijkstra Algorithm

When the controller starts, it initializes all the modules and subsequently queries the counters field of the OpenFlow switches to retrieve the node and edge information. With this information, it goes through all the nodes and edges in the topology and selects the path with the least transmission cost and determines that is the best possible route.

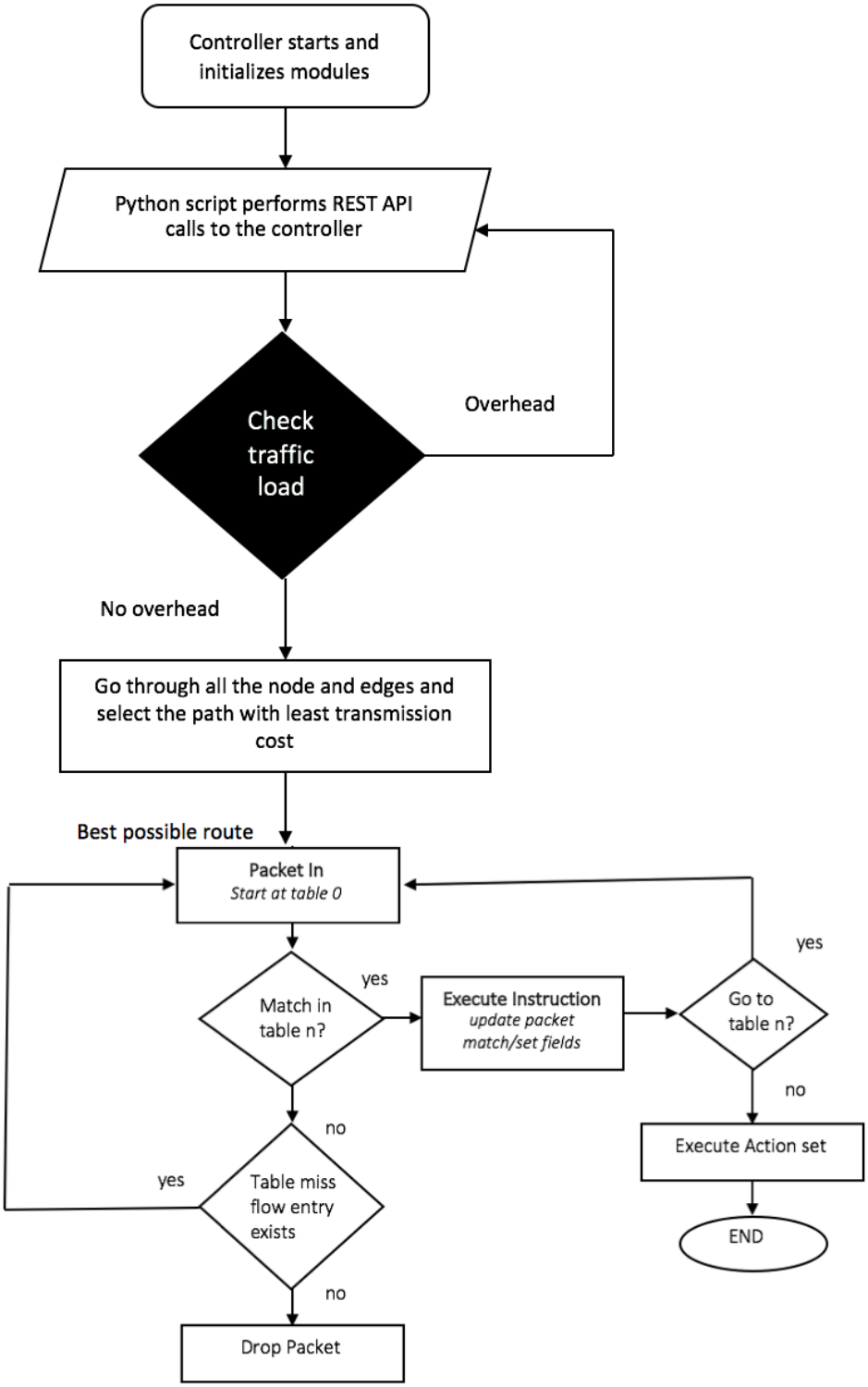


Figure 3.15: Flowchart of Improved Extended Dijkstra Algorithm

As depicted in Figure 3.15, when the controller starts, it first initializes all the modules. These includes the topology discovery, routing and other modules responsible for the primary function of the controller. Unlike ED-SDN, mED-SDN introduces a congestion control component and uses the lightweight REST API mechanism to update the flow entries of the OpenFlow switches. The complete python script for mED-SDN can be found in Appendix C.

3.2.8 Performance Evaluation

Performance of load balancing algorithm can be evaluated using a number of quantitative metrics. For this research work, throughput and end-to-end delay were used for evaluating the performance of ED-SDN, mED-SDN and the Round Robin load balancing approach.

3.2.9.1 Transmission Latency

Transmission latency is delay experienced by a packets during data transmission. The transmission latency can indicate the congestion condition of a link and load condition of the switch in some way. The SDN controller can collect the transmitted bytes in this period and the transmission rate *tr_Rate* at corresponding OpenFlow switches port. Transmission latency can be calculated as follows:

$$Latency_i = \frac{\text{Byte}}{tr_{Rate}} \tag{3.2}$$

Where:

tr_{Rate} = the transmission rate at corresponding OpenFlow switch

If one path consists of several links L_1, L_2, L_3 ... L_n with the corresponding transmission latency $Latency_1$, $Latency_2$, $Latency_3$ ... $Latency_n$, then the total latency of this path can be presented as:

$$Latency = \sum_{1-i}^{n} Latency_i \tag{3.3}$$

3.4.2 Throughput

Throughput refers to how much data can be transferred from the source to the receiver(s) in a given amount of time.

$$Throughput = \frac{Number\ of\ packets\ sent}{Time\ taken} \tag{3.4}$$

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction

In this section, the performance of the ED-SDN and that of the mED-SDN are discussed and pertinent results and analysis are reported. A number of graphs illustrating the performance of the algorithms under a 4-node, 8-node and 12-node network sizes are generated, presented and discussed. Furthermore, to test for efficiency and effectiveness of mED-SDN, it was compared with the popular round-robin load balancing approach. The network testing utility tool, Iperf was used as a testing tool to generate TCP data streams in the simulation and to measure the corresponding throughput and latency.

