SUDAN UNIVERSITY OF SCIENCE & TECHNOLOGY

COLLEGE OF ENGINEERING

ELECTRONICS ENGINEERING DEPARTMENT

Designing Dynamic Consistency for Multi-Controller Software Defined Network Topologies

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

Prepared By:

1. Ahmed Mahjoub MohamedAhmed
2. Alnazeer Mohamed Musa
3. Mohamed Abd-Elmonem Ahmed
4. Mohamed Wida'a-tlla Alameen

Supervised By:

Dr. Ahmed Abdalla Mohamed

October 2017

قال تعالي :

{ يَا أَيُّهَا الَّذِينَ آمَنُوا إِذَا قِيلَ لَكُمْ تَفَسَّحُوا فِي الْمَجَالِسِ فَافْسَحُوا يَفْسَحِ اللَّهُ لَكُمْ وَإِذَا قِيلَ انْشُزُوا فَانْشُزُوا يَرْفَعِ اللَّهُ الَّذِينَ آمَنُوا مِنْكُمْ وَالَّذِينَ أُوتُوا الْعِلْمَ دَرَجَاتٍ وَاللَّهُ بِمَا تَعْمَلُونَ خَبِيرٌ }

سورة المجادلة

Dedication

This thesis is dedicated with all sincerity

To our families …

To the ones who brought love to our lives …

To our friends and colleagues whom we spent with the most beautiful moments …

To everyone who've supported us …

Acknowledgment

First thank to Allah who’ve guided us through our lives till we reached this point.

We would like to express our special thanks of gratitude to our families who helped us through the way.

A debt of gratitude to (Dr. Ahmed Abdalla Mohamed Ali), Our beloved thesis adviser, without his expertise the completion of this study could not have been possible.

We would also like to thank everyone who supported us academically regardless of that support

Abstract

Motivated by the internet of the future, which will likely be considerably larger in size as well as highly decentralized, the trending research topic in the field of Software Defined Networking is the distribution of the control plane in order to meet the needs of the internet. In this research the concept of dynamic controllers is introduced into Software Defined Networking, these controllers can tune their own configurations in real-time in order to enhance the performance of the network. We used the consistency models in the context of distributed Software Defined Networks controllers as our tunable configuration. The project uses Mininet emulation environment, and POX as a controller to control this environment, the output of this research is a performance comparison of a proof-of-concept distributed load-balancing application when it runs on top of our proposed dynamic controller versus the usual static controller. The results showed that using dynamic consistency over the traditional static consistency is more beneficial in terms in network efficiency.

Abstract in Arabic

بدافع من إنترنت المستقبل والذي من المرجح أن يكون أكبر بكثير في الحجم وكذلك غير مركزي للغاية , سيصبح اهم مواضيع البحث في الوقت الحالي في مجال الشبكات المعرفه بالبرمجيات , هو توزيع مستوى التحكم للتماشى مع احتياجات الإنترنت , في هذا البحث يتم إدخال مفهوم وحدات التحكم الديناميكية في الشبكات المعرفه بالبرمجيات , يمكن لهذه الوحدات تعديل الضبط الخاص بها لحظياً من أجل تعزيز أداء الشبكة .استخدمنا انواع التطابق في سياق الشبكات المعرفه بالبرمجيات كالضبط الذي سيتم التحكم فيه ديناميكياً. يستخدم المشروع محاكي Mininet كالبيئة التي سيتم فيها عملية المحاكاة ،والمتحكم POX كوحدة تحكم للسيطرة على هذه ألبيئة. ناتج هذا البحث هو مقارنة أداء تطبيق "موازنة التحميل" عندما يتم استخدام التطابق الديناميكي مقابل استخدام التطابق الإستاتيكي المعتاد. وأظهرت النتائج أن استخدام التطابق الديناميكي على التطابق الإستاتيكي التقليدي هو أكثر فائدة من حيث كفاءة الشبكة.

Table of contents

List of Figures

List of Abbreviation

API	Application Programmable Interface
SDN	Software-Defined Networking
ONF	Open Networking Foundation
NBI	Northbound Interface
CDPI	Control to Data-Plane Interface
NFV	Network Function Virtualization (NFV)
GNS3	Graphical Network Simulator-3
NS3	Network Simulator 3
GUI	Graphical User Interface
QoS	Quality of Service
CLI	Command Line Interface
SNMP	Simple Network Management Protocol
SPOF	Single Point of Failure
DISCO	Distributed SDN Control plane
AMQP	The Advanced Message Queuing Protocol
ICMP	Internet Control Message Protocol

Chapter 1 Introduction

Chapter 1

Introduction

1.1. Introduction:

This chapter provides a brief overview of the literature review, problem definition, proposed solution, aim and objectives, in addition to the thesis outline.

"Software-Defined Networking (SDN)" is a term of the programmable networks paradigm. In short, SDN refers to the ability of software applications to program individual network devices dynamically and therefore control the behavior of the network as a whole. SDN is a set of techniques used to facilitate the design, delivery, and operation of network services in a deterministic, dynamic, and scalable manner,(Haleplidis et al., January 2015) its common deployment model is by employing a point of logically centralized network control which then orchestrates, mediates, and facilitates communication between applications wishing to interact with network elements and wishing to convey information to those applications. The controller then exposes and abstracts network functions and operations via modern, application friendly and bidirectional programmatic interfaces. (Gray and Nadeau, August 2013)

Among many benefits, SDN eliminates the rigidity present in traditional network and make it easier to build application for enterprise networks, data centers, internet exchange points, home networks and backbone/WAN. basically because it enables customizing the data plane to perform functions other than match-action like traffic shaping.

SDN changes the way of designing, configuring and managing networks. By decoupling the control plane from the data plane the chance of creating secure network is increased, and with a centralized controller the overall view and management of a network is becoming much easier. While this simplifies the implementation of the control logic, it has scalability limitations as the size and dynamics of the network increase. To overcome these limitations, several approaches have been proposed that fall into two categories, hierarchical and fully distributed approaches. In hierarchical solutions,(S.H. Yeganeh and Ganjali, 2012) distributed controllers operate on a partitioned network view, while decisions that require network-wide knowledge are taken by a logically centralized root controller. In distributed approaches,(Koponen, 2010) controllers operate on their local view or they may exchange synchronization messages to enhance their knowledge. Distributed solutions are more suitable for supporting dynamic SDN applications.

1.2. Problem statement:

The design of SDN applications that run on top of distributed controllers is a non-trivial task due to the complexity of handling controllers' state synchronization which in-turn can affect the applications' performance. Inconsistency between these distributed controllers can significantly degrade the performance of SDN applications.

Furthermore, beside application performance degradation, inconsistency can create other severe problems in the network such as forwarding loops, black holes and isolation and reachability violation.

Performance degradation can also be caused by using inappropriate consistency policies because certain network states need certain consistency policies. That's why using static consistency policies become inappropriate when the network state changes.

1.3. Proposed solution:

The solution to this problem is to emulate a SDN with multi-controllers using Mininet network emulator and then implement a dynamic mechanism for maintaining a consistent view of the network among all controllers and altering the consistency policies according to the network state, and then a proof-of-concept distributed load balancing SDN application is implemented and finally compare its performance when run on-top of: (1) static consistency controllers and (2) dynamic consistency controllers.

1.4. Objectives:

The main objectives of this research are to:

- Emulate a testbed topology with two controllers.
- Implement an interface between the controllers.
- Design and test the load balancing application that supports static and dynamic consistency.
- Run a performance test on the load balancing application when applying static and dynamic consistency then compare the results.

1.5. Thesis outlines:

The rest of this thesis is organized as follows:

Chapter 2: A theoretical background of the proposed work is presented. Also this chapter presents some SDN-related concepts that is relevant to this thesis, it also explores some technologies and concepts that forms the road map of SDN. Then a review of the limitations of multi-controller architectures is discussed and an overview of the related work is performed on those aspects.

Chapter 3: Describes the tools and technologies used in the implementation phase. Both network virtualization and SDN tools were used. In this chapter, all the steps taken to implement the network are explained, and a step by step description for the design of static consistency and dynamic consistency modules.

Chapter 4: Shows the results obtained from testing scenarios. This chapter verifies benefits of using dynamic consistency over static consistency to the proposed network topology.

Chapter 5: Aims to draw the final remarks and conclusions of the presented work. Proposed optimizations and complementary future work are also presented.

Chapter 2 LITERATURE REVIEW

Chapter 2

Literature Review

2.1. Introduction:

Software defined networking is a promising network Architecture. SDN has emerged as an efficient network technology capable of supporting the dynamic nature of future network functions and intelligent applications while lowering operating costs through simplified hardware, software, and management. The term software-defined networking (SDN) has been coined in recent years. However, the concept behind SDN has been evolving since 1996.

SDN implementation opens up a means for new innovation and new applications. Dynamic topology control (i.e., adjusting switch usage depending on load and traffic mapping) becomes possible with the global network view. (Sezer et al., July 2013)

In SDN it is possible to use distributed-central controllers to achieve high efficiency and scalability. Using multi-controller without achieving consistency between controllers can cause problems. Our research will focus on how to achieve consistency between multi-controllers.

In this chapter we define the software Defined Network and present the architecture of SDN, as well as network management and control. We discuss previous researches related to the context of this thesis and covering the benefits and limitations of SDN.

2.2. Software-Defined Networking:

Software Defined Networking (SDN) is an emerging network architecture where network control is decoupled from forwarding and is directly programmable. This migration of control, formerly tightly bound in individual network devices, into accessible computing devices enables the underlying infrastructure to be abstracted for applications and network services, which can treat the network as a logical or virtual entity.

By centralizing network state in the control layer, SDN gives network managers the flexibility to configure, manage, secure, and optimize network resources via dynamic, automated SDN programs. Moreover, they can write these programs themselves and not wait for features to be embedded in vendors' proprietary and closed software environments in the middle of the network.

In addition to abstracting the network, SDN architectures support a set of APIs that make it possible to implement common network services, including routing, multicast, security, access control, bandwidth management, traffic engineering, quality of service, processor and storage optimization, energy usage, and all forms of policy management, custom tailored to meet business objectives. For example, an SDN architecture makes it easy to define and enforce consistent policies across both wired and wireless connections.(Jammal et al., October 2014)

2.3. Traditional Network vs. SDN

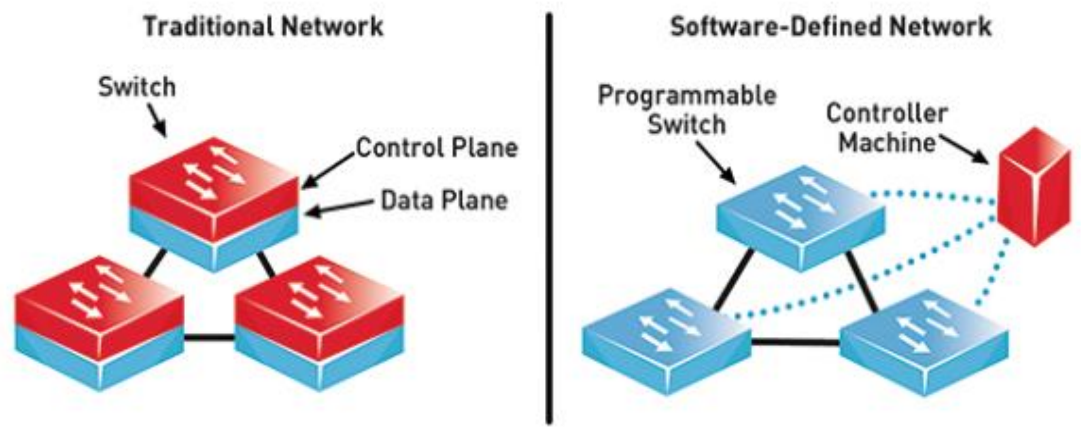
2.3.1. Traditional Network

Network Devices in traditional network consists of Control plane and Data plane. The control plane provides information used to build a forwarding table. The data plane consults the forwarding table to make a decision on where to send frames or packets entering the device. Both of these planes are exist directly on the networking device. (HumayunKabir, Augest 2013)

2.3.2. Software Defined Network

According to the Open Networking Foundation (ONF) the definition of SDN is an emerging network architecture where network control is decoupled from forwarding and is directly programmable. (Foundation, 28/5/2017)

Control Plane functions are removed from individual networking devices and hosted on a centralized server (Controller). The SDN controller can be a server running SDN software. The Controller communicates with a physical or virtual switch Data Plane through a protocol called OpenFlow. OpenFlow conveys the instructions to the data plane on how to forward data. The network device must run the OpenFlow protocol for this to be possible.(HumayunKabir, Augest 2013)



(HumayunKabir, Augest 2013)

Figure 2-1 Traditional and software-defined network

2.4. SDN Architecture:

The aim of SDN is to provide open interfaces enabling development of software that can control the connectivity provided by a set of network resources and the flow of network traffic though them, along with possible inspection and modification of traffic that may be performed in the network.

Figure 2-2 depicts a logical view of the SDN architecture. Network intelligence is (logically) centralized in software-based SDN controllers, which maintain a global view of the network. As a result, the network appears to the applications and policy engines as a single, logical switch.

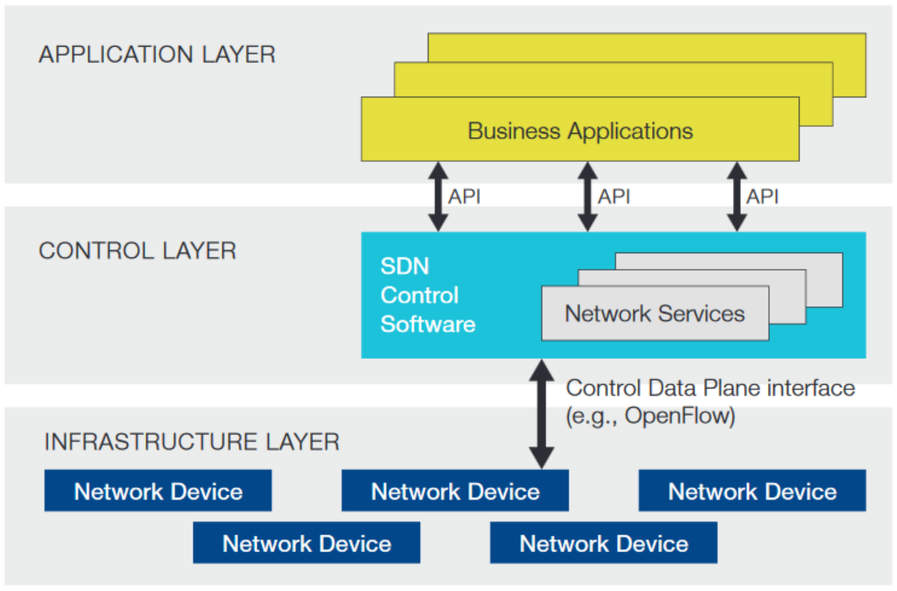


Figure 2-2 Logical view of the SDN architecture

The Figure is a graphical representation of the architectural components and their interactions.(hoang, 28/5/2017)

2.5. Architectural components

2.5.1. SDN Application:

SDN Applications are programs that explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN Controller via a Northbound Interface (NBI).

