DEVELOPMENT OF A MODIFIED EAST-WEST INTERFACE FOR DISTRIBUTED CONTROL PLANE IN SOFTWARE DEFINED NETWORK FOR WIDE AREA NETWORKS

BY

ABDULFATAI DARE ADEKALE

DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING

AHMADU BELLO UNIVERSITY ZARIA, NIGERIA

JULY, 2018

DEVELOPMENT OF A MODIFIED EAST-WEST INTERFACE FOR DISTRIBUTED CONTROL PLANE IN SOFTWARE DEFINED NETWORK FOR WIDE AREA NETWORKS

BY

Abdulfatai Dare ADEKALE, B.Tech., (LAUTECH), 2012

P15EGCP8013

fataiadekale@gmail.com

A DISSERTATION SUBMITTED TO THE SCHOOL OF POSTGRADUATE 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 COMPUTER ENGINEERING
FACULTY OF ENGINEERING
AHMADU BELLO UNIVERSITY, ZARIA
NIGERIA

JULY, 2018

DECLARATION

I declare that this dissertation titled “Development of a Modified East-West Interface for Distributed Control Plane in Software Defined Network for Wide Area Networks” was carried out by me in the Department of Computer Engineering, Ahmadu Bello University Zaria. 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.

Abdulfatai Dare ADEKALE	_____	_____
(Student)	Signature	Date

CERTIFICATION

This thesis entitled "DEVELOPMENT OF A MODIFIED EAST-WEST INTERFACE FOR DISTRIBUTED CONTROL PLANE IN SOFTWARE DEFINED NETWORK FOR WIDE AREA NETWORKS " by Abdulfatai Dare ADEKALE meets the regulations governing the award of the degree of Master of Science (M.Sc.) in Computer Engineering of the Ahmadu Bello University, and is approved for its contribution to knowledge and literary presentation.

Dr. E. A. Adedokun	_____	_____
(Chairman, Supervisory Committee)	Signature	Date
Prof. M. B. Mu'azu	_____	_____
(Member, Supervisory Committee)	Signature	Date
Prof. M. B. Mu'azu	_____	_____
(Head of Department)	Signature	Date
Prof. S. Z. Abubakar	_____	_____
(Dean, School of Postgraduate Studies)	Signature	Date

DEDICATION

This research work is dedicated to Almighty Allah (S.W.T), my beloved parents, and my entire family.

ACKNOWLEDGMENT

All praises are due to Almighty Allah, the most gracious and most merciful for providing His infinite blessings and guidance for the completion of this research work.

Foremost, my sincere gratitude goes to the chairman of my supervisory committee, Dr. E. A. Adedokun, for his tireless effort, patience, guidance, advice, support and encouragement to the successful completion of this dissertation. I would like to thank my co-supervisor: Prof. M. B. Mu'azu, for his contributions, guidance, support and relentless efforts. The quality of this research work would not be possible without your assistance and supervision.

I appreciate all the lecturers of Computer Engineering Department, Ahmadu Bello University, namely; Dr. T. H. Sikiru, Dr. I. J. Umoh, Dr. M. B. AbdulRazaq, Dr. I. A. Bello, Mrs. B. Yahaya, Mr. I. Yusuf, Mrs. Z. M. Abubakar and other staff for the valuable contributions and encouragement.

My sincere thanks to Mr. Bashir O. Sadiq and Mr. Tijjani A. Salawudeen for all the research discussions, encouragement, assistance and all the fun we had together in NLNG Laboratory. I thank Mr. Ajayi Oreofe for his support, brotherly advice and guidance. I praise the enormous amount of help, valuable contributions, suggestions and constructive criticism of the Computer-Control Research Group.

I am grateful to my friends, my class governor Mr. Lukman and the entire P15 and P16 sets of the Computer Engineering Department; thanks for the support. To Mr. Aliyu Garba, I say thank you for the programming discussions.

Last but not the least, special thanks and deepest appreciation to my parents: Mr. R. Adekale and Mrs. K. B. Adekale for their endless love, unconditional support, kind advice, understanding, and prayers. To my siblings Idris Adekale, Lukman Adekale, Khadijah Adekale, and AbdulQadir Adekale, I say a big thank you for your support and prayers. My gratitude goes to my uncles Mr. Afeez Adekale (Last-born) and Mr. Hassan Zakari (Sir T.), and the entire Adekale family.

I am grateful to everyone, for the support and love. May Almighty Allah bless you all.

Abdulfatai Dare ADEKALE

July 2018

ABSTRACT

This research is aimed at the development of a modified East-West Interface for distributed control plane in Software Defined Network (SDN) for Wide Area Network (WAN). The East-West Interface is important in achieving communication across a Software Defined WAN, to enable scalability and distribution of the control plane. The Communication Interface for Distributed Control Plane (CIDC) was developed to ensure communication in WANs. However, the interface does not address the synchronization of different modules in the controller. This synchronization enables a consistent high availability, policy updating and efficient communication among controllers, needed for SDN to scale in a WAN environment. The modified-CIDC (mCIDC) is therefore developed using the ISyncService that ensures synchronization of state across the different modules in the controller, and among controllers in the WAN. The performance of the mCIDC and CIDC was compared using captured TCP (Transmission Control Protocol) packets, TCP errors and inter-controller communication overload (ICO). The results indicated that for Claranet_2, mCIDC showed a better performance in minimizing the number of Captured TCP Packets, TCP Errors and ICO by 26.55%, 17.89%, and 19.35% respectively when compared with CIDC. While for Claranet_3; 15.82%, 21.60% and 29.25% for Captured TCP Packets, TCP Errors and ICO respectively. For network policies with Claranet_2, mCIDC indicated a better performance in minimize Captured TCP Packets, TCP Errors and ICO by 16.34%, 17.77% and 44.99% respectively. While network policies with Claranet_3; 15.51%, 29.85% and 22.98% for Captured TCP Packets, TCP Errors and ICO respectively. This shows that the mCIDC ensures communication by transmitting the necessary required packets (information) among controllers with reduced TCP errors and fewer overloads.

TABLE OF CONTENTS

CHAPTER FOUR: RESULTS AND DISCUSSION

CHAPTER FIVE: CONCLUSION AND RECOMMENDATIONS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

LIST OF ABBREVIATIONS

ACL	Access Control List
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BGP	Border Gateway Protocol
CIDC	Communication Interface for Distributed Control Plane
CLI	Command Line Interface
CPU	Central Processing Unit
DISCO	Distributed SDN Control Plane
DHT	Distributed Hash Table
DMC	Distributed Multi-Domain Controllers
EWBridge	East-West Bridge
FEs	Forwarding Elements
ForCES	Forwarding and Control Element Separation
GMPLS	Generalized Multiprotocol Label Switching
GNS3	Graphical Network Simulator 3
ICO	Inter-controller Communication Overload
IDE	Integrated Development Environment
LAN	Local Area Network
LLDP	Link Layer Discovery Protocol
mCIDC	Modified Communication Interface for Distributed Control Plane
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NFV	Network Function Virtualization
NIB	Network Information Base
NOS	Network Operating System
ODL	OpenDaylight

ONF	Open Networking Foundation
ONOS	Open Network Operating System
OVSDB	Open vSwitch Database
PCEP	Path Computation Element Protocol
QOS	Quality of Service
REST	Representational State Transfer
RPCs	Remote Procedure Calls
SDN	Software Defined Network
SPOF	Single Point of Failure
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TOR	Top of Rack
UDP	User Datagram Protocol
VM	Virtual machine
WAN	Wide Area Network

CHAPTER ONE

INTRODUCTION

1.1 Background of Research

Computer networks need network devices such as routers and switches to transmit information over the network. These network devices require protocols and policies for effective communication (Nunes *et al.,* 2014). Traditional networks are faced with many challenging concerns (Chen *et al.,* 2015) such as; implementing high-level policies in the network, and manually configuring each network device using low-level configuration commands while considering the network conditions. There is also the need for network environments to be dynamic (Fault-tolerant, Load changes, Automatic Reconfiguration and Response Mechanism), making it challenging to enforce required network policies (Kreutz *et al.,* 2015). In addition to the increasing complexity of Network devices, Traditional networks, also known as legacy networks, are vertically integrated (Kreutz *et al.,* 2015). The control plane and data plane are bundled inside the networking devices, preventing innovation and network flexibility. Another challenge posed by traditional networks is "Internet ossification"; the difficulty in evolving the Internet in terms of physical infrastructures, protocols and performance (Nunes *et al.,* 2014). Emerging Internet applications and services have become increasingly complex and demanding, as such the Internet needs to evolve to address these challenges (Nunes *et al.,* 2014).

Software defined networking (SDN) is a new networking paradigm that addresses the limitations of traditional networks by simplifying network management and enable innovation and evolution (Kreutz *et al.,* 2015). SDN separates the control plane from the data plane. While the control plane decides how to handle the traffic, the data plane forwards traffic according to decisions that the control plane makes (Feamster *et al.,* 2013). SDN has gained immense interest from the industry and academia (Jarraya *et al.,* 2014).

It transfers the intelligence from the traditional network devices to a centralized control plane and enables network programming. SDN is now envisioned for multi-datacenter environments e.g. B4 (Jain *et al.,* 2013) and WANs (Wide Area Networks). This can only be achieved by distributing the SDN control plane. A distributed control plane is necessary to tackle single point of failure (SPOF); which makes SDN architecture highly vulnerable to attacks (Kreutz *et al.,* 2013). It also addresses the challenges of scalability and performances in large networks.

The distributed control plane involves the use of multiple controllers, and this is divided into: logically centralized and logically distributed control planes (Blial *et al.,* 2016). The logically centralized control plane balances charges between controllers and uses a shared database to unify decisions. However, it requires extensive synchronization between controllers and it is not suitable for large and highly distributed networks (e.g. Multi-domain networks). The logically distributed control plane (logically distributed controllers) is suitable for large distributed networks, where each controller manages its domain and distributes the necessary data to other controllers. The primary use of this category is in large data centers and WAN networks that suffer from the high cost and latency, due to the complexity of the infrastructure and protocol e.g. border gateway protocol (BGP), and multi-protocol label switching (MPLS) that handle the traffic (Benamrane et al., 2017).

In a logically distributed SDN architecture (e.g. WAN), the communication between multiple controllers is of primary importance (Benamrane et al., 2017). The communication between the controllers at the control plane is handled by the East-West Interface (or API). Currently, there is no standard for the East-West Interface (Jarraya *et al.,* 2014). Existing research on East-West interface does not consider several characteristics of a real WAN such as different network policies, with high availability.

1.2 Significance of Research

Networks and network technologies have evolve over the years, providing a need for effective and efficient network management across all kinds of networks (small, medium and large networks). This network management must be reliable, scalable and provide fine-grained performance. This necessitate the development of SDN for WAN; providing forwarding, distribution and specification abstractions. This development will enable simple, scalable, cost-effective, efficient, secure, and enhance connectivity across the network(s). The significance of this research work is to ensure communication in WAN with different policies through policy updating and high availability; encouraging the development of SD-WAN. Previous researchers have not considered consistent high availability across controllers for East-West Communication.

1.3 Statement of Problem

The logically distributed architecture control plane address the issue of scalability in large networks such as WAN by providing an East-West Interface for communication among this networks. The East-West Interface needs to provide scalability through efficient and effective communication among controllers in a WAN network; considering real-time characteristics of WAN. Therefore, there is a need for an Interface that considers high availability for policy updating and decision making, and ensures communication among WANs with different policies. This research is aimed at developing a modified East-West Interface for WANs by incorporating high availability across the controllers.

1.4 Aim and Objectives

The aim of this research is to develop a modified East-West interface for distributed control plane in Software Defined Network (SDN) for Wide Area Networks (WANs).

To achieve this aim, the following objectives were set:

1. Emulation of the network environment on a virtual machine (VM) running Ubuntu 16.04 LTS necessary for the implementation of the East-West communication interface for distributed control plane (CIDC) developed by Benamrane et al., 2017.
2. Development of an improved East-West interface based on the network emulated in (1) called the "Modified-CIDC" (mCIDC) to connect the different WANs and network policies.
3. Evaluation of performance of the mCIDC with CIDC using captured TCP packets, TCP errors, and inter-controller communication overload (ICO) as performance metrics.

1.5 Scope of the study

The research study considers the use of the same controller across the different WAN network.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

The literature review comprises of the review of fundamental concepts and the review of similar works.

2.2 Review of Fundamental Concepts

In this section; fundamental concepts such as traditional networks, SDN, SDN Architecture, SDN Communication Interface, Controller, Control Architecture, SDN Controllers, OpenFlow, WAN, CIDC, Simulation and Emulation Tools, Iperf and Wireshark amongst others are reviewed.

2.2.1 Traditional Networks (Legacy Networks)

Traditional networks are usually hardware-centric networks; networks in which networking devices (routers, switches) have purpose-built application-specific integrated circuits (ASICs), and chips (Farhady *et al.*, 2015). These networks are closed systems in which the networking devices have the control and forwarding mechanisms embedded within it and support manufacturer-specific control interfaces (Kreutz *et al.*, 2015). Therefore, deploying new protocols, applications and services in traditional networks becomes a challenge, because all the networking devices need to be updated or replaced.

In the vertically integrated traditional networks, the control plane provides information used to build a forwarding table. The data plane consults the forwarding table. The forwarding table is used by the network device to make a decision on where to send frames or packets entering the device. In the traditional network scenario, routers execute a specific algorithm to send the incoming packets to the next hop. For example, to access one website, the data packets have to move through 10 routers, i.e. 10 routers have to run the routing algorithm to serve one request made by the end user. In practice, there are

million web users accessing the same web content simultaneously, which create a lot of processing overhead at the router. Figure 2.1 shows a Traditional network device with its components (Masoudi & Ghaffari, 2016).

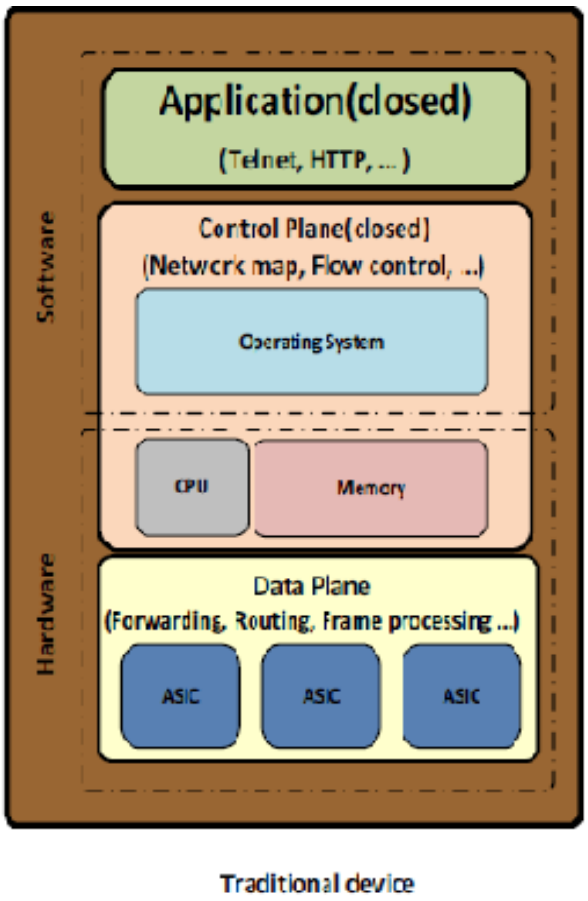

Figure 2.1: Components of Traditional Network Device (Masoudi & Ghaffari, 2016)

2.2.2 Software Defined Networking (SDN)

Open Networking Foundation (ONF) defines SDN as ‘‘an emerging network architecture where the network control is decoupled and separated from the forwarding mechanism and is directly programmable”. In SDN; the network intelligence is logically centralized in software-based controllers (the control plane), and network devices become simple packet forwarding devices (the data plane) that can be programmed via an open interface e.g. ForCES, OpenFlow (Nunes *et al.*, 2014). SDN is dynamic, manageable and cost-effective.

It promises network evolution and innovation by simplifying and allowing the development of new protocols, services and applications. SDN is a network architecture with four pillars (Kreutz *et al.*, 2015):

a) The control and data planes are decoupled.

b) Forwarding decisions are flow-based, instead of destination-based. The flow abstraction allows unifying the behaviour of different types of network devices, including routers, switches, firewalls and middleboxes.

c) Control logic is moved to an external entity, the so-called SDN controller or Network Operating System (NOS).

d) The network is programmable through software applications running on top of the NOS that interact with the underlying data plane devices, and controls the network resources.

Figure 2.2 shows the schematics of SDN and its components (Farhady *et al.*, 2015).

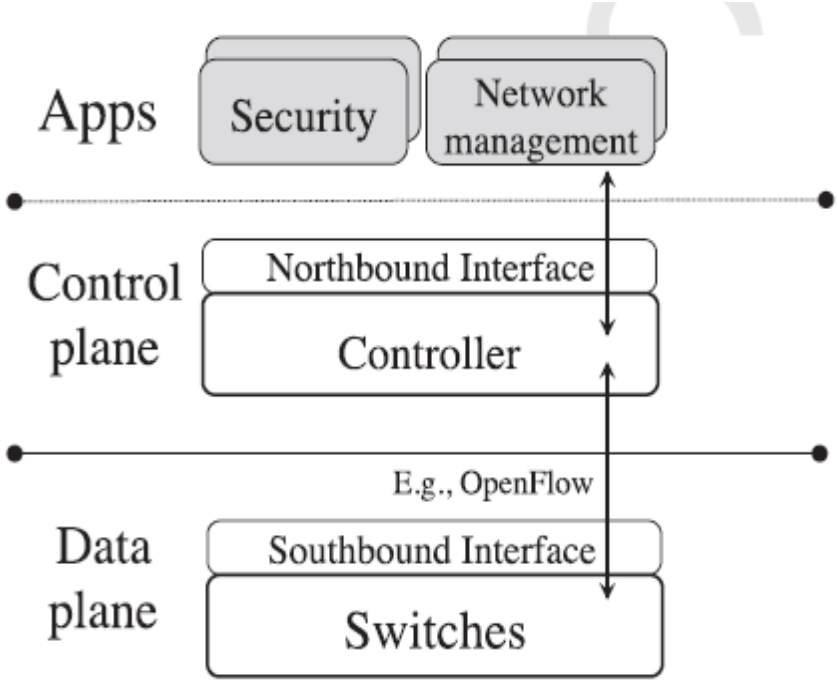

Figure 2.2: Components of SDN (Farhady et al., 2015)

2.2.3 SDN Architecture

SDN is an emerging architecture that decouples the network control and forwarding functions. The SDN architecture is vertically split into three main functional layers as shown in Figure 2.3 (Jarraya *et al.,* 2014). These functional layers are:

a) Infrastructure Layer: Also known as the data plane. It consists mainly of Forwarding Elements (FEs) including physical and virtual switches, routers and middleboxes accessible via an open interface and allows packet switching and forwarding.

b) Control Layer: Also known as the control plane. It consists of a set of software-based SDN controllers providing consolidated control functionality through open APIs, to supervise the network forwarding behavior through an open interface. Three communication interfaces allow the controllers to interact: southbound, northbound and east/westbound interfaces.

c) Application Layer: It mainly consists of the end-user business applications that consume the SDN communications and network services. Examples of such business applications include network visualization and security business applications. This is also called application plane.

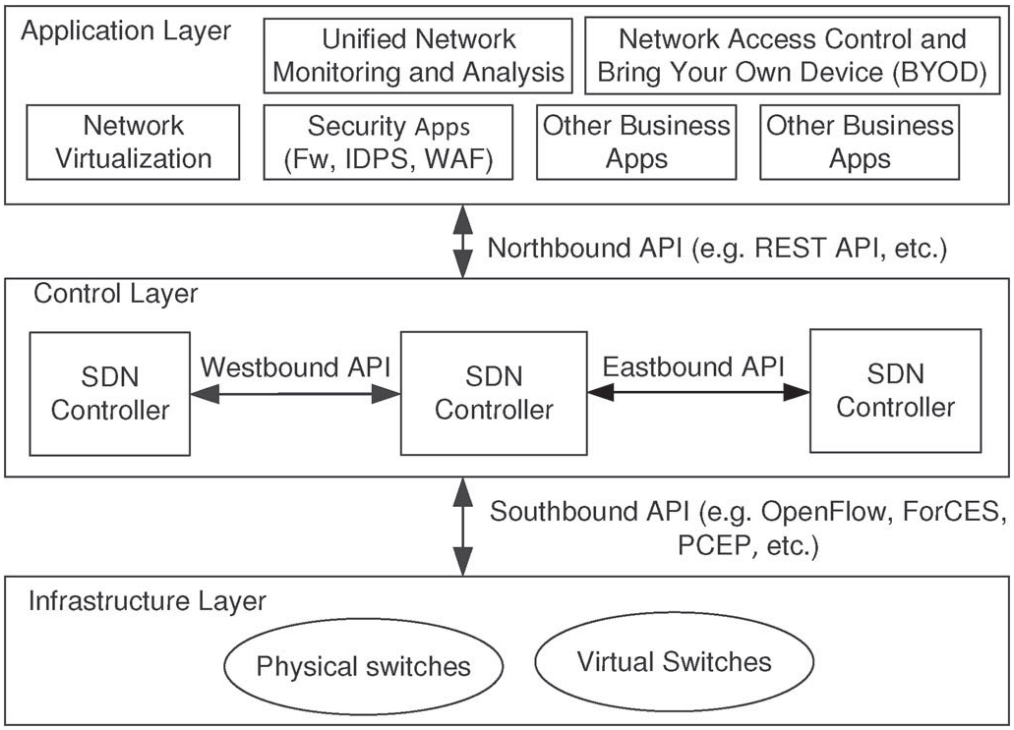

Figure 2.3: SDN Architecture (Jarraya *et al.*, 2014)

2.2.4 SDN Communication Interface

The controller in the control plane interacts with the functional layers (planes) through three open interfaces or APIs (Application Program Interfaces): Southbound API, East/West Bound API, and Northbound API (Jarraya *et al.*, 2014).

2.2.4.1 Southbound Interfaces

Southbound Interfaces (or Southbound APIs) connects the control plane to the infrastructure layer, allowing interaction between the controller and the forwarding elements (Jarraya *et al.*, 2014). OpenFlow is a protocol maintained by ONF; a foundational element for building SDN solutions and can be viewed as a promising implementation of such an interaction. OpenFlow is the most widely accepted and deployed open southbound standard for SDN, but it is not the only available southbound interface for SDN. Other Southbound APIs are (Kreutz *et al.*, 2015): ForCES, OVSDB, POF, OpFlex, OpenState, ROFL, HAL and PAD.

2.2.4.2 Northbound Interfaces

This communication interface provide interaction between the control plane and the application layer, enabling the programmability of the controllers by exposing universal network abstraction data models and other functionalities within the controllers for use by applications at the application layer (Jammal *et al.,* 2014). The northbound interface is mostly a software ecosystem, not a hardware one as is the case of the southbound APIs (Kreutz *et al.,* 2015). It allows programming and managing of the network. Northbound API is an important API that support and enable innovative applications. Northbound APIs can enable basic network functions like path computation, loop avoidance, routing and security. Network applications such as load balancers, software defined security services and orchestration applications can be optimized through the northbound interface. At the time of this study, there is no standard northbound interface. Different SDN controllers have different Northbound APIs. Existing controllers such as Floodlight, Trema, NOX, Onix and OpenDaylight propose and define their own northbound (Kreutz *et al.,* 2015).These various Northbound APIs can be broadly divided into three areas namely; RESTful APIs, specialized ad-hoc APIs and programming languages (Vijay Poonam and Vasudevan Deepika, 2016). RESTful APIs is the most commonly used kind of API to implement Northbound Interface.

2.2.4.3 East/Westbound Interfaces

This interface is an envisioned communication interface, which is not currently supported by an accepted standard (Jarraya *et al.,* 2014). The main purpose of this interface is to synchronize states for high availability, by enabling communication between groups or federations of controllers (Wibowo *et al.,* 2017). East/westbound APIs are very important in large networks or multi-domain networks with multiple controllers. The functions of these interfaces include import/export data between controllers, algorithms for data

consistency models, and monitoring /notification capabilities (Chen *et al.*, 2015). East/Westbound APIs are essential components of distributed controllers. To identify and provide common compatibility and interoperability between different controllers, it is necessary to have standard east/westbound interfaces (Kreutz *et al.,* 2015). These interfaces create a more scalable and dependable distributed control plane. For controllers to be interoperable with subordinate controllers and non-SDN controllers, the east/westbound interfaces must accommodate different controller interfaces with their specific set of services, and the diverse characteristics of the underlying infrastructure, including the diversity of technology, the geographic span and scale of the network, and the distinction between WAN and local area network (LAN) – potentially across administrative boundaries (ONF, 2014). (Jarschel *et al.*, 2014) suggest a finer distinction between eastbound and westbound horizontal interfaces, referring to westbound interfaces as SDN-to-SDN protocols and controller APIs while eastbound interfaces would be used to refer to standard protocols used to communicate with legacy network control planes e.g. path computation element protocol (PCEP), generalized multi-protocol label switching (GMPLS). Figure 2.4 shows an East-West Interface for different SDN domains (Xie *et al.*, 2015).