4.2 Simulation Scenario

Five simulation cases were conducted for each test. For each case, the bandwidth of the edge and the capacity of a node were set to randomly be within the range shown in Table 4.1.

Table 4.1: Simulation Parameters (Jiang *et al.,* 2014)

Parameter	Setting
Bandwidth on Edges	100Mbps ~ 1Gps
Capacity of nodes	3Gbps ~ 7Gbps
Number of switches	12
Number of edges	25
Controller	OpenDaylight controller
Testing tool	IPerf
Testing time	30 secs

Table 4.2: Simulating Parameters for Larger Abilene Network Topology

Parameter	Setting
Bandwidth on Edges	100Mbps ~ 1Gps
Capacity of nodes	3Gbps ~ 7Gbps
Number of switches	18
Number of edges	38
Controller	OpenDaylight controller
Testing tool	iPerf
Testing time	30 secs

4.3 Throughput test on Abilene Network Topology

The focus in this section is to study the average throughput in the network under various node settings. Figure 4.1 illustrates the throughput utilization which is measured by the average transfer rate from the server side using iPerf for the 4, 8 and 12 node scenario of the Abilene network topology after five (5) simulation cases that ran for 30 seconds each. We can see that when the number of nodes are relatively light, all three approaches perform similarly. However, the TCP control mechanisms such as flow control, congestion control and error control mechanisms limit the throughput.

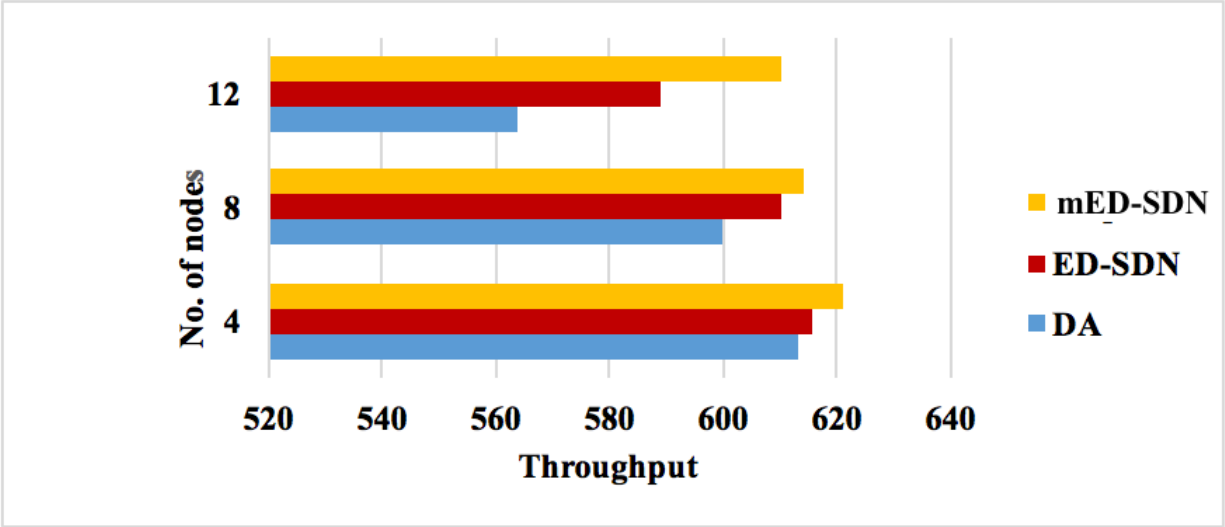


Figure 4.1: Throughput Test on Abilene Network Topology

As the number of host node increase, so does the controller-to-switch traffic (initiated by the controller) increase, since the extended and the classical Dijkstra algorithm are not aware of the congestion status in the network, the transfer rate becomes slower than the proposed approach. Furthermore, the proposed approach has relatively smaller overhead considering the fact that it leverages the REST API technology for obtaining topology information. The results suggest that the proposed approach gives a better performance in comparison to the other two. The average throughput recorded for the 12-node case for DA, ED-SDN, mED-SDN was 564, 589 and 610 Mbps respectively.

4.4 Throughput test on a larger Abilene network scenario

To test for the effectiveness of the algorithm and further study the impact of the number of switches on the performance of DA, ED-SDN and mED-SDN under the Abilene network topology, the number of nodes were increased by six (6) more nodes. The Figure 4.2 illustrates the output from iPerf on the larger network scenario.

Using the parameters for a larger Abilene network network topology as discussed in Table 4.3, throughput tests were carried out for the three approaches. An arbitrary node was selected as the server to carry out the throughput tests using iPerf.

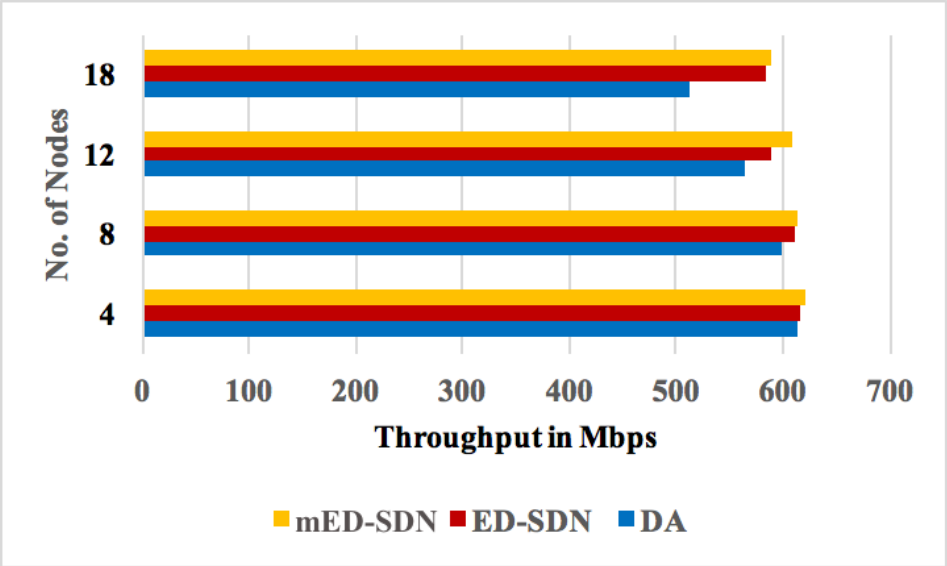


Figure 4.2: Throughput Test on Larger Abilene Network Topology

Under the new traffic conditions, DA experiences the most throughput set back, with an average of 512 Mbps for the 18-node scenario, ED-SDN and mED-SDN performed similarly with a throughput of 585.5 Mbps and 590.4 Mbps respectively.