2.5.2. SDN controller:

The control plane is placed at a central device called the Controller (sometimes a distributed-central controllers). The controller is responsible for translating the requirements from the SDN Application layer down to the SDN Data paths, providing the SDN Applications with an abstract view of the network, building the forwarding base and figuring out how a packet should be forwarded through a network.

2.5.3. Data plane:

Represents the forwarding devices on the network (routers, switches, etc.). It uses the southbound APIs to interact with the control plane by receiving the forwarding rules and policies to apply them to the corresponding devices.

2.5.4. SDN Control to Data-Plane Interface (CDPI):

The SDN CDPI (also called Southbound API) is the interface defined between an SDN Controller and an SDN Data path, which provides at least (i) programmatic control of all forwarding operations, (ii) capabilities for advertisement, (iii) statistics reporting, and (iv) event notification. One value of SDN lies in the expectation that the CDPI is implemented in an open, vendor-neutral and interoperable way. A common open standard SDN protocol, and one of the most popular options for southbound APIs is OpenFlow.

2.5.5. SDN Northbound Interface (NBI):

SDN NBI is the interface between SDN Application and SDN Controller and typically provide abstract network views and enable direct expression of network behavior and requirements.

2.6. SDN Controllers:

The control plane (Controller) presents an abstract view of the complete network infrastructure, enabling the administrator to apply custom policies/protocols across the network hardware. SDN controllers are based on protocols, such as OpenFlow, that allow servers to tell switches where to send packets.(Jammal et al., October 2014)

Open source SDN controllers have evolved over the years. Some examples of this controllers,

2.6.1. NOX

NOX was developed by Nicira and donated to the research community and hence becoming open source in 2008.The first highly popular OpenFlow controller was NOX. NOX is often used in academic network research to develop SDN applications. NOX being programmed primarily in C++. (Gray and Nadeau, August 2013)

2.6.2. POX

NOX's successor, POX, was built as a friendlier alternative and has been used and implemented by a number of SDN developers and engineers. Compared to NOX, POX performs well compared to NOX applications written in Python, POX has an easier development environment to work with and a reasonably well written API and documentation it's also provides a web based GUI and is written in Python.(Gray and Nadeau, August 2013)

2.6.3. Beacon

Beacon has been in development since early 2010, and has been used in several research projects, networking classes, and trial deployments. Beacon is a very well written and organized SDN controller written in Java and it’s also runs on many platforms.(Beacon, 3/9/2017)

2.6.4. Floodlight

Floodlight is a very popular SDN controller contribution from Big Switch Networks to the open source community. Floodlight is based on Beacon from Stanford University. Floodlight is an Apache-licensed, Java-based OpenFlow controller. Floodlight has a very active community and has a large number of features that can be added to create a system that best meets the requirements of a specific organization.(Gray and Nadeau, August 2013)

2.6.5. RYU

RYU is an open source controller. The name comes from a Japanese word meaning "flow”. Ryu provides software components with well-defined API that make it easy for developers to create new network management and control applications. RYU controller written in python programming language. (Ryu, 28/5/2017)

2.6.6. OpenDayLight

OpenDayLight is a Linux Foundation collaborative project that has been highly supported by Cisco, Big Switch, and several other networking companies. The goal of the project is to promote software defined network (SDN) and network function virtualization (NFV). Like Floodlight, OpenDayLight is written in Java programming language and is a popular, well-supported SDN controller. (ODL, 3/9/2017)

2.7. SDN Emulators:

We can use multiple emulators to implement SDN networks like:

2.7.1. GNS3

Graphical Network Simulator-3 (GNS3) GNS3 offers an easy way to design, build and test networks of any size in a virtual environment without the need for hardware.(GNS-3, 16/7/2017)

2.7.2. NS3

Network Simulator3 (NS3) is a network simulator for Internet systems, targeted primarily for research and educational use. ns-3 is free software and is publicly available for research, development, and use.(NS-3, 28/5/2017)

2.7.3. Mininet

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 Software-Defined Networking. Mininet supports research, development, learning, testing, debugging, and any other tasks that could benefit from having a complete experimental network on a laptop or other PC. (overview, 28/5/2017)

2.7.4. EstiNet

EstiNet network simulator and emulator It became a commercial software on 2011 EstiNet's user-friendly GUI provides users a convenient way to construct a simulated network and a visual display for simulation result observation and debugging. (EstiNet, 28/5/2017)

2.8. SDN Elements:

2.8.1. Separation of control plane

Separation of control plane means that the decision about how to handle traffic is not made by the entity handling the traffic, and all policy decisions in the network are made by a centralized controller.

Separation of control requires a well---defined and standard API Between the controller and the network device, so that the two can be logically separated.

OpenFlow Is defined as a network protocol so that the physical separation can also be achieved on top of the logical separation.

The Controller owns all the network policies, and uses the mechanisms in the API to enforce those policies. The OpenFlow Protocol must therefore offer enough control and visibility to enforce those policies. The Split of responsibility between the controller and the network device is evolving. Complex Processing may be offloaded to the data path however; the policy decision must rest in the control function. (HumayunKabir, December , 2014)

2.8.2. Logical centralization

A Logically centralized control plane is a departure from traditional network protocols which are mainly distributed. However, Experience has shown that some traffic engineering problems, such as QoS and load balancing, can better be solved with a global view of the network and its policies. Many Other aspects of networking can also benefit from global optimization.

This Centralization means that the controller needs to be able to fully control all network devices within the policy domain. This Also means that the network devices must offer APIs for the controller to derive the topology, and to implement the monitoring and control of network resources across multiple devices.

2.8.3. Programmability

Programmability enables automation software that can react and reprogram the network without involving humans in the critical path. Previous Interfaces to network devices were mostly designed for human interaction (the CLI) or narrow management functions (SNMP: Simple Network Management Protocol).

The SDN Framework must be flexible enough to handle all kinds of network devices, rather than one API per device type. This Means dealing with both hardware and software devices, simple forwarding devices and devices with rich and complex behavior.

Having a common framework means that programs which are complex to build can be more easily repurposed to a different context. This Also means that a lot of developments and management tools, such as inspection and debugging tools, can be common.

2.8.4. Flow entries

A Key principle of network design has been the separation of network layers, where the operation of each layer is done without using information from other layers. However, more and more products violate those assumptions and now operate in a cross-layer manner.

The OpenFlow API Needs to offer flexible control and visibility of packet processing which is decoupled from the protocol definitions. The API should enable the collapsing of network layers as needed. Packet Processing should be enabled at any granularity desired, as fine or as coarse as desired and suitable for the deployment.

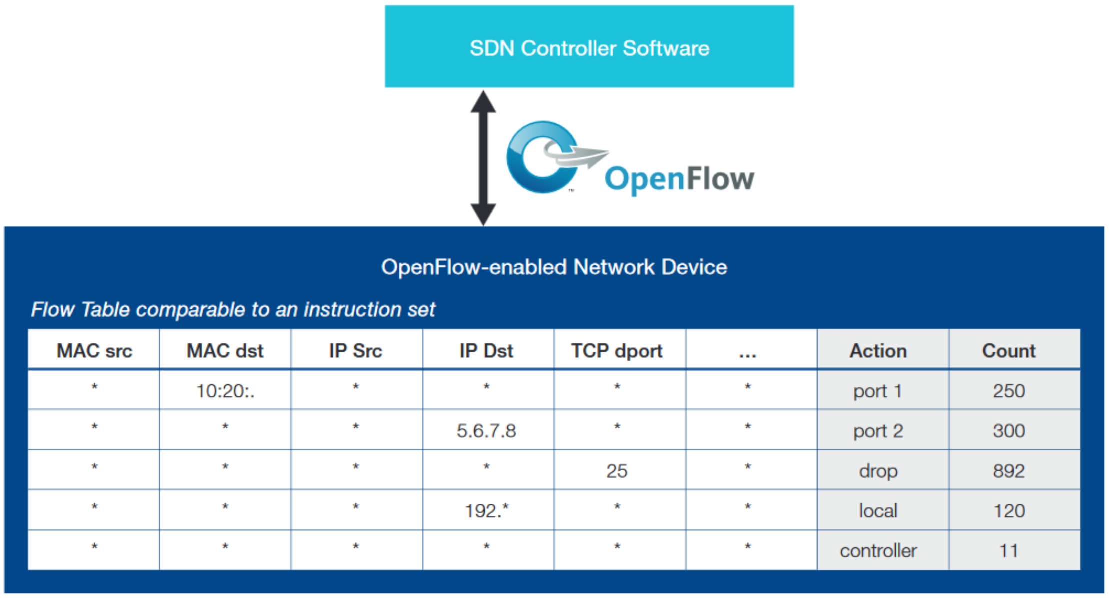
The Solution adopted by OpenFlow Was the concept of a flow entry. The Flow entry match describes a pattern of header values, though some header fields may be omitted (wildcard) or bit-masked. Flow entries enable to select related packets with flexible granularity and across protocol layers, and unrelated flow patterns can be used. In OpenFlow, most processing is attached to flow entries, and therefore flow entries are one of the most important concepts of the API.

2.9. OpenFlow:

OpenFlow is the first standard communications interface defined between the controls and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based).

It is the absence of an open interface to the forwarding plane that has led to the characterization of today's networking devices as monolithic, closed, and mainframe-like. No other standard protocol does what OpenFlow does, and a protocol like OpenFlow is needed to move network control out of the networking switches to logically centralized control software.

OpenFlow can be compared to the instruction set of a CPU. As shown in Figure 2-3, the protocol specifies basic primitives that can be used by an external software application to program the forwarding plane of network devices, just like the instruction set of a CPU would program a computer system.


SDN Controller Software
OpenFlow
OpenFlow-enabled Network Device
Flow Table comparable to an instruction set

MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Figure 2-3 Example of OpenFlow

The OpenFlow protocol is implemented on both sides of the interface between network infrastructure devices and the SDN control software. As shown in the Figure 2-3.(Foundation, April 2012)

OpenFlow uses the concept of flows to identify network traffic based on pre-defined match rules that can be statically or dynamically programmed by the SDN control software. It also allows IT to define how traffic should flow through network devices based on parameters such as usage patterns, applications, and cloud resources. Since OpenFlow allows the network to be programmed on a per-flow basis, an OpenFlow-based SDN architecture provides extremely granular control, enabling the network to respond to real-time changes at the application, user, and session levels. Current IP-based routing does not provide this level of control, as all flows between two endpoints must follow the same path through the network, regardless of their different requirements.

The OpenFlow protocol is a key enabler for software-defined networks and currently is the only standardized SDN protocol that allows direct manipulation of the forwarding plane of network devices. While initially applied to Ethernet-based networks, OpenFlow switching can extend to a much broader set of use cases. OpenFlow-based SDNs can be deployed on existing networks, both physical and virtual. Network devices can support OpenFlow-based forwarding as well as traditional forwarding, which makes it very easy for enterprises and carriers to progressively introduce OpenFlow-based SDN technologies, even in multi-vendor network environments.

The Open Networking Foundation is chartered to standardize OpenFlow and does so through technical working groups responsible for the protocol, configuration, interoperability testing, and other activities, helping to ensure interoperability between network devices and control software from different vendors. OpenFlow is being widely adopted by infrastructure vendors, who typically have implemented it via a simple firmware or software upgrade. OpenFlow-based SDN architecture can integrate seamlessly with an enterprise or carrier's existing infrastructure and provide a simple migration path for those segments of the network that need SDN functionality the most.(Foundation, April 2012)

2.10. Challenges of adopting a Single controller network architecture:

Three critical requirements are not achievable in an SDN-enabled centralized network, which was the main tendency for early proposed SDN architectures, using just one controller: first, efficiency that is not enough established with just one centralized controller, second, scalability that is one of the most issues that pushes network architects to consider the idea of multi-controllers, and, third, high availability, which has two items, redundancy and security. Redundancy is one of the most significant aspects of any design. One controller could fail anytime and, for this reason, abandon the network without its control plane. Security is considered an important item. If an attacker compromises the controller, subsequently it loses the entire management over the network. Clearly, if we have multiple controllers, we can certainly minimize the issue, because they will team up to identify that another one is misbehaving and for that reason separate the attacker from the network.(Blial et al., April 2016)

2.11. Single Controller versus huge networks:

The majority of current SDN architectures, relies on this single or master/slave controllers that is a physically centralized control plane. This centralization, adapted for datacenters, is not suitable for wide multi-technology multi-domain networks. Because the centralized SDN controller represents a Single Point of Failure (SPOF), which makes SDN architectures highly vulnerable to disruptions and attacks.

Also recent studies conducted on the networks of many real-world data centers showed that such networks necessitate the handling of about 150 million flows per second. On the other hand, today's OpenFlow controllers are known handle at most 6 million flows per second on a high end dedicated server with 4 cores. Therefore, implementation of SDN for one of such data center networks requires a controller running either on an appropriate mainframe computer with sufficiently many cores or a server cluster where each server is composed of limited cores. Implementation of the controller on a cluster offers a number of benefits. First, this platform is scalable, as an increasing load on the controller is easily handled by introducing new servers to the cluster. Second, the cluster offers more reliability than an implementation on a single mainframe.