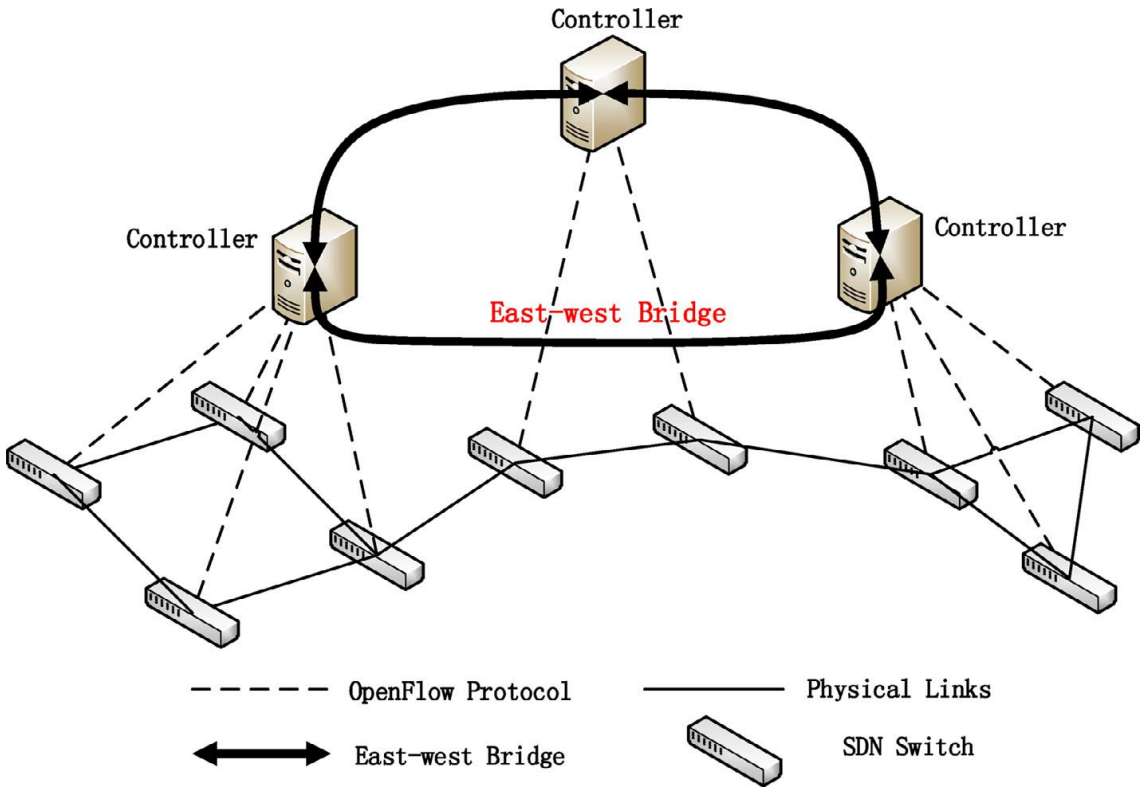

Figure 2.4: East-West Interface for Different SDN Domains (Xie *et al*., 2015)

2.2.5 Controller

The controller is the essential component in the control plane. SDN controllers in an SDN are the "brains" of the network (Stancu *et al.,* 2015). It lies between network devices and the applications. The SDN controller is a software entity that opens SDN APIs to applications, and translates application's request to actions on network devices using proprietary protocol or OpenFlow -like protocol with or without the support of the network operating system NOS (Yin *et al.,* 2012). The NOS is a software responsible for monitoring and controlling a network system. The SDN controller can reside in NOS or run separately on a server in the network (Yin *et al.,* 2012).

The controller is responsible for maintaining all the network protocols, policies and distributing appropriate instructions to the network devices (Jammal *et al.,* 2014). Two of the most well-known protocols used by SDN Controllers to communicate with the switches/routers are OpenFlow and Open vSwitch Database (OVSDB). The controller is

responsible for determining how packets are handled and managing switch flow table. The controller is responsible to maintain the global viewpoint of the whole network and imposes control constraints on each flow by running a set of user-defined control applications. If the controller fails or becomes the performance bottleneck, the network will lose the advantages of SDN (Xie *et al.,* 2015).

2.2.6 Control Architecture

The control architecture affects the performance and scalability of the controller. There are two categories of control architecture: single control plane and multiple control planes.

2.2.6.1 Single Control Plane

The control plane is managed by a physically single centralized controller. All the switches are managed centrally in the network, since they are all connected to the same instance. The single centralized controller represents a single point of failure, making the network highly vulnerable. SDN controller has a limited resource when handling an immense amount of request (Oktian *et al.,* 2017); this creates scalability issue for the single controller.

2.2.6.2 Multiple Control Plane

This involves having multiple controllers sharing the network load equally. In this architecture; a controller can take over from a crash controller, solving the single point of failure problem in the single control plane. Currently, the multiple-control plane has two different implementation methods (Xie *et al.,* 2015):

a) **Logically centralized but physically distributed control planes**: These controllers synchronize their local views about the network with each other, maintaining a global and consistent network with a view to making an optimal decision. This control plane balances charges between controllers and uses a shared database to unify decisions. Examples of the logically centralized control plane are: ONOS (Open Network Operating System),

HyperFlow, Onix and ODL (OpenDaylight). An illustration of a logically centralized but physically distributed control plane is shown in Figure 2.5 (Xie *et al.*, 2015).

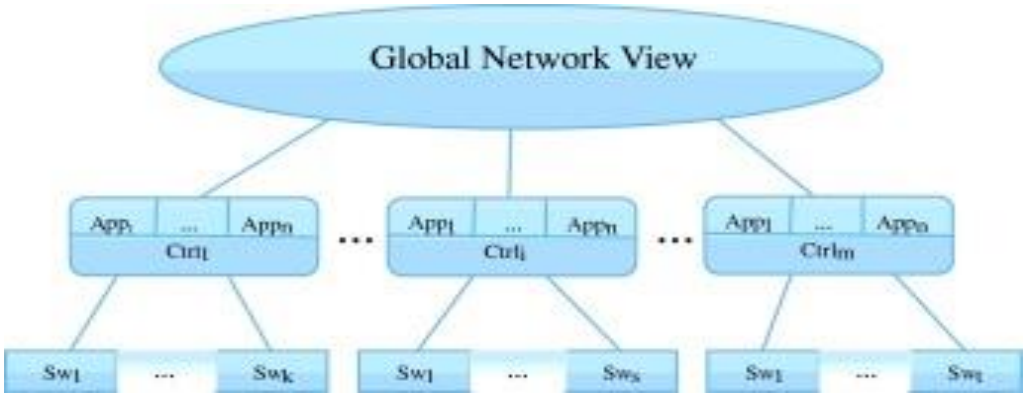

Figure 2.5: Logically Centralized but Physically Distributed Control Architecture (Xie *et al.*, 2015)

b) **Logically distributed control planes**: A local view of the whole network is usually sufficient to achieve given functions of software-defined networks, pursuing global view is not necessary. The controllers in the logically distributed control plane make decisions only using their local view. In this architecture, each controller manages its own domain and distributes the necessary data to other controllers. This architecture is suitable for large and distributed networks (e.g. large data centers and WAN networks), because it does not require extensive synchronization between controllers. Synchronizing network state can lead to network overload, due to frequent change of the whole network state. And, the inconsistent control state of the SDN significantly degrades the performance of many applications (Levin *et al.*, 2012). Logically distributed control plane can be design using two methods: Hierarchical (vertical) and Flat (Horizontal). Examples of the logically distributed control plane are: Kandoo. Figure 2.6 illustrates the two design methods for a logically distributed control plane (Wibowo *et al.*, 2017).

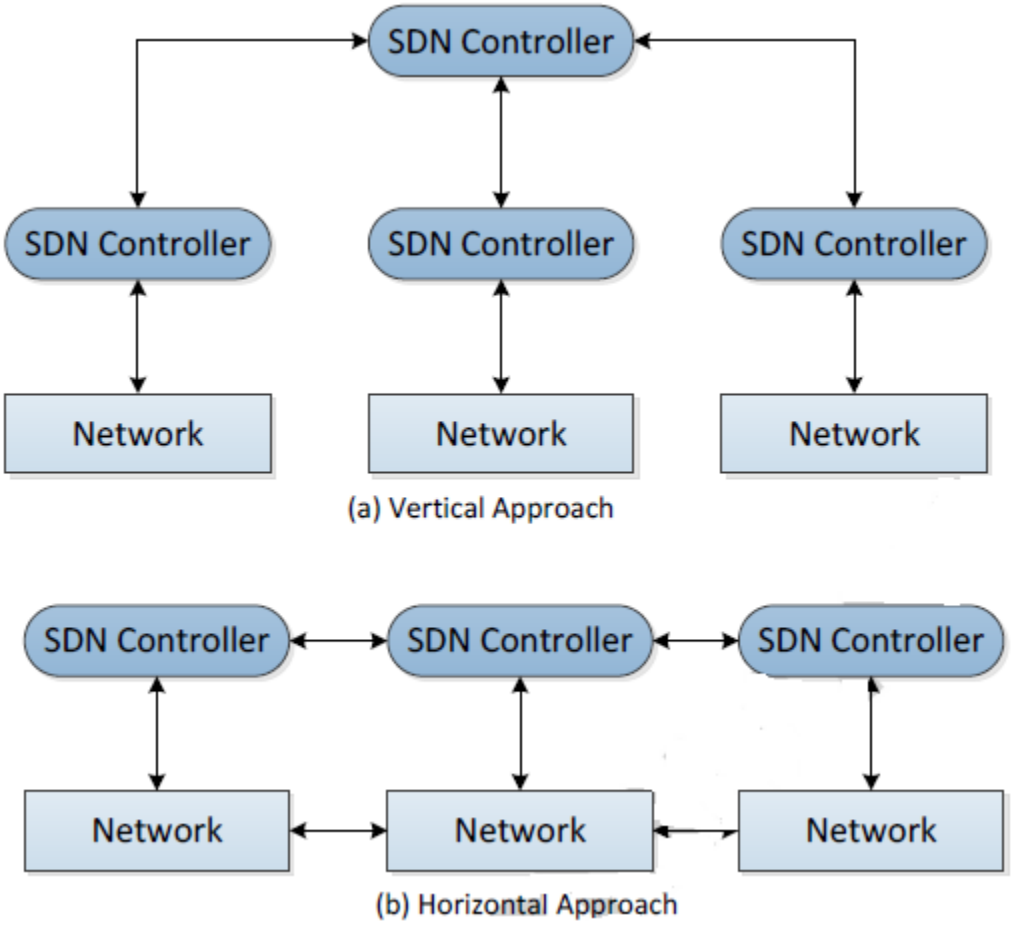

Figure 2.6: Logically Distributed Control Planes (Wibowo et al., 2017)

2.2.7 SDN Controllers

There are different kinds of SDN controllers. These controllers support OpenFlow and OVSDB; to control traffic and configure network devices. Some Open source OpenFlow controllers are shown in Table 2.1 (Wibowo *et al.*, 2017):

Table 2.1: Summary of Open Source SDN Controller Platforms (Wibowo *et al*., 2017)

Controller Name	Programming Language	Organization	Architecture
Beacon	Java	Stanford University	Centralized Multi-threaded
ONOS	Java	ON. Lab	Distributed
Maestro	Java	Rice University	Centralized Multi-threaded
RYU	Python	NTT	Centralized Multi-threaded
POX	Python	Nicira	Centralized
Open Contrail	Python	Juniper	Distributed
Floodlight	Java	Big Switch Network	Centralized Multi-threaded
OpenDaylight	Java	Linux Foundation	Distributed

2.2.7.1 Floodlight

This is based on Beacon and is Apache-licensed. It is one of the open source projects supported by Big Switch Networks; the controller can work with networks containing OpenFlow and non-OpenFlow components. It also integrates OpenStack cloud orchestration. According to Project Floodlight; Floodlight is designed to work with the growing number of switches, routers, virtual switches, and access points that support the OpenFlow standard. It is a Java-based controller, and uses a centralized, multi-threaded architecture. Its Northbound API is the REST (Representational State Transfer) API, and supports OpenFlow (version 1.0 -1.3) for southbound communication. The floodlight controller is used in this research work. The controller is used because of the numerous features it offers, this includes:

a) A module loading system that makes it simple to extend and enhance
b) Easy to use and set up with minimal dependencies

c) It supports a broad range of virtual- and physical- OpenFlow switches

d) High performance due to multi-threaded architecture

e) Support of an open community of developers.

Figure 2.7 shows the architecture of the Floodlight Controller (Ryan Wallner & Robert Cannistra, 2013).

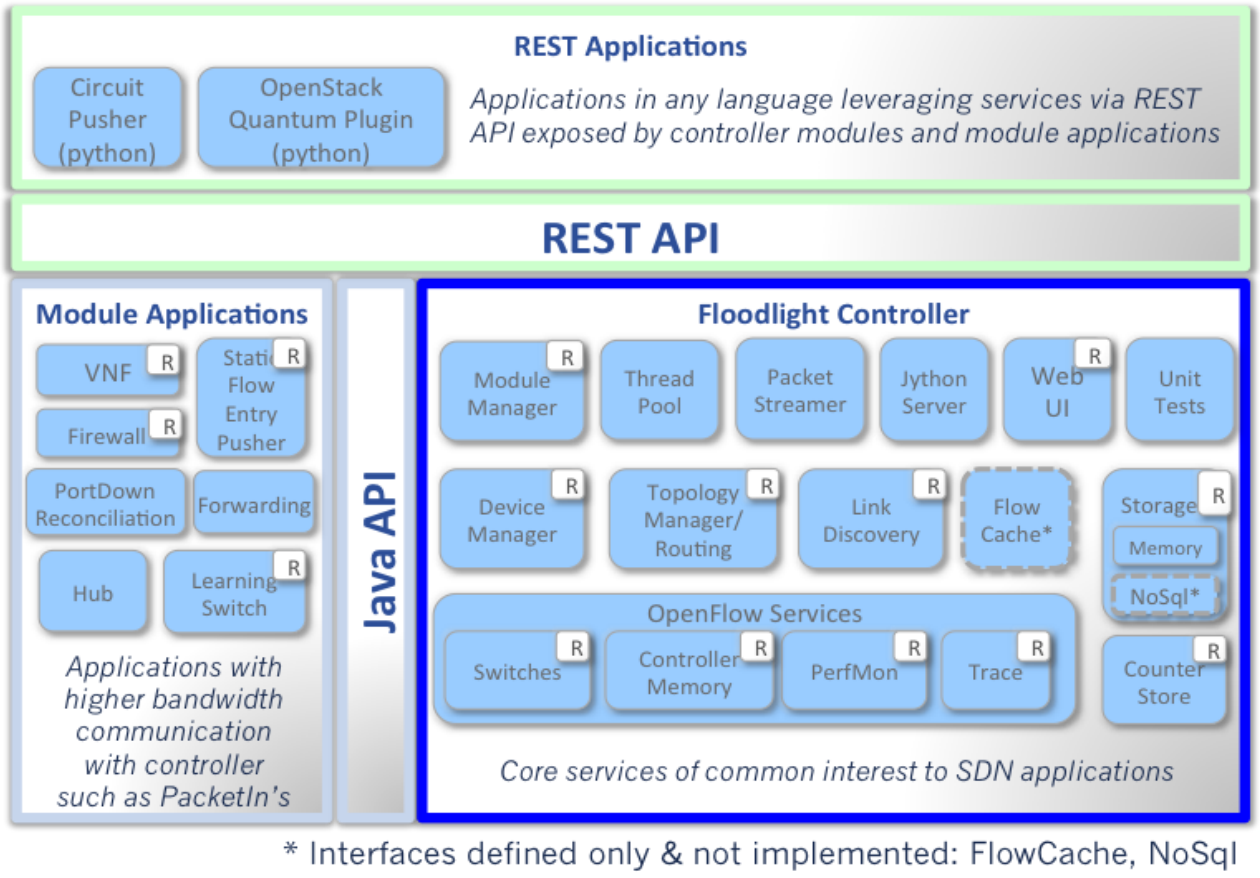

Figure 2.7: Architecture of Floodlight Controller (Ryan Wallner & Robert Cannistra, 2013)

2.2.8 Open Flow

OpenFlow was first released by Stanford University in 2008 (McKeown *et al.*, 2008). It is an open source communication protocol (OpenFlow Protocol) use for communication between the controllers (in the control plane) and networking devices (switches in the data plane) via the Southbound Interface. Since 2011, the OpenFlow switch specification has

been maintained and updated by the ONF. OpenFlow provides software-based access to the switch and router flow tables to enable dynamic network traffic management. The OpenFlow protocol provides management tools to manage features such as topology configuration or packet filtering (Wibowo *et al.,* 2017). Controllers and Switches that support OpenFlow are called OpenFlow Controllers and OpenFlow Switches respectively. OpenFlow-enabled switches can be managed by one or more OpenFlow controllers.

2.2.8.1 Open Flow Protocol

The OpenFlow controller manages the switch flow table by adding, modifying and removing flow entries over the secure channel using the OpenFlow protocol. The secure channel is the OpenFlow channel (southbound interface) that connects the switches to the controller. The OpenFlow channel is usually encrypted using transport layer security (TLS), but can also operate directly over TCP (Jammal *et al.,* 2014). An OpenFlow-enabled switch contains flow and group tables that include a number of entries, depending on the network device. An OpenFlow-enabled switch contains at least one flow table and group table, use to perform packet lookups and forwarding (Wibowo *et al.,* 2017). The flow table consists of a set of flow entries. Each flow entry is used to match to the packet header fields, counters, and a set of instructions for matching packets (Jammal *et al.*, 2014).

2.2.9 SDN and WAN

A WAN is a network that extends over a large geographical distance. It is a large and distributed network. WAN is usually faced with challenges such as scalability, performance and reliability. These challenges can be overcome using the multiple control plane architecture (multiple controller architecture). The multiple controller approach consists of logically centralized and fully distributed or multi-domain approaches (Xia *et al.,* 2015). The fully distributed approach (logically distributed control plane) is most suitable because it does not require extensive synchronization between controllers. Also,

WAN are large networks that consists of multiple networks – the distributed control plane architecture supports the separation of the WAN into multiple SDN domains -Software Defined Wide Area Networks (SD-WAN). These multiple domains form the multi-domain architecture. To improve scalability in WAN; challenges such as global network knowledge (network traffic and state), high availability, real-time monitoring, decision making, fine-grained control and policy updating must be addressed (Shuhao and Baochun, 2015).

2.2.9.1 Multi-Domain

A domain in SDN can be referred to an SDN administrative domain. Multi-domain SDN requires interconnection of controllers in different domains to exchange information across the domain. Multi-domain SDN will enable the interconnection of global SDN domains, introduce interoperability between domains, and provide better provisioning of cross domain services. Each domain in a multi-domain is managed by one or more controllers. The east/westbound interface is very important in multi-domain communication. According to (Egilmez, 2014), the fully distributed SDN controller architectures, both vertical and horizontal approaches could be utilized for multi-domain SDN communication. Figure 2.8 illustrates a Multi-domain Software Defined - Wide Area Network (SD-WAN) (Phemius *et al.,* 2014).

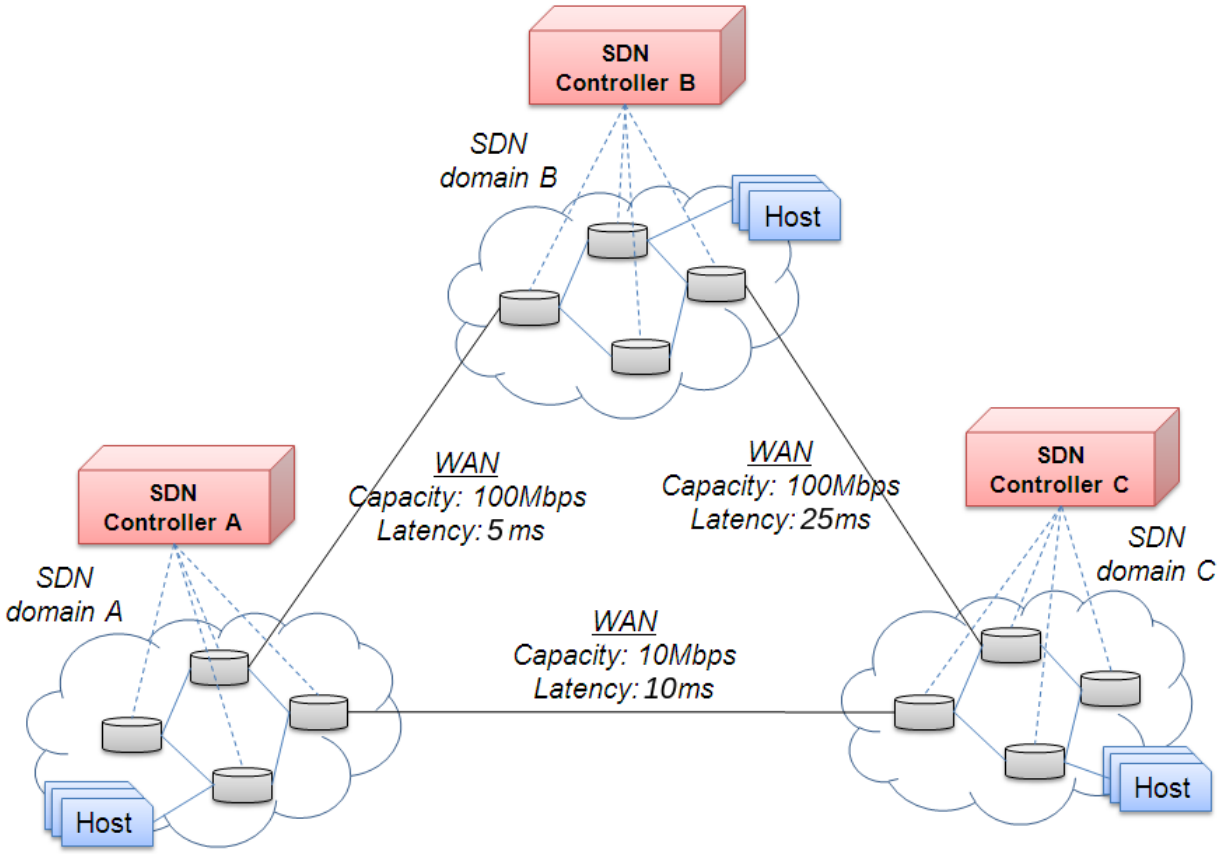

Figure 2.8: Multi-domain SD-WAN (Phemius *et al*., 2014)

2.2.10 CIDC

The CIDC uses the logically distributed control plane architecture to extend SDN for large multi-domain networks such as WAN (Benamrane et al., 2017). The CIDC interface is used by each Floodlight controller to synchronize communication with neighboring controllers. The CIDC interface is composed of four essential modules; Consumer, Producer, Data Updater and Data Collector. These modules are connected with Floodlight controller modules to communicate with the core elements of the controller. Each controller plays the role of a Consumer for external events (events outside its network) and Producer for local events (events within its network). The CIDC interface uses Netty framework to reduce resource consumption. The interface uses several communication modes; Notification (the Producer notifies all remote controllers when changes occur in its domain), Service (the Producer share any activated services such as SSL, Firewall and

Load Balancer) and Full (the Producer shares all events and services). Figure 2.9 shows the algorithm for the CIDC. The CIDC uses an event-driven paradigm (actions are made by events). The controller is active, when new events such as Packet-in are received. These new events are caught by event listeners and observers in the controller. There are listeners and observers for each kind of event.