In a typical network however, it is very unlikely that all the switches are highly active at the same time, but protocols such as the Control Datagram Packet (CDP) and OpenFlow messages

exchanged between the switches and the controller transmitted across the links can reduce the network throughput. Furthermore, DA experienced significant throughput set back due to the fact that it only considers distance as the single factor for determining the shortest path, as the number of nodes increase, the number of hops to reach the destination also increases, therefore a packet has to transverse through more network hops, subsequently resulting to drop in the transmission rate.

4.5 Latency test on the Abilene network

The bandwidth of an edge and the capacity of a node were set randomly to be within the range shown in Table 4.1. The simulation results of the latency tests are shown in Figure 4.5. By the simulation results, we can see the proposed approach has less end-to-end latency than the original ED-SDN, DA experiences significant degradation of latency when the nodes increase, partly due to the increase in the number of hops a packet travels from the server node. The average latency experienced on the 12-node Abilene topology for DA, ED-SDN and mED-SDN were measured at 7.5, 6.8 and 5.4 milliseconds respectively.

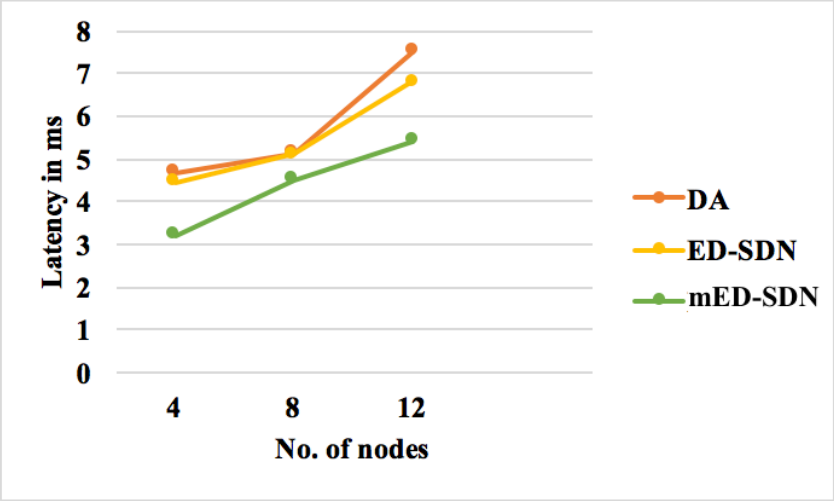


Figure 4.5: Latency Test Result on Abilene Network

4.6 Comparison of mED-SDN and Round Robin approach

Further comparison tests were conducted with a common load balancing method which is the Round Robin approach. As seen in Figure 4.6, the round robin approach has a significant low throughput around 50% less than the proposed algorithm. Furthermore, the round robin approach may deflect a request to a farther server which would cause significant latency. Therefore, if the requests are deflected to the farther servers through which packets transverse switches and routers, the throughput would reduce therefore causing the latency to increase significantly. As shown in the figure below, the proposed algorithm is superior with respect to network throughput. mED-SDN attained an average of 612Mbps under 12-node Abilene network topology while round robin was at 315Mbps.