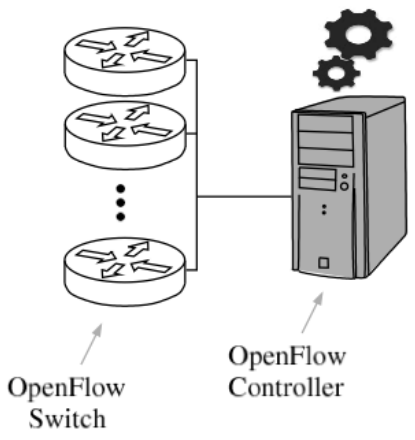


Figure 2-4 Connecting Switches to the controller

2.12. A distributed Multi-domain SDN network architecture:

2.12.1. Multi-domain networks:

Are generally decomposed into administrative or geographical domains interconnected with a large variety of network technologies from high-capacity leased lines to limited-bandwidth satellite links, or from costly but highly secured links to cheap but unsecured ones. The distributed and heterogeneous nature of these environments call for a distributed multi-domain network control plane which should be lightweight, adaptable to user or network requirements, and robust to failures. Current state of the art distributed SDN solutions are not suitable, as they do not provide a fine grain mean to control and adapt inter-controller information exchanges. (A. Dixit et al., 2013)

2.12.2. Distributed SDN control plane for multi-domain SDN networks:

It relies on a per domain organization, where each controller is in charge of an SDN domain, and communicates with neighbor domains using inter-domain channel to exchange aggregated network-wide information for end-to-end flow management purposes. This can be seen in the figure:

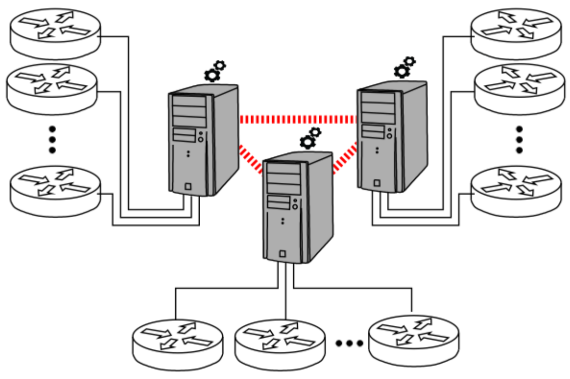


Figure 2-5 Multi-domain SDN network

2.12.3. Trade-offs of distributing the control plane:

The design of SDN applications that run on top of distributed controllers is a non-trivial task due to the complexity of handling controllers' state synchronization which in-turn can affect the application's performance. Levin et al. studied the impact of the inconsistent controllers on the SDN application performance, they found that inconsistency can significantly degrade the performance of SDN applications.(Levin et al., 2012)

Brewer theorem (also known as CAP theorem) stated that it is impossible for a distributed system to simultaneously provide the following guarantees: Consistency, Availability, and Partition tolerance, and that there is always a trade-off between the system's consistency and availability in the presence of network partitions. For example, in the case of a data store cluster that is comprised of a set of distributed nodes. At one point of time, those nodes got partitioned due to a network failure, while new data update requests continued to arrive at some of the nodes. If those nodes continued to handle the requests, the stored data might become inconsistent but the cluster will still be available. Otherwise the data would remain consistent but it would be said that the cluster is unavailable. As the CAP theorem applies to any distributed system, we believe that in SDN that would imply that designing an SDN application that runs on top of physically distributed controllers encounters a trade-off between consistency and availability, in case of network partitions. Panda and et al., investigated how these trade-offs apply to software-defined networks. They concluded that availability and partition tolerance are identical in networks and data stores, however the notion of consistency may differ. In data store systems, the consistency of data across replicas is the primary concern, but in SDN it is the consistent application of policies across the network. (Brewer, 2000)

2.13. Consistency in Software-Defined Networks:

2.13.1. Consistency Models

In distributed systems, the consistency of data among different nodes (different nodes holding copies of same data are known as replicas) is governed by the consistency model being employed. Tanenbaum and Van

Steel, presented different consistency models for distributed data stores. They defined the consistency model as a contract between the applications and the data store, which embodies that if applications agree to obey certain rules, the store promises to work correctly. In the light of their work, the main consistency models of many distributed systems can be categorized into: strong, weak or eventual. In the presence of network partitioning, a strong consistency model would favor consistency to availability, a weak consistency model would favor availability to consistency. While an eventual consistency model, would be in the favor of availability, relax its consistency requirements so that replicas will eventually converge to the same state (i.e. become consistent) in case of no further updates were served. To the best of our knowledge, we believe that most of the research conducted in the area of distributed controllers can be categorized as either strong or eventually consistent controllers.(Tanenbaum and Steen, 2007)

2.13.2. Impact of Inconsistency in Networks

As aforesaid, Levin et al. studied the impact of inconsistency among distributed controllers on the SDN(Levin et al., 2012)application performance, the less consistent the controllers become, and the lower the application performs. Levin et al. only studied the impact of inconsistency on application performance. We also confirm that the inconsistent state information among the controllers or between the controllers and switches had an impact on the application performance.(Aslan and Matrawy, 2015)However, beside application performance degradation, inconsistency can create other severe problems in the network such as forwarding loops, black holes and isolation and reachability violation, which we discuss later in this subsection. Guo et al. identified some of these problems. An example of an application that would employ strong consistency is a security-sensitive firewall application. Such application would probably employ a strong consistency model as it must ensure certain policies are met, and any inconsistency in such policies could led to illegitimate traffic traversing restricted links. On the other hand, an application that would tolerate using a weaker consistency model is a load-balancing application. Such application could tolerate some inconsistency between the controllers, as long as they agree on the least-loaded server in order to avoid creating forwarding loops which we discuss in details next. In such case, non-strongly consistent load-balancers must develop techniques to ensure that all the controllers agree on the least-loaded server, or to be designed in a way to avoid such conflicts. Indeed, this can complicate the design of such SDN application.

Further, it is of great importance to highlight the following critical problems that can occur as result of inconsistency between the SDN controllers. It is worth noting that those problems can also be caused by other means e.g. implementation or misconfiguration bugs (even in case of non-distributed controllers). However, in this paper we are only considering the case where those problems were caused as result of controllers' inconsistency.

1) Forwarding loops.

2) Black holes.

3) And Isolation and Reachability Violation.(Khurshid et al., 2012)

2.14. Related work:

Parallel open source initiatives such as NOX, Beacon, Floodlight, Ryu etc. Researchers mainly focused on improving the performance of a specific controller, like Maestro and NOX or demonstrating the improvement offered by OpenFlow against a classic L2 paradigm.(Koponen, 2010)

Several attempts have been done to tackle the problem of scaling SDNs. A first class of solutions, such as DIFANE and DevoFlow, address this problem by extending data

Plane mechanisms of switches with the objective of reducing the load towards the controller. DIFANE tries to partly offload forwarding decisions from the controller to special switches, called authority switches. Using this approach, network operators can reduce the load on the controller and the latencies of rule installation. DevoFlow, similarly, introduces new mechanisms in switches to dispatch far fewer 'important' events to the control plane.

A second class of solutions proposes to distribute controllers. HyperFlow, Onix, and Devolved controllers try to distribute the control plane while maintaining a logically

Centralized using a distributed file system, a distributed hash table and a pre-computation of all possible combinations Respectively. These approaches, despite their ability to distribute the SDN control plane, impose a strong requirement: a consistent network-wide view in all the controllers.(Koponen, 2010)

On the contrary, Kandoo proposes a hierarchical distribution of the

Controllers based on two layers of controllers:

1. the bottom layer, a group of controllers with no interconnection, and no knowledge of the network-wide state, and
2. The top layer, a logically centralized controller that maintains the network wide state.(S.H. Yeganeh and Ganjali, 2012)

Google has presented their experience with B4 (Jain, 2013), a global SDN deployment interconnecting their datacenters with a centralized Traffic Engineering service and clusters of controllers in each data center. In addition, the work of D. Levin et al(Levin et al., 2012) analyzes the trade-off between centralized and distributed control states in SDN, while the work of B. Heller et al proposes a method to optimally place a single controller in an SDN network.

Distributed SDN Control plane (DISCO)(Phemius et al., 2014) is a distributed controllers' platform that was designed for multi domain SDN networks, it is built on top of Floodlight(Floodlight, 15/6/2015)SDN controllers, and employs an AMQP-based publish/subscribe messaging module. To support other functionalities such as QoS, DISCO uses agents that can be dynamically be added at the different controllers.

Chapter 3 Methodology

Chapter 3

Methodology

3.1. Introduction:

This chapter demonstrates how to implement a dynamic mechanism on software defined networks for maintaining a consistent view of the network among all controllers and allowing the controllers to tune their own configurations in real-time according to the network state in order to enhance the performance of the applications running on top of them. This could alleviate some of the emerging challenges in SDN that could have an impact on the performance, security, or scalability of the network. The development and testing environment will be Mininet emulator. Mininet is an emulator which works based on Linux operating system used for network simulation and testing. Also we will use Pox controller this controller will be programmed using python programming language.

To give a better understanding of how this project works this chapter is divided into five sections, the first section abstracts a short brief about our proposed solution to solve the research problem, the second section gives an abstraction of what was done throughout the project, the third section gives the proposed topology, the fourth section describes the environment tools and technology in which the project is implemented, the last section contains the configuration and software.

3.2. Proposed Dynamic Consistency:

The problem of inconsistency between different controllers in SDN network can degrade the performance of SDN applications, also inconsistency can create other severe problems in the network such as forwarding loops, black holes and isolation and reachability violation. To overcome this problem, we will implement a dynamic mechanism for maintaining a consistent view of the network among all controllers this will let the controllers to be Dynamic Controllers. The dynamic controller is the controller that can autonomously and dynamically tune its configuration in order to achieve a certain level of performance measured in predefined metrics and based on its requirements. In the case where the tunable configuration is the consistency level, we call it a dynamically consistent controller. In other words, a dynamically consistent controller is one that can tune its level of consistency in order to reach the desired level of performance based on specific metrics.

There are a number of reasons, which we believe are enough, for justifying and implementing the concept of dynamic consistency in SDN controllers. Dynamically consistent controllers can:

1) Reduce the complexity at the applications. Without dynamic consistency, application developers would need to implement application-specific consistency models directly into their applications as every application has different requirements. In turn this could contribute to a lower application implementation cost. Nevertheless, the need for dynamically consistent controllers becomes more apparent in case of deploying multiple applications with different requirements where application developers ought to implement multiple consistency models.

2) React rapidly to the changing network conditions. By tracking the applications' performance in real time, dynamically consistent controllers can tune the consistency level in order to maintain a certain performance level based on pre-defined metrics. In other words, dynamically consistent controllers could provide the applications with robustness and reliability against sudden changes in network conditions.

3) Reduce the overhead of controller's state distribution by eliminating unnecessary state distribution messages without compromising the application performance, especially in case strong consistency is not a requirement, or network states do not have to be replicated to all the controllers.

3.3. Research Activities:

1. Studying literatures and related works.
2. Determining an appropriate testbed topology.
3. Installing Ubuntu as the operating system by using VM.
4. Representing the topology by using mininet emulator.
5. Configure and using Pox controller.
6. Implement an interface between the controllers.
7. Design and test the load balancing application with static consistency
8. Implement the dynamic consistency module and integrate it with the load balancing application.
9. Run a performance test on the load balancing application when applying static and dynamic consistency then compare the results.

3.4. The Design structure for the Dynamic Controllers:

At the core of a dynamic controller, lies down the adaptation module. The dynamic module is one that is given a current state for the network, calculates application-specific performance indicators and apply a dynamic strategy in order to find suitable values for the tunable parameters that would maintain the required level of application performance.

The tunable consistency module is one that implements the tunable consistency model and provides a configurable consistency level. In other words, this module encapsulates the complexity of maintaining distributed information across multiple controllers and provides other modules with a uniform interface that can be used to change the consistency of such information in between strong and weak consistency levels.

3.5. Studying SDN Concept:

In this part enough information has been gathered in order to start the implementation of the project by studying courses and reading books and papers.

Based on this literature review both emulator, controller and other necessary tools has been selected, these tools will be discussed later.

3.5.1. Choosing the topology:

A suitable network topology was chosen for testing the load balancing application and to clarify our idea about the concept of the dynamic consistency, the network will consist of two domains each domain has a controller, 32 clients and a server, all connected to a single switch, and the two domains are connected together using a cable between the switches. Each switch will forward the traffic it receives according to the application running on the controller connected to it.

This topology was implemented in Mininet emulator as shown in the figure:

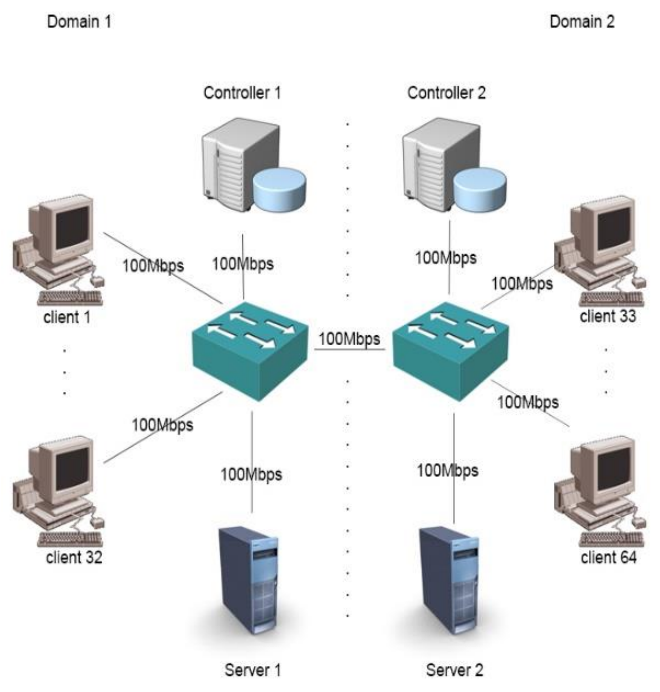


Figure 3-1 Our Topology

3.5.2. Choosing the suitable environment:

Ubuntu was chosen to run the virtual topology using Mininet (the emulator) because it's stable and reliable operating system.

3.5.3. Choosing the emulator:

Mininet was chosen because it allows the user to create, interact with, customize and share a software-defined network (SDN) prototype to simulate a network topology that uses OpenFlow switches, in addition to that is widely used in research. Mininet is an SDN emulator that runs virtual switches that support OpenFlow protocol.

3.5.4. Choosing the controller:

There is a collection of SDN controllers available to implement OpenFlow protocol to control the switches and act as the aggregated control plane, this collection includes NOX, POX, OpenDayLight, Floodlight… Etc.

POX controller was chosen because it has an easier development environment to work with and a reasonably well written API and documentation.

3.6. Environment tools and technology:

Here a brief information about tools and technologies that used in this project:

3.6.1. Oracle VM VirtualBox:

VirtualBox is a cross-platform virtualization application. It can be installed on our existing Intel or AMD-based computers, whether they are running Windows, Mac, Linux or Solaris operating systems. This virtualization software actually create a special environment called Virtual Machine by using system resources like disk space & memory from our existing operating system (Host OS) in which entire operating system (Guest OS) could run. It can run multiple operating systems in our existing computer at the same time. For Example, we can run Linux on our Windows system or run Windows & Linux on our Mac system.

Here Oracle VM VirtualBox is used to run Ubuntu operating system in our Laptop that represent the whole topology.

3.6.2. Mininet network emulator:

Mininet is a software emulator for prototyping a large network on a single machine. Mininet can be used to quickly create a realistic virtual network running actual kernel, switch and software application code on a personal computer. Mininet allows the user to quickly creates a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and its switches support OpenFlow.

One feature of Mininet is Miniedit; it allows editing and configuring the topology through graphical user interface (GUI).

Mininet is used in the project as the emulation environment, it installed in our laptop and acts as the emulation of the topology.

3.6.3. POX Controller:

POX is an open source development platform for Python-based software-defined networking (SDN) control applications. The project goal for POX controller is to use it to create "an archetypal, modern SDN controller."

POX acts as the controller that is able to configure the switches, it can manipulate and redirect the flows on each switch using OpenFlow protocol, which enables rapid development and prototyping. the controller can be thought of as an aggregation of all the control planes of the switches.

The official Mininet emulator come with POX installed.