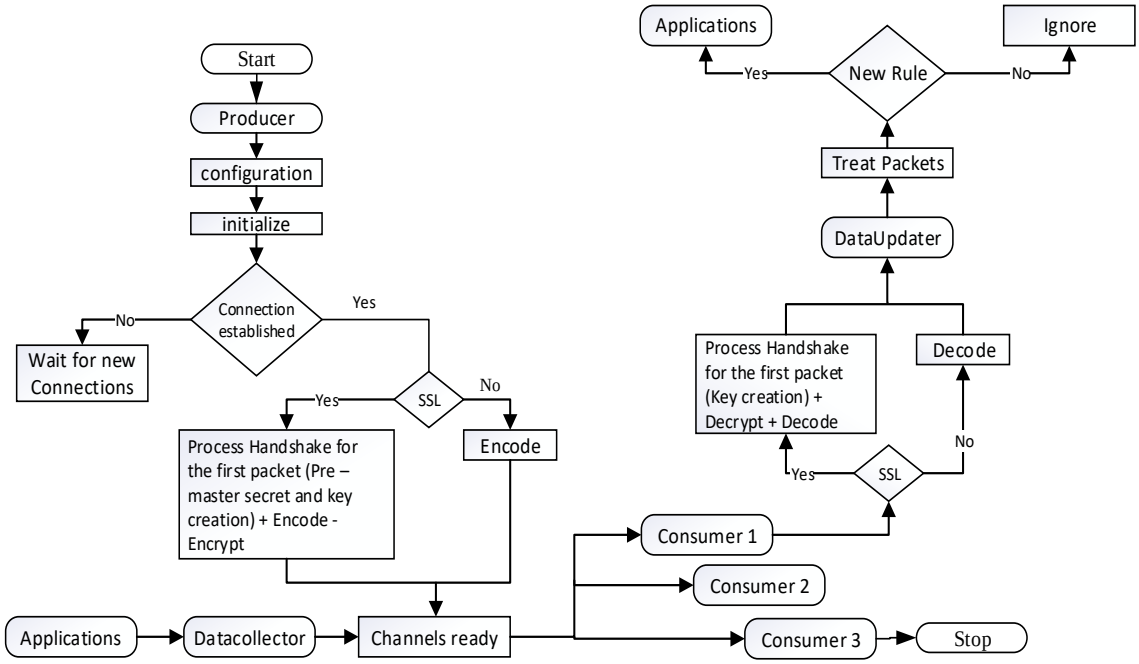

Figure 2.9: CIDC Algorithm (Benamrane et al., 2017)

2.2.11 Simulation and Emulation Tools

Simulation and emulation software is of particular importance for fast prototyping and testing without the need for expensive physical devices (Kreutz *et al.,* 2015). Some Simulation and emulation tools includes: Mininet, ns-3, OMNeT++, EstiNet 8.0, Trema, Mirage, GNS3, and Iperf.

2.2.11.1 Mininet Network Emulator

Mininet is a network emulator which creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and SDN. Mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM,

cloud or native), in seconds, with a single command. Mininet (Lantz *et al.,* 2010) is the first system that provides a quick and easy way to prototype, and evaluates SDN protocols and applications. One of the key properties of Mininet is its use of software-based OpenFlow switches in virtualized containers, providing the exact same semantics of hardware-based OpenFlow switches (Kreutz *et al.,* 2015). Networks emulated on Mininet can be move directly to practical networks composed of real hardware devices. Mininet can be used to support basic network topologies and create custom topologies, using python scripts.

2.2.11.2 Graphical Network Simulator-3 (GNS3)

GNS3 is a graphical network simulator that allows emulation of complex networks. It was first released in 2008, and allows the combination of virtual and real devices, used to simulate complex networks. GNS3 supports both emulated and simulated devices. It allows the virtualization of hardware devices. GNS3 supports cisco devices and network technologies with SDN, NFV, Linux and Docker.

2.2.11.3 Iperf

Iperf is a popular network tool that was developed for measuring TCP and user datagram protocol (UDP) bandwidth performance. It is a widely-used tool for network performance measurement and tuning. It enables the tuning of various parameters and characteristics of the TCP/UDP protocol, providing an insight into the network's bandwidth availability, delay, jitter and data loss. It is significant as a cross-platform tool that can produce standardized performance measurements for any network. Iperf has client and server functionality, and can create data streams to measure the throughput between the two ends in one or both directions.

2.2.12 Wireshark

Wireshark is a widely-used packet analyzer. It is free and open source. It is used for network troubleshooting and analysis. Wireshark is cross-platform; it runs on Linux, macOS, Windows and some other Unix-like operating systems. Wireshark uses pcap (packet capture) to capture packets in network traffic. It is used for software and communication protocol development and education. It supports different networking protocol, and is a standard network protocol analyzer used across many commercial and non-profit enterprises, government agencies and educational institutions.

2.2.13 Performance Metrics

Synchronization metrics such as Captured TCP Packets, TCP Errors and Inter-controller Communication Overload (ICO) that could affect response time and controller performance in a distributed architecture are considered as performance metrics. These performance metrics are used to evaluate the necessary data to synchronize during communication.

2.2.13.1 Captured TCP Packets

This is the TCP Packets captured using Wireshark Analyzer, between clients and server. The TCP Packets represents the necessary information transmitted across different controller domains.

2.2.13.2 TCP Errors

These are the TCP errors during conversations between client and server. These errors include: Out-of Order, Dup Ack, Lost Segment, Fast Retransmission, Windows Update and so on.

2.2.13.3 Inter-Controller Communication Overload (ICO)

This is the total rate of bidirectional traffic (Kbit/s) exchanged between controllers, during transmission.

2.3 Review of Similar Works

The communication of controllers in multi-domain networks (such as WAN) is very significant and this is best achieved by the East/Westbound interface. For better understanding, some details about East/Westbound communication interface in distributed control plane and multi-domain, from the operational research perspective are presented.

Koponen *et al.*, (2010) presented the design and implementation of Onix, a distributed control platform for large-scale production networks. In their work, four components in the network (physical infrastructure, connectivity infrastructure, onix and control logic) were controlled by Onix. The network state was stored in the Network Information Base (NIB), and control applications were read and written to the NIB. Onix provided scalability and resilience, by replicating and distributing the NIB between multiple running instances. The network state consistency was provided using a replicated transactional database (for stronger consistency state) and a memory-based one-hop DHT- Distributed Hash Table (for the volatile inconsistent state). The evaluation results showed that Onix could partition the workload over multiple Onix instances. So, in case there was an overhead inside an Onix instance, already assigned switches could be reassigned to another Onix instance. However, not all applications require a network-wide view. This network-wide consistency cannot be maintained in large networks such as WAN.

Tootoonchian and Ganjali, (2010) presented HyperFlow, a logically centralized but physically distributed event-based control plane for OpenFlow. The HyperFlow network is composed of OpenFlow switches (forwarding elements), NOX controllers (decision elements) running an instance of the HyperFlow controller application and an event propagation system (based on publish/subscribe messaging paradigm). A distributed file system called WheelFS, was used to implement the publish/subscribe system and built a global network view. HyperFlow-based controllers operate more smoothly under heavy

load synchronization and kept minimal latency in comparison to NOX controllers. Also, HyperFlow could keep an acceptable amount of consistency among controllers for some 1000 arriving events per second that was 1000 events that include switch and host connecting and disconnecting the network and a link state change. Nevertheless, it imposed a strong requirement of consistent network-wide view in all controllers and this generated large control traffic.

Yeganeh and Ganjali, (2012) designed and implemented Kandoo, a distributed control plane for scaling and efficiently offloading control applications. In their work, they proposed two layers of controllers: Local controllers and a Root controller. The local controllers executed local applications (frequent events) and did not have knowledge of the network-wide state. While, the logically centralized root controller runs non-local applications (rare events) and maintained network-wide state. The local controllers controlled one or more switches while the root controller controlled all local controllers. The root controller loads were considerably lower during the experimental setup; the controller handled fewer events and did not query the Top of Rack (TOR) switches. Results about elephant flow detection problem were obtained, using different applications. This elephant flow was a large continuous flow over a network link that decreased the total bandwidth after a certain amount of time. The final result showed that Kandoo scales significantly better than a traditional OpenFlow network, while solving the elephant flow problem increasingly. The local controller scaled linearly with the size of the network and efficiently shielded the root controller from frequent events. However, the partial network-wide consistency and synchronization were not suitable for large and highly distributed networks such as WAN because each controller was required to distribute data to other controllers in the network.

Dixit *et al.*, (2013) proposed ElastiCon, an elastic distributed controller architecture for dynamically adapting the controller resource pool to traffic conditions and shifting load across controllers. They achieve their objectives by designing algorithms for switch migration (migrating switches from the overloaded controller to lightly-loaded one), controller load balancing (monitoring and automatically balancing the load across controllers) and elasticity. The controller pool is scale up (down) if the aggregate traffic load is greater (smaller) than aggregate controller capacity. The switch migration algorithm is based on a 4-phase migration protocol, to ensure minimal disruption to ongoing flows during migration. ElastiCon is implemented by extending Floodlight and enhancing Mininet. Hazelcast was used to implement the distributed data store, for building global network view. It was noted that ElastiCon added controllers at 10,000 and 11,000 Packet-In messages per second and removes them at 9,000 and 7,000 Packet-In messages per second. The migration time increases marginally as the load on the controller increases. Also, the study shows that the load balancing via switch migration can improve the performance. The evaluation process also indicates that the switch migration process takes about 20ms, which proves the speed of the process. However, this research is more focus on load shifting, than Inter-controller communication. The centralized distributed data store does not scale well in large networks.

Lin *et al.*, (2013) designed a high-performance mechanism called East-West Bridge (EWBridge), for heterogeneous NOSes (Network Operating Systems) to exchange network view in multi-domain networks. The EWBridge is enabled in all kinds of NOSes by adding three modules (Network Virtualization, East-West Bridge and LLDP extension). The EWBridge initiates the controller discovery process (for all controllers to learn the addresses of their peers), and controllers establish a virtual full mesh topology based on TCP/SSL. Messages are delivered for update by the publish/subscribe system and each

database stores the entire topology locally. EWBridge is implemented in two open-source NOSes: Floodlight and NOX. Two data centers are emulated using Mininet to show the functionality of EWBridge. Evaluation of EWBridge shows that; it is faster in data query and speed of network view updates than DHT (Distributed Hash Table). Nevertheless, EWBrdige focuses on enterprise, data center and intra-domain networks. It is not efficiently scalable for large networks,

Berde *et al.*, (2014) developed an experimental system – Open Network Operating System (ONOS). ONOS adopts a distributed SDN control platform architecture. In their work; they propose two prototypes. Prototype 1 focused on implementing a global network view for scalability and fault-tolerance. Prototype 2 focused on improving performance. The network view has three components: a graph database (Titan), a key-value store (Cassandra), and a graph API to expose network state to the application layer (Blueprints). The results of the evaluation study of ONOS prototype 1 showed that ONOS can dynamically add switches, efficiently control hundreds of switches and hosts, and deal instantly with network failures. Prototype 2 addresses the issues in Prototype 1 such as excessive data store operations, which slows the network and the lack of notifications and messaging across nodes, which is essential for the proper communication between the controllers. Prototype 2 improves the performance of Prototype 1, while keeping the global network view consistent. A median latency response of 53.1ms and throughput of 18,832 paths/sec was achieved during the experiment. The target latency response (10 – 100ms) was met, but the target throughput was not (1M path/sec). But, ONOS is focused primarily on service provider networks, and does not take into consideration WAN requirements such as high availability support and network policies.

Krishnamurthy *et al.*, (2014) proposed Pratyaastha as an efficient elastic distributed SDN control plane. A controller assignment algorithm was developed to efficiently assign state

partitions and switches to distributed controller instances by minimizing inter-controller communication to keep flow setup latencies low, and minimizing number of machines used to run controller instances to keep operating costs low. The controller assignment algorithm was implemented using the Optaplanner planning engine. Pratyaastha's assignment algorithm and pooling technique help to avoid costly Remote Procedure Calls (RPCs) between controllers, when handling events from 80% of the switches (switches and state partitions assigned to same controller instance). The algorithm avoids overload by keeping the controller instance processing responsibilities within its CPU (Central Processing Unit) capacity. Pratyaastha gave a 33% and 42% decrease cost when compared with Local CPU + Memory and CPU only, respectively. Nevertheless, this work does not focus on control traffic and inter-controller communication needed for a WAN, such as high availability support, network policies, real-time monitoring and fine grained control.

Phemius *et al.*, (2014) proposed DISCO, an extensible DIstributed SDN COntrol plane for WAN and overlay networks. DISCO was implemented on top of Floodlight OpenFlow Controller and the AMQP protocol. The DISCO controller is composed of two parts: Intra-domain and Inter-domain. The Intra-domain monitors the network and manages flow prioritization, while the Inter-domain manages communication with other DISCO controllers. An Extended Database in each controller is used to store network topology information from both domains. The inter-domain enables the exchange of aggregated network-wide information using Messenger and Agents. The AMQP protocol serves as a base for implementing the Messenger. The Messenger Module builds channels between neighboring controllers to share information. Each Agent publishes and consumes message through Messenger, to ensure consistency in the system. DISCO uses four main Agents (Connectivity, Monitoring, Reachability and Reservation Agent). A typical enterprise network with domain A, B and C were emulated using Mininet to evaluate

DISCO. The evaluation study of DISCO has shown that; after the Monitoring Agent discovers the failure, the Connectivity and the Reachability Modules take charge of failure recovery, working together with the Messenger. Also, DISCO can migrate a virtual machine, from one domain to another inside a DISCO architecture, with low latency and high reachability. But, network policies such as QoS (Quality of Service) and ACLs (Access Control Lists) were not considered during the experimental evaluation, preventing the real-time application of DISCO for WAN.

Fu *et al.*, (2014) proposed ORION, a hybrid hierarchical control plane for large-scale networks. ORION architecture consists of two layers: Bottom layer and Top layer. The Bottom Layer (Area controller layer) is connected to physical switches, and responsible for intra-area topology and routing. The Top layer (Domain controller layer) treats area controllers as devices, and synchronizes the global abstracted network view through a distributed database. Communication between area controllers and domain controllers is established via TCP, which is used to send requests and distributed rules. Routing rules are distributed through the Publish/Subscribe Mechanism. Intra-area and Inter-area single-link failures are prevented with the fast reroute module. Other modules in ORION include device management, topology, routing, storage and vertical communication channel modules. ORION has made a theoretical and an experimental evaluation to test the performance of its control plane. The theoretical evaluation shows that the computing time of ORION has a linear growth, which is much lower than the traditional Dijkstra routing algorithm. The feasibility and effectiveness of Orion are verified with a prototype system built using Java, parts of the area controller were built using Floodlight Controller (which uses OpenFlow). The domain controllers do not use OpenFlow to communicate with the area controllers. ORION was run on a server with multiple virtual controllers, and Mininet was use to simulate the data plane. In the experimental evaluation; simulating 200 switches,

ORION single area controller handled 8114 new flows per second. Increasing the number of areas (controllers) in ORION; shows the overall flow setup rate of ORION is scalable with a gradual increase in delay time and low overhead. The fast reroute results for intra- and inter-area single link failure was acceptable. However, ORION focuses on a large scale intra-domain network and does not consider inter-domain communications. The distributed database provides a partial consistency and synchronization, which is not suitable for WAN.

Bilal *et al.*, (2016) introduced a Distributed approach in the Multi-domain controllers architecture (DMC), which interconnects heterogeneous networks to form a WAN while preserving the privacy of their domains. DMC is implemented on top of the RYU control platform. DMC architecture and implementation is divided into 3 phase: the controller design based on events and connected via REST Interface, the routing module for computing intra- and inter-domain routes, and the monitoring module to ensure end-to-end connectivity between nodes in the case of link failure across domains. In experimenting DMC; the network is emulated using Mininet. The Mininet is hosted on a dedicated Virtual Machine (VM) and Controllers VMs are hosted on a single machine. LAN Segmentation is used to achieve a feel of WAN, and User terminals (hosts) are provided with 100Mbps links, to connect to the network domains. The Flow setup delay results obtained an average response time of 5.2 milliseconds, from the controller to the switch. The controller shares less than 1KB data with other controllers during packet exchange, decreasing overheads, enhancing scalability and resulting in a very smooth communication. The experiment gave a link recovery time of 1.5s, showing how the monitoring module senses and reacts to link failure, and computes an alternative route in minimal time. Nevertheless, DMC focuses more on the privacy of domains, and does not consider Inter-controller communication among WANs.

Benamrane *et al.*, (2017) implemented and evaluated an East-West Interface called Communication Interface for Distributed Control Plane (CIDC), for communication between controllers in logically distributed SDN control plane architectures. Two Controllers were used; a cluster of OpenDaylight (ODL) controllers to build a logically centralized control plane and a modified adaptive Floodlight controller to support logically distributed control plane using CIDC. The CIDC Interface is used by each controller to synchronize its stats and services with its neighboring controllers; for inter-controller communication. Several communication modes (Notification, Service and Full) was used to customize the role of each controller in the network. The Interface was emulated with four different topologies (Gridnet, Claranet, Marwan and Hibernia Uk) obtained from Internet Topology Zoo, Each topology was implemented as a large ring WAN topology. The CIDC shares selected events to exchange based on modes, showing stable delay (does not exceed 4ms) upon addition of a number of connected domains (especially hosts and flows phase), while ODL peak 30ms or more. The necessary information is shared based on the desired mode, resulting in less system resource utilization when compared to ODL. However, Network policies such as QoS (Quality of Service) and ACLs (Access Control Lists) were not considered. And, high availability support across different controller modules was not considered.

In view of the limitations identified from the reviewed of works, it is evident that an East-West Interface is important in establishing communication in a large and highly distributed network, such as WAN. In order to achieve a scalable control plane in a logically distributed architecture; the interface must address the requirements of WAN environment such as global knowledge of network traffic and state (high availability support), real-time monitoring, decision making, policy updating and fine-grained control.

CHAPTER THREE

MATERIALS AND METHODS

3.1 Introduction

In this chapter, the methods, materials and procedures employed for the successful completion of this research are presented. This chapter detailed the steps of the methodology adopted in this research as highlighted in section 1.7.

3.2 Materials

The materials used in this research work are listed and discussed as follows:

3.2.1 Hardware

This research work was emulated on a 64-bit Windows 8 operating system, x64-based processor, Intel® Core ™ i5-3470 CPU @ 3.20GHz (4 logical cores) with 8.00GB RAM, and 500GB Hard disk.

3.2.2 VM Workstation Pro v12.0.1.3

The network environment used in this research work is emulated on a Virtual Machine (VM). Considering features such as faster speed, support for nested virtualization and easy integration with GNS3; VMware was selected as the choice of VM. The research was performed on VMware Workstation Pro v12.0.1.3. The VMware was installed on the 64-bit Windows operating system. Ubuntu 16.04 LTS was installed on the VM as the operating system for the network environment; with 1.5GB RAM, Java 1.7 (and above) and 2 cores of CPU. The installation steps/procedures of the VMware and Ubuntu 16.04 LTS is captured in Appendix A.

3.2.3 Floodlight Controller

Floodlight controller master version was installed on the Ubuntu VM as the controller for the network. This controller was the latest version as at the time of this research study. The controller was cloned from GitHub using Ubuntu Command-Line Interface (CLI). Eclipse

Oxygen 4.7 was installed for programming and launching the floodlight controller. The floodlight controller installation steps are shown in Appendix B.

3.2.4 Mininet

Mininet version 2.2 was installed on the Ubuntu VM using the CLI, with *Open vSwitch* 2.5.4. The Mininet is the software that does the actual emulation of the network on the VM. Steps for installing Mininet are found in Appendix C.

3.3 Methods

The methodology used to achieve the set objectives in this research work are presented and discussed clearly. The methodology adopted are as follows:

1. Emulation of the network environment on a virtual machine (VM) running Ubuntu 16.04 LTS necessary for the implementation of the East-West communication interface for distributed control plane (CIDC) developed by Benamrane *et al.*, 2016. This is achieved using the following steps:
    a. Install virtual machine (VM) running Ubuntu 16.04 LTS. Each VM represents an emulated WAN environment.
    b. Install Floodlight Controller on each VM, to manage each WAN.
    c. Download the network topology (Claranet) from Internet Topology Zoo, and convert from graph mL into a python script.
    d. Configure the Claranet topology on each VM using Mininet.
    e. Interconnect the emulated VMs using GNS3.
    f. Set the link between switches and host to 1Gbps for the bandwidth and 30ms for the one-way delay.
    g. Develop the CIDC using Eclipse Integrated Development Environment (IDE).
    h. Inject traffic into the network using Iperf.
2. Develop the improved East-West Interface (mCIDC), using the following steps:

a. Repeat Step 1 (a) – (g).
b. Develop mCIDC using Eclipse IDE, floodlight services and dependencies.
c. Develop and configure network policies (Quality of Service and Access Control Lists) for each VM.
d. Repeat Step 1 (h).

3. Evaluate the performance of the mCIDC with CIDC:
   a. Compare both mCIDC and CIDC using captured TCP packets, TCP errors and ICO, with no network policies.
   b. Compare both mCIDC and CIDC using captured TCP packets, TCP errors and ICO, with network policies.

3.3.1 Claranet Network Topology

The research is carried out using standard network topology; obtained from Internet Topology Zoo. The Claranet topology which consists of fifteen (15) switches and fifteen (15) hosts was used. The topology was obtained and converted from GraphML to python (.py) format using a parser. The parser was installed inside the Mininet directory as shown in Appendix D. The python script was modified to accept multiple controllers (Appendix D). Figure 3.1 shows the Claranet Network topology.

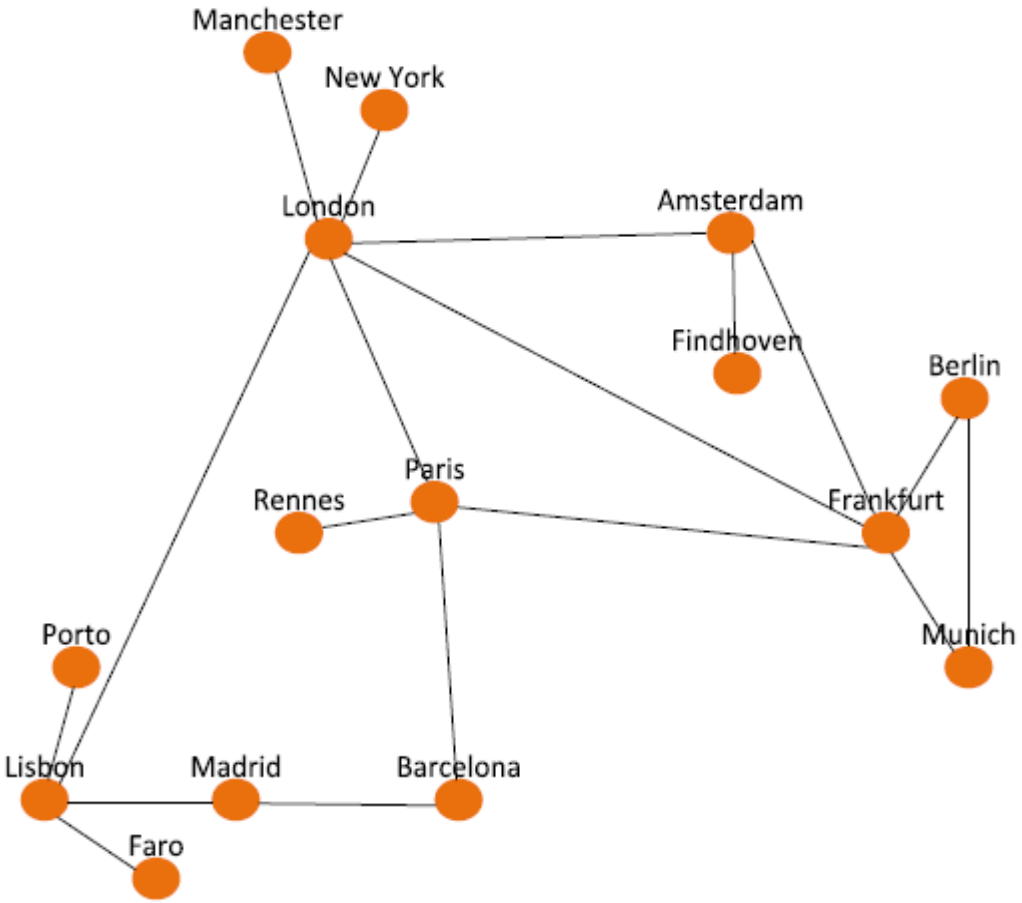

Figure 3.1: Claranet Network Topology (Benamrane et al., 2017)

3.3.2 Connection of VMs in Graphical Network Simulator 3 (GNS3)

The Ubuntu VM is cloned into three VMs (Ubuntu 1, Ubuntu 2 and Ubuntu 3) with each VM containing the network topology, Mininet, and floodlight controller. The VMs are emulated using GNS3 version 2.0.3. Each VM has two network adapters (one connected to the Custom adapter and the other Network Address Translation (NAT)). The GNS3 contains the Ethernet switch (Switch 1) that connects all the VMs. GNS3 is installed on the physical device (the Windows operating system). GNS3 installation is found in Appendix E. Figure 3.2 shows the connection of the VMs in GNS3.