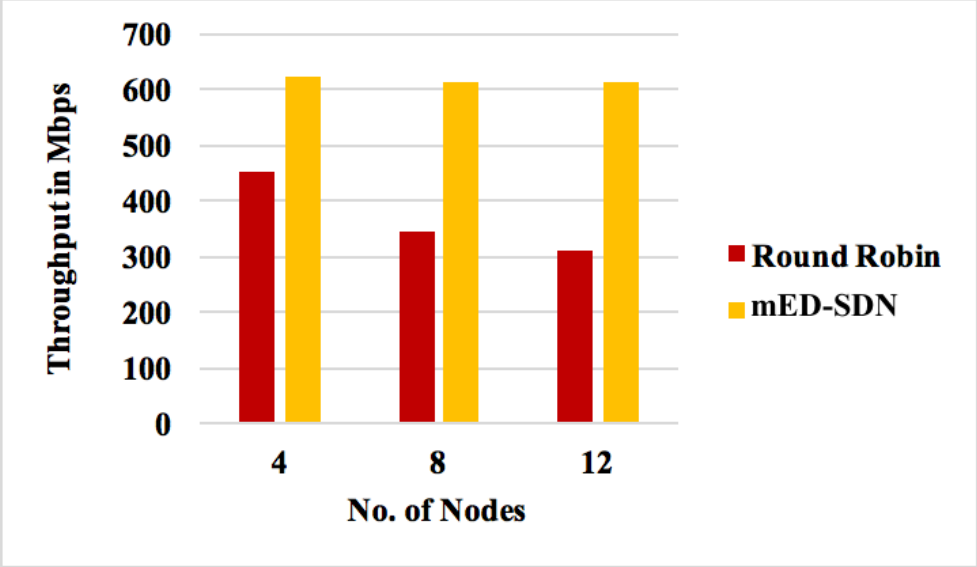


Figure 4.6: mED-SDN and Round Robin Throughput Test Comparison

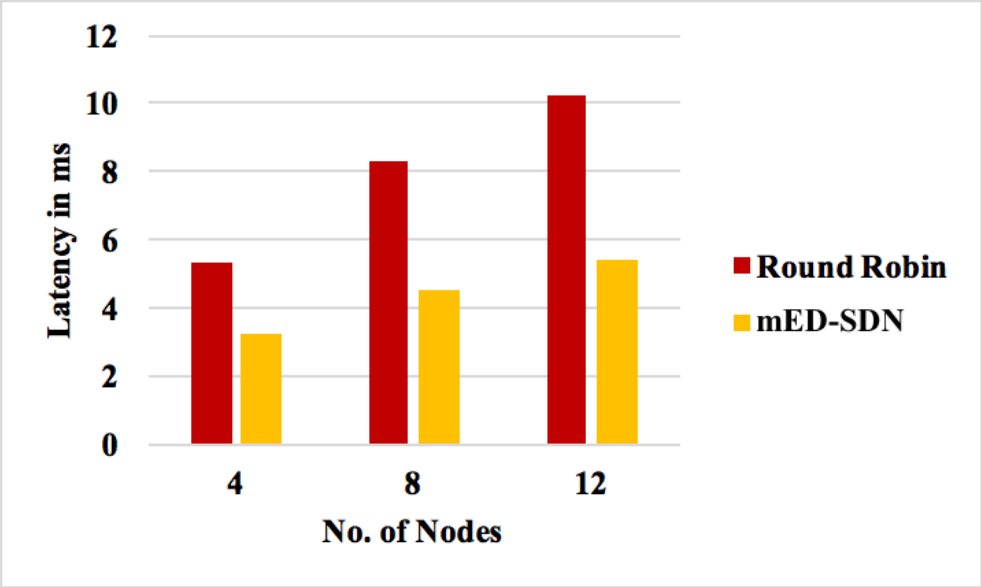


Figure 4.7. mED-SDN and Round Robin Latency Test Comparison

As mentioned in sub-section 4.6, the Round Robin approach may deflect a client's request to a farther remote server that may be congested. As depicted in Figure 4.7, when the Round Robin load balancer was applied to the Abilene network with 12 nodes, the latency increased significantly to an average of 10.2 milliseconds, this is because of the static nature of this method. However, mED-SDN had a corresponding latency of about 5.4 milliseconds. Therefore mED-SDN has a 50% improvement with respect to the Round Robin method.

CHAPTER FIVE

SUMMARY, CONCLUSION AND RECOMMENDATION

5.1 Introduction

This chapter presents a summary of the research, the limitations that were observed during the course of the research work, the recommendation for future work and the conclusion.

5.2 Summary

Most SDN controllers today utilize Dijkstra Shortest Path First algorithm for allocating resources and path finding. The classical Dijkstra algorithm can return a shortest path from the source node to the destination node given a weighted, directed graph G = (V, E) and a single source node, where V is the set of nodes and E is the set of edges, each of which is associated with a non-negative weight (or length). Jiang *et al.,* (2014) extended the original algorithm to consider both edge weights and node weights for a graph derived from the underlying SDN topology. ED-SDN took account of the inherent latency experienced when packets transverse a node and was implemented using POX controller. ED-SDN is useful for deriving the best routing path to send a packet from a specific source node to another node (i.e., the destination node) in a SDN environment but the mechanism for retrieving network status incurred some overhead and lacked any congestion control approach in the case of a congested link. mED-SDN addresses this problem by utilizing the REST API, a light weight mechanism for retrieving network status and topology information from the network topology and introduces a congestion control component in the case of traffic congestion.

5.3 Conclusion

This research is aimed at the development of an efficient load balancer in the context of SDN, by modifying ED-SDN using the northbound plug-in that performs REST requests to retrieve topology information and a function that performs control congestion in the SDN topology. The modified ED-SDN (mED-SDN) is a python-based script, which is imported as a component into the controller. Using iPerf, throughput and latency tests were carried out on a 4-node, 8-node and 12-node scenario respectively on the Abilene network topology. Furthermore, this research presented a use case in which the implemented method achieved better performance than the Round Robin approach and the traditional shortest-path routing mechanism used by most SDN controllers in terms of latency and throughput.

5.4 Significant Contributions

The significant contributions of this research work are as follows:

a) Development of a python-based load balancer for Software Defined Networks by modifying ED-SDN using the REST API of the controller and congestion control strategy.
b) The mED-SDN algorithm produced a 24.3% and 12.5% improvement with respect to the latency and throughput respectively when compared with ED-SDN.
c) Compared the results of the proposed approach with the round robin method and proved superiority with respect to throughput and latency.

5.5 Limitations

The limitation(s) experienced during the course of this research work are as follows:

a) The connectivity between node and a host was assumed to be 100 Mbps. This may not necessarily be the case in a Data Center with redundant high capacity links between hosts.

5.6 Recommendations for Further Work

The following possible areas are recommended for future work in the context of load balancing in Software Defined Networks.

a) A single centralized controller can suffer from issues such as availability and scalability. To address this problem, replicated or distributed controllers can be used to ensure redundancy and service availability.
b) Extending load balancing algorithms in the context of SDN to support traditional networks with only regular switches, or hybrid networks with both OpenFlow and regular switches can be explored with the aim of aiding the SDN transition.

REFERENCES

Astuto, B. and Nguyen, X. (2014). A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE International Conference on Advanced Information Networking and Applications*, 355 – 359.

Bindhu, M., Gun, P., and Rosuol, P. (2015) Proposed Approach of Extended Johnson Algorithm Analysis for Load Balancing and Congestion Control in Software Defined Networking. *Journal of Innovative Technology and Education, Vol. 3(1), 51 - 59.*

Brandon, H., Jaehyeok, S. and Ashis, T. (2008). OpenFlow Switch Specification. *2008 IEEE International Conference on Trust, Security and Privacy in Computing Communications*, 1 - 6.

Cui Chen-Xiao and Xu Ya-Bin (2016). Research on Load Balance Method in SDN. *International Journal of Grid and Distributed Computing, Vol. 9*(5), 25-66.

Deep, G. and Hong, J. (2016). Round Robin Load Balancer using Software Defined Networking (SDN). *Capstone Team Research Project, Vol. 5*, 1 - 9.

Diego, K., Fernando M., Ramos., V., (2014). Software-Defined Networking: A Comprehensive Survey. *IEEE International Conference on Advanced Information Networking and Applications, Vol. 103(1), 1 - 63.*

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik, 1*(1), 269-271.