3.6.4. PuTTY:

PuTTY is a free and open-source terminal emulator, serial console and network file transfer application. It supports several network protocols, and raw socket connection. It can also connect to a serial port. PuTTY was originally written for Microsoft Windows, but it has been ported to various other operating systems.

PuTTY will allow us to easily access the VM through the host machine.

3.6.5. Xming:

Xming is an X display server (X is the windowing capability of Unix, Linux, etc. It is the graphical user interface of most operating systems but not Microsoft's operating systems.) for Microsoft Windows operating systems, including Windows XP or later.

3.6.6. RabbitMQ:

Is an open source message broker software (sometimes called message-oriented middleware) that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover. Client libraries to interface with the broker are available for all major programming languages.

The Advanced Message Queuing Protocol (AMQP) is an open standard application layer protocol for message-oriented middleware. The defining features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security

3.7. Preparation of Virtual Machine and Network Emulator

In this part all steps to build and integrate the system will be explained in details.

3.7.1. Set up the Mininet network simulator:

The easiest way to get started using the Mininet network simulator is to use the Mininet virtual machine. It is based on the Ubuntu Linux Server operating system and comes with all the software and tools required to support Mininet already installed.

The Mininet virtual machine is downloaded from the Mininet site (Mininet, 15/3/2017). This file is a compressed ZIP archive containing two files so, after downloading it, the files are decompressed to import them to VirtualBox in the next step.

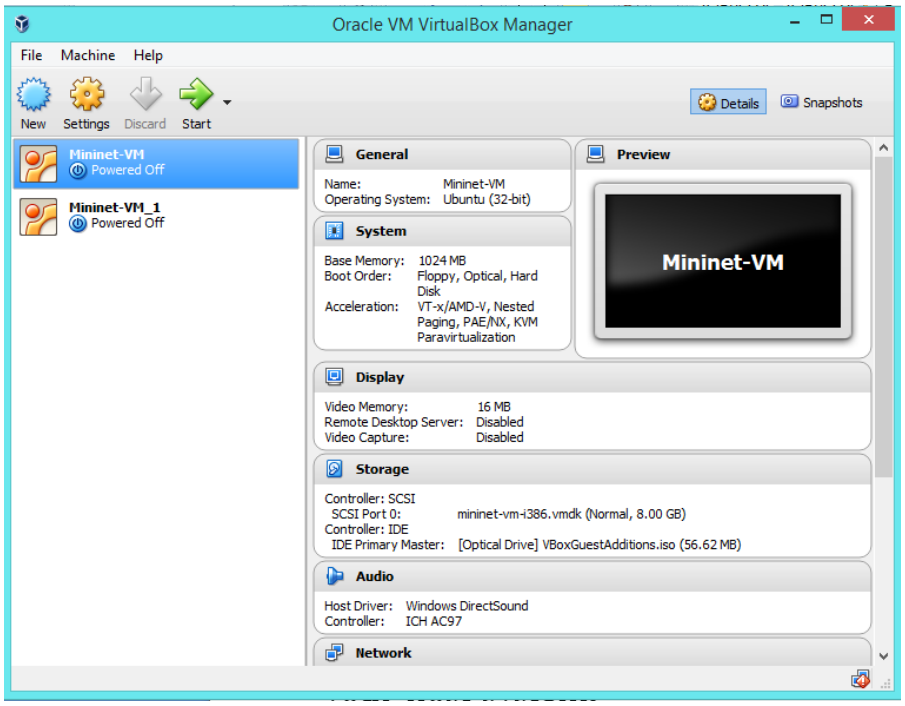
The decompression process will create a folder named *mininet-2.1.0-130919-ubuntu-13.04-server-amd64-ovf*. The folder will contain the following files:

```
mininet-vm-x86_64.vmdk
mininet-2.1.0-130919-ubuntu-13.04-server-amd64.ovf
```

3.7.2. Import the virtual machine into VirtualBox:

Next, a version of the Mininet virtual machine that will run in VirtualBox is created by importing the Mininet virtual machine into the VirtualBox program.

Start the VirtualBox manager application on your host system, to import the Mininet virtual machine, use the VirtualBox menu command:



Name: Mininet-VM
Operating System: Ubuntu (32-bit)

Base Memory: 1024 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

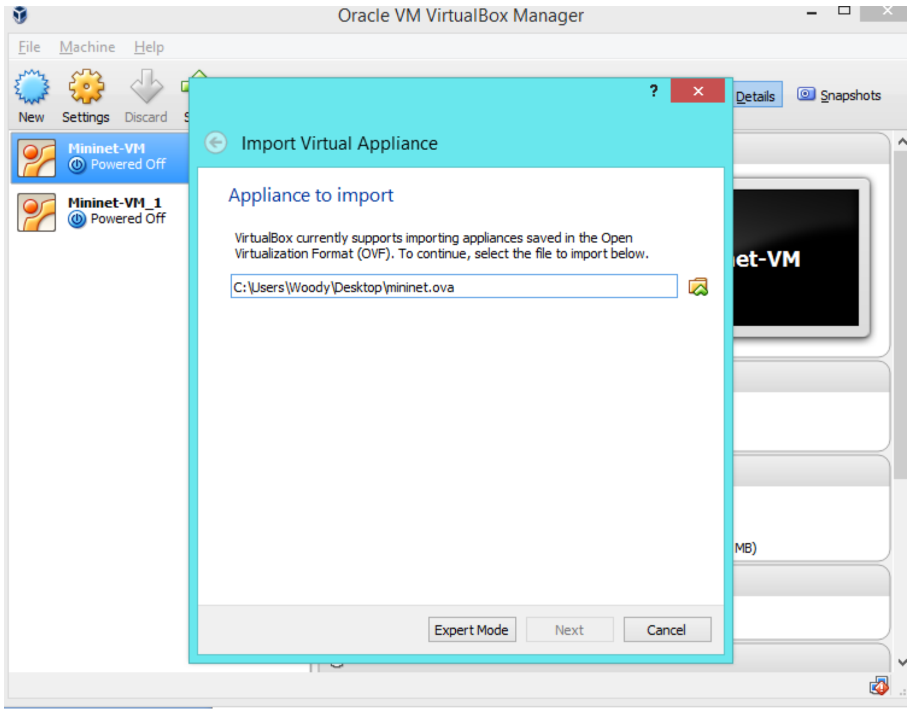
Video Memory: 16 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Controller: SCSI
SCSI Port 0: mininet-vm-i386.vmdk (Normal, 8.00 GB)
Controller: IDE
IDE Primary Master: [Optical Drive] VBoxGuestAdditions.iso (56.62 MB)

Host Driver: Windows DirectSound
Controller: ICH AC97

Figure 3-2 Virtual box main window

File → Import Appliance



Appliance to import

VirtualBox currently supports importing appliances saved in the Open Virtualization Format (OVF). To continue, select the file to import below.

C:\Users\Woody\Desktop\mininet.ova

Figure 3-3 Importing the Mininet-vm

Navigate to the folder containing the *mininet-2.1.0-130919-ubuntu-13.04-server-amd64.ovf* file and select it.

Then, click the “Next” button to get to the *Appliance Settings* screen. Use the default settings. Finally click on the “Import” button and the *Mininet* VM will appear in the VirtualBox window.

3.7.3. Add a Host-only Adapter in VirtualBox

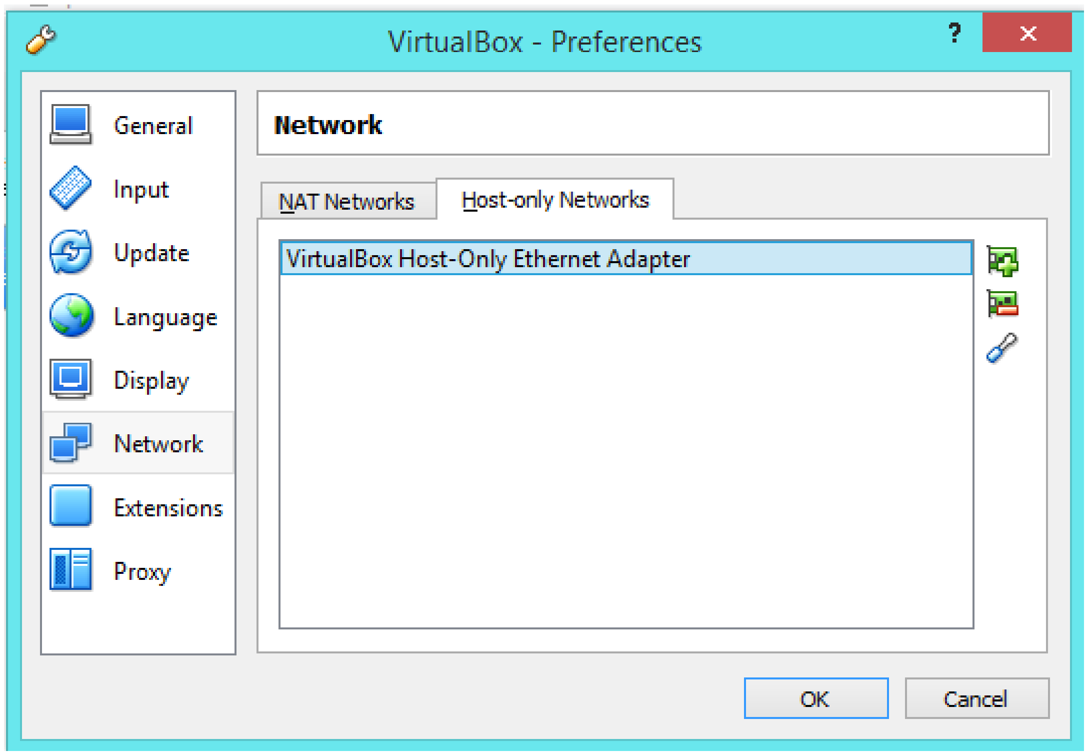
To use Mininet in the way recommended by the Mininet setup notes, a “host only” network interface must be created in VirtualBox. This creates a loopback interface on the host computer that can be used to connect the virtual machine to the host computer (or to other virtual machines). This is needed so that the host computer can run remote X11 sessions on the virtual machine in the later steps.

Open the VirtualBox preferences panel. Use the VirtualBox menu command:

```
VirtualBox → Preferences.
```

Click on the “Network” icon in the Preferences panel. Then. click on the small green “plus” sign on the right side of the window to add a new network adapter. An adapter called *Virtual Host-Only Ethernet Adapter* will be created

.

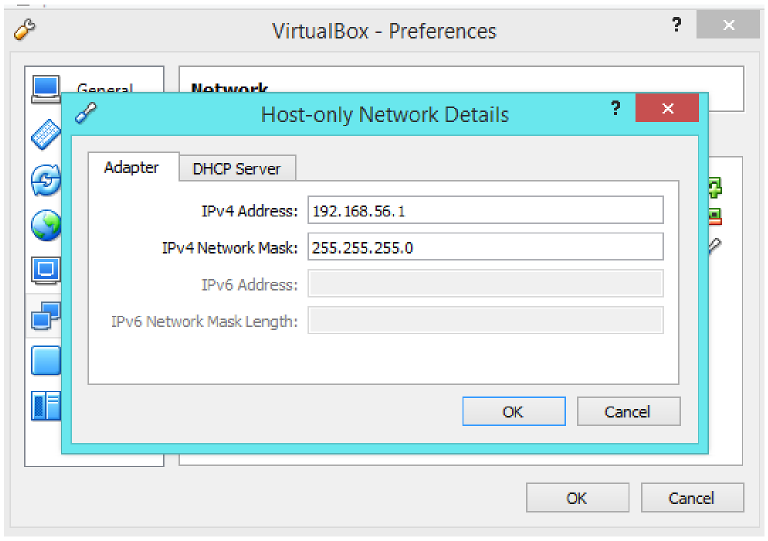


VirtualBox Host-Only Ethernet Adapter

Figure 3-4 Creating a host-only adaptor

Check the settings by clicking on the small "screwdriver" icon on the right side of the window to edit the adapter's configuration. Make a note of the IP address.

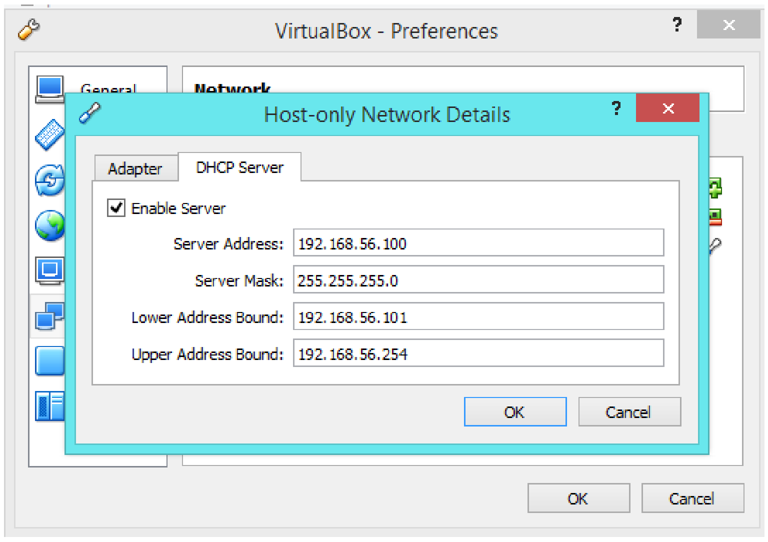
In this case the default IP address used by VirtualBox for the first host-only adapter is 192.168.56.1/24.



IPv4 Address: 192.168.56.1
IPv4 Network Mask: 255.255.255.0
IPv6 Address:
IPv6 Network Mask Length:

Figure 3-5 Configuring the host-only adaptor

The DHCP server is enabled on the interface and we see that the Lower Address Bound is 192.168.56.101/24. So, we know that the IP address of the virtual interface connected to the host-only network on the virtual machine will be assigned that IP address.