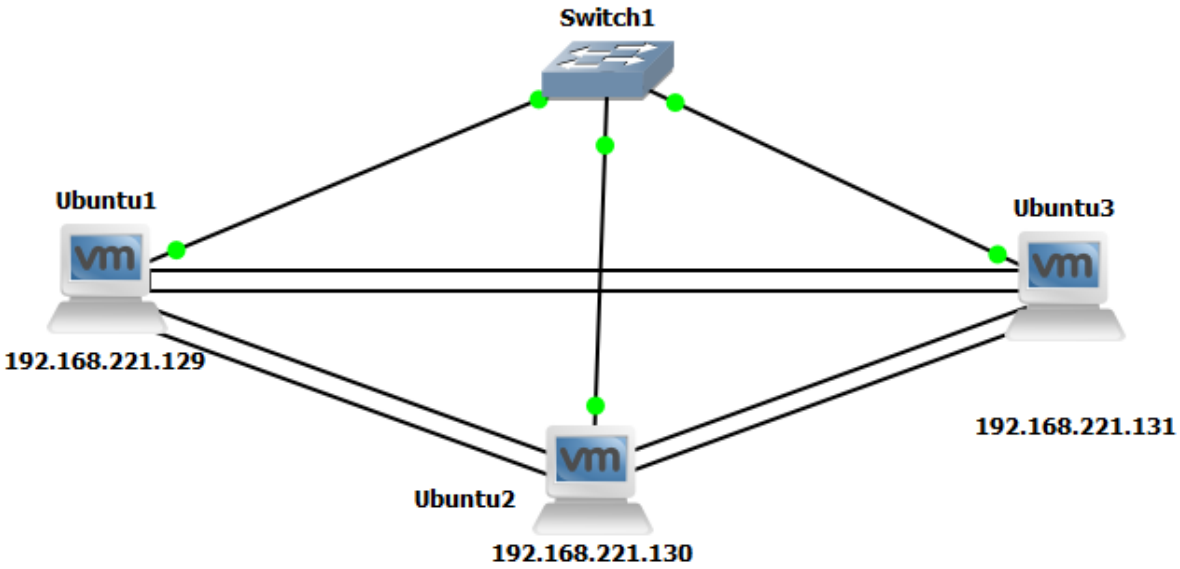

Figure 3.2: GNS3 VMs Connection

3.3.3 CIDC Operation

The default modules and states such as forwarding, thread pool, topology manager, etc. are initialized on starting the controller. The Producer starts by reading the configuration to recognize the IP addresses of the remote consumers (Benamrane et al., 2017). The Producer notifies all remote consumers when new changes occur in its domain, and shares all events. After the Producer initializes the connections by sending a Hello message to the consumers, each consumer that responds and joins a list of connections. The channels are configured and ready for any new events from the Data-Collector. When changes occur (new events are detected), the Data-Collector sends network status to all connected Consumers to notify remote controllers that a local status has changed and to synchronize data between controllers. The Data-Updater receives external data from the Consumer and updates the system. The CIDC interface codes programmed using Eclipse Oxygen 4.7 is contained in Appendix F. Figure 3.3 shows a snippet of the CIDC codes.

```
ScheduledExecutorService ses = threadPool.getScheduledExecutor();
        mythread = new SingletonTask (ses,new Runnable() {
            public void run() {
                try{
                    logger.info("Waiting 10 sec");
                Thread.sleep(10000);
                logger.info("Producer :: "+myConf.getMode()+" mode is configured");
                producer();

                }
                catch(Exception e){
                logger.error("Exception detected "+e.getMessage());
                }
        }
        });
        mythread.reschedule(1,  TimeUnit.SECONDS);

        }
        public void producer() throws IOException{
            // Configure the client.
                bootstrap = new ClientBootstrap( new NioClientSocketChannelFactory(
                            Executors.newCachedThreadPool(),
                            Executors.newCachedThreadPool()));
```

Figure 3.3: Snippet of CIDC codes (Benamrane et al., 2017)

3.3.4 Modification of CIDC

The modification of the CIDC (mCIDC) is carried out using the floodlight ISyncService. The ISyncService synchronizes updates between controllers state by providing access to updates published by all other modules in the controller in an efficient manner. Figure 3.4 shows the flowchart for the mCIDC. The mCIDC codes are shown in Appendix G.

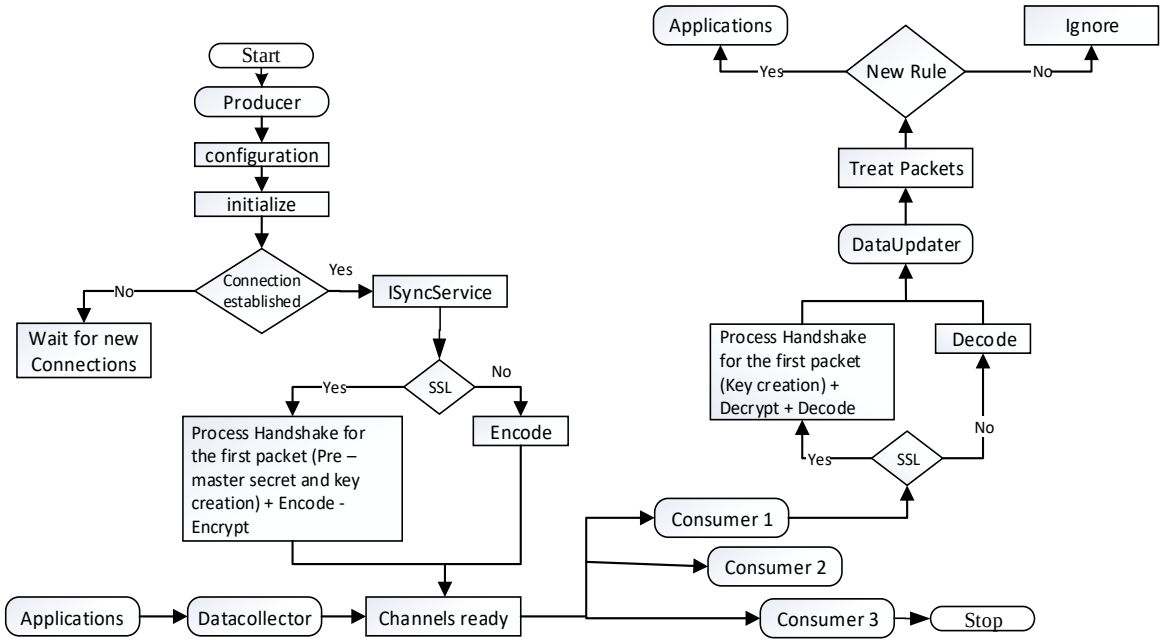

Figure 3.4: Flowchart mCIDC

3.3.5 Information Exchange by ISyncService

The ISyncService allows the controllers to share and exchange state information. The Producer of the CIDC is linked with the floodlight ISyncService to share link discovery and topology information across every module in the controller, and with neighboring controllers. The link discovery and topology updates are store in the StoreLD and StoreTopo, respectively and is accessed by the controllers. Figure 3.5 shows a snippet of mCIDC code.

```
try {
        Producer.syncService.registerStore("LDUpdates", Scope.GLOBAL);
        Producer.storeLD = Producer.syncService.getStoreClient("LDUpdates", String.class,
String.class);
        Producer.storeLD.addStoreListener(this);
} catch (SyncException e) {
        throw new FloodlightModuleException("Error while setting up sync service", e);
}
Producer.ldhaworker = new LDHAWorker(Producer.storeLD, controllerID);
linkserv.addListener(Producer.ldhaworker);
haworker.registerService("LDHAWorker", Producer.ldhaworker);
```

Figure 3.5: Snippet of mCIDC codes

3.3.6 Development of Network Policies

Network policies are developed, configured and set on the different WAN. These policies are provided by Access Control Lists (ACLs) and simple Quality of Service (QoS) settings. Each VM (WAN) contains different network policies. Appendix H contains the network policies for this research work.

3.3.7 Experimental Description

The VMs are powered "ON" from the GNS3 interface, starting all three VMs. Each VM represents a WAN domain. Each domain controller is launched from eclipse using floodlight launch (Appendix I). The network topology in each domain is started using Mininet (from the CLI) as shown in Figure 3.6. OpenFlow 1.3 is used as the southbound protocol. After the successful establishment of OpenFlow connections between the nodes (hosts and switches) and the controllers, then new traffic is injected into the network using the Iperf tool, for a default time of 10 seconds. This is achieved by simulating a scenario

where switches must request their controller for new events processing, by fixing one of the emulated hosts as a server (host 14) and the remaining hosts as clients. New flow from the client passes through the switch; sending a packet-In message to its controller. The controller installs forwarding rule in the switch flow table. These operations result in network changes, causing the controller to synchronize with its neighboring controllers. Wireshark analyzer was used for network analysis. This experimental description is carried out on CIDC and mCIDC; without network policies (notification mode) and with network policies (full mode). The Experiment is carried out on the network topology as shown in Table 3.1. The modules in both CIDC and mCIDC are created using the steps in Appendix J.

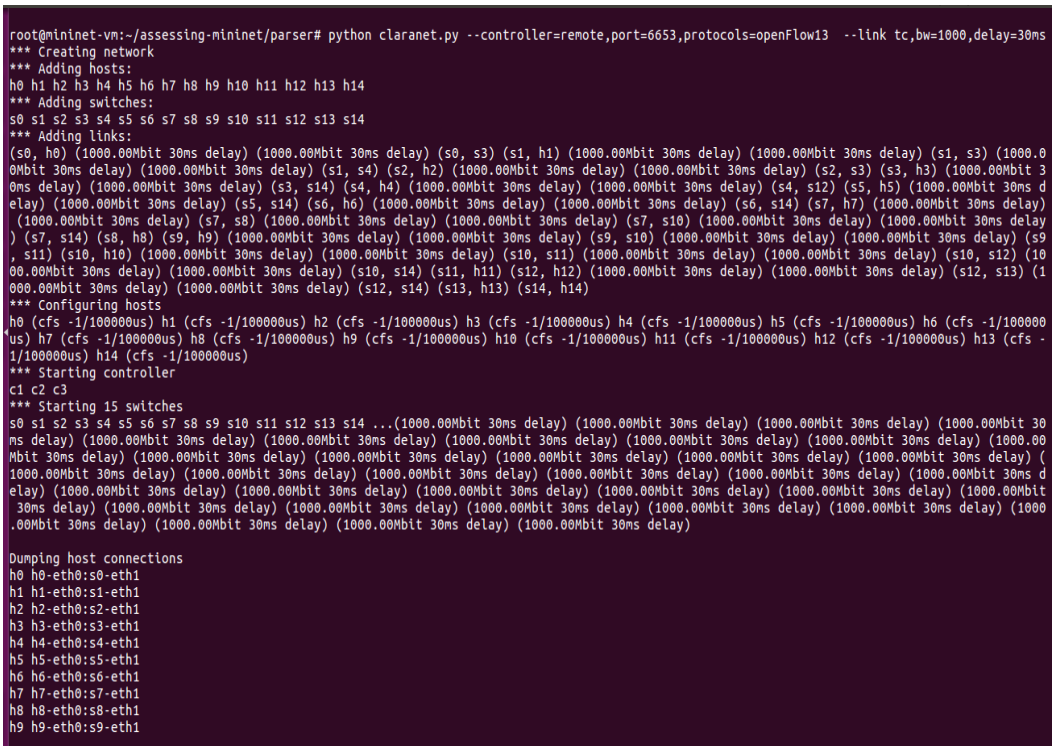

Figure 3.6: Start Network Topology with Mininet

Table 3.1: Evaluated Network Topology

Topology	Pseudo-name	Domain	Nodes per domain	Nodes
Claranet	Claranet_2	2	15	30
Claranet	Claranet_3	3	15	45

3.3.8 Performance Evaluation

The performance metrics are combined with network policies to evaluate the performance of the controller in policy updating and decision making. The performance metrics include captured TCP packets, TCP errors and ICO. This performance metrics are used to compare the CIDC and mCIDC, with and without network policies. The percentage improvement from the performance of mCIDC over CIDC was obtained using equation (3.1).

$$Percentage\ Improvement = \frac{Parameter\_CIDC\ -\ Parameter\_mCIDC}{\text{Parameter\_CIDC}} \qquad 3.1$$

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction

In this chapter, the results and discussion for the research work are presented. The performance of the CIDC and mCIDC are evaluated using the performance metrics as discussed in subsection 3.3.8. The performances with and without network policies are reported appropriately.

4.2 Experimental Connection

The connection of the floodlight controllers across the VMs launched inside Eclipse Floodlight Interface is shown in Figure 4.1.

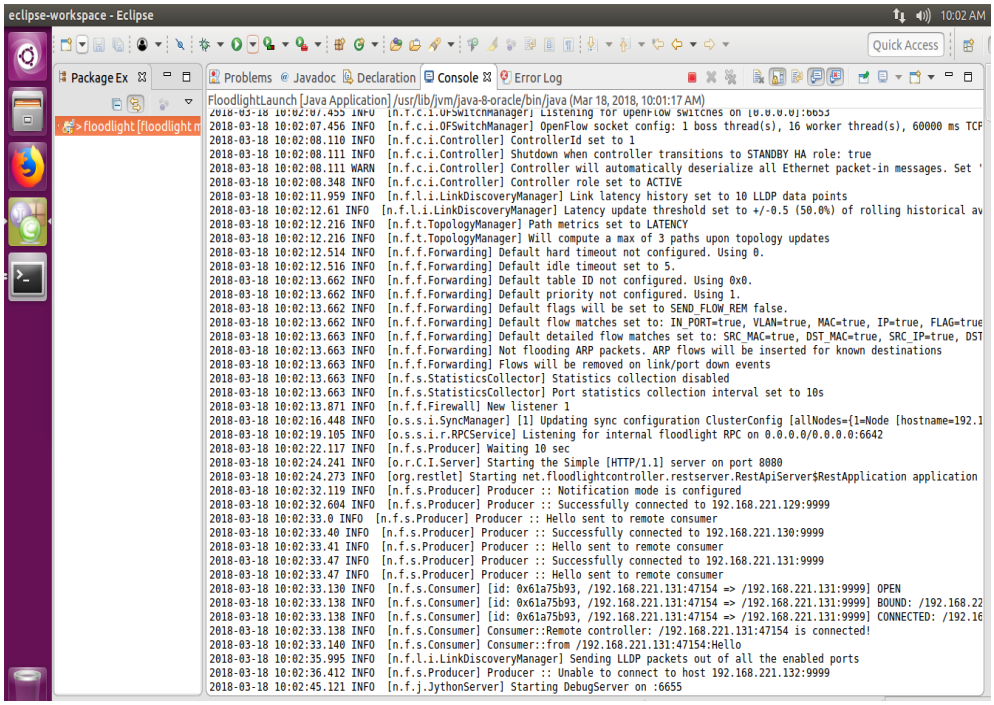

Figure 4.1: Connection of Three controllers on Eclipse Floodlight Interface

4.3 Performance Analysis without Network Policies

This subsection reports the performance analysis for CIDC and mCIDC without network policies. Results for both CIDC and mCIDC interface are presented here.

4.3.1 Result Analysis of CIDC

The results for CIDC Claranet_2 and Claranet_3 network topology are presented in Table 4.1 and 4.2 respectively. Figures 4.2 and 4.3 show the plots of captured TCP packets against TCP errors for Claranet_2 and Claranet_3 respectively. It observed that as the number of packets increases, the TCP errors generated during transmission also increases. Also, the ICO decreases with an increase TCP errors.

Table 4.1: Result Analysis of CIDC Claranet_2 without Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	10,156	127	97
2	10,466	176	87
3	10,685	193	79
4	10,693	199	77
5	10,700	203	74
6	10,928	256	70
7	11,178	290	65
8	11,182	292	65
9	11,381	315	61
10	11,541	410	59

Table 4.2: Result Analysis of CIDC Claranet_3 without Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	13,409	140	155
2	13,590	185	149
3	13,603	190	145
4	14,062	218	136
5	14,153	229	130
6	14,400	416	124
7	14,408	418	123
8	14,466	453	112
9	14,572	510	103
10	14,804	616	92

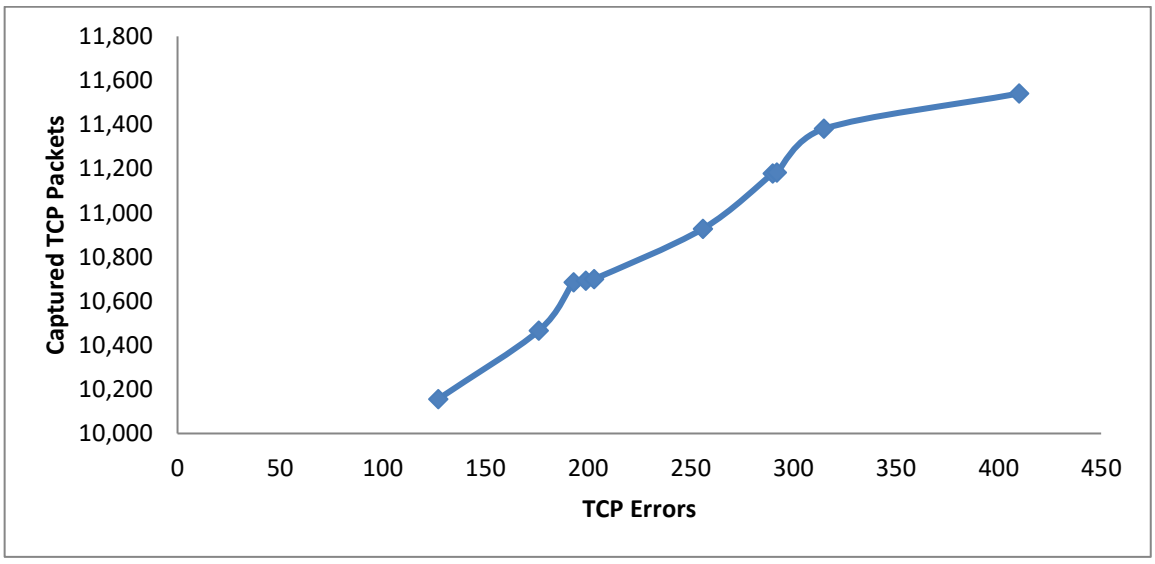

Figure 4.2: Plot of Captured TCP Packets against TCP Errors for CIDC Claranet_2 without Network Policies

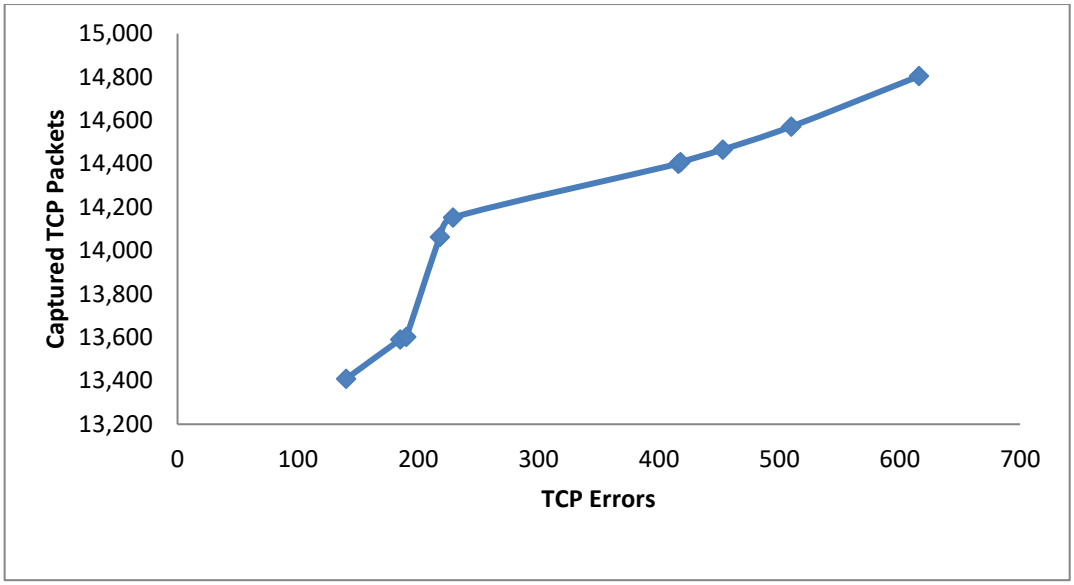

Figure 4.3: Plot of Captured TCP Packets against TCP Errors for CIDC Claranet_3 without Network Policies

Figures 4.4 and 4.5 show the plots of ICO against TCP Errors for Claranet_2 and Claranet_3, respectively. It can be seen that as the number of TCP Errors increases, the ICO decreases due to congestion of the TCP window (pipe).

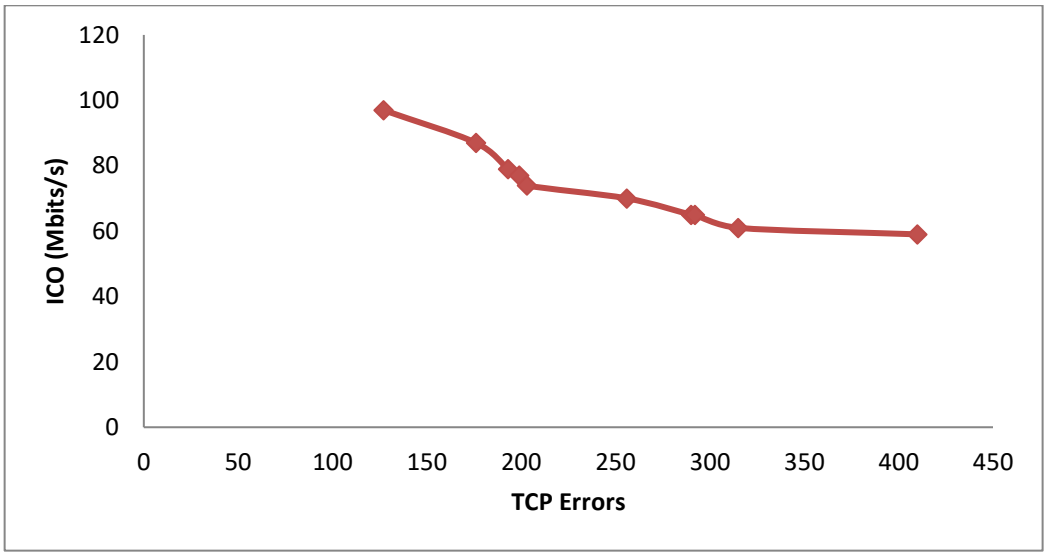

Figure 4.4: Plot of ICO against TCP Errors for CIDC Claranet_2 without Network Policies

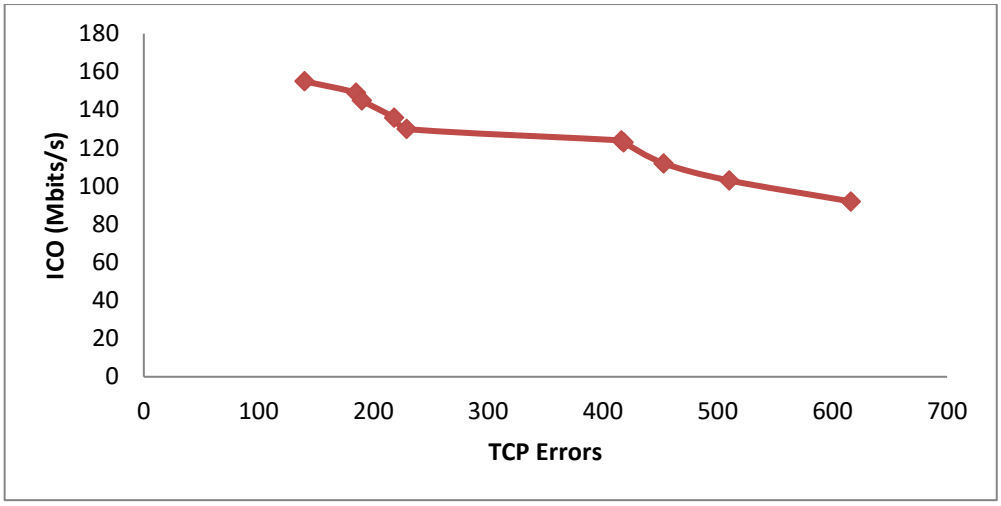

Figure 4.5: Plot of ICO against TCP Errors for CIDC Claranet_3 without Network Policies

4.3.2 Result Analysis for mCIDC

The results for mCIDC Claranet_2 and Claranet_3 network topology are presented in Table 4.3 and 4.4 respectively. Figures 4.6 and 4.7 show the plot of captured TCP packets against TCP errors for Claranet_2 and Claranet_3, respectively. It is observed that as the number of packets increases, the TCP errors generated during transmission also increases.