Diyar, J., Khirota G. and Ibrahim T. (2014) Getting network traffic from network devices in an SDN environment using OpenFlow. *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing Communications, 1 - 6.*

Doan H. and Minh P. (2014). On Software-defined networking and the design of SDN Controllers. *iNEXT: Center for innovation in IT Swervices and Applications University of Technology, Syndey.*

Gunjan, P. (2013). OpenFlow based dynamic load balanced switching COEN233. *Project Report., 82*(1), 31-60.

Handigol, N., Nick M. and Heller, W., (2013) Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow *CS 561: Networking Report, University of Washington, Vol. 7.*

Hardeep, U. and Dane, B., (2010). OpenFlow Based Load Balancing. Paper presented at the Eighth Annual Symposium on Combinatorial Search, 1-7

Hong, Z., Jie, C. and Farid, Y., (2016). LBBSRT: An efficient Load Balancing Scheme Based on Server Response Time. Paper presented at the Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS), 47-51

Hui, L., Yao, S. and Fin, T. (2013). LABERIO: Dynamic Load Balanced Routing in OpenFlow enabled networks. *IEEE International Conference on Advanced Information Networking and Applications*, 290 - 297.

Idris, B., Upena, D. and Raji S., (2016). Performance Analysis of Proposed Network Architecture: OpenFlow vs. Traditional Network. *International Journal of Computer Science and Information Security (IJCSIS), Vol. 14*(No. 3), 30 - 39.

Inderpreet, K., Stern, R., Ben-Yair, A., Kraus, S. and Netanyahu, N. (2016). PHA Star: finding the shortest path with A Star in an unknown physical environment. *Journal of Artificial Intelligence Research* 631-670.

Jiang Jehn-Ruey, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen (2014). Extending Dijkstra Shortest Path Algorithm for Software Defined Networking. *Department of Computer Science and Information Engineering. National Central University Jhongli City, Taiwan*

Manar, J., Abdullah, S., Rasool, A., (2014) Software define networking: State of the art and research challenges. *Department of Electrical and Computer Engineering, Western University Canada*, 74 - 98.

Martin, C., Nate, F. and Arjun G. (2014). Abstraction for SDN. *University of Massachusetts Amherst*, 289-296.

Mckeown, N., Balarishnan H., Parulka, G. and Rexfod, J., (2008). OpenFlow enabling innovation on Campus Networks. *ACM SIGCOMM Commun., Vol. 38*, 69 - 74.

Michael, D., (2011) The REST API lifecycle: From planning to production. *International Journal of Advances in Engineering Research, Vol. 2*, 333 - 339.

Open Networking Foundation (O.N.F.) OpenFlow Switch Specification Version 1.4.0. 2013. *Available online: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf (accessed on 12 January 2017)*

Payal, B., Dan, G. and Jane, S. (2014). A comparative study of static and dynamic Load Balancing. *International Journal of Advance Research in Computer Science and Management Studies, Vol. 12*, 385 - 392.

Pongsakorn, U. Kohei, I. and Susumu, D., (2015). Designing of SDN-Assisted Bandwidth and Latency Aware Route Allocation. *2015 IEEE 13th International Conference on Trust, Security and Privacy in Computing Communications*, 1 - 6.

Sandeep S., Meenakshi, S., & Sam, S. (2008). Performance Analysis of Load Balancing Algorithms. *International Journal of Computer, Control and Information Engineering, Vol. 2* (No. 2), 367 - 370.

Senthil G. and Rajani, S. (2015). Dynamic Load Balancing using Software Defined Networks *International Journal of Electrical and Computer Engineering (IJECE), Vol. 3*, 0975 - 8887.

Seonhyeok K., Jaehyeok, S. and Ashis, T. (2015). Congestion Prevention Mechanism Based on Q-learning for Efficient Routing in SDN. *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing Communications*, 1 - 6.

Smriti Bhandarkar, K. A. K. (2015). Load Balancing in Software-defined Network (SDN) based on Traffic Volume. *Advances in Computer Science and Information Technology (ACSIT), Vol. 2*(7), 72 - 76.

Tim, H. (2011). Path computation enhancement in Software Defined Networks. *ChenDu College of University of Electronic Science and Technology of China*, 128 - 132.

Wei Z., Li L., and Min L. (2016). REST API Design Patterns of SDN Northbound API. *28th International Conference on Advanced Information Networking and Applications Workshops.*

Widhi Y., Jehn-Ruey, J., & Achmad., B, (2015). The Extended Dijkstra-based Load Balancing for OpenFlow Network. *International Journal of Electrical and Computer Engineering (IJECE), Vol. 5*(2), 289-296.

Wolfgang, B. and Michael, M. (2014). Software-defined Networking: Using OpenFlow: Protocols, Applications and Architectural Design Choices. *2013 IEEE 13th International Conference on Trust, Security and Privacy in Computing Communications*, 1 - 6.

Yaunhao Z., & Limin., X. (2014). A Load Balancing Strategy for SDN Controller based Distribution Decision. *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing Communications*, 1 - 6.

Yu, L., Deng, P., (2013). OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support. *Florida International University, Miami, Vol. 3*, 223 - 229.

Marconett., I. Deng, P. and Jing, F. (2013). OpenFlow control architecture using Optical FlowBroker for multi-domain software-defined optical. *International Journal of Grid and Distributed Computing, Vol. 3*, 223 - 229.

APPENDIX A

MININET SCRIPT FOR ABILENE NETWORK TOPOLOGY

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.node import Node
from mininet.node import CPULimitedHost
from mininet.link import TCLink
from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.util import dumpNodeConnections

class GeneratedTopo( Topo ):
"Internet Abilene Topology."

def __init__( self, **opts ):

"Create a topology."