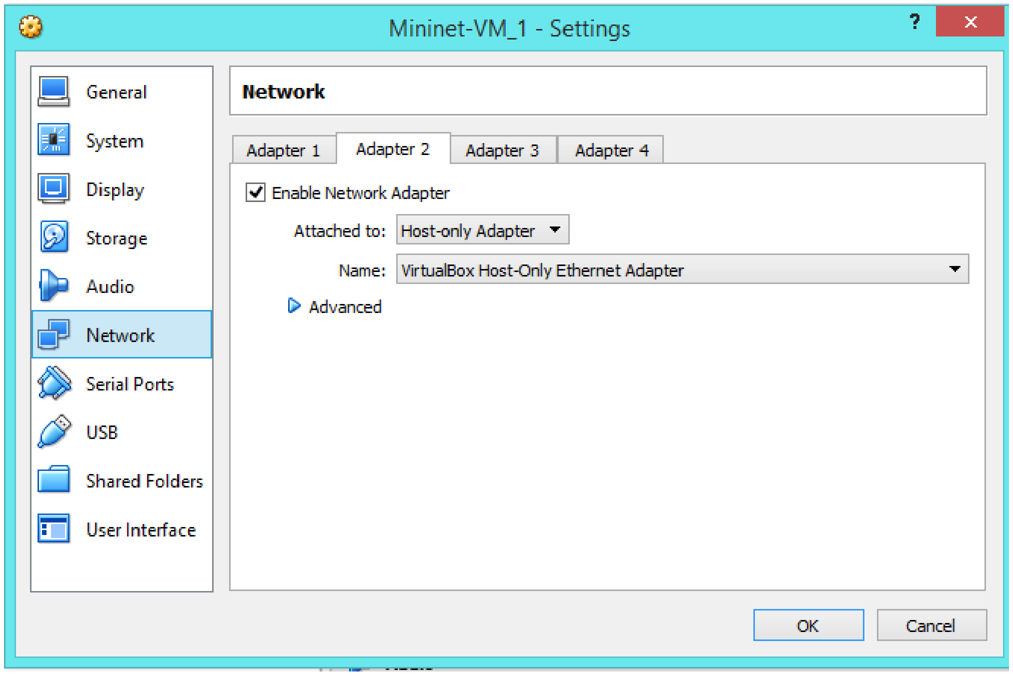
☑ Enable Server
Server Address: 192.168.56.100
Server Mask: 255.255.255.0
Lower Address Bound: 192.168.56.101
Upper Address Bound: 192.168.56.254

Figure 3-6 Configuring the DHCP server

3.7.4. Adding a Network Adapter to Mininet virtual machine

In the VirtualBox Manager window, click on the Mininet virtual machine and then click on the "Settings" icon on the top of the window. Click on the "Network" icon in the settings panel that appears.

Click on the "Adapter 2" tab and, in the "Attached to:" field, select "Host-only network". This allows other programs running on your host computer to connect to the VM using SSH. Since only one host-only network is currently created, VirtualBox will automatically select the "*Virtual Host-Only Ethernet Adapter*" host-only network.



Network

☑ Enable Network Adapter

Attached to: Host-only Adapter

Name: VirtualBox Host-Only Ethernet Adapter

Figure 3-7 Adding the created adaptor to Mininet

Click the "OK" button.

Now the network settings are configured for the Mininet virtual machine.

3.7.5. Start the Mininet virtual machine

In the VirtualBox manager, select the Mininet virtual machine and then click the “Start” button to start the Mininet VM.

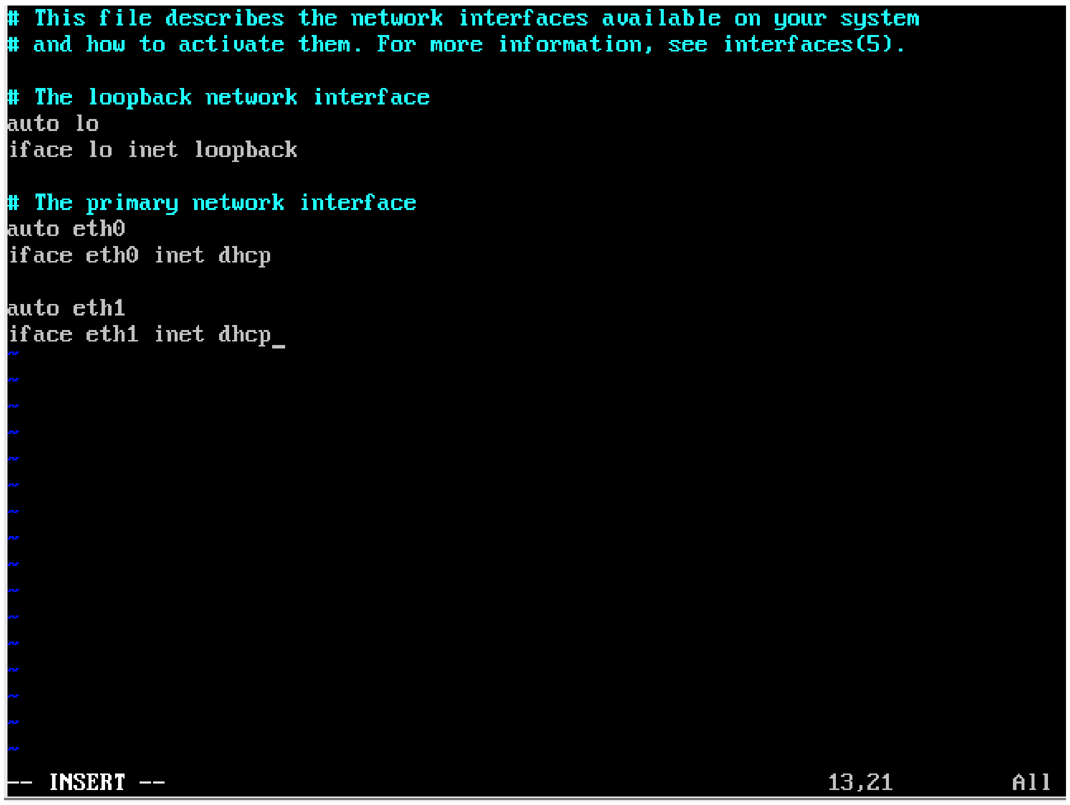
The VM will boot up and present a login prompt, log in with userid=mininet and password=mininet

The first step is to configure the new host-only interface to request an IP from the DHCP every time the machine is booted, by editing the */etc/network/interfaces* file.

```
sudo vi /etc/network/interfaces
```

Then add the following lines to the file:

```
auto eth1
iface eth1 inet dhcp
```



```
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet dhcp

auto eth1
iface eth1 inet dhcp_
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                              13,21          All
```

Figure 3-8 Editing mininet interface file

Save and restart the VM and changes will be applied.

For future use, we note the following information:

-Host-only network address: 192.168.56.0/24

-Virtual Machine’s virtual interface IP address on host-only network:192.168.56.101/24

3.7.6. Using SSH to connect to the Mininet VM:

Putty the SSH client software on the host computer will be used to connect to the Mininet virtual machine. This accomplishes two things:

From the host computer, we can connect to remote X applications running on the Mininet VM, such as *xterm* and *wireshark*.

We can use an easier-to-use *terminal* window or *xterm* window to interact with the Mininet virtual machine. Working with the VirtualBox console window is difficult because:

The VirtualBox console window captures your mouse whenever you use it and you have to use the appropriate “host key” to escape from the virtual machine and return control to your host computer.

You cannot cut-and-paste text from the virtual machine console window to a program on your host computer.

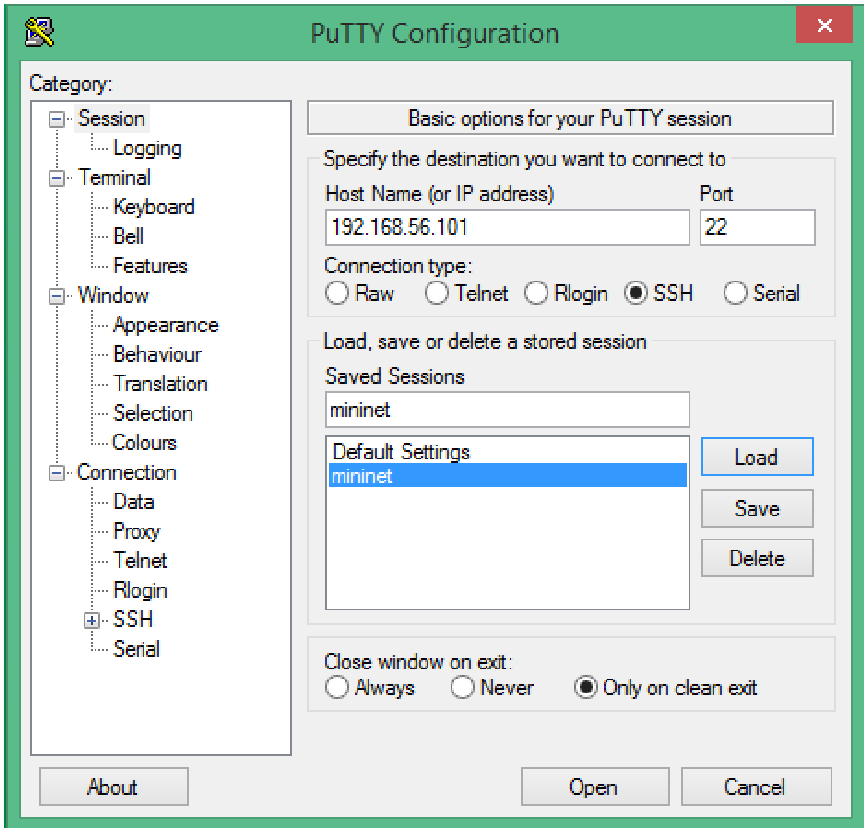
We need to set up an SSH connection to the virtual machine with X11 forwarding enabled so that we can run X applications on the Mininet virtual machine but display the applications on the X Server running on our host computer. Then we can set up an Xterm and stop using the virtual machine console window.

3.7.7. SSH and X11 configuration

On the Mininet VM, SSH forwarding is already enabled (in the */etc/ssh/sshd_config* file). So you do not need to make any SSH configuration changes on the virtual machine.

On the host computer we run Putty and use the Virtual Machine’s virtual interface IP address as the session ip then enable X11 forwarding from the SSH tap

Finally, Xming which is our X11 application is launched and the open button is clicked to start the SSH session.



Specify the destination you want to connect to

Host Name (or IP address): 192.168.56.101

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions: mininet

Default Settings
mininet

Close window on exit: Always Never [✓ Only on clean exit]

Figure 3-9 Putty SSH configuration

3.8. Configuration of Dynamic Load Balancing in the Testbed

In this step we set up the Mininet virtual machine on our host computer and we verified that we can communicate properly with it. We are ready to work with Mininet.

3.8.1. The Topology:

A suitable network topology for testing the load balancing application is simulated, the network consists of two domains each domain has a controller, 32 clients and a server, all connected to a single switch, and the two domains are connected together using a cable between the switches. *As shown in Figure 3-1*

Each switch will forward the traffic it receives according to the application running on the controller connected to it.

Topology generation will be done in two steps, the first one is to simulate a simplified version of the proposed topology using MiniEdit which is a simple GUI editor for Mininet.

It works by dragging and dropping network elements, then an executable Python file representing the topology will be generated to be edited in the second step.

MiniEdit is launched using the command

```
Sudo python mininet/examples/miniedit.py
```

The simplified network consists of two domains each domain has a controller, 2 clients and a server.

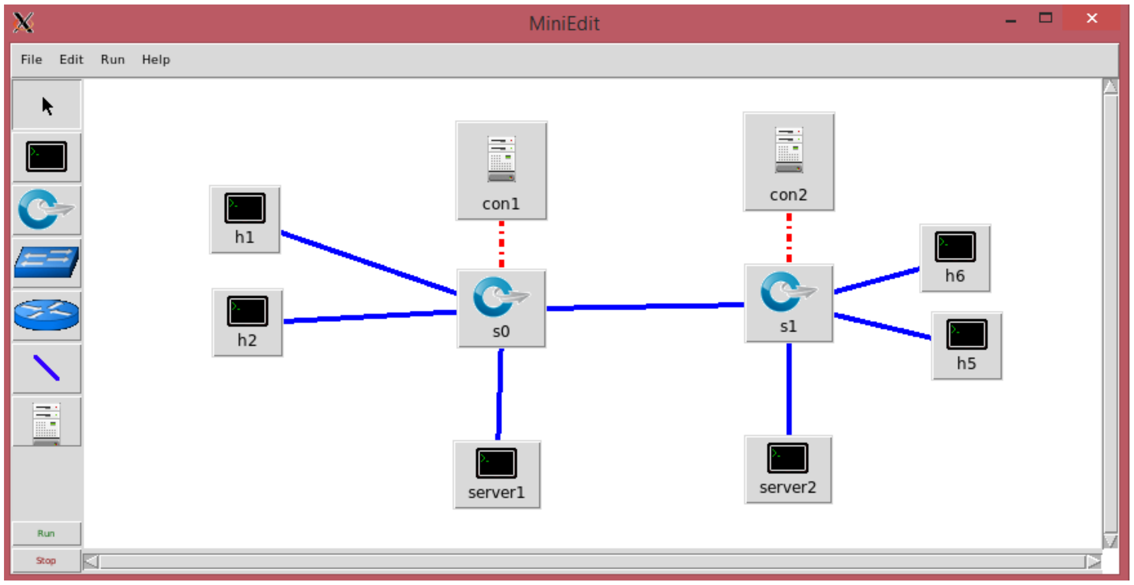


Figure 3-10 Creating the topology using MiniEdit

the executable Python file is generated by clicking:

```
File → Export level 2 script
```

and then saving it for further editing, the file was saved in the directory

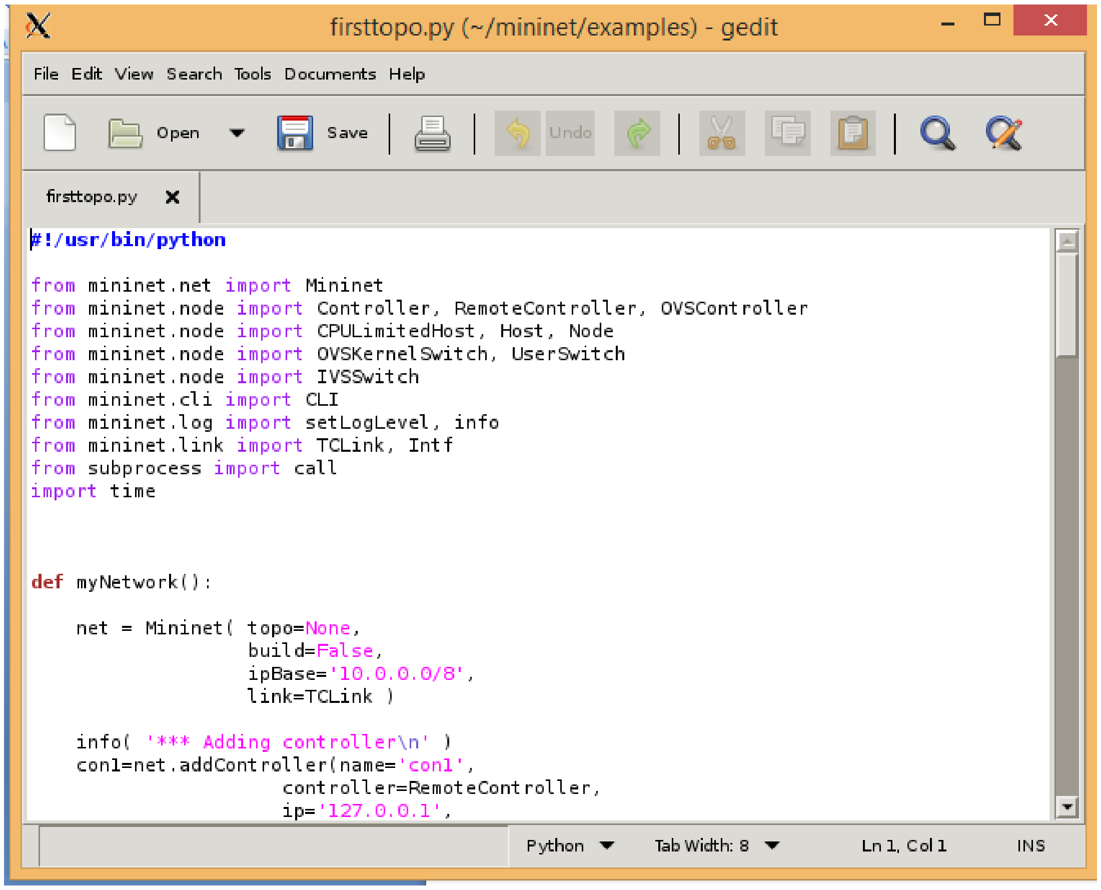
"mininet/examples" under the name "firsttopo.py"