Table 4.3: Result Analysis of mCIDC Claranet_2 without Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	7,527	81	77
2	7,594	101	74
3	7,607	107	72
4	7,656	134	67
5	7,864	187	60
6	7,911	202	56
7	8,018	233	52
8	8,541	288	48
9	8,559	297	46
10	8,709	395	40

Table 4.4: Result Analysis of mCIDC Claranet_3 without Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	11,127	104	123
2	11,569	187	110
3	11,599	206	104
4	11,604	208	102
5	11,704	241	95
6	11,738	276	89
7	12,090	287	75
8	12,320	353	70
9	12,538	373	66
10	12,800	416	61

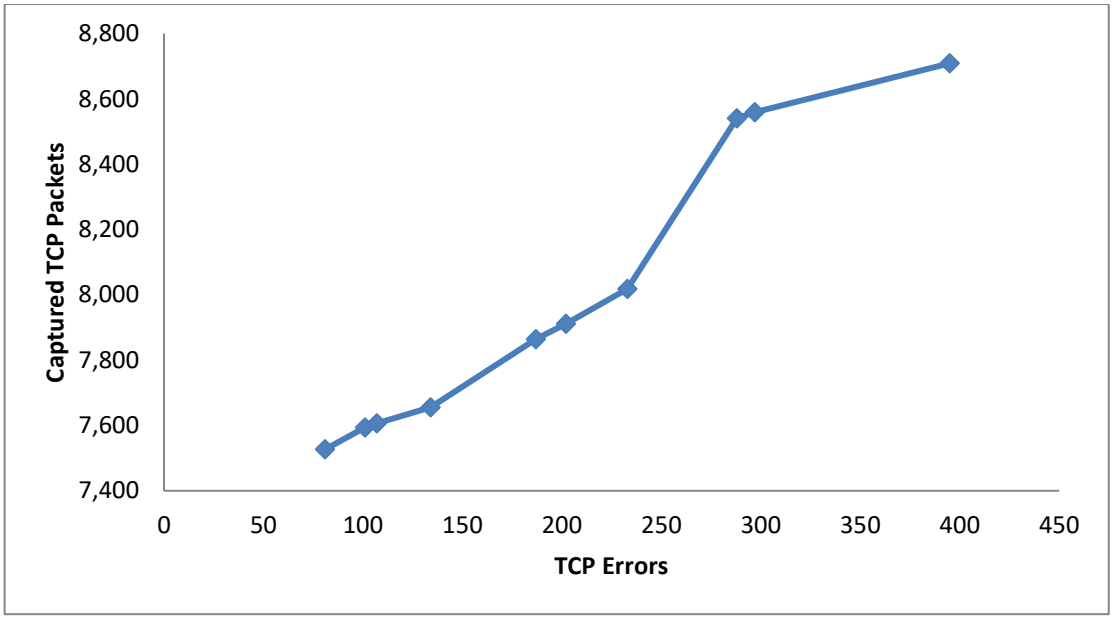

Figure 4.6: Plot of Captured TCP Packets against TCP Errors for mCIDC Claranet_2 without Network Policies

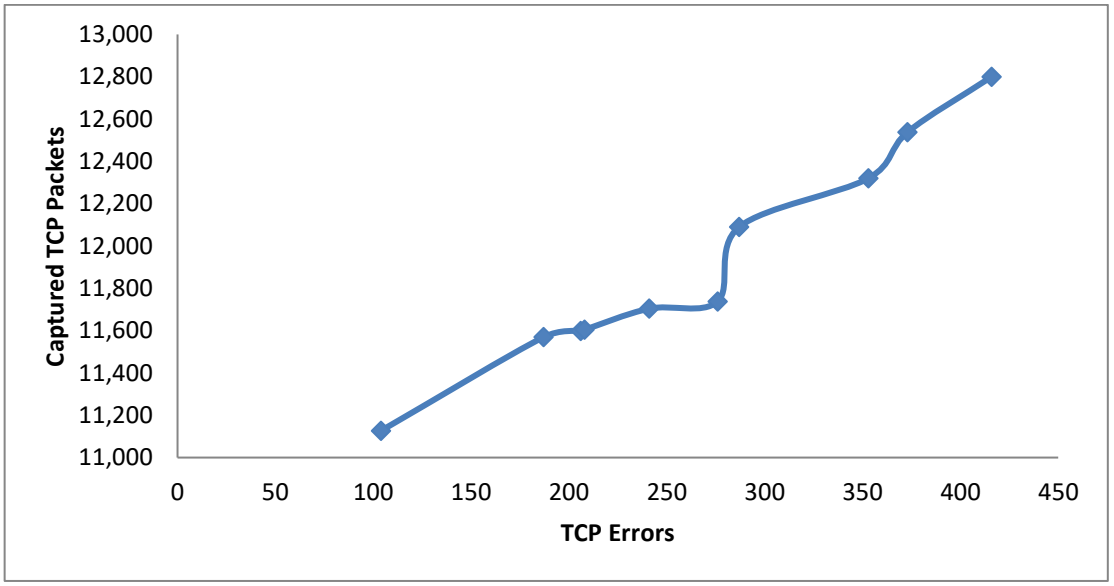

Figure 4.7: Plot of Captured TCP Packets against TCP Errors for mCIDC Claranet_3 without Network Policies

Figures 4.8 and 4.9 show the plots of ICO against TCP errors for Claranet_2 and Claranet_3, respectively. It is observed that as the number of TCP errors increases, the ICO decreases due to congestion of the TCP window.

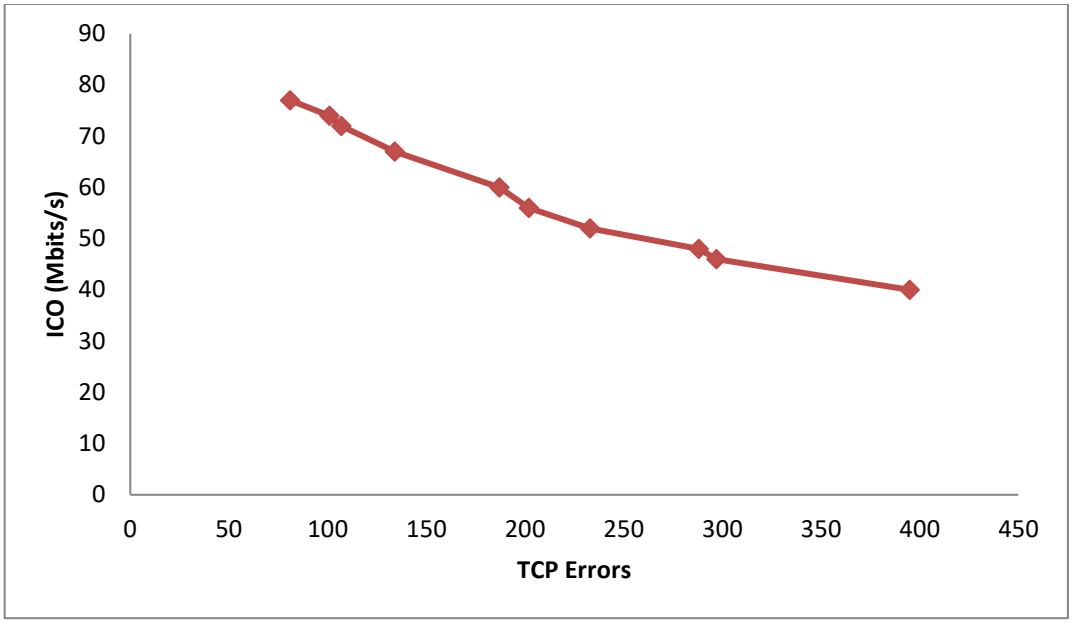

Figure 4.8: Plot of ICO against TCP Errors for mCIDC Claranet_2 without Network Policies

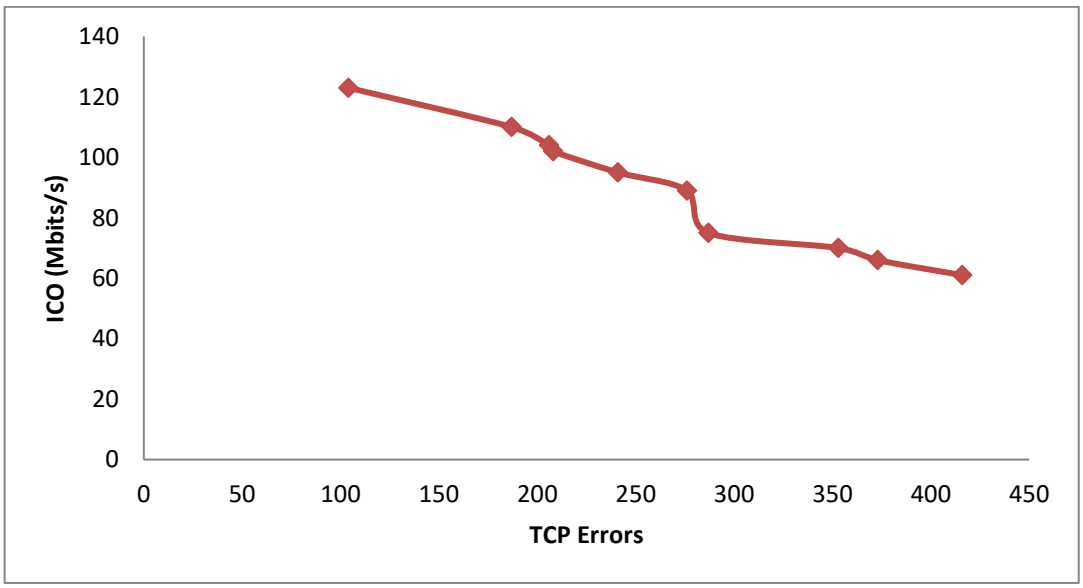

Figure 4.9: Plot of ICO against TCP Errors for mCIDC Claranet_3 without Network Policies

4.4 Performance Analysis with Network Policies

This subsection reports the performance analysis for CIDC and mCIDC with network policies. Results for both CIDC and mCIDC interface are presented here.

4.4.1 Result Analysis of CIDC

The results for CIDC Claranet_2 and Claranet_3 network topology are presented in Table 4.5 and 4.6 respectively. Figures 4.10 and 4.11 show the plot of captured TCP packets against TCP errors for Claranet_2 and Claranet_3, respectively. It is observed that as the number of packets increases, the TCP errors generated during transmission also increases.

Table 4.5: Result Analysis of CIDC Claranet_2 with Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	16,271	483	225
2	16,347	518	213
3	16,441	578	207
4	16,553	616	200
5	16,583	642	196
6	16,620	669	191
7	16,724	705	184
8	16,729	709	182
9	16,732	711	181
10	16,764	729	175

Table 4.6: Result Analysis of CIDC Claranet_3 with Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	19,332	729	284
2	20,244	857	269
3	20,304	905	262
4	20,318	911	260
5	20,354	934	256
6	20,528	969	248
7	20,721	999	240
8	20,890	1145	223
9	20,954	1193	215
10	20,971	1205	210

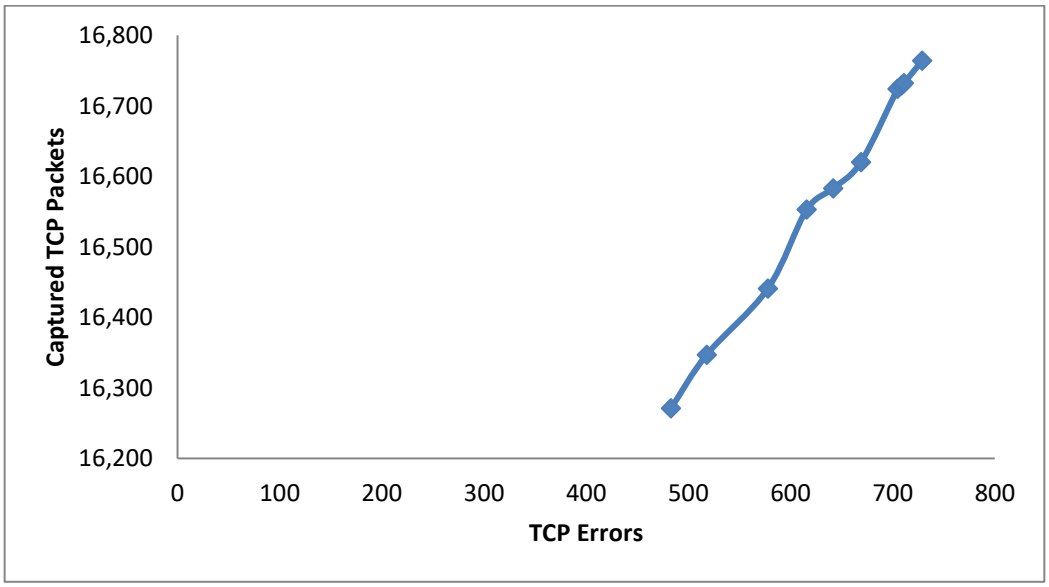

Figure 4.10: Plot of Captured TCP Packets against TCP Errors for CIDC Claranet_2 with Network Policies

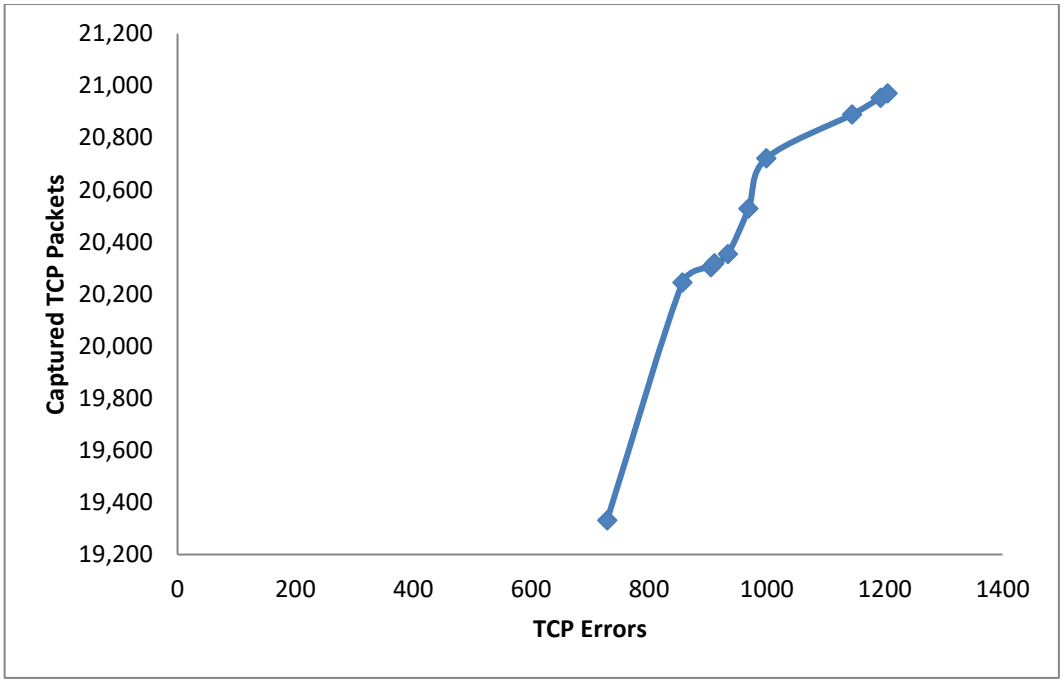

Figure 4.11: Plot of Captured TCP Packets against TCP Errors for CIDC Claranet_3 with Network Policies

Figures 4.12 and 4.13 show the plots of ICO against TCP Errors for Claranet_2 and Claranet_3, respectively. It can be seen that as the number of TCP Errors increases, the ICO decreases due to congestion of the TCP window.

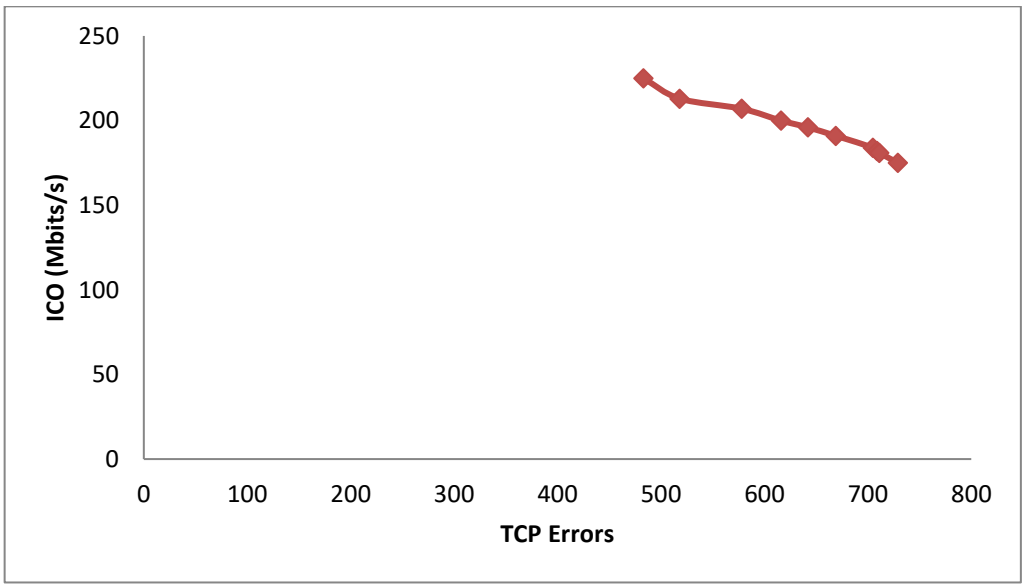

Figure 4.12: Plot of ICO against TCP Errors for CIDC Claranet_2 with Network Policies

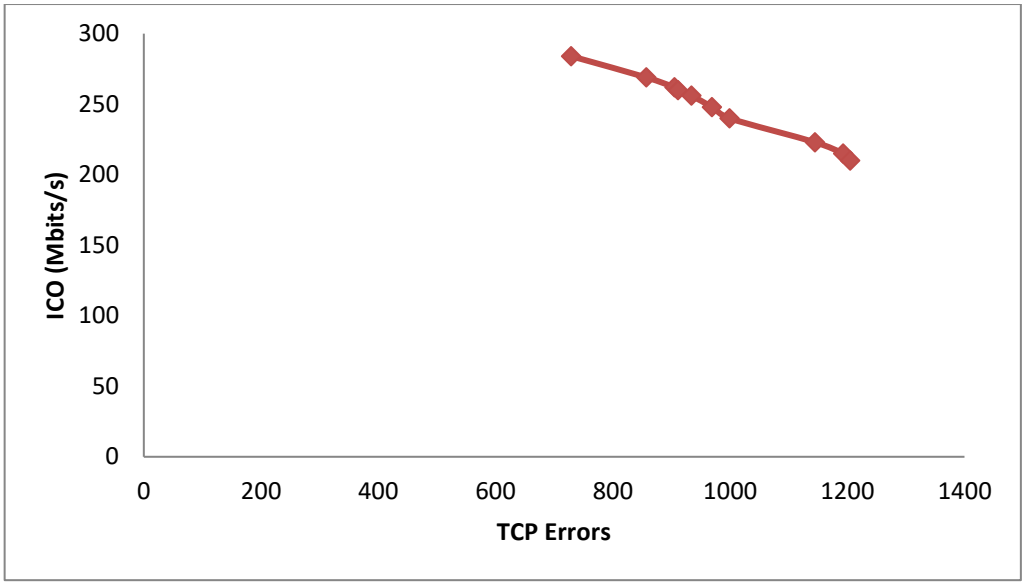

Figure 4.13: Plot of ICO against TCP Errors for CIDC Claranet_3 with Network Policies

4.4.2 Result Analysis of mCIDC

The results for mCIDC Claranet_2 and Claranet_3 network topology are presented in Table 4.7 and 4.8 respectively. Figures 4.14 and 4.15 show the plot of captured TCP packets against TCP errors for Claranet_2 and Claranet_3, respectively. It is observed that as the number of packets increases, the TCP errors generated during transmission also increases.

Table 4.7: Result Analysis of mCIDC Claranet_2 with Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	13,099	310	148
2	13,147	348	142
3	13,521	417	130
4	13,640	453	122
5	13,707	483	115
6	13,946	511	102
7	14,335	642	89
8	14,381	669	80
9	14,386	672	78
10	14,505	750	69

Table 4.8: Result Analysis of mCIDC Claranet_3 with Network Policies

S/N	Captured TCP Packets	TCP Errors	ICO (Mbits/s)
1	16,271	481	225
2	16,845	569	213
3	16,935	578	211
4	16,959	598	206
5	17,003	629	198
6	17,340	687	190
7	17,794	783	177
8	17,841	817	169
9	17,911	859	160
10	17,983	911	151

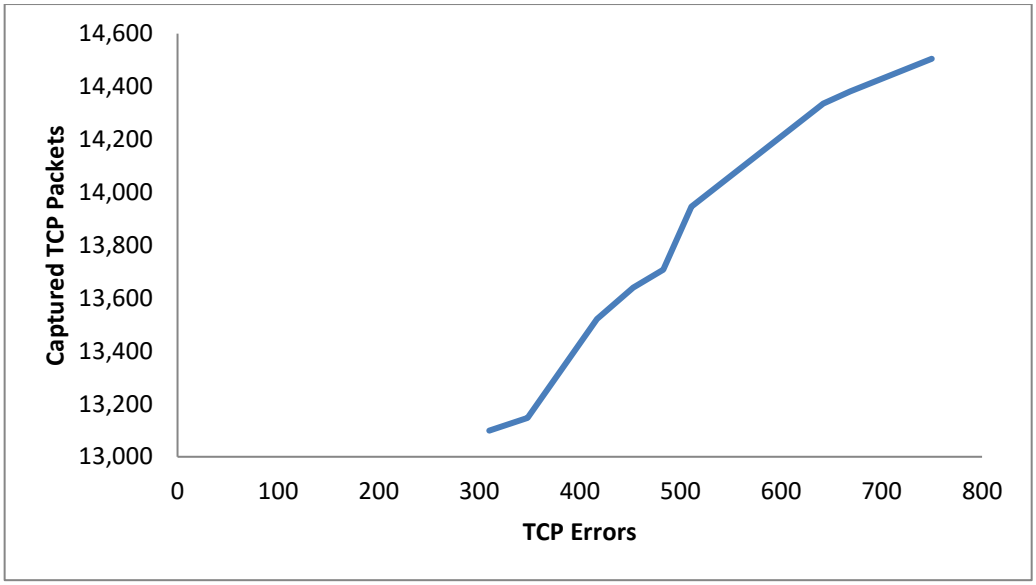

Figure 4.14: Plot of Captured TCP Packets against TCP Errors for mCIDC Claranet_2 with Network Policies

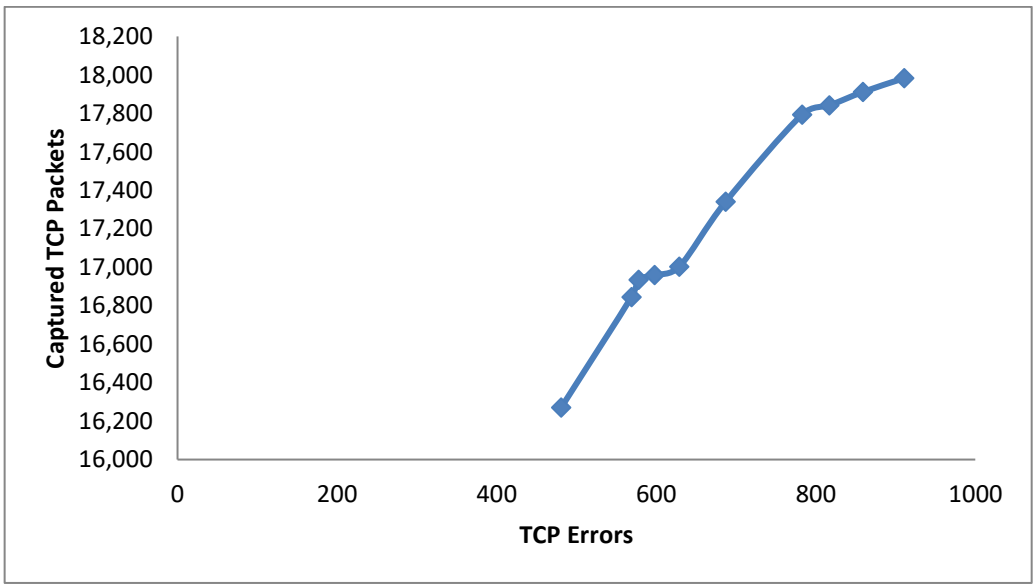

Figure 4.15: Plot of Captured TCP Packets against TCP Errors for mCIDC Claranet_3 with Network Policies

Figures 4.16 and 4.17 show the plots of ICO against TCP errors for Claranet_2 and Claranet_3, respectively. It is observed that as the number of TCP errors increases, the ICO decreases due to congestion of the TCP window.