# Initialize Topology
Topo.__init__( self, **opts )

# add nodes, switches first...
NewYork = self.addSwitch( 's0' )
Chicago = self.addSwitch( 's1' )
WashingtonDC = self.addSwitch( 's2' )
Seattle = self.addSwitch( 's3' )
Sunnyvale = self.addSwitch( 's4' )
LosAngeles = self.addSwitch( 's5' )
Denver = self.addSwitch( 's6' )
KansasCity = self.addSwitch( 's7' )
Houston = self.addSwitch( 's8' )
Atlanta = self.addSwitch( 's9' )
Indianapolis = self.addSwitch( 's10' )

# ... and now hosts
NewYork_host = self.addHost( 'h0' )
Chicago_host = self.addHost( 'h1' )
WashingtonDC_host = self.addHost( 'h2' )
Seattle_host = self.addHost( 'h3' )
Sunnyvale_host = self.addHost( 'h4' )
LosAngeles_host = self.addHost( 'h5' )
Denver_host = self.addHost( 'h6' )
KansasCity_host = self.addHost( 'h7' )
Houston_host = self.addHost( 'h8' )
Atlanta_host = self.addHost( 'h9' )
```

```
Indianapolis_host = self.addHost( 'h10' )


# add edges between switch and corresponding host
self.addLink( NewYork , NewYork_host )
self.addLink( Chicago , Chicago_host )
self.addLink( WashingtonDC , WashingtonDC_host )
self.addLink( Seattle , Seattle_host )
self.addLink( Sunnyvale , Sunnyvale_host )
self.addLink( LosAngeles , LosAngeles_host )
self.addLink( Denver , Denver_host )
self.addLink( KansasCity , KansasCity_host )
self.addLink( Houston , Houston_host )
self.addLink( Atlanta , Atlanta_host )
self.addLink( Indianapolis , Indianapolis_host )

# add edges between switches
self.addLink( NewYork , Chicago, bw=1, delay='0.806374975652ms')
self.addLink( NewYork , WashingtonDC, bw=1, delay='0.605826192092ms')
self.addLink( Chicago , Indianapolis, bw=1, delay='1.34462717203ms')
self.addLink( WashingtonDC , Atlanta, bw=1, delay='0.557636936322ms')
self.addLink( Seattle , Sunnyvale, bw=1, delay='1.28837123738ms')
self.addLink( Seattle , Denver, bw=1, delay='1.11169346865ms')
self.addLink( Sunnyvale , LosAngeles, bw=1, delay='0.590813628707ms')
self.addLink( Sunnyvale , Denver, bw=1, delay='0.997327682281ms')
self.addLink( LosAngeles , Houston, bw=1, delay='1.20160833263ms')
self.addLink( Denver , KansasCity, bw=1, delay='0.223328790403ms')
self.addLink( KansasCity , Houston, bw=1, delay='1.71325092726ms')
self.addLink( KansasCity , Indianapolis, bw=1,
delay='0.240899959477ms')
self.addLink( Houston , Atlanta, bw=1, delay='1.34344500256ms')
self.addLink( Atlanta , Indianapolis, bw=1, delay='0.544962634977ms')

topos = { 'generated': ( lambda: GeneratedTopo() ) }
```

APPENDIX B

COMPLETE PYTHON SOURCE CODE FOR THE EXTENDED DIJKSTRA ALGORITHM

```
import requests
import json
import unicodedata
from subprocess import Popen, PIPE
import time
import networkx as nx
from sys import exit
import os


# Parses JSON Data To Find all the host information
def hostInformation(data):
    global switch #Store all hosts IP address and the MAC of the
switch to which it is connected
    global deviceMAC # All hosts IP addresses and corresponding MAC
addresses
    global hostPorts # ALL hosts IP, Last octet of the MAC of switch
to which it is connected and the port number to which it is connected
    switchDPID = ""
    for i in data['devices']:
     if(i['ipv4']):
          ip = i['ipv4'][0].encode('ascii','ignore')
          mac = i['mac'][0].encode('ascii','ignore')
          deviceMAC[ip] = mac
          for j in i['attachmentPoint']:
               for key in j:
                    temp = key.encode('ascii','ignore')
                    if(temp=="switch"):
                         switchDPID = j[key].encode('ascii','ignore')
                          switch[ip] = switchDPID
                    elif(temp=="port"):
                          portNumber = j[key]
                          switchShort = switchDPID.split(":")[7]
                          hostPorts[ip+ "::" + switchShort] =
str(portNumber)
    print("\n\nSwitch: " + str(switch))
    print("\n\ndeviceMAC: " + str(deviceMAC))
    print("\n\nhostPort: " + str(hostPorts))


# Parses JSON Data to Find all the link information

def linksInformation(data,s):
    global switchLinks
    global linkPorts
```

```
global G
global linkLatency

links=[]
count = 0;
for i in data:
  src = i['src-switch'].encode('ascii','ignore')
  dst = i['dst-switch'].encode('ascii','ignore')

  srcPort = str(i['src-port'])
  dstPort = str(i['dst-port'])

  latency = str(i['latency'])

  srcTemp = src.split(":")[7]
  dstTemp = dst.split(":")[7]

  G.add_edge(int(srcTemp,16), int(dstTemp,16))

  tempSrcToDst = srcTemp + "::" + dstTemp
  tempDstToSrc = dstTemp + "::" + srcTemp

  portSrcToDst = str(srcPort) + "::" + str(dstPort)
  portDstToSrc = str(dstPort) + "::" + str(srcPort)

  linkPorts[tempSrcToDst] = portSrcToDst
  linkPorts[tempDstToSrc] = portDstToSrc

  linkLatency[tempSrcToDst] = latency
  linkLatency[tempDstToSrc] = latency

  if (src==s):
        links.append(dst)
  elif (dst==s):
        links.append(src)
  else:
        continue

switchID = s.split(":")[7]
switchLinks[switchID]=links
print("The Graph: ")
print(list(G.edges()))
#print("The link is: ")
#print(links)
#print("The switchID:")
#print(switchID)
print("\n\nThe switchLinks: "),
print(switchLinks)
print("\n\nThe linkPort: "),
print(linkPorts)
```

```
    print("\n\nThe linkLatency: "),
    print(linkLatency)