The second step is editing the python code to simulate our proposed topology, we will use Gedit as our python coding environment, the command to install Gedit is:

```
Sudo apt-get install gedit
```

After installation is complete the topology is edited using the

```
Sudo gedit mininet/examples/firsttopo.py
```


firsttopo.py (~/mininet/examples) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
firsttopo.py
Python
Tab Width: 8
Ln 1, Col 1
INS

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
import time


def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8',
                   link=TCLink )

    info( '*** Adding controller\n' )
    con1=net.addController(name='con1',
                      controller=RemoteController,
                      ip='127.0.0.1',
```

Figure 3-11 Editing topology code using Gedit

The code is edited to generate all 64 clients, start the servers and generate the traffic. (As shown in appendix A)

3.8.2. The testing scenario:

The characteristics of the traffic generated by the clients, which consists of ICMP packets, is as follows:

Flow arrival rates at the switches are 2 and 1 flows/sec. In order to simulate a sudden network change, we change the parameters of traffic shortly after 30 secs to 10 flows/sec from each domain.

This test will be run on the network when applying static and dynamic consistency and the captured results will be analyzed and compared

3.8.3. The controller:

The controllers handling the switches are **Pox** controllers, Pox controllers are pre-installed in mininet and to start a controller first you have to navigate to “/pox” directory.

The command to run any program over a pox controller has three common attributes:

--port: Specifies the TCP port the controller will use to listen for connections on

--log.LEVEL : Specifies the level of logging the controller will operate according to.

Openflow.of_0X : Specifies the version of Openflow .

The rest of the command varies according to the attributes needed by the program running on the controller.

Communication between the two controllers will be accomplished using RabbitMQ which is a messaging broker. It gives applications a common platform to send and receive messages.

3.8.4. The Program “load balancing”:

The basic concept of load balancing is that when a flow arrives at the switch with no rules to match, the switch will notify its controller which in-turn decides where to assign the flow (i.e. which server should handle it).

The decision is based on the controllers’ network view of the least-loaded server. Each controller will negotiate the number of flows assigned to its domain server with the other controller and they both agree on the least loaded server; the negotiation process is done over a logical "RabbitMQ" channel connecting the controllers.

The programs implemented to achieve load balancing receives two attributes:

--ip: which is the IP of the load balancing service that will be requested by the clients.

10.1.1.3 for domain 1
10.1.1.4 for domain 2

--servers: which is a list of two IPs the first is the domains' server IP and the other is the other domains' service IP.

10.0.0.32 for server 1
10.0.0.64 for server 2

--operation mode: 1 for static load balancing and 2 for dynamic load balancing

3.8.5. Static load balancing:

When applying static consistency, the controllers will negotiate every 4 seconds to agree on the least loaded server

The commands used for running the two controllers are:

```
sudo ./pox.py log.level --ERROR  openflow.of_01 --port=6633 misc.newiploadbalancer --
ip=10.1.1.3 --servers=10.0.0.32,10.1.1.4 --operation_mode=1
sudo ./pox.py log.level --ERROR  openflow.of_01 --port=6634 misc.newiploadbalancer --
ip=10.1.1.4 --servers=10.0.0.64,10.1.1.3 --operation_mode=1
```

The following flow chart describes the operation of the static load balancing program:

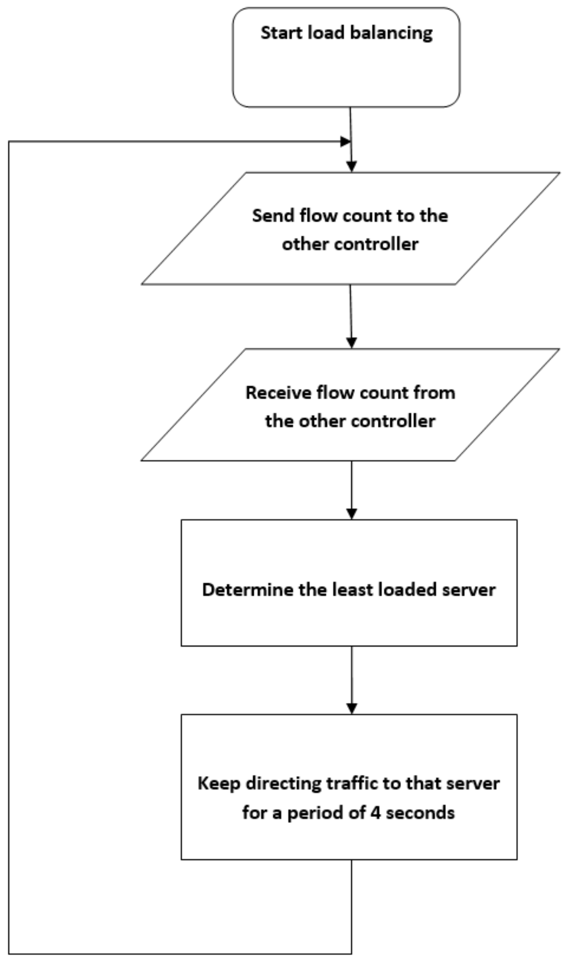

Start load balancing
Send flow count to the other controller
Receive flow count from the other controller
Determine the least loaded server
Keep directing traffic to that server for a period of 4 seconds

Figure 3-12 Static load balancing Flow chart

The controllers exchange the flow count assigned to their local domain server using the RabbitMQ logical channel connecting them, then they decide the least loaded server accordingly.

This process is repeated every synchronization period of four seconds.

3.8.6. Dynamic load balancing:

When applying dynamic consistency, the controllers will negotiate every X seconds to agree on the least loaded server

The synchronization period X is determined by the adaption module every two seconds.

The commands used for running the two controllers are:

```
 sudo ./pox.py log.level --ERROR  openflow.of_01 --port=6633 misc.newiploadbalancer --
ip=10.1.1.3 --servers=10.0.0.32,10.1.1.4 --operation_mode=2
```