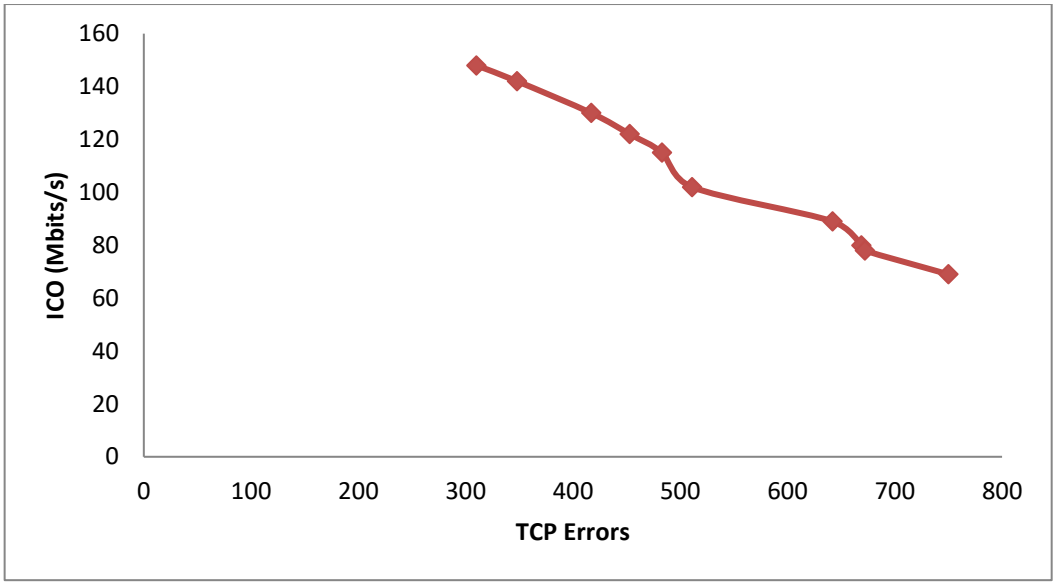

Figure 4.16: Plot of ICO against TCP Errors for mCIDC Claranet_2 with Network Policies

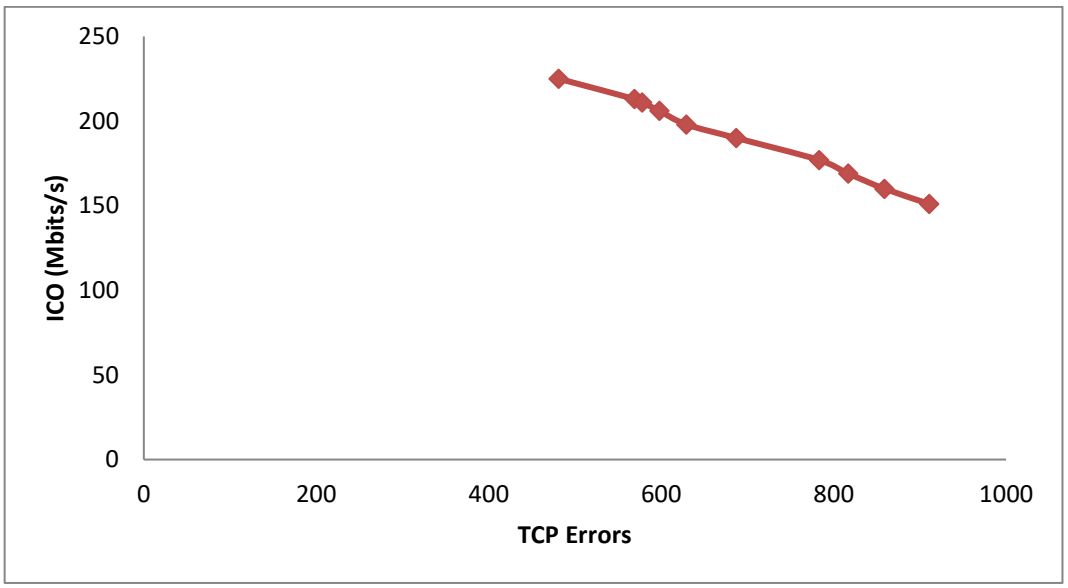

Figure 4.17: Plot of ICO against TCP Errors for mCIDC Claranet_3 with Network Policies

4.5 Performance Percentage Improvement of mCIDC over CIDC

The performance improvement of mCIDC over CIDC is determined by taking the average results of each performance metrics for both Claranet_2 and Claranet_3 network topology. The performance improvements are evaluated using equation (3.1).

4.5.1 Comparison of CIDC and mCIDC without Network Policies

The plots obtained from the analysis of CIDC and mCIDC are compared using Figures 4.2 to 4.9 and Tables 4.1 to 4.4. Table 4.9 shows the performance improvement of mCIDC over CIDC without network policies.

Table 4.9: Performance Improvement of mCIDC over CIDC without Network Policies

Performance Metrics	**Claranet_2**			**Claranet_3**		
	CIDC	**mCIDC**	**Percentage Improvement (%)**	**CIDC**	**mCIDC**	**Percentage Improvement (%)**
Average Captured TCP Packets	10,891	7999	26.55	14,147	11,909	15.82
Average TCP Errors	246	202	17.89	338	265	21.60
Average ICO	73.4	59.2	19.35	126.5	89.5	29.25

It is observed that without network policies, the percentage improvement of mCIDC over CIDC for Claranet _2 are 26.55%, 17.89%, and 19.35% for captured TCP packets, TCP errors and ICO respectively. While for Claranet_3, improvements are 15.82%, 21.60%, and 29.25% for captured TCP packets, TCP errors and ICO, respectively. This shows that mCIDC transmits the required necessary information between controllers with reduced TCP errors and ICO.

4.5.2 Comparison of CIDC and mCIDC with Network Policies

The plots obtained from the analysis of CIDC and mCIDC are compared using Figures 4.10 to 4.17 and Tables 4.5 to 4.8. Table 4.10 shows the performance improvement of mCIDC over CIDC with network policies.

Table 4.10: Performance Improvement of mCIDC over CIDC with Network Policies

Performance Metrics	**Claranet_2**			**Claranet_3**		
	CIDC	**mCIDC**	**Percentage Improvement (%)**	**CIDC**	**mCIDC**	**Percentage Improvement (%)**
Average Captured TCP Packets	16,576	13,867	16.34	20,462	17,288	15.51
Average TCP Errors	636	523	17.77	985	691	29.85
Average ICO	195.4	107.5	44.99	246.7	190	22.98

It is observed that with network policies, the percentage improvement of mCIDC over CIDC for Claranet _2 are 16.34%, 17.77%, and 44.99% for captured TCP packets, TCP errors and ICO, respectively. While for Claranet_3, improvements are 15.51%, 29.85%, and 22.98% for captured TCP packets, TCP errors and ICO, respectively. This shows that the mCIDC transmits the required information (reduced captured TCP packets) between neighboring controllers with reduced TCP errors and ICO.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

5.1 Summary

The East-West interface is very important in large networks or multi-domain networks with multiple controllers (such as WAN). The main purpose of East –West interface was to synchronize states for high availability, by enabling communication between groups or federations of controllers. This was achieved through monitoring /notification capabilities and exchange of data between controllers. The CIDC was developed to provide common compatibility and interoperability between different floodlight controllers, in a distributed control plane network (WAN network). The CIDC did not synchronize with all the modules of the floodlight controller, leading to inadequate consistent high availability and inefficient policy updates among controllers. In order to overcome this problem, the mCIDC with ISyncService was developed. The ISyncService provided high availability by synchronizing states among different modules in the controller and across different controllers in the WAN. The synchronized states included link discovery and topology. The CIDC and mCIDC were developed on an Ubuntu 16.04 LTS VM using Eclipse Oxygen 4.7. Analyses were carried out on the performance of the CIDC and mCIDC using the following performance metrics: captured TCP packets, TCP errors and ICO. The performance metrics were evaluated with and without network policies. The results indicated that for Claranet_2, the performance of mCIDC in minimizing number of captured TCP packets, TCP errors and ICO was better than CIDC by 36.16%, 21.78%, and 23.99% respectively. While for Claranet_3, the performance of mCIDC was better than CIDC by 18.79%, 27.55%, and 41.34% for captured TCP packets, TCP errors and ICO, respectively. For network policies with Claranet_2, mCIDC indicated better performance than CIDC in minimizing captured TCP packets, TCP errors and ICO by 19.54%, 21.61%,

and 18.17%, respectively. While network policies with Claranet_3, the mCIDC was better by 18.36%, 42.55%, and 29.84% for captured TCP packets, TCP errors and ICO, respectively.

5.2 Conclusion

The research developed the mCIDC with ISyncService for consistent high availability among controllers by minimizing number of TCP packets, TCP errors, and ICO needed for communication among multiple controllers in a WAN. This also helps to improve policy updating and decision making in the network. The comparisons of the performance of mCIDC and CIDC were carried out for captured TCP packets, TCP errors and ICO. The mCIDC outperformed the CIDC in all the metrics.

5.3 Limitation

The research work was unable to prevent the *Open vSwitch* from disconnecting and connecting during transmission among controllers. Hence, there was the need to run ten experiments per scenario, for accurate measurements.

5.4 Significant Contributions

A lot of research work has been done on improving East-West communication in a distributed control plane but most are focus on enterprise and intra-domain networks. The significant contributions of this research work are as follows:

a) Development of a mCIDC with ISyncService for communication in multiple WANs, while providing high availability and improved policy updating and decision-making.
b) The developed mCIDC showed a performance improvement without network policies over CIDC with Claranet_2 having a 26.55%, 17.89% and 19.35% for captured TCP packets, TCP errors and ICO. While for Claranet_3, improvements

were 15.82%, 21.60%, and 29.25% for captured TCP packets, TCP errors and ICO, respectively. Also, a performance improvement with network policies for Claranet_2; were 16.34%, 17.77%, and 44.99% for captured TCP packets, TCP errors and ICO. While for Claranet_3, improvements were 15.51%, 29.85%, and 22.98% for captured TCP packets, TCP errors and ICO, respectively.

5.5 Recommendations for Further Work

The following possible areas of further work are recommended for consideration for future research:

a) The application of the developed controller on a real network with multiple WANs to determine its scalability when other WAN parameters are considered.
b) The research only considered link discovery and topology for synchronization states, more states can be synchronized in further works.

REFERENCES

Benamrane Fouad, Ben Mamoun Mouad & Benaini Redouane (2017). An East-West interface for distributed SDN control plane: Implementation and evaluation. *Computers and Electrical Engineering*, 57, 162-175. http://dx.doi.org/10.1016/j.compeleceng.2016.09.012

Berde P., Gerola M., Hart J., Higuchi Y., Kobayashi M., Koide T., Lantz B., O'Connor B., Radoslavov P., Snow W., & Parulkar G. (2014). ONOS: towards an open, distributed SDN OS. Paper presented at the *Proceedings of the third workshop on hot topics in software defined networking*, (pp. 1-6) ACM.

Bilal S., Shieh M., Tung L., & Paul Lin B. (2016). DMC: Distributed Approach in Multi-Domain Controllers. *Proceedings of the Eleventh International Network Conference* (INC), pp. 31-36.

Blial O., Ben Mamoun M., & Benaini R. (2016). An Overview on SDN Architectures with Multiple Controllers. *Journal of Computer Networks and Communications, Article ID* 9396525, 8 pages, http://dx.doi.org/10.1155/2016/9396525

Chen J., Zheng X., & Rong C. (2015). Survey on Software Defined Networking. In: Qiang W., Zheng X., Hsu CH. (eds) *Cloud Computing and Big Data. Lecture Notes in Computer Science, vol 9, pg 106. Springer*, Cham.

Dixit A., Hao F., Mukherjee S., Lakshman T., & Kompella R. (2013). Towards an elastic distributed SDN controller. Paper presented at the *ACM SIGCOMM Computer Communication Review* (vol. 43, No.4, pp. 7-12). ACM.

Egilmez H. E. (2014). Distributed QoS architectures for multimedia streaming over software defined networks. IEEE Transactions on Multimedia, 16(6), 1597-1609

Farhady H., Lee H., & Nakao A. (2015). Software-Defined Networking: A survey. *Computer Networks*, 81, 79-95. http://dx.doi.org/10.1016/j.comnet.2015.02.014

Feamster N., Rexford J., & Zegura E. (2013). The road to SDN: An Intellectual History of Programmable Networks. *ACM SIGCOMM Computer Communication Review*, 44(2), 87-98.

Fu Y., Bi J., Chen Z., Gao K., Zhang B., Chen G., & Wu J. (2015). A Hybrid Hierarchical Control Plane for Flow-Based Large-Scale Software-Defined Networks. IEEE *Transactions on Network and Service Management*, 12(2), 117-131.

Jain S.,, Kumar A., Mandal S., Ong J., Poutievski L., Singh A., Venkata S., Wanderer J., Zhou J., Zhu M., Zolla J., Hölzle U., Stuart S., & Vahdat A. (2013). B4:

Experience with a Globally-Deployed Software Defined WAN. In *ACM SIGCOMM Computer Communication Review* (vol. 43, No.4, pp. 3-14).

Jammal M., Singh T., Shami A., Asal R., & Li Y. (2014). Software defined networking: State of the art and research challenges. *Computer Networks*, 72, 74-98. doi: http://dx.doi.org/10.1016/j.comnet.2014.07.004

Jarraya Y., Madi T., & Debbabi M. (2014). A Survey and a Layered Taxonomy of Software-Defined Networking. Communications Surveys &Tutorials, IEEE, 16(4), 1955-1980. doi: 10.1109/COMST.2014.2320094

Jarschel M., Zinner T., HoQfeld T., Tran-Gia P., & Kellerer W. (2014). Interfaces, attributes, use cases: A compass for SDN. IEEE *Commun. Mag*., vol. 52, no. 6, pp. 210–217.

Koponen T., Casado M., Gude N., Stribling J., Poutievski L., Zhu M., Ramanathan R., Iwata Y., Inuoe H., Hama T., & Shenker S. (2010). Onix: A Distributed Control Platform for Largescale Production Networks. Paper presented at *Proceedings of the 9th USENIX conference on operating systems design and implementation* (OSDI10); p. 1–6.

Kreutz D., Ramos F., & Verissimo P. (2013). Towards Secure and Dependable Software-Defined Networks. *In Proceedings of the second ACM SIGCOMM Workshop on Hot Topics in software defined networking* (pp. 55-60). ACM.

Kreutz D., Ramos F. M. V., Esteves Verissimo P., Esteve Rothenberg C., Azodolmolky S., & Uhlig S. (2015). Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE, 103(1), 14-76. doi: 10.1109/JPROC.2014.2371999

Krishnamurthy A., Chandrabose S. P., & Gember-Jacobson A. (2014). Pratyaastha: an efficient elastic distributed SDN control plane. Paper presented at the *Proceedings of the third workshop on hot topics in software defined networking*, Chicago, Illinois, USA. http://dl.acm.org/citation.cfm?doid=2620728.2620748.

Lantz B., Heller B., & McKeown N. (2010). A network in a laptop: rapid prototyping for software-defined networks. *In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, (p. 19). ACM.

Levin D., Wundsam A., Heller B., Handigol N., & Feldmann A. (2012). Logically centralized? State distribution trade-offs in software defined networks," in: *Proceedings of ACM on Hot Topics in Software Defined Networking (HotSDN)* Helsinki, Finland.

Lin P., Bi J., and Wang Y. (2013). East-West Bridge for SDN network peering. *Frontiers in internet technologies Springer Berlin Heidelberg*; p.170–81.

Masoudi Rahim and Ghaffari Ali (2016). Software defined networks: A survey. *Journal of Network and Computer Applications*, 67, 1-25. http://dx.doi.org/10.1016/j.jnca.2016.03.016

Mckeown, N., Anderson, T., Balakrishnan, H.,Parulkar, G., Peterson, L., Rexford, J., Shenker, S., And Turner, J. (2008). OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR* 38, 2, 69–74.

Nunes B. A. A., Nguyen X.-N., Turletti T., Mendonca M., & Obraczka K. (2014). A survey of software-defined networking: Past, present, and future of programmable networks. IEEE *Communications Surveys and Tutorials*, 16(3), 1617-1634. doi: 10.1109/SURV.2014.012214.00180

Oktian Y., Lee S., Lee H., & Lam J. (2017). Distributed SDN Controller System: A Survey on Design Choice. *Computer Networks*, 121, 100-111. doi: 10.1016/j.comnet.2017.04.038.

Open Networking Foundation (ONF), ‘‘SDN architecture,’’ Jun. 2014. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/ technical-reports/TR_SDN_ARCH_1.0_6062014.pdf

Phemius K., Bouet M., & Leguay J. (2014). Disco: Distributed Multi-Domain SDN Controllers. *Network operations and management symposium (NOMS)* IEEE; p.1–4.

Ryan Wallner & Robert Cannistra (2013). An SDN approach: Quality of Service using Big Switch’s Floodlight Open-source Controller. in *Proceedings of the Asia-Pacific Advanced Network*, Vol. 35, pp. 14-19. doi:http://dx.doi.org/10.7125/APAN.35.2

Shuhao Liu & Baochun Li (2015). On Scaling Software-Defined Networking in Wide-Area Networks. *Tsinghua Science and Technology*, ISSN 1007-0214 01/10, pp. 221-232, Vol. 20, No. 3.

Stancu A., Halunga S., Vulpe A., Suciu G., Fratu O., & Popovici E. (2015). A Comparison between Several Software Defined Networking Controllers. in *Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), 12th International Conference.*

Tootoonchian A. & Ganjali Y. (2010). HyperFlow: a distributed control plane for OpenFlow. *Proceedings of the 2010 internet network management conference on research on enterprise networking*, (p. 3-3).

Vijay Poonam & Vasudevan Deepika (2016). The Northbound APIs of Software Defined Networks. *International Journal of Engineering Sciences & Research Technology (IJSERT) ISSN: 2277-9655*.

Wibowo F., Gregory M., Ahmed K., & Gomez K. (2017). Multi-Domain Software Defined Networking: Research Status and Challenges. *Journal of Network and Computer Applications*, 87, 32-45. http://dx.doi.org/10.1016/j.jnca.2017.03.004

Xia W., Wen Y., Foh C., Niyato D., & Xie H. (2015). A Survey on Software-Defined Networking. IEEE Communications Surveys & Tutorials, 17(1), 27-51. doi: 10.1109/COMST.2014.2330903.

Xie J., Guo D., Hu Z., Qu T., & Lv P (2015). Control plane of software defined networks: A survey. *Computer Communications*, 67, 1-10. http://dx.doi.org/10.1016/j.comcom.2015.06.004

Yeganeh S. H. & Ganjali Y., (2012). Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. *Proc. HotSDN '12 Wksp*., pp. 19–24.

Yin H., Xie H., Tsou T., Lopez D., Aranda P., & Sidi R. (2012). SDNi: A Message Exchange Protocol for Software Defined Networks (SDNs) across Multiple Domains. from https://datatracker.ietf.org/doc/draft-yin-sdn-sdni/

APPENDICES

APPENDIX A

Installation of VMware Workstation 12 Pro

Steps

1. Download the VMware workstation from VMware, and double click on it.
2. Accept the License Agreement and click Next to continue.
3. Choose the installation path where you want to install. Click Next to continue
   Note: You can also install feature Enhanced Keyboard Driver by checking box.
4. You can check additional options and Click Next to continue.
5. Choose where you want to create the shortcut and Click Next to continue.
6. Click Install to start the installation process.
7. After Installation; click on the License to install the License or Click Finish.
8. On Clicking the License; insert the License Key or choose I want to try VMware Workstation for 30 days. Enter your email address and Click Next to continue.
9. Then Click Finish to close the installer.

Installation of Ubuntu 16.04 LTS

Steps

1. Download Ubuntu 16.04 LTS desktop image.
2. Open VMware Workstation 12 Pro.
3. Launch VMware Workstation New Virtual Machine (VM) installation wizard.
4. On the New Virtual Machine Wizard box, select Typical (default option) and click Next.
5. Select the installation media or source by browsing to the downloaded Ubuntu ISO file, and click Next.
6. Enter your details in the personalize Linux dialog box. This includes Name, Username and password (your Ubuntu login credentials).
7. Enter the VM Name and Location.
8. Specify disk capacity
9. On the ready to create VM dialog box; click on Customize Hardware (Memory, Network Adapters and Other devices) or leave them as defaults.
10. Check the box Power on the VM after creation. Else, Power on the VM manually.
11. Ubuntu OS installation Starts – Enter username and password to login.