# Parses JSON Data To Find the list of all shortest paths from source
to destination

def findSwitchRoute():
    global path
    pathKey = ""
    nodeList = []
    src = int(switch[h2].split(":",7)[7],16)
    dst = int(switch[h1].split(":",7)[7],16)
    #print("Source = "),
    #print src;
    #print("Destination = "),
    #print dst;
    #temp = nx.all_shortest_paths(G, source=src, target=dst,
weight=None)

   for currentPath in nx.all_shortest_paths(G, source=src, target=dst,
weight=None):
     #print ("currentPath: "),
     #print(currentPath);
     for node in currentPath:

          tmp = ""
          if node < 17:
               #print "str of hex of node:",
               #print str(hex(node))
               pathKey = pathKey + "0" +
str(hex(node)).split("x",1)[1] + "::"
               tmp = "00:00:00:00:00:00:00:0" +
str(hex(node)).split("x",1)[1]
          else:
               pathKey = pathKey + str(hex(node)).split("x",1)[1] +
"::"
               tmp = "00:00:00:00:00:00:00:" +
str(hex(node)).split("x",1)[1]
          nodeList.append(tmp)

     pathKey=pathKey.strip("::")
     path[pathKey] = nodeList
     pathKey = ""
     nodeList = []

    print("\n\nThe path: "),
    print path;
```

```
############################################################
# Computation of total switch latency in all shortest paths

def totalPathLatency():
    global path;
    url = "http://192.168.56.101:8080/wm/core/switch/all/flow/json"
    getResponse(url,"findSwitchLatency")
    findLinkLatency()
    print "\n\nThe total PathLatency: ", pathLatency

#########################################################################
##
# Computation of total latency offered by the links in the shortest
path
def findLinkLatency():
    global linkLatency
    global pathLatency

    for key in pathLatency:
      temp1 = key.split('::')
      length = len(temp1)
      count = 1
      for i in temp1:
            temp2 = i + '::' + temp1[count]
            pathLatency[key] = str(int(pathLatency[key]) +
int(linkLatency[temp2]))
            count += 1
            if count == length:
                  break

############################################################
# Calculate the total switch latency
def findSwitchLatency(data):
    global path
    global pathLatency
    global tempLatency
    for key in path:
      for switch in path[key]:
            tempSec = int(data[switch]['flows'][0]['duration_sec'])
            tempByte = int(data[switch]['flows'][0]['byte_count'])
            if tempByte == 0: tempByte = 1;
            tempLatency += 100 * (tempSec/tempByte)
      #print "Key: ", key
      pathLatency[key] = tempLatency
      tempLatency = 0


############################################################
# Computation total transmission rate in a path
```

```
def costComputation(data,key):
    global cost
    port = linkPorts[key]
    port = port.split("::")[0]
    for i in data:
     if i['port']==port:
          cost = cost + (int)(i['bits-per-second-tx'])

############################################################
# Computation total transmission rate in a path

def getLinkCost():
    global portKey
    global cost
    global linkPorts

    for key in path:
     #print("Key: "),
     #print(key)
     start = switch[h2]
     src = switch[h2]
     srcShortID = src.split(":")[7]
     mid = path[key][1].split(":")[7]
     for link in path[key]:
          temp = link.split(":")[7]

          if srcShortID==temp:
               continue
          else:
               portKey = srcShortID + "::" + temp
               #print("portkey: "),
               #print(portKey)
               portNumber = linkPorts[portKey].split("::")[0]
               stats =
"http://192.168.56.101:8080/wm/statistics/bandwidth/" + src + "/" +
portNumber +"/json"
               #print stats
               getResponse(stats,"costComputaion")
               srcShortID = temp
               src = link
               #print "srcShortID: ",
               #print srcShortID;
               #print "src: ",
               #print src;
     portKey = start.split(":")[7] + "::" + mid + "::" +
switch[h1].split(":")[7]
     finalLinkTX[key] = cost
     cost = 0
     portKey = ""
    #print("portkey: "),
```

```
    #print(portKey)
    #print("cost: "),
    #print(cost)
    print("finalLinkTX: "),
    print(finalLinkTX)

#####################################################
# Setting up of Flow Rule
#####################################################
def systemCommand(cmd):
    terminalProcess = Popen(cmd, stdout=PIPE, stderr=PIPE, shell=True)
    terminalOutput, stderr = terminalProcess.communicate()
    #print "\n***", terminalOutput, "\n"

#####################################################
# creating and pushing the flow rules
def flowRule(currentNode, flowCount, inPort, outPort, staticFlowURL):
    flow = {
      'switch':"00:00:00:00:00:00:00:" + currentNode,
        "name":"flow" + str(flowCount),
        "cookie":"0",
        "priority":"32768",
        "in_port":inPort,
      "eth_type": "0x0800",
      "ipv4_src": h2,
      "ipv4_dst": h1,
      "eth_src": deviceMAC[h2],
      "eth_dst": deviceMAC[h1],
        "active":"true",
        "actions":"output=" + outPort
    }

    jsonData = json.dumps(flow)

    cmd = "curl -X POST -d \'" + jsonData + "\' " + staticFlowURL

    systemCommand(cmd)

    flowCount = flowCount + 1

    flow = {
      'switch':"00:00:00:00:00:00:00:" + currentNode,
        "name":"flow" + str(flowCount),
        "cookie":"0",
        "priority":"32768",
        "in_port":outPort,
      "eth_type": "0x0800",
      "ipv4_src": h1,
      "ipv4_dst": h2,
      "eth_src": deviceMAC[h1],
```

APPENDIX C

COMPLETE PYTHON SOURCE CODE FOR THE MODIFIED EXTENDED DIJKSTRA ALGORITHM

```
class Graph:
def __init__(self):
    self.nodes = set()
    self.edges = defaultdict(list)
    self.distances = {}

def add_node(self, value):
    self.nodes.add(value)

def add_edge(self, from_node, to_node, distance):
    self.edges[from_node].append(to_node)
    self.edges[to_node].append(from_node)
    self.distances[(from_node, to_node)] = distance


def dijsktra(graph, initial):
    visited = {initial: 0}
    path = {}

nodes = set(graph.nodes)

while nodes:
    min_node = None
        for node in nodes:
        if node in visited:
    if min_node is None:
    min_node = node
        elif visited[node] < visited[min_node]:
    min_node = node

if min_node is None:
    break


# Function responsible for retrieving bandwidth info

def costComputation(data,key):
    global cost
    port = linkPorts[key]
    port = port.split("::")[0]
    for i in data:
     if i['port']==port:
           cost = cost + (int)(i['bits-per-second-tx'])
```