```
sudo ./pox.py log.level --ERROR  openflow.of_01 --port=6634 misc.newiploadbalancer --
ip=10.1.1.4 --servers=10.0.0.64,10.1.1.3 –operation_mode=2
```

The following flow charts describe the operation of the dynamic load balancing program and the adaptation module:

[The adaptation module (figure 3-13) is responsible for determining the synchronization period used by the controllers, every two seconds the module calculates the relative difference in flows between the two servers and then decides whether do keep, increase or decrease the synchronization period, according to a pre-defined acceptable relative difference region]

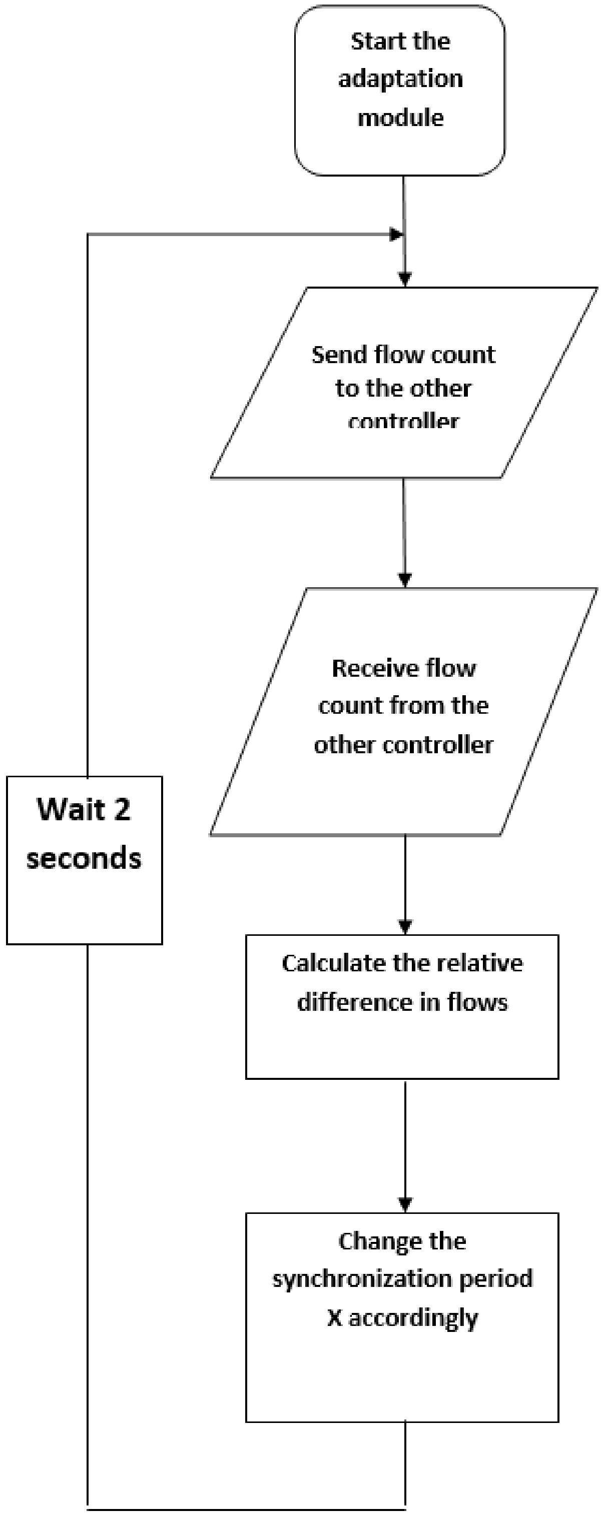

Start the adaptation module
Send flow count to the other controller
Receive flow count from the other controller
Calculate the relative difference in flows
Change the synchronization period X accordingly
Wait 2 seconds

Figure 3-13 Adaptation module flow chart

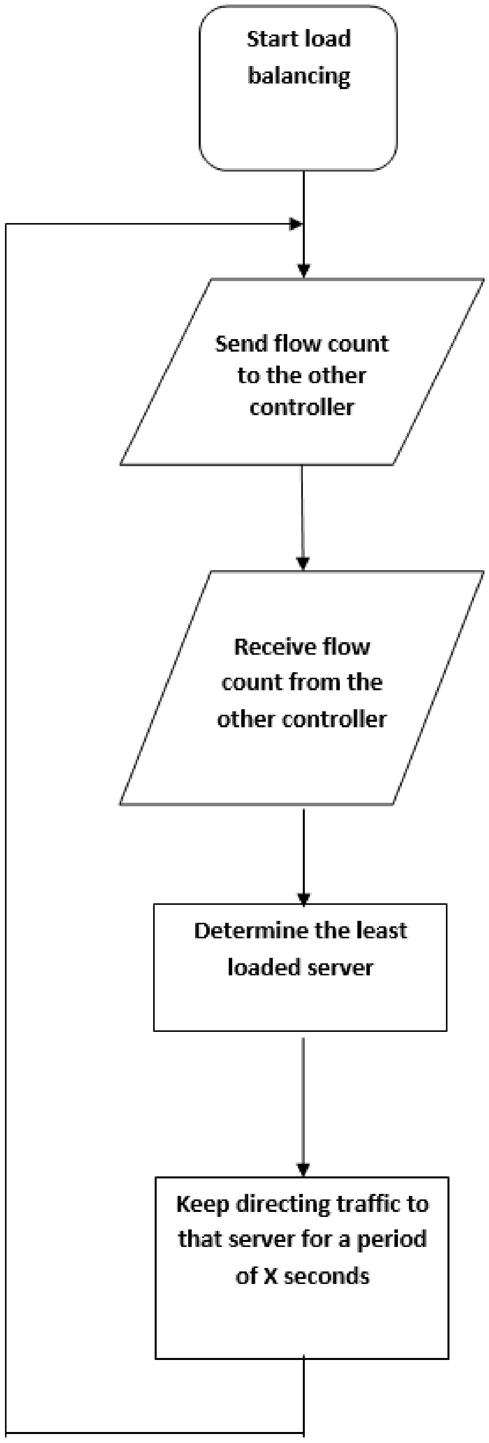

Start load balancing
Send flow count to the other controller
Receive flow count from the other controller
Determine the least loaded server
Keep directing traffic to that server for a period of X seconds

Figure 3-14 Dynamic load balancing flow chart

[The controllers exchange the flow count assigned to their local domain server using the RabbitMQ logical channel connecting them, then they decide the least loaded server accordingly.

This process is repeated every synchronization period which is determined by the adaptation module.]

Chapter 4 Results and Discussion

Chapter 4

Results and Discussion

4.1. Introduction

This chapter verifies the previous setup and check the efficiency of the proposed technology. The testing process will start with generating low traffic from the clients to their domain service IP for a period of time, this test will be performed twice, once when the controllers are behaving in a static manner and second when the controllers are running our proposed dynamic consistency module then the results will be logged and compared. After that the same process will be repeated but with high traffic generating from the clients and the results will be captured and compared. Finally, a fail over test will be done by disconnecting a server from the topology when high traffic is applied and see how will the load balancing service perform in both conditions.

The relative difference is used as a control and a performance parameter, it is calculated using the following equation:

$$\text{Relative difference} = \frac{|number\ of\ flows\ on\ server\ 1 - number\ of\ flows\ on\ server\ 2\ |}{\text{number of flows on server 1+ number of flows on server 2}}$$

The result value is between 0 and 1, 0 means no difference and 1 means complete difference

To control the dynamic consistency module a relative difference threshold region of 0.07 to 0.15 is set to define the normal region, this region is tunable to achieve different levels of performance, the more the readings of the relative difference stay inside this region the more efficient the load balancing application is.

4.2. The performance when applying low traffic:

Requests arrival rates at the two switches are 2 AND 1 respectively the relative difference between the two servers was logged every 2 secs, the following plots show the results of the two technologies:

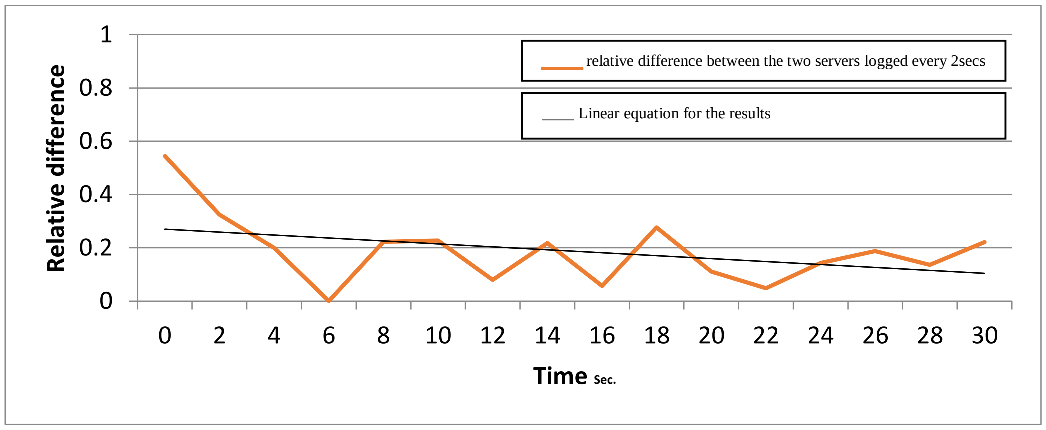


Figure 4-1 Static at low traffic

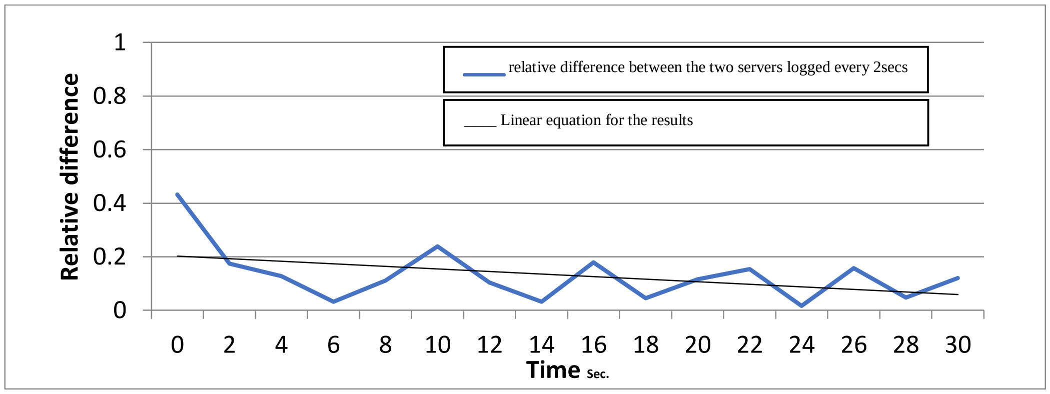


Figure 4-2 Dynamic at low traffic

The results indicate clearly that both techniques are performing excellently but there is one down side to applying dynamic consistency when the network is facing low traffic time, the relative difference gets lower than the normal region more frequently which means that the frequency of changing between the servers is more than needed. This is caused by the tries of the adaptation module to stay inside the threshold region after facing a rise in the relative difference.

4.3. The performance when applying high traffic:

Requests arrival rates at the two switches are 10 flows/sec from each domain, the relative difference between the two servers was logged every 2 secs, the following plots show the results of the two technologies:

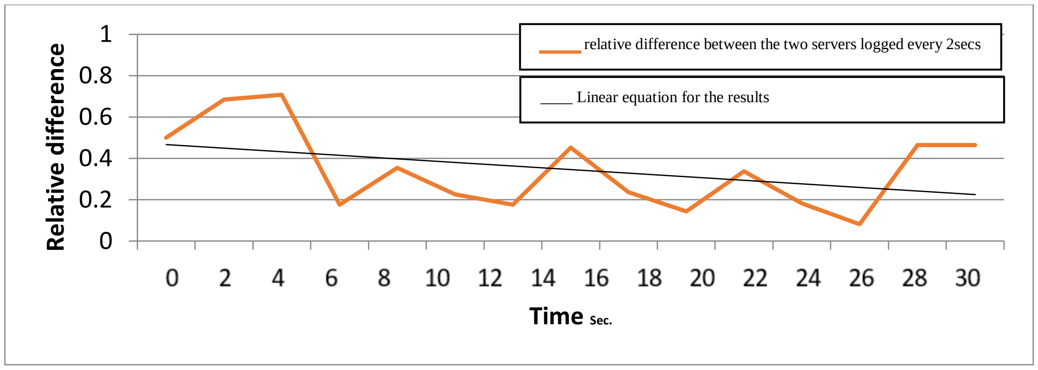


Figure 4-3 Static at high traffic

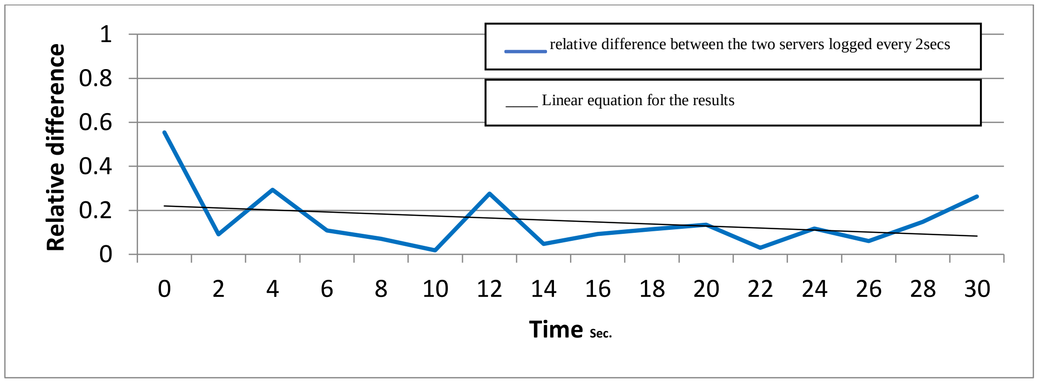
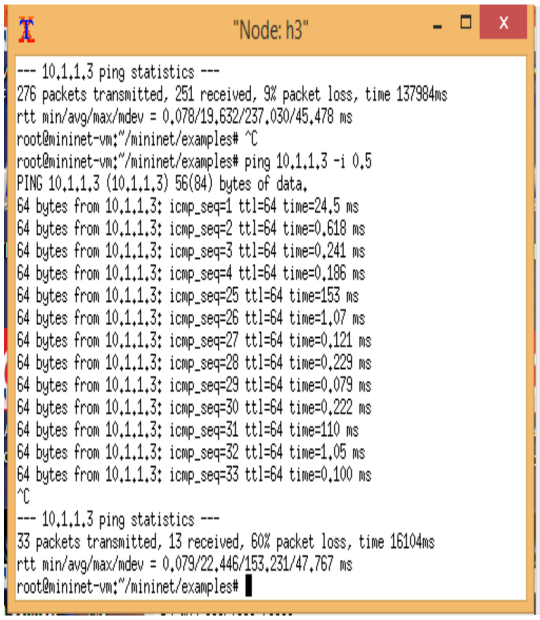


Figure 4-4 Dynamic at high traffic

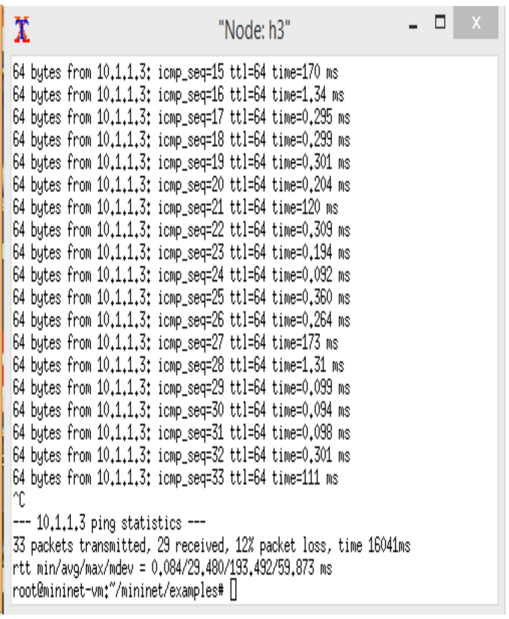
The results prove that the proposed dynamic consistency technique is more efficient and more flexible to network state changes than using the static consistency technique, the static 4 seconds synchronization period became too much when the network was flooded with high traffic.

4.4. Fail over test:

The fail over test was done by inspecting the ping from one client to the load balancing service after flooding the network with high traffic then dropping the link to the server that's serving the clients at the moment, the following screen captures shows the ping statistics from the client terminal:



```
--- 10.1.1.3 ping statistics ---
276 packets transmitted, 251 received, 9% packet loss, time 137984ms
rtt min/avg/max/mdev = 0.078/19.632/237.030/45.478 ms
root@mininet-vm:~/mininet/examples# ^C
root@mininet-vm:~/mininet/examples# ping 10.1.1.3 -i 0.5
PING 10.1.1.3 (10.1.1.3) 56(84) bytes of data.
64 bytes from 10.1.1.3: icmp_seq=1 ttl=64 time=24.5 ms
64 bytes from 10.1.1.3: icmp_seq=2 ttl=64 time=0.618 ms
64 bytes from 10.1.1.3: icmp_seq=3 ttl=64 time=0.241 ms
64 bytes from 10.1.1.3: icmp_seq=4 ttl=64 time=0.186 ms
64 bytes from 10.1.1.3: icmp_seq=25 ttl=64 time=153 ms
64 bytes from 10.1.1.3: icmp_seq=26 ttl=64 time=1.07 ms
64 bytes from 10.1.1.3: icmp_seq=27 ttl=64 time=0.121 ms
64 bytes from 10.1.1.3: icmp_seq=28 ttl=64 time=0.229 ms
64 bytes from 10.1.1.3: icmp_seq=29 ttl=64 time=0.079 ms
64 bytes from 10.1.1.3: icmp_seq=30 ttl=64 time=0.222 ms
64 bytes from 10.1.1.3: icmp_seq=31 ttl=64 time=110 ms
64 bytes from 10.1.1.3: icmp_seq=32 ttl=64 time=1.05 ms
64 bytes from 10.1.1.3: icmp_seq=33 ttl=64 time=0.100 ms
^C
--- 10.1.1.3 ping statistics ---
33 packets transmitted, 13 received, 60% packet loss, time 16104ms
rtt min/avg/max/mdev = 0.079/22.446/153.231/47.767 ms
root@mininet-vm:~/mininet/examples# 
```



```
64 bytes from 10.1.1.3: icmp_seq=15 ttl=64 time=170 ms
64 bytes from 10.1.1.3: icmp_seq=16 ttl=64 time=1.34 ms
64 bytes from 10.1.1.3: icmp_seq=17 ttl=64 time=0.295 ms
64 bytes from 10.1.1.3: icmp_seq=18 ttl=64 time=0.299 ms
64 bytes from 10.1.1.3: icmp_seq=19 ttl=64 time=0.301 ms
64 bytes from 10.1.1.3: icmp_seq=20 ttl=64 time=0.204 ms
64 bytes from 10.1.1.3: icmp_seq=21 ttl=64 time=120 ms
64 bytes from 10.1.1.3: icmp_seq=22 ttl=64 time=0.309 ms
64 bytes from 10.1.1.3: icmp_seq=23 ttl=64 time=0.194 ms
64 bytes from 10.1.1.3: icmp_seq=24 ttl=64 time=0.092 ms
64 bytes from 10.1.1.3: icmp_seq=25 ttl=64 time=0.360 ms
64 bytes from 10.1.1.3: icmp_seq=26 ttl=64 time=0.264 ms
64 bytes from 10.1.1.3: icmp_seq=27 ttl=64 time=173 ms
64 bytes from 10.1.1.3: icmp_seq=28 ttl=64 time=1.31 ms
64 bytes from 10.1.1.3: icmp_seq=29 ttl=64 time=0.099 ms
64 bytes from 10.1.1.3: icmp_seq=30 ttl=64 time=0.094 ms
64 bytes from 10.1.1.3: icmp_seq=31 ttl=64 time=0.098 ms
64 bytes from 10.1.1.3: icmp_seq=32 ttl=64 time=0.301 ms
64 bytes from 10.1.1.3: icmp_seq=33 ttl=64 time=111 ms
^C
--- 10.1.1.3 ping statistics ---
33 packets transmitted, 29 received, 12% packet loss, time 16041ms
rtt min/avg/max/mdev = 0.084/29.480/193.492/59.873 ms
root@mininet-vm:~/mininet/examples# 
```

Figure 4-5 Fail over test

the statistics showed a packet loss of 60% until the requests where redirected to the other server when using static consistency against 12% packet loss when applying dynamic consistency, because the service checks for server aliveness every synchronization period, that's why the dynamic consistency module behaved faster than the static consistency module.

Chapter 5 Conclusion and Recommendations

Chapter 5

Conclusion and Recommendations

5.1. Conclusion

Results are captured live and plotted for further analysis, the plots are compared to insure that using dynamic consistency over the traditional static consistency is more beneficial in terms in network efficiency.

Our results showed that dynamic controllers were more resilient to sudden changes in the network conditions than the static ones.

This thesis proposed a new way of achieving consistency in multi controller SDN topologies and proved that the proposed technique is better than the traditional technique being used,

5.2. Recommendations for Future work

We believe that future work should include implementing the distributed load balancing application over different types of controllers which has advantages over the POX to compare which type is better.

Another important addition is distributing the topology on multiple devices to notice the effect of network delays on the results.

Also a great addition would be running other services on the servers (HTTP, FTP).

Implementation on a real operational network

References:

A. DIXIT, F. H., MUKHERJEE, S., LAKSHMAN, T. & KOMPELLA, R. 2013. Towards an elastic distributed SDN controller.

ASLAN, M. & MATRAWY, A. 2015. On the impact of network state collection on the performance of SDN applications.

BEACON. 3/9/2017. *Beacon home website* [Online]. Available: https://openflow.stanford.edu/display/Beacon/Home

BLIAL, O., MAMOUN, M. B. & BENAINI, R. April 2016. An Overview on SDN Architectures with multiple controllers.

BREWER, E. 2000. Towards robust distributed systems.

ESTINET. 28/5/2017. *EstiNet* [Online]. Available: http://www.estinet.com/ns/?page_id=21140

FLOODLIGHT. 15/6/2015. *Project Floodlight.* [Online]. Available: http://www.projectfloodlight.org/.

FOUNDATION, O. N. 28/5/2017. *ONF* [Online]. Available: https://www.opennetworking.org/sdn-resources/sdn-definition

FOUNDATION, O. O. N. April 2012. *Software-Defined Networking: The New Norm for Networks*.

GNS-3. 16/7/2017. *Graphical Network Simulator-3* [Online]. Available: https://www.gns3.com/software.

GRAY, K. & NADEAU, T. D. August 2013. *SDN: Software Defined Networks* O'Reilly Media.

HALEPLIDIS, E., PENTIKOUSIS, K., DENAZIS, S. & SALIM, J. H. January 2015. *Software-Defined Networking (SDN): Layers and Architecture Terminology*.

HOANG, S. 28/5/2017. *Open Networking Foundation (ONF)* [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/SDN-architecture-overview-1.0.pdf.

HUMAYUNKABIR, M. Augest 2013. Software Defined Networking (SDN): A Revolution in Computer Network.

HUMAYUNKABIR, M. December , 2014. *The Evolution of SDN and OpenFlow: A Standards Perspective*.

JAIN, S. 2013. Experience with a Globally-Deployed Software Defined WAN.

JAMMAL, M., SINGH, T. & SHAMI, A. October 2014. Software defined networking: State of the art and research challenges.

KHURSHID, A., ZHOU, W., CAESAR, M. & GODFREY, P. 2012. Veriflow: verifying network-wide invariants in real time.

KOPONEN, T. 2010. *A Distributed Control Platform for Large scale Production Networks*.

LEVIN, D., WUNDSAM, A., HELLER, B., HANDIGOL, N. & FELDMANN, A. 2012. Logically centralized: state distribution trade-offs in software defined networks.

MININET. 15/3/2017. *Mininet* [Online]. Available: http://mininet.org/download. .

NS-3. 28/5/2017. *NS-3* [Online]. Available: https://www.nsnam.org/.

ODL. 3/9/2017. *OpenDayLight website :* [Online]. Available: https://www.opendaylight.org/.

OVERVIEW, M. 28/5/2017. *Mininet* [Online]. Available: http://mininet.org/overview/.

PHEMIUS, K. E., BOUET, M. & LEGUAY, J. E. E. 2014. Distributed Multi-domain SDN Controllers.

RYU. 28/5/2017. *Ryu SDN Framework* [Online]. Available: https://osrg.github.io/ryu/.

S.H. YEGANEH & GANJALI, Y. 2012. *Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications*.

SEZER, S., SCOTT-HAYWARD, S. & CHOUHAN, P. K. July 2013. *Are We Ready for SDN*.

TANENBAUM, A. & STEEN, M. V. 2007. *Distributed systems*.

Appendix

Appendix A

Topology generation code

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call
import time
from time import sleep

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8',
                          link=TCLink )
```

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

con1=net.addController(name='con1',

                controller=RemoteController,

                ip='127.0.0.1',

                protocol='tcp',

                port=6633)


con2=net.addController(name='con2',

                controller=RemoteController,

                ip='127.0.0.2',

                protocol='tcp',

                port=6634)


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

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

s0 = net.addSwitch('s0', cls=OVSKernelSwitch)

h=[]

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

for x in range(1,65):

 h.append(net.addHost('h'+str(x), cls=Host, ip='10.0.0.'+str(x), defaultRoute=None))

#h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)

#h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)

#h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)

#h5 = net.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)

#h6 = net.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)


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

for x in range(0,32):

 net.addLink(s0, h[x], bw=100 )

for x in range(32,64):
 net.addLink(s1, h[x], bw=100 )


net.addLink(s0, s1, bw=1000 )


info( '*** Starting network\n')
net.build()
info( '*** Starting controllers\n')
for controller in net.controllers:
    controller.start()


info( '*** Starting switches\n')
net.get('s1').start([con2])
net.get('s0').start([con1])


info( '*** Post configure switches and hosts\n')
s1.cmd('ifconfig s1 10.0.0.201')
s0.cmd('ifconfig s0 10.0.0.200')
#time.sleep(5) # delays for 5 seconds


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


while True:


 testtype=raw_input("Press Enter To Start The Test")
 if testtype == "1":
  time100s=time.time() + 30
  x=0
  y=0
  while time.time() <= time100s :
   if time.time() > time100s:
```

```
   break
  info(time100s-time.time())
  info('\n')
  h[x].cmd('ping 10.1.1.3 -c 1 -s 1 &')
  time.sleep(0.4)
  h[(x+1)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')

  x=(x+2)%32
  h[y+32].cmd('ping 10.1.1.4 -c 1 -s 1 &')
  y=(y+1)%32
  time.sleep(0.4)

elif testtype == "2":
 time100s=time.time() + 30
 x=0
 y=0
 while time.time() <= time100s :
  if time.time() > time100s:
   break
  ts=time.time()
  h[x%32].cmd('ping 10.1.1.3 -c 1 -s 1 &')
  h[y+32].cmd('ping 10.1.1.4 -c 1 -s 1 &')

  sleep(0.08)
  h[(x+1)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
  h[(y+1+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

  sleep(0.08)
  h[(x+2)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
  h[(y+2+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')
```

```
sleep(0.08)
h[(x+3)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+3+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
h[(x+4)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+4+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
h[(x+5)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+5+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
h[(x+6)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+6+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
h[(x+7)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+7+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
h[(x+8)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+8+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
h[(x+9)%31].cmd('ping 10.1.1.3 -c 1 -s 1 &')
h[(y+9+32)%63].cmd('ping 10.1.1.4 -c 1 -s 1 &')

sleep(0.08)
```

```
        x=(x+10)%32

        y=(y+10)%32

        info(time.time()-ts)

        info('\n')


      else:

        break

    #h[0].cmd('iperf -c 10.1.1.3 -k 2')

    #time.sleep(6) # delays for 5 seconds


    #h[40].cmd('iperf -c 10.1.1.4 -l 20B &')

    CLI(net)

    net.stop()


if __name__ == '__main__':

    setLogLevel( 'info' )

    myNetwork()
```

Appendix B

The load balancing application running on the controllers

```
#-*- coding:utf-8 -*-

from pox.core import core    # the POX core object
import pox.openflow.libopenflow_01 as of
from pox.lib.revent import *    # event system
from pox.lib.util import dpidToStr
from pox.lib.packet.ethernet import ethernet, ETHER_BROADCAST    # handle ethernet
from pox.lib.packet.arp import arp    # handle arp
from pox.lib.packet.ipv4 import ipv4
from pox.lib.addresses import IPAddr    # ip address
from pox.lib.addresses import EthAddr    # ethernet address
import time
import pika
import math
import csv
ThisController=''
OtherController=''

log = core.getLogger()

IDLE_TIMEOUT = 0    # in seconds
HARD_TIMEOUT = 2  # infinity

class LoadBalancer (EventMixin):

  class Server:
    def __init__ (self, ip, mac, port):
      self.ip = IPAddr(ip)    # set the ip address
      self.mac = EthAddr(mac)    # set the mac address
      self.port = port

    def __str__(self):
      return','.join([str(self.ip), str(self.mac), str(self.port)])

  def __init__ (self, connection, service_ip,Sender,Reciver, operation_mode,servers = [] ):
    self.connection = connection
    self.listenTo(connection)
    self.mac = self.connection.eth_addr
    self.service_ip = IPAddr(service_ip)
    self.serversip = [IPAddr(a) for a in servers]
    self.sender=Sender
    self.reciver=Reciver
    self.sync=4
    self.opmode=operation_mode
    w=self.sender+"finaldhigh"
#    print(w)
    fo = open(w,"w")
    fo.close()
    self.lasts=0
    self.lasto=0
    try:
      self.log = log.getChild(dpid_to_str(self.connection.dpid))
    except:
      # Be nice to Python 2.6 (ugh)
      self.log = log
    self.log.warning("serevers %s ",[str(a) for a in servers])

    self.c=0

    # Initialize the server list

    self.live_servers = {}

    if self.sender=="TX3":

     self.last_server = 0

    else:

     self.last_server = 1

    self.counter=0

    self.StaticCounter=0

    self.OtherCounter=0


    self.adaptation_module()

    self.outstanding_probes = {} # IP -> expire_time

    self.SendArps()

    self.change=1


    core.callDelayed(self.sync,self.set_next_server)


    self.reset=0


  def _do_expire (self):

    t = time.time()

    #print(".")

    # Expire probes

    for ip,expire_at in self.outstanding_probes.items():

      #print(ip)

      if t > expire_at:

        #print("Server %s down", ip)

        self.outstanding_probes.pop(ip, None)
        if ip in self.live_servers:
         if ip==self.serversip[0]:
            self.counter=1000000000
            del self.live_servers[ip]
  """
  def _do_probe (self):
    self._do_expire()

    server = self.serversip.pop(0)
    self.serversip.append(server)

    r = arp()
    r.hwtype = r.HW_TYPE_ETHERNET
    r.prototype = r.PROTO_TYPE_IP
    r.opcode = r.REQUEST
    r.hwdst = ETHER_BROADCAST
    r.protodst = server
    r.hwsrc = self.mac
    r.protosrc = self.service_ip
    e = ethernet(type=ethernet.ARP_TYPE, src=self.mac,
                 dst=ETHER_BROADCAST)
    e.set_payload(r)
    #self.log.debug("ARPing for %s", server)
    msg = of.ofp_packet_out()
    msg.data = e.pack()
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    msg.in_port = of.OFPP_NONE
    self.connection.send(msg)
    core.callDelayed(self.sync, self._do_probe)
  """


  def SendArps (self):
   self._do_expire()
   for a in self.serversip:
    server = a


    r = arp()
    r.hwtype = r.HW_TYPE_ETHERNET
    r.prototype = r.PROTO_TYPE_IP
    r.opcode = r.REQUEST
    r.hwdst = ETHER_BROADCAST
    r.protodst = server
    r.hwsrc = self.mac
    r.protosrc = self.service_ip
    e = ethernet(type=ethernet.ARP_TYPE, src=self.mac,
                     dst=ETHER_BROADCAST)
    e.set_payload(r)
    #self.log,warning("ARPing for %s", server)
    msg = of.ofp_packet_out()
    msg.data = e.pack()
    msg.actions.append(of.ofp_action_output(port = of.OFPP_FLOOD))
    msg.in_port = of.OFPP_NONE
    self.connection.send(msg)
    self.outstanding_probes[server] = time.time() + 0.5
   core.callDelayed(self.sync,self.SendArps)

  def get_next_server (self):
    #print("we choose %s" % self.serversip[self.last_server])
    return self.serversip[self.last_server]

  def set_next_server (self):

    rconnection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))

    channel = rconnection.channel()

    channel.queue_declare(queue='TX3')
    channel.queue_declare(queue='TX4')
    #print("hi %s   %s" % (self.sender,self.reciver))
    def callback(ch, method, properties, body):
     #print(" [x] Received %s" % body)
     self.StaticCounter=self.counter
     self.OtherCounter=int(body)

     if int(body) > self.counter:
       if self.last_server != 0:
         self.resetcounter()
       self.last_server=0

       print(" [X] agreed on this domains' server %i" %(self.sync))
     elif int(body) < self.counter:
       self.last_server=1

       print(" [X] agreed on the other domains' server %i" %(self.sync))
     #print("we choose %s" % self.live_servers.keys()[self.last_server])
     rconnection.close()
```

```
   core.callDelayed(self.sync,self.set_next_server)


  channel.basic_consume(callback,
                  queue=self.reciver,
                   no_ack=True)


  #print(' [*] Waiting for messages. To exit press CTRL+C')
  channel.basic_publish(exchange='',
                   routing_key=self.sender,
                   body=str(self.counter))


  channel.start_consuming()


def resetcounter(self):
     self.counter=self.counter-self.reset
     self.reset=self.counter


def adaptation_module(self):
  rconnection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))


  channel = rconnection.channel()


  channel.queue_declare(queue='TX3')
  channel.queue_declare(queue='TX4')
  #print("hi %s   %s" % (self.sender,self.reciver))
  def callback(ch, method, properties, body):
    #print(" [x] Received %s" % body)
    self.StaticCounter=self.counter
    self.OtherCounter=int(body)
```

```
      rconnection.close()


    channel.basic_consume(callback,
                      queue=self.reciver,
                       no_ack=True)


    #print(' [*] Waiting for messages. To exit press CTRL+C')
    channel.basic_publish(exchange='',
                       routing_key=self.sender,
                       body=str(self.counter))


    channel.start_consuming()


    self.lasts=self.StaticCounter
    self.lasto=self.OtherCounter


    x=abs(self.lasts-self.lasto)
    y=self.lasts + self.lasto
    if y==0:
     RDiff=0.25
    else:
     RDiff= x/float(y)
    RDiffTh=0.1
    eplus=0.05
    eminus=0.03
    w=self.sender+"finaldhigh"
#    print(w)
    fo = open(w,"a")
    wr=csv.writer(fo, quoting=csv.QUOTE_ALL)
    wr.writerow([self.c,RDiff])
```

```
  fo.close()
  self.c=self.c+2
  if self.opmode==str(2):
   if 0 <= RDiff and RDiff < RDiffTh - eminus:
    self.sync=pow(2,min(math.log(self.sync,2)+1,2))
    print(1)
   elif RDiffTh - eminus <= RDiff and RDiff <= RDiffTh + eplus:
    self.sync=self.sync
    print(2)
   elif RDiffTh + eplus < RDiff and RDiff <= 1:
    self.sync=pow(2,max(math.log(self.sync,2)-1,0))
    print(3)
  print("\n %i %f %i %i %i" % (abs(x),RDiff,self.lasts,self.lasto,self.sync))
  core.callDelayed(2,self.adaptation_module)

def handle_arp (self, packet, in_port):    # function No.2

  # Get the ARP request from packet
  arp_req = packet.next
  #if arp_req.protosrc != self.serversip[1]:
    #print("req :  %s is asking whos is %s from port %s" %(arp_req.protosrc,self.service_ip,in_port))
  # Create ARP reply
  arp_rep = arp()
  arp_rep.opcode = arp.REPLY
  arp_rep.hwsrc = self.mac    # hardware source
  arp_rep.hwdst = arp_req.hwsrc    # hardware destination
  arp_rep.protosrc = self.service_ip    # software source
  arp_rep.protodst = arp_req.protosrc    # software destination

  # Create the Ethernet packet
  eth = ethernet()
  eth.type = ethernet.ARP_TYPE
  eth.dst = packet.src
  eth.src = self.mac
  eth.set_payload(arp_rep)

  # Send the ARP reply to client
  msg = of.ofp_packet_out()
  msg.data = eth.pack()
  msg.actions.append(of.ofp_action_output(port = of.OFPP_IN_PORT))
  msg.in_port = in_port
  self.connection.send(msg)

def handle_request (self, packet, event):    # function No.3

  # Get the next server to handle the request
  serverip = self.get_next_server()
  if serverip == self.serversip[0]:
     self.counter=self.counter+1

  mac,sport=self.live_servers[serverip]
  "First install the reverse rule from server to client"
  #print("%s %s" %(serverip , sport))
  msg = of.ofp_flow_mod()
  msg.idle_timeout = IDLE_TIMEOUT
  msg.hard_timeout = HARD_TIMEOUT
  msg.buffer_id = None

  # Set packet matching
  # Match (in_port, src MAC, dst MAC, src IP, dst IP)
```

```
msg.match.in_port = sport

msg.match.dl_src = mac

msg.match.dl_dst = packet.