APPENDIX B

Floodlight Controller Installation

The following steps where used to install floodlight controller:

1. Open Ubuntu CLI (ALT+CTRL+T).
2. Install dependencies for Floodlight Master:
   $ sudo apt-get install build-essential ant maven python-dev
3. Install git: $ sudo apt install git
4. Install Floodlight from github with the following commands (sequentially):
   $ git clone git://github.com/floodlight/floodlight.git
   $ cd floodlight
   $ git submodule init
   $ git submodule update
   $ ant

Note: Floodlight can be run on the terminal (CLI): $ java –jar target/floodlight.jar

Floodlight can be developed in eclipse: by manually starting eclipse or running on the CLI
$ ant eclipse

APPENDIX C

Mininet Installation

The following steps were used to install Mininet from the CLI:

1. Open the Ubuntu VM CLI.
2. sudo apt-get install Mininet
3. sudo mn –c
4. sudo apt-get install git
5. git clone git://github.com/mininet/mininet
6. cd mininet
7. git tag (this displays the available versions of Mininet)
8. git checkout –b version (replace version with the selected version i.e. 2.2)
9. cd
10. mininet/util/install.sh –a
11. sudo mn

APPENDIX D

Importing Network Topology

These are the steps for importing the network topology from Internet Topology Zoo, and converting it to python (.py) format:

1. Open the CLI
2. cd mininet
3. git clone https://github.com/uniba-ktr/assessing-mininet.git
4. cd assessing-mininet/parser
5. wget http://topology-zoo.org/files/Claranet.graphml (to import the topology)
6. ./GraphML-Topo-to-Mininet-Network-Generator.py –f Claranet.graphml –o Claranet.py (to convert to .py format)

Modified Claranet python Script

```
#!/usr/bin/python

"""

Custom topology for Mininet, generated by GraphML-Topo-to-Mininet-Network-Generator.

"""

from mininet.topo import Topo

from mininet.net import Mininet

from mininet.node import RemoteController, Controller

from mininet.node import Node, OVSSwitch

from mininet.node import CPULimitedHost

from mininet.link import TCLink, Intf

from mininet.cli import CLI

from mininet.log import setLogLevel

from mininet.util import dumpNodeConnections


class GeneratedTopo( Topo ):

    "Internet Topology Zoo Specimen."
```

```
def __init__( self, **opts ):
    "Create a topology."
    # Initialize Topology
    Topo.__init__( self, **opts )
    # add nodes, switches first...
    Faro = self.addSwitch( 's0', cls= OVSSwitch, protocols= OpenFlow13 )
    Madrid = self.addSwitch( 's1', cls= OVSSwitch, protocols= OpenFlow13 )
    Porto = self.addSwitch( 's2', cls= OVSSwitch, protocols= OpenFlow13)
    Lisbon = self.addSwitch( 's3', cls= OVSSwitch, protocols= OpenFlow13 )
    Barcelona = self.addSwitch( 's4', cls= OVSSwitch, protocols= OpenFlow13 )
    Manchester = self.addSwitch( 's5', cls= OVSSwitch, protocols= OpenFlow13 )
    NewYork = self.addSwitch( 's6', cls= OVSSwitch, protocols= OpenFlow13)
    Amsterdam = self.addSwitch( 's7', cls= OVSSwitch, protocols= OpenFlow13 )
    Eindhoven = self.addSwitch( 's8', cls= OVSSwitch, protocols= OpenFlow13 )
    Berlin = self.addSwitch( 's9', cls= OVSSwitch, protocols= OpenFlow13 )
    Frankfurt = self.addSwitch( 's10', cls= OVSSwitch, protocols= OpenFlow13 )
    Munich = self.addSwitch( 's11', cls= OVSSwitch, protocols= OpenFlow13 )
    Paris = self.addSwitch( 's12', cls= OVSSwitch, protocols= OpenFlow13 )
    Rennes = self.addSwitch( 's13', cls= OVSSwitch, protocols= OpenFlow13)
    London = self.addSwitch( 's14', cls= OVSSwitch, protocols= OpenFlow13 )
    # ... and now hosts
    Faro_host = self.addHost( 'h0' )
    Madrid_host = self.addHost( 'h1' )
    Porto_host = self.addHost( 'h2' )
    Lisbon_host = self.addHost( 'h3' )
    Barcelona_host = self.addHost( 'h4' )
    Manchester_host = self.addHost( 'h5' )
```

```
NewYork_host = self.addHost( 'h6' )
Amsterdam_host = self.addHost( 'h7' )
Eindhoven_host = self.addHost( 'h8' )
Berlin_host = self.addHost( 'h9' )
Frankfurt_host = self.addHost( 'h10' )
Munich_host = self.addHost( 'h11' )
Paris_host = self.addHost( 'h12' )
Rennes_host = self.addHost( 'h13' )
London_host = self.addHost( 'h14' )
# add edges between switch and corresponding host
self.addLink( Faro , Faro_host )
self.addLink( Madrid , Madrid_host )
self.addLink( Porto , Porto_host )
self.addLink( Lisbon , Lisbon_host )
self.addLink( Barcelona , Barcelona_host )
self.addLink( Manchester , Manchester_host )
self.addLink( NewYork , NewYork_host )
self.addLink( Amsterdam , Amsterdam_host )
self.addLink( Eindhoven , Eindhoven_host )
self.addLink( Berlin , Berlin_host )
self.addLink( Frankfurt , Frankfurt_host )
self.addLink( Munich , Munich_host )
self.addLink( Paris , Paris_host )
self.addLink( Rennes , Rennes_host )
self.addLink( London , London_host )

# add edges between switches
```

```
        self.addLink( Faro , Lisbon, bw=10, delay='30ms')
        self.addLink( Madrid , Lisbon, bw=10, delay='30ms')
        self.addLink( Madrid , Barcelona, bw=10, delay='30ms')
        self.addLink( Porto , Lisbon, bw=10, delay='30ms')
        self.addLink( Lisbon , London, bw=10, delay='30ms')
        self.addLink( Barcelona , Paris, bw=10, delay='30ms')
        self.addLink( Manchester , London, bw=10, delay='30ms')
        self.addLink( NewYork , London, bw=10, delay='30ms')
        self.addLink( Amsterdam , Eindhoven, bw=10, delay='30ms')
        self.addLink( Amsterdam , Frankfurt, bw=10, delay='30ms')
        self.addLink( Amsterdam , London, bw=10, delay='30ms')
        self.addLink( Berlin , Frankfurt, bw=10, delay='30ms')
        self.addLink( Berlin , Munich, bw=10, delay='30ms')
        self.addLink( Frankfurt , Munich, bw=10, delay='30ms')
        self.addLink( Frankfurt , Paris, bw=10, delay='30ms')
        self.addLink( Frankfurt , London, bw=10, delay='30ms')
        self.addLink( Paris , Rennes, bw=10, delay='30ms')
        self.addLink( Paris , London, bw=10, delay='30ms')
topos = { 'generated': ( lambda: GeneratedTopo() ) }
# HERE THE CODE DEFINITION OF THE TOPOLOGY ENDS
# the following code produces an executable script working with a remote controller
# and providing ssh access to the the mininet hosts from within the ubuntu vm
c1_ip= '192.168.118.129'
c2_ip= '192.168.118.133'
c3_ip= '192.168.118.131'

def myNetwork():
```

```
    "Create network and run simple performance test"
    topo = GeneratedTopo()
    net = Mininet(topo=topo, controller= None, switch=OVSSwitch, host=CPULimitedHost, link=TCLink, autoSetMacs=True)
    #Add SDN Controllers to the network
    c1 = net.addController ('c1',controller=RemoteController, ip= c1_ip, port=6653)
    c2 = net.addController ('c2',controller=RemoteController, ip= c2_ip, port=6653)
    c3 = net.addController ('c3',controller=RemoteController, ip= c3_ip, port=6653)
    return net


def connectToRootNS( network, switch, ip, prefixLen, routes ):
    "Connect hosts to root namespace via switch. Starts network."
    "network: Mininet() network object"
    "switch: switch to connect to root namespace"
    "ip: IP address for root namespace node"
    "prefixLen: IP address prefix length (e.g. 8, 16, 24)"
    "routes: host networks to route to"
    # Create a node in root namespace and link to switch 0
    root = Node( 'root', inNamespace=False )
    intf = TCLink( root, switch ).intf1
    root.setIP( ip, prefixLen, intf )
    # Start network that now includes link to root namespace
    network.start()
    # Add routes from root ns to hosts
    for route in routes:
        root.cmd( 'route add -net ' + route + ' dev ' + str( intf ) )
```

```
def sshd( network, cmd='/usr/sbin/sshd', opts='-D' ):
    "Start a network, connect it to root ns, and run sshd on all hosts."
    switch = network.switches[ 0 ]  # switch to use
    ip = "# our IP address on host network
    routes = [ '10.0.1.0/16' ]  # host networks to route to
    connectToRootNS( network, switch, ip, 16, routes )
    for host in network.hosts:
        host.cmd( cmd + ' ' + opts + '&' )

    # DEBUGGING INFO
    print
    print "Dumping host connections"
    dumpNodeConnections(network.hosts)
    print
    print "*** Hosts are running sshd at the following addresses:"
    print
    for host in network.hosts:
        print host.name, host.IP()
    print
    print "*** Type 'exit' or control-D to shut down network"
    print
    print "*** For testing network connectivity among the hosts, wait a bit for the controller
to create all the routes, then do 'pingall' on the mininet console."
    print

CLI( network )
    for host in network.hosts:
```

```
        host.cmd( 'kill %' + cmd )
    network.stop()

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

APPENDIX E

Installing GNS3 on Windows Operating System

Steps:

1. Download GNS3 for windows from the gns3 website.
2. After creating an account and Login, download GNS3-all-in-one package.
3. Then, click Run to start the GNS3 installation and Next to continue.
4. Click I agree button to agree to the License Agreement.
5. Select the Start Menu folder for the GNS3 shortcut, and click Next.
6. Choose components or leave all selected at their default selection, and click Next.
7. Choose Install Location and click Install.
8. The installation wizard will guide you through installing the components selected.
9. On successful installation, an Installation Complete message is displayed. Click Next to continue.
10. Leave the Start GNS3 checkbox enabled and click Finish

Installing GNS3 VM

Steps:

1. Download GNS3 VM, from gns3 website.
2. Unzip the downloaded zip file.
3. In VMware Workstation, click Open a Virtual Machine
4. Navigate to the directory where the extracted GNS3 VM.ova is located and click Open.
5. Leave the VM name as GNS3 VM and click Import.
6. GNS3 VM will show as available in VMware Workstation after the Import. Leave all settings at their defaults.

APPENDIX F

CIDC CODES

Producer

```
/**
By Benamrane Fouad
*/
package net.floodlightcontroller.secureSDNi;


import static org.jboss.netty.channel.Channels.pipeline;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.InetSocketAddress;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import javax.net.ssl.SSLEngine;
import net.floodlightcontroller.core.IFloodlightProviderService;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
```

```
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;
import net.floodlightcontroller.core.util.SingletonTask;
import net.floodlightcontroller.threadpool.IThreadPoolService;;
import org.jboss.netty.bootstrap.ClientBootstrap;
import org.jboss.netty.channel.Channel;
import org.jboss.netty.channel.ChannelEvent;
import org.jboss.netty.channel.ChannelFuture;
import org.jboss.netty.channel.ChannelHandlerContext;
import org.jboss.netty.channel.ChannelPipeline;
import org.jboss.netty.channel.ChannelPipelineFactory;
import org.jboss.netty.channel.ChannelStateEvent;
import org.jboss.netty.channel.ExceptionEvent;
import org.jboss.netty.channel.MessageEvent;
import org.jboss.netty.channel.SimpleChannelUpstreamHandler;
import org.jboss.netty.channel.socket.nio.NioClientSocketChannelFactory;
import org.jboss.netty.handler.codec.frame.DelimiterBasedFrameDecoder;
import org.jboss.netty.handler.codec.frame.Delimiters;
import org.jboss.netty.handler.codec.string.StringDecoder;
import org.jboss.netty.handler.codec.string.StringEncoder;
import org.jboss.netty.handler.ssl.SslHandler;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Producer implements IFloodlightModule,IHAControllerService,
IStoreListener<String>, IHAWorkerService {
```

```
        protected static List<Channel> CH;
        protected IFloodlightProviderService floodlightProvider;
        protected static Logger logger;
        protected SingletonTask mythread;
        protected Runnable SS;
        protected IThreadPoolService threadPool;
        private ClientBootstrap bootstrap;
        protected static ChannelFuture future;
        private static Producer netClient=null;
        protected conf myConf;
        protected static ChannelFuture connection;

        public static synchronized Producer getnetClient(){
            if(netClient==null){
                    netClient=new Producer();
            }
            return netClient;
        }
@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
        Collection<Class<? extends IFloodlightService>> l =
                        new ArrayList<Class<? extends IFloodlightService>>();
                l.add(IFloodlightProviderService.class);
                l.add(IThreadPoolService.class);
                return l;
            }
@Override
```

```
public Collection<Class<? extends IFloodlightService>> getModuleServices() {
                        return null;
                }
@Override
public Map<Class<? extends IFloodlightService>, IFloodlightService>
getServiceImpls() {
                        return null;
                }
@Override
public void init(FloodlightModuleContext context)
                                throws FloodlightModuleException {
                        myConf = new conf();
                   CH= new ArrayList<Channel>();
              floodlightProvider =
context.getServiceImpl(IFloodlightProviderService.class);
                logger = LoggerFactory.getLogger(Producer.class);
                threadPool=context.getServiceImpl(IThreadPoolService.class);
                   }
@Override
public void startUp(FloodlightModuleContext context)
                        throws FloodlightModuleException {
                   ScheduledExecutorService ses = threadPool.getScheduledExecutor();
                mythread = new SingletonTask (ses,new Runnable() {
                   public void run() {
 try{
                        logger.info("Waiting 10 sec");
                     Thread.sleep(10000);
                    logger.info("Producer :: "+myConf.getMode()+" mode is configured");
```

```
                producer();
            }
            catch(Exception e){
                logger.error("Exception detected "+e.getMessage());
            }
        }
    });
    mythread.reschedule(1,  TimeUnit.SECONDS);
}
public void producer() throws IOException{
    // Configure the client.
      bootstrap = new ClientBootstrap( new NioClientSocketChannelFactory(
                Executors.newCachedThreadPool(),
                Executors.newCachedThreadPool()));
        // Configure the pipeline factory.
        bootstrap.setPipelineFactory(new SecureProducerPipelineFactory());
        // Start the connection attempt.
        for(String h:myConf.getL()){
      future = bootstrap.connect(new InetSocketAddress(h,
myConf.getPort()));
        if(!future.awaitUninterruptibly().isSuccess()) {
 logger.info("Producer :: Unable to connect to host " + h + ":" + myConf.getPort());
            }
 else{ logger.info("Producer :: Successfully connected to " + h + ":" +
myConf.getPort());
        CH.add(future.getChannel());
        future.getChannel().write("Hello \n");
        logger.info("Producer :: Hello sent to remote consumer");
```

```
}
        } }
        // Wait until the connection attempt succeeds or fails.
        //Channel channel = future.awaitUninterruptibly().getChannel();
        /*if (!future.isSuccess()) {
            future.getCause().printStackTrace();
            bootstrap.releaseExternalResources();
            return;
        }
        */
        // Read commands from the stdin.
        /*ChannelFuture lastWriteFuture = null;
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        for (;;) {
            String line = in.readLine();
            if (line == null) {
                break;
            }
            // Sends the received line to the server.
            lastWriteFuture = channel.write(line + "\r\n");
            // If user typed the 'bye' command, wait until the server closes
            // the connection.
            if (line.toLowerCase().equals("bye")) {
                channel.getCloseFuture().awaitUninterruptibly();
                break;
            }
        }
```

```
            // Wait until all messages are flushed before closing the channel.
            if (lastWriteFuture != null) {
                lastWriteFuture.awaitUninterruptibly();
            }
            // Close the connection.  Make sure the close operation ends because
            // all I/O operations are asynchronous in Netty.
            channel.close().awaitUninterruptibly();
            // Shut down all thread pools to exit.
            bootstrap.releaseExternalResources();*/
    // }
        public static List<Channel> getCH() {
                return CH;
        }
        public void setCH(List<Channel> cH) {
                CH = cH;
        }
        public static ChannelFuture getFuture() {
                return future;
        }
        public static void setFuture(ChannelFuture future) {
                Producer.future = future;
        }
        public class SecureProducerPipelineFactory implements
        ChannelPipelineFactory {
        public ChannelPipeline getPipeline() throws Exception {
        ChannelPipeline pipeline = pipeline();
        // Add SSL handler first to encrypt and decrypt everything.
```

```
                // In this example, we use a bogus certificate in the server side
                // and accept any invalid certificates in the client side.
                // You will need something more complicated to identify both
                // and server in the real world.
                if(myConf.isSSL()){
SSLEngine engine =
                    SecureSslContextFactory.getClientContext().createSSLEngine();
                engine.setUseClientMode(true);
                pipeline.addLast("ssl", new SslHandler(engine));
                }
                // On top of the SSL handler, add the text line codec.
                pipeline.addLast("decoder", new StringDecoder());
                pipeline.addLast("encoder", new StringEncoder());
                // and then business logic.
                pipeline.addLast("handler", new SecureProducerHandler());
                        return pipeline;
                }
                }
        public class SecureProducerHandler extends SimpleChannelUpstreamHandler {
                         //protected boolean SSL
@Override
        public void handleUpstream(
                        ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
                            /*if (e instanceof ChannelStateEvent) {
                               logger.info(e.toString());
                            }
                            super.handleUpstream(ctx, e);*/
```

```
                }

@Override
                public void channelConnected(
        ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
                    // Get the SslHandler from the pipeline
                    // which were added in SecureChatPipelineFactory.
                        if(myConf.isSSL()){
                    SslHandler sslHandler = ctx.getPipeline().get(SslHandler.class);
                    // Begin handshake.
                    sslHandler.handshake();
                   }
                        logger.info("Producer:: Remote
controller"+ctx.getChannel().getRemoteAddress().toString()+" is connected");
                }
@Override
            public void messageReceived(
                    ChannelHandlerContext ctx, MessageEvent e) throws Exception {
                        if(!e.getMessage().equals("Hello \n"))
                                    logger.info("Producer::from
"+e.getRemoteAddress().toString()+":"+e.getMessage().toString().replace("\n", ""));
                                super.messageReceived(ctx, e);
                }
@Override
                public void exceptionCaught(
                        ChannelHandlerContext ctx, ExceptionEvent e) {
                    logger.warn(
                            "Unexpected exception from downstream.",
```

```
                                    e.getCause());
                                e.getChannel().close();
                            }
                    }
}
```

Consumer

```
/*By Benamrane Fouad
**/
package net.floodlightcontroller.secureSDNi;

import static org.jboss.netty.channel.Channels.pipeline;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import javax.net.ssl.SSLEngine;
import org.jboss.netty.bootstrap.ServerBootstrap;
import org.jboss.netty.channel.ChannelEvent;
import org.jboss.netty.channel.ChannelFuture;
import org.jboss.netty.channel.ChannelFutureListener;
import org.jboss.netty.channel.ChannelHandlerContext;
import org.jboss.netty.channel.ChannelPipeline;
```

```
import org.jboss.netty.channel.ChannelPipelineFactory;
import org.jboss.netty.channel.ChannelStateEvent;
import org.jboss.netty.channel.ExceptionEvent;
import org.jboss.netty.channel.MessageEvent;
import org.jboss.netty.channel.SimpleChannelUpstreamHandler;
import org.jboss.netty.channel.group.ChannelGroup;
import org.jboss.netty.channel.group.DefaultChannelGroup;
import org.jboss.netty.channel.socket.nio.NioServerSocketChannelFactory;
import org.jboss.netty.handler.codec.string.StringDecoder;
import org.jboss.netty.handler.codec.string.StringEncoder;
import org.jboss.netty.handler.ssl.SslHandler;
import org.projectfloodlight.openflow.types.DatapathId;
import org.projectfloodlight.openflow.types.EthType;
import org.projectfloodlight.openflow.types.IPv4AddressWithMask;
import org.projectfloodlight.openflow.types.IpProtocol;
import org.projectfloodlight.openflow.types.MacAddress;
import org.projectfloodlight.openflow.types.OFPort;
import org.projectfloodlight.openflow.types.TransportPort;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import net.floodlightcontroller.core.IFloodlightProviderService;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;
import net.floodlightcontroller.core.util.SingletonTask;
import net.floodlightcontroller.dataCollector.DataTracker;
```

```
import net.floodlightcontroller.firewall.FirewallRule;
import net.floodlightcontroller.firewall.FirewallRule.FirewallAction;
import net.floodlightcontroller.firewall.IFirewallService;
import net.floodlightcontroller.loadbalancer.ILoadBalancerService;
import net.floodlightcontroller.loadbalancer.LBMember;
import net.floodlightcontroller.loadbalancer.LBPool;
import net.floodlightcontroller.loadbalancer.LBVip;
import net.floodlightcontroller.threadpool.IThreadPoolService;

public class Consumer implements IFloodlightModule{
        protected conf myConf;
        protected IFloodlightProviderService floodlightProvider;
        protected static Logger logger;
        protected IThreadPoolService threadPool;
        protected SingletonTask mythread;
        protected Runnable SS;
        protected IThreadPoolService threadPoolService;
        protected IFirewallService firewallService;
        protected ILoadBalancerService lbService;
        protected FirewallRule newRule;
        protected LBVip newVip;
        protected LBPool newPool;
        protected LBMember newMember;
        public static DataTracker datatracker;
        public static Consumer Consumer = null;
        protected static List<String> receivedVipNames;
        protected static List<String> receivedPoolNames;
```

```
protected static List<Integer> receivedMemberIPs;
protected static List<Integer> receivedRulesIds;
static final ChannelGroup channels = new DefaultChannelGroup();
protected String message;
public static List<Integer> getReceivedRulesIds() {
        return receivedRulesIds;
}
public static List<String> getReceivedVipNames() {
        return receivedVipNames;
}
public static List<String> getReceivedPoolNames() {
        return receivedPoolNames;
}
public static List<Integer> getReceivedMemberIPs() {
        return receivedMemberIPs;
}
public String getMessage() {
        return message;
}
public void setMessage(String message) {
        this.message = message;
}
public static void setReceivedRulesIds(List<Integer> receivedRulesIds) {
        Consumer.receivedRulesIds = receivedRulesIds;
}
public static synchronized Consumer getConsumer(){
        if(Consumer ==null){
```

```
                    Consumer = new Consumer();
                }
                return Consumer;
        }
        @Override
        public Collection<Class<? extends IFloodlightService>> getModuleServices() {
                // TODO Auto-generated method stub
                return null;
        }
        @Override
public Map<Class<? extends IFloodlightService>, IFloodlightService>
getServiceImpls() {
                // TODO Auto-generated method stub
                return null;
        }
        @Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
                Collection<Class<? extends IFloodlightService>> l =
                    new ArrayList<Class<? extends IFloodlightService>>();
                  l.add(IFloodlightProviderService.class);
                  l.add(IThreadPoolService.class);
                  l.add(IFirewallService.class);
                  l.add(ILoadBalancerService.class);
                  return l;
        }
        @Override
        public void init(FloodlightModuleContext context)
```

```
                throws FloodlightModuleException {
floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
        threadPool = context.getServiceImpl(IThreadPoolService.class);
        logger = LoggerFactory.getLogger(Consumer.class);
        threadPoolService = context.getServiceImpl(IThreadPoolService.class);
        myConf=new conf();
        firewallService = context.getServiceImpl(IFirewallService.class);
        lbService = context.getServiceImpl(ILoadBalancerService.class);
        newRule= new FirewallRule();
        newVip = new LBVip();
        newPool= new LBPool();
        newMember = new LBMember();
        datatracker = new DataTracker();
        receivedRulesIds = new ArrayList<Integer>();
        receivedMemberIPs = new ArrayList<Integer>();
        receivedPoolNames = new ArrayList<String>();
        receivedVipNames = new ArrayList<String>();
}
@Override
public void startUp(FloodlightModuleContext context)
                throws FloodlightModuleException {
ScheduledExecutorService ses = threadPoolService.getScheduledExecutor();
        mythread = new SingletonTask (ses, new Runnable() {
                public void run() {
                        try{
                                Consumer();
                        }catch (Exception e) {
```

```
                                        throw new RuntimeException(e);
                                    }
            }
                    });
                    mythread.reschedule(2,TimeUnit.SECONDS);
            }
public void Consumer(){
            ServerBootstrap bootstrap = new ServerBootstrap(
              new NioServerSocketChannelFactory(
                    Executors.newCachedThreadPool(),
                    Executors.newCachedThreadPool()));
        // Configure the pipeline factory.
        bootstrap.setPipelineFactory(new SecureConsumerPipelineFactory());
        // Bind and start to accept incoming connections.
        bootstrap.bind(new InetSocketAddress(myConf.getPort()));
            }
            public class SecureConsumerPipelineFactory implements
    ChannelPipelineFactory {
                    public ChannelPipeline getPipeline() throws Exception {
    ChannelPipeline pipeline = pipeline();
    // Add SSL handler first to encrypt and decrypt everything.
    // In this example, we use a bogus certificate in the server side
    // and accept any invalid certificates in the client side.
    // You will need something more complicated to identify both
    // and server in the real world.
    // Read SecureChatSslContextFactory
    // if you need client certificate authentication.
```

```
    if(myConf.isSSL()){
    SSLEngine engine =
       SecureSslContextFactory.getServerContext().createSSLEngine();
    engine.setUseClientMode(false);

    pipeline.addLast("ssl", new SslHandler(engine));
    }
    // On top of the SSL handler, add the text line codec.
    pipeline.addLast("decoder", new StringDecoder());
    pipeline.addLast("encoder", new StringEncoder());
    // and then business logic.
    pipeline.addLast("handler", new DataUpdater());
    return pipeline;
}
}
        public class DataUpdater extends SimpleChannelUpstreamHandler {
                final ChannelGroup channels = new DefaultChannelGroup();
            @Override
            public void handleUpstream(
                ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
               if (e instanceof ChannelStateEvent) {
                  logger.info(e.toString());
               }
               super.handleUpstream(ctx, e);
            }
            @Override
            public void channelConnected(
```

```
            ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
            if(myConf.isSSL()){
          // Get the SslHandler in the current pipeline.
          // We added it in SecureChatPipelineFactory.
          final SslHandler sslHandler = ctx.getPipeline().get(SslHandler.class);
          // Get notified when SSL handshake is done.
          ChannelFuture handshakeFuture = sslHandler.handshake();
          handshakeFuture.addListener(new Greeter(sslHandler));
            }
            logger.info("Consumer::Remote controller:
"+e.getChannel().getRemoteAddress().toString()+" is connected!");
        }
        @Override
        public void channelDisconnected(
            ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
          // Unregister the channel from the global channel list
          // so the channel does not receive messages anymore.
          channels.remove(e.getChannel());
          logger.info("Consumer::Remote controller:
"+e.getChannel().getRemoteAddress().toString()+" is disconnected from the server.");
        }
        @Override
        public void messageReceived(
            ChannelHandlerContext ctx, MessageEvent e) throws Exception {
            message=e.getMessage().toString();
                    message = message.replace("\n", "");
                    if(message.contains("Hello")){
                            e.getChannel().write("Hello \n");
```

```
                    logger.info("Consumer::from
"+e.getRemoteAddress().toString()+":"+message);
                }
                else if(message.contains("Firewall") && myConf.isFirewall()){
                    logger.info("New remote firewall rule received ");
                    if(!firewallService.isEnabled())
                    {
                        firewallService.enableFirewall(true);
                    }
                        treatMessage(message);
            }
                else if(message.contains("LoadBalancer Vip created") &&
myConf.isLoadBalancer()){
                logger.info("New remote Load Balancer vip received");
                treatMessageVip(message);
            }
                else if(message.contains("LoadBalancer Pool created") &&
myConf.isLoadBalancer()){
                logger.info("New remote Load Balancer pool received");
                treatMessagePool(message);
            }
                else if(message.contains("LoadBalancer Member created") &&
myConf.isLoadBalancer()){
                logger.info("New remote Load Balancer Member received");
                treatMessageMember(message);
            }
            else logger.info("Consumer::from "+e.getRemoteAddress().toString()+":
"+message);
```

```
        }
    public void treatMessageMember(String Member){
        String [] listMessage;
        listMessage = Member.split(",");
        String id = listMessage[1];
        newMember.setId(id);
        int ip = Integer.parseInt(listMessage[2]);
        if(!receivedMemberIPs.contains(ip))
                receivedMemberIPs.add(ip);
        newMember.setAddress(ip);
        short port = Short.parseShort(listMessage[3]);
        newMember.setPort(port);
        String poolId = listMessage[4];
        newMember.setPoolId(poolId);
        lbService.createMember(newMember);
    }
        public void treatMessagePool(String pool){
                String [] listMessage;
                listMessage = pool.split(",");
                String id = listMessage[1];
                newPool.setId(id);
                String name = listMessage[2];
                if(!receivedPoolNames.contains(name))
                receivedPoolNames.add(name);
                newPool.setName(name);
                byte protocol = Byte.parseByte(listMessage[3]);
                newPool.setProtocol(protocol);
```

```
            String vipId= listMessage[4];
            newPool.setVipId(vipId);
            lbService.createPool(newPool);
    }
    public void treatMessageVip(String vip){
            String[] listMessage;
            listMessage = vip.split(",");
            String id=listMessage[1];
            newVip.setId(id);
            String name = listMessage[2];
            if(!receivedVipNames.contains(name))
            receivedVipNames.add(name);
            newVip.setName(name);
            byte protocol = Byte.parseByte(listMessage[3]);
            newVip.setProtocol(protocol);
            int ip = Integer.parseInt(listMessage[4]);
            newVip.setAddress(ip);
            short port = Short.parseShort(listMessage[5]);
            newVip.setPort(port);
            lbService.createVip(newVip);
    }
    @SuppressWarnings("static-access")
    public void treatMessage(String msg){
            String[] listMessage;
            listMessage = msg.split(",");
            int ID=Integer.parseInt(listMessage[1]);
            newRule.setRuleid(ID);
```

```
                    if(!receivedRulesIds.contains(ID))
                            receivedRulesIds.add(ID);
                    /*if(!datatracker.getRuleIds().contains(ID))
                    {*/
                    DatapathId id = DatapathId.of(listMessage[2]);
                    newRule.setDpid(id);
                    OFPort port;
                    if (listMessage[3].equals("any"))
                            port =OFPort.ANY;
                    else if (listMessage[3].equals("all"))
                            port =OFPort.ALL;
                    else if (listMessage[3].equals("flood"))
                            port =OFPort.FLOOD;
                    else port = OFPort.of(Integer.parseInt(listMessage[3]));
                    newRule.setIn_port(port);
                    MacAddress Mac = MacAddress.of(listMessage[4]);
                    newRule.setDl_src(Mac);
                    Mac = MacAddress.of(listMessage[5]);
                    newRule.setDl_dst(Mac);
                    EthType type = EthType.of(Integer.parseInt(listMessage[6]));
                    newRule.setDl_type(type);
                    IPv4AddressWithMask ip =
IPv4AddressWithMask.of(listMessage[7]);
                    newRule.setNw_src_prefix_and_mask(ip);
                    ip = IPv4AddressWithMask.of(listMessage[8]);
                    newRule.setNw_dst_prefix_and_mask(ip);
                    IpProtocol proto =
IpProtocol.of(Short.parseShort(listMessage[9]));
```

```
                newRule.setNw_proto(proto);
                TransportPort tcpPort =
TransportPort.of(Integer.parseInt(listMessage[10]));
                newRule.setTp_src(tcpPort);
                tcpPort = TransportPort.of(Integer.parseInt(listMessage[11]));
                newRule.setTp_dst(tcpPort);
                boolean anyS = stringToBoolean(listMessage[12]);
                newRule.setAny_in_port(anyS);
                anyS = stringToBoolean(listMessage[13]);
                newRule.setAny_dl_src(anyS);
                anyS = stringToBoolean(listMessage[14]);
                newRule.setAny_dl_dst(anyS);
                anyS = stringToBoolean(listMessage[15]);
                newRule.setAny_dl_type(anyS);
                anyS = stringToBoolean(listMessage[16]);
                newRule.setAny_nw_src(anyS);
                anyS = stringToBoolean(listMessage[17]);
                newRule.setAny_nw_dst(anyS);
                anyS = stringToBoolean(listMessage[18]);
                newRule.setAny_nw_proto(anyS);
                anyS = stringToBoolean(listMessage[19]);
                newRule.setAny_tp_src(anyS);
                anyS = stringToBoolean(listMessage[20]);
                newRule.setAny_tp_dst(anyS);
                newRule.setPriority(Integer.parseInt(listMessage[21]));
                FirewallAction action = FirewallAction.valueOf(listMessage[22]);
                newRule.setAction(action);
```

```
                firewallService.addRule(newRule);
                //}
                //else logger.info("Rule already existe");
        }
        public boolean stringToBoolean(String msg)
        {
                if(msg.equals("true"))
                        return true;
                else return false;
        }
@Override
public void exceptionCaught(
        ChannelHandlerContext ctx, ExceptionEvent e) {
    logger.warn("Unexpected exception from downstream.",
            e.getCause());
    e.getChannel().close();
}
private final class Greeter implements ChannelFutureListener {
    private final SslHandler sslHandler;
    Greeter(SslHandler sslHandler) {
        this.sslHandler = sslHandler;
    }
    public void operationComplete(ChannelFuture future) throws Exception {
        if (future.isSuccess()) {
            // Once session is secured, send a greeting.
            future.getChannel().write(
                    "Welcome to " + InetAddress.getLocalHost().getHostName() +
```

```
                    " secure chat service!\n");
                future.getChannel().write(
                    "Your session is protected by " +
                    sslHandler.getEngine().getSession().getCipherSuite() +
                    " cipher suite.\n");
                // Register the channel to the global channel list
                // so the channel received the messages from others.
                channels.add(future.getChannel());
            } else {
                future.getChannel().close();
            }
        }
    }
}
```

APPENDIX G

mCIDC Codes

```
/**
By Benamrane Fouad
Adekale Abdulfatai
*/
package net.floodlightcontroller.secureSDNi;

import static org.jboss.netty.channel.Channels.pipeline;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import javax.net.ssl.SSLEngine;

import org.jboss.netty.bootstrap.ClientBootstrap;
import org.jboss.netty.channel.Channel;
import org.jboss.netty.channel.ChannelEvent;
```

```
import org.jboss.netty.channel.ChannelFuture;
import org.jboss.netty.channel.ChannelHandlerContext;
import org.jboss.netty.channel.ChannelPipeline;
import org.jboss.netty.channel.ChannelPipelineFactory;
import org.jboss.netty.channel.ChannelStateEvent;
import org.jboss.netty.channel.ExceptionEvent;
import org.jboss.netty.channel.MessageEvent;
import org.jboss.netty.channel.SimpleChannelUpstreamHandler;
import org.jboss.netty.channel.socket.nio.NioClientSocketChannelFactory;
import org.jboss.netty.handler.codec.string.StringDecoder;
import org.jboss.netty.handler.codec.string.StringEncoder;
import org.jboss.netty.handler.ssl.SslHandler;
import org.sdnplatform.sync.IStoreClient;
import org.sdnplatform.sync.IStoreListener;
import org.sdnplatform.sync.ISyncService;
import org.sdnplatform.sync.ISyncService.Scope;
import org.sdnplatform.sync.error.SyncException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import net.floodlightcontroller.core.IFloodlightProviderService;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;
import net.floodlightcontroller.core.util.SingletonTask;
import net.floodlightcontroller.linkdiscovery.ILinkDiscoveryService;
```

```
import net.floodlightcontroller.secureSDNi.linkdiscovery.LDHAWorker;
import net.floodlightcontroller.secureSDNi.topology.TopoHAWorker;
import net.floodlightcontroller.storage.IStorageSourceService;
import net.floodlightcontroller.threadpool.IThreadPoolService;
import net.floodlightcontroller.topology.ITopologyService;

public class Producer implements IFloodlightModule, IStoreListener<String>,
IHAWorkerService {

        protected static List<Channel> CH;
        protected IFloodlightProviderService floodlightProvider;
        protected static Logger logger;
        protected SingletonTask mythread;
        protected static IHAWorkerService haworker;
        protected static ILinkDiscoveryService linkserv;
        protected static ITopologyService toposerv;
        protected static ISyncService syncService;
        protected static IStoreClient<String,String> storeFT;
        protected static IStoreClient<String, String> storeLD;
        protected static IStoreClient<String, String> storeTopo;
        private static String controllerID;
        protected static LDHAWorker ldhaworker;
        protected static TopoHAWorker topohaworker;
        private static Map<String, IHAWorker> workers = new HashMap<>();
        protected Runnable SS;
        protected IThreadPoolService threadPool;
        private ClientBootstrap bootstrap;
```

```
        protected static ChannelFuture future;
        private static Producer netClient=null;
        protected conf myConf;
        protected static ChannelFuture connection;

public static synchronized Producer getnetClient(){
            if(netClient==null){
                netClient=new Producer();
            }
            return netClient;
        }

@Override
public Collection<Class<? extends IFloodlightService>> getModuleDependencies() {
        Collection<Class<? extends IFloodlightService>> l =
                    new ArrayList<Class<? extends IFloodlightService>>();
                l.add(IFloodlightProviderService.class);
                l.add(IThreadPoolService.class);
                l.add(IStorageSourceService.class);
                        l.add(ISyncService.class);
                return l;
            }

@Override
public Collection<Class<? extends IFloodlightService>> getModuleServices() {
        Collection<Class<? extends IFloodlightService>> l = new ArrayList<>();
                l.add(IHAWorkerService.class);
```

```
                return l;
        }
        /**
         * Gets the specified HAWorker object.**/
@Override
public IHAWorker getService(String serviceName) {
                synchronized (workers) {
                        return workers.get(serviceName);
                }
        }
@Override
public Map<Class<? extends IFloodlightService>, IFloodlightService>
getServiceImpls() {
        Map<Class<? extends IFloodlightService>, IFloodlightService> m = new
HashMap<>();
                /**
                 * We are the class that implements the service**/
                m.put(IHAWorkerService.class, this);
                return m;
        }
        /**
         * Returns the keys of the workers hashmap, which holds the objects
         * corresponding to the registered HAWorker classes.
         **/
@Override
public Set<String> getWorkerKeys() {
                synchronized (workers) {
                        return workers.keySet();
```

```
                }
        }
@Override
public void init(FloodlightModuleContext context)
        throws FloodlightModuleException {
                myConf = new conf();
        CH= new ArrayList<Channel>();
    floodlightProvider =
context.getServiceImpl(IFloodlightProviderService.class);
    logger = LoggerFactory.getLogger(Producer.class);
    threadPool=context.getServiceImpl(IThreadPoolService.class);
    haworker = this;
    linkserv = context.getServiceImpl(ILinkDiscoveryService.class);
    toposerv = context.getServiceImpl(ITopologyService.class);
    syncService = context.getServiceImpl(ISyncService.class);
    controllerID = new String("C" + floodlightProvider.getControllerId());
        }
@Override
public void keysModified(Iterator<String> keys,
org.sdnplatform.sync.IStoreListener.UpdateType type) {
        }
@Override
public void registerService(String serviceName, IHAWorker haw) {
        synchronized (workers) {
        workers.putIfAbsent(serviceName, haw);
                }
        }
```

```
@Override
public void startUp(FloodlightModuleContext context)
                throws FloodlightModuleException {
                        logger.info("LDHAWorker is starting...");
        try {
                Producer.syncService.registerStore("LDUpdates", Scope.GLOBAL);
                Producer.storeLD = Producer.syncService.getStoreClient("LDUpdates",
String.class, String.class);
                Producer.storeLD.addStoreListener(this);
        } catch (SyncException e) {
        throw new FloodlightModuleException("Error while setting up sync service", e);
        }
                Producer.ldhaworker = new LDHAWorker(Producer.storeLD,
controllerID);
                linkserv.addListener(Producer.ldhaworker);
                haworker.registerService("LDHAWorker", Producer.ldhaworker);

        logger.info("TopoHAWorker is starting...");
        try {
                Producer.syncService.registerStore("TopoUpdates", Scope.GLOBAL);
                Producer.storeTopo = Producer.syncService.getStoreClient
                ("TopoUpdates", String.class, String.class);
                Producer.storeTopo.addStoreListener(this);
        } catch (SyncException e) {
        throw new FloodlightModuleException("Error while setting up sync service", e);
                }
                Producer.topohaworker = new TopoHAWorker(Producer.storeTopo,
controllerID);
                toposerv.addListener(Producer.topohaworker);
```

```
                haworker.registerService("TopoHAWorker", Producer.topohaworker);

        ScheduledExecutorService ses = threadPool.getScheduledExecutor();
            mythread = new SingletonTask (ses,new Runnable() {
                public void run() {
                    try{
                        logger.info("Waiting 10 sec");
                        Thread.sleep(10000);
                      logger.info("Producer:: "+myConf.getMode()+" mode is
configured");
                        producer ();
                    }
                    catch(Exception e){
                        logger.error("Exception detected "+e.getMessage());
                    }
            }
                });
                mythread.reschedule(1,  TimeUnit.SECONDS);
               /** try {
                    Producer.syncService.registerStore("StoreUpdates", Scope.GLOBAL);
                    Producer.storeFT = Producer.syncService.getStoreClient("StoreUpdates",
String.class, String.class);
                    Producer.storeFT.addStoreListener(this);
                } catch (SyncException e) {
                    throw new FloodlightModuleException ("Error while setting up sync
service", e);
                }**/
            }
```

```
public void Producer () throws IOException{
// Configure the client.
bootstrap = new ClientBootstrap( new NioClientSocketChannelFactory(
                    Executors.newCachedThreadPool(),
                    Executors.newCachedThreadPool()));
// Configure the pipeline factory.
                bootstrap.setPipelineFactory(new SecureProducerPipelineFactory());
// Start the connection attempt.
                for (String h:myConf.getControllerID()){
                future = bootstrap.connect(new InetSocketAddress(h,
myConf.getPort()));
                if(!future.awaitUninterruptibly().isSuccess()) {
logger.info ("Producer:: Unable to connect to host " + h + ":" + myConf.getPort());
                    }
else{ logger.info("Producer:: Successfully connected to " + h + ":" + myConf.getPort());
                CH.add(future.getChannel());
                future.getChannel().write("Hello \n");
                logger.info ("Producer:: Hello sent to remote consumer");
            }
                } }
                // Wait until the connection attempt succeeds or fails.
                //Channel channel = future.awaitUninterruptibly().getChannel();
                /*if (!future.isSuccess()) {
                    future.getCause().printStackTrace();
                    bootstrap.releaseExternalResources();
                    return;
                }
```

```
                */
                // Read commands from the stdin.
                /*ChannelFuture lastWriteFuture = null;
BufferedReader in = new BufferedReader (new InputStreamReader(System.in));
                for (;;) {
                    String line = in.readLine();
                    if (line == null) {
                        break;
                    }
                    // Sends the received line to the server.
                    lastWriteFuture = channel.write(line + "\r\n");
                    // If user typed the 'bye' command, wait until the server closes
                    // the connection.
                    if (line.toLowerCase().equals("bye")) {
                        channel.getCloseFuture().awaitUninterruptibly();
                        break;
                    }
                }
                // Wait until all messages are flushed before closing the channel.
                if (lastWriteFuture != null) {
                    lastWriteFuture.awaitUninterruptibly();
                }
                // Close the connection.  Make sure the close operation ends because
                // all I/O operations are asynchronous in Netty.
                channel.close().awaitUninterruptibly();
                // Shut down all thread pools to exit.
                bootstrap.releaseExternalResources();*/
```

```
// }

        public static List<Channel> getCH() {
                return CH;
        }
        public void setCH(List<Channel> cH) {
                CH = cH;
        }
        public static ChannelFuture getFuture() {
                return future;
        }
        public static void setFuture(ChannelFuture future) {
                Producer.future = future;
        }
        public class SecureProducerPipelineFactory implements
        ChannelPipelineFactory {
        public ChannelPipeline getPipeline() throws Exception {
        ChannelPipeline pipeline = pipeline();
        // Add SSL handler first to encrypt and decrypt everything.
        // In this example, we use a bogus certificate in the server side
        // and accept any invalid certificates in the client side.
        // You will need something more complicated to identify both
        // and server in the real world.
        if(myConf.isSSL()){
        SSLEngine engine =
           SecureSslContextFactory.getClientContext ().createSSLEngine ();
        engine.setUseClientMode(true);
```

```
                pipeline.addLast("ssl", new SslHandler(engine));
                }
                // On top of the SSL handler, add the text line codec.
                pipeline.addLast("decoder", new StringDecoder());
                pipeline.addLast("encoder", new StringEncoder());
                // and then business logic.
                pipeline.addLast("handler", new SecureProducerHandler());
                        return pipeline;
                }
                }
        public class SecureProducerHandler extends SimpleChannelUpstreamHandler
         //protected boolean SSL
@Override
        public void handleUpstream (
                        ChannelHandlerContext ctx, ChannelEvent e) throws Exception
{
        /*if (e instanceof ChannelStateEvent) {
                                logger.info(e.toString());
                            }
                            super.handleUpstream(ctx, e);*/
                        }
        @Override
        public void channelConnected (
                ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
                        // Get the SslHandler from the pipeline
```

```
                        // which were added in SecureChatPipelineFactory.
                            if(myConf.isSSL()){
        SslHandler sslHandler = ctx.getPipeline().get(SslHandler.class);
                        // Begin handshake.
                        sslHandler.handshake();
                       }
        logger.info ("Producer:: Remote
controller"+ctx.getChannel().getRemoteAddress().toString()+" is connected");
                    }
@Override
        public void messageReceived (
                ChannelHandlerContext ctx, MessageEvent e) throws Exception {
                        if(!e.getMessage().equals("Hello \n"))
        logger.info("Producer::from
"+e.getRemoteAddress().toString()+":"+e.getMessage().toString().replace("\n", ""));
                                super.messageReceived(ctx, e);
                    }
@Override
        public void exceptionCaught(
                    ChannelHandlerContext ctx, ExceptionEvent e) {
                        logger.warn(
                            "Unexpected exception from downstream.",
                            e.getCause());
                        e.getChannel().close();
                    }
                }
```

APPENDIX H

NETWORK POLICIES

Note:

1. **If action is not specified, it implies ALLOW rule.**
2. **To use the Curl command, run sudo apt install curl**
3. **To list all ACL rules on a network:**
   Curl http://<controller_ip>:8080/wm/acl/rules/json | python -mjson.tool

Network 1 (VM1)

1. Curl –X POST –d ‘{“src-ip”: “10.0.0.11/16”, “dst-ip”: “10.0.0.9/16”, “nw-proto”:“udp”,“action”:“ALLOW”}’
   http://<controller_ip>:8080/wm/acl/rules/json
   Curl –X POST –d ‘{“src-ip”: “10.0.0.9/16”, “dst-ip”: “10.0.0.11/16”, “nw-proto”:“udp”, “action”:“ALLOW”}’
   http://<controller_ip>:8080/wm/acl/rules/json
2. Curl –X POST –d ‘{“switch id”:“00:00:00:00:00:00:00:14”}’
   http://<controller_ip>:8080/wm/acl/rules/json
3. Curl –X POST –d ‘{“src-ip”:“10.0.0.13/16”,“dst-ip”:“10.0.0.12/16”,“action” :“DENY”}’
   http://<controller_ip>:8080/wm/acl/rules/json
4. Curl –X POST –d ‘{“src-ip”:“10.0.0.13/16”,“dst-ip”:“10.0.0.12/16”,“action” :“ALLOW”}’
   http://<controller_ip>:8080/wm/acl/rules/json

Network 2 (VM2)

1. Curl –X POST –d ‘{“src-ip”: “10.0.0.10/16”, “dst-ip”: “10.0.0.14/16”, “nw-proto”:“udp”,“action”:“ALLOW”}’
   http://<controller_ip>:8080/wm/acl/rules/json
   Curl –X POST –d ‘{“src-ip”: “10.0.014./16”, “dst-ip”: “10.0.0.10/16”, “nw-proto”:“udp”, “action”:“ALLOW”}’
   http://<controller_ip>:8080/wm/acl/rules/json
2. Curl –X POST –d ‘{“switch id”:“00:00:00:00:00:00:00:14”}’
   http://<controller_ip>:8080/wm/acl/rules/json
3. Curl –X POST –d ‘{“src-ip”:“10.0.0.13/16”,“dst-ip”:“10.0.0.12/16”,“action” :“ALLOW”}’
   http://<controller_ip>:8080/wm/acl/rules/jsonCurl –X POST –d ‘{“src-ip”:“10.0.0.13/16”,“dst-ip”:“10.0.0.12/16”,“action” :“DENY”}’
   http://<controller_ip>:8080/wm/acl/rules/json

Network 3 (VM3)

1. Curl –X POST –d ‘{“src-ip”: “10.0.0.4/16”, “dst-ip”: “10.0.0.12/16”, “nw-proto”:“udp”,“action”:“DENY”}’ http://<controller_ip>:8080/wm/acl/rules/json
   Curl –X POST –d ‘{“src-ip”: “10.0.0.12/16”, “dst-ip”: “10.0.0.4/16”, “nw-proto”:“udp”, “action”:“DENY”}’ http://<controller_ip>:8080/wm/acl/rules/json
2. Curl –X POST –d ‘{“switch id”:“00:00:00:00:00:00:00:14”}’ http://<controller_ip>:8080/wm/acl/rules/json
3. Curl –X POST –d ‘{“src-ip”:“10.0.0.14/16”,“dst-ip”:“10.0.0.3/16”,“action” :“DENY”}’ http://<controller_ip>:8080/wm/acl/rules/jsonCurl –X POST –d ‘{“src-ip”:“10.0.0.3/16”,“dst-ip”:“10.0.0.14/16”,“action” :“ALLOW”}’ http://<controller_ip>:8080/wm/acl/rules/json

APPENDIX I

Floodlight launch from eclipse

1. Open eclipse
2. Open File
3. Import from Existing Workspace
4. Open Folder containing Floodlight controller (i.e. Home → Floodlight)
5. Highlight floodlight (in the box Projects)
6. Finish (Note: Do not Interrupt 'Building Workspace')
7. Right-Click "Floodlight" in project explorer
8. Run as – "Run configuration"
9. Click new launch configuration, under Java Application
10. Name - FloodlightLaunch
11. Main Class – net.floodlightcontroller.core.Main
12. Click Apply
13. Close the window
14. Run FloodlightLaunch from Java Application

APPENDIX J

Creating a module

1. Open Eclipse
2. Open File
3. Import from Existing Workspace
4. Select “Floodlight” from the Home directory (or directory containing floodlight)
5. Highlight floodlight (in the box Projects)
6. Finish (Note: Do not Interrupt ‘Building Workspace’)
7. Inside the project explorer, select click on “Floodlight”
8. Select “src/main/java” folder
9. Right-click New
10. Select Class
11. Package module name – net.floodlightcontroller.name of package (for example; net.floodlightcontroller.secureSDNi)
12. Name – Producer (this is the class name)
13. Click interface to Add “IFloodlightModule” (optional: you can select more interfaces to add necessary interfaces)
14. Click Finish
15. The skeleton of the package module is created.

Registering a module

1. Open “Floodlight”
2. Open “src/main/resource”
3. Open “floodlightdefault.properties”
4. Add the module name (for example; net.floodlightcontroller.secureSDNi.Producer)
5. Note: Previous line before your module name; add a “ \ ” after the comma (,).
6. Save
7. Open the folder “META-INF-SERVICES” under “src/main/resource”
8. Open “net.floodlightcontroller.core.module.IFloodlightModule”
9. Add the module name (for example; net.floodlightcontroller.secureSDNi.Producer)
10. Save