```
# Function responsible for retrieving latency info

def totalPathLatency():
    global path;
    url = "http://192.168.56.101:8080/wm/core/switdch/all/flow/json"
    getResponse(url,"findSwitchLatency")
    findLinkLatency()
    print "\n\nThe total PathLatency: ", pathLatency

nodes.remove(min_node)
current_weight = visited[min_node]

for edge in graph.edges[min_node]:
    weight = current_weight + graph.distance[(min_node, edge)]
if edge not in visited or weight < visited[edge]:
return visited, path

########################################
# Authenticating to the controller

auth=$(curl –sk –H ‘Content-Type:application/json’
–d ‘{“login”:{“user”:sdn”, “$password”:”opendaylight”}}’
https://192.168.56.1:6653/sdn/v2.0/auth


########################################
# Method To Get REST Data In JSON Format

def getResponse(url,choice):

    response = requests.get(url)

    if(response.ok):
      jData = json.loads(response.content)
      if(choice=="deviceInfo"):
            hostInformation(jData)
      elif(choice=="linksInformation"):
            linksInformation(jData,switch[h2])
      elif(choice=="costComputaion"):
            costComputaion(jData,portKey)
      elif(choice=="findSwitchLatency"):
            findSwitchLatency(jData)
    else:
      response.raise_for_status()
```

```
##################################
# Method To Perform Load Balancing

def loadbalance():
    global LBType
    linkURL = "http://192.168.56.101:8080/wm/topology/links/json"
    getResponse(linkURL,"linksInformation")
    findSwitchRoute()
    if(LBType == 1):
     getLinkCost()
    if(LBType == 2):
     totalPathLatency()
    addFlow()


########################################
# Pushing the flow rules into the switches in the path
def addFlow():
    global LBType

    Deleting Flow
    cmd = "curl -X DELETE -d \'{\"name\":\"flow1\"}\'
http://192.168.56.101:8080/wm/staticflowpusher/json"
    systemCommand(cmd)

    cmd = "curl -X DELETE -d \'{\"name\":\"flow2\"}\'
http://192.168.56.101:8080/wm/staticflowpusher/json"
    systemCommand(cmd)

    flowCount = 1
    staticFlowURL = "http://192.168.56.101/wm/staticflowpusher/json"

    if(LBType == 1):
     count = 1
     for key in finalLinkTX:
           if count == 1:
                maximum = int(finalLinkTX[key])
                maxKey = key
                count = 2
           else:
                current = int(finalLinkTX[key])
                if current >= maximum:
                     maximum = current
                     maxKey = key
     shortestPath = maxKey
    if(LBType == 2):
     shortestPath = min(pathLatency, key=pathLatency.get)
    print "\n\nBest Shortest Path: ",shortestPath

    currentNode = shortestPath.split("::",2)[0]
```

```
# Port Computation

    port = linkPorts[currentNode+"::"+nextNode]
    outPort = port.split("::")[0]
    inPort = hostPorts[h2+"::"+switch[h2].split(":")[7]]

    flowRule(currentNode,flowCount,inPort,outPort,staticFlowURL)

    flowCount = flowCount + 2

    bestPath = path[shortestPath]
    previousNode = currentNode

    for currentNode in range(0,len(bestPath)):
      if previousNode == bestPath[currentNode].split(":")[7]:
            continue
      else:
            port =
linkPorts[bestPath[currentNode].split(":")[7]+"::"+previousNode]
            inPort = port.split("::")[0]
            outPort = ""
            if(currentNode+1<len(bestPath) and
bestPath[currentNode]==bestPath[currentNode+1]):
                  currentNode = currentNode + 1
                  continue
            elif(currentNode+1<len(bestPath)):
                  port =
linkPorts[bestPath[currentNode].split(":")[7]+"::"+bestPath[currentNod
e+1].split(":")[7]]
                  outPort = port.split("::")[0]
            elif(bestPath[currentNode]==bestPath[-1]):
                  outPort =
str(hostPorts[h1+"::"+switch[h1].split(":")[7]])


    flowRule(bestPath[currentNode].split(":")[7],flowCount,str(inPort)
,str(outPort),staticFlowURL)
            flowCount = flowCount + 2
            previousNode = bestPath[currentNode].split(":")[7]
```

APPENDIX D

THROUGHPUT AND LATENCY TEST RESULTS

Table D1: Simulation results on 4-node Abilene Topology

	mED-SDN		ED-SDN		DA	
No.	**Thoughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**
1	621.6	3.220	615.6	4.432	613.6	4.600
2	621.8	3.223	614.8	4.433	613.8	4.610
3	621.1	3.224	616.1	4.432	612.8	4.612
4	620.8	3.220	615.8	4.443	613.8	4.610
5	620.9	3.210	614.2	4.434	612.2	4.612

Table D2: Simulation results on 8-node Abilene Topology

	mED-SDN		ED-SDN		DA	
No.	**Thoughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**
1	614.2	4.455	610.2	5.123	599.6	5.211
2	613.9	4.511	610.3	5.124	599.8	5.213
3	614.2	4.510	610.4	5.122	600.2	5.231
4	614.1	4.511	609.5	5.122	600.4	5.222
5	614.3	4.512	609.5	5.123	600.3	5.223

Table D3: Simulation results on 12-node Abilene Topology

	mED-SDN		ED-SDN		DA	
No.	**Thoughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**
1	610.2	5.411	589.2	6.791	564.1	7.544
2	610.3	5.412	589.1	6.782	564.2	7.533
3	610.3	5.421	589.4	6.811	564.4	7.532
4	610.1	5.434	589.4	6.811	564.1	7.531
5	610.4	5.421	589.1	6.812	564.3	7.532

Table D4: mED-SDN and Round Robin Throughput and Latency Test on 12-node Topology

	mED-SDN		DA	
No.	**Thoughput (Mbps)**	**Latency (Milliseconds)**	**Throughput (Mbps)**	**Latency (Milliseconds)**
1	612.2	5.322	315.2	10.003
2	612.3	5.322	314.9	10.001
3	612.3	5.312	315.3	10.006
4	612.1	5.311	315.2	9.994
5	612.3	5.322	315.4	9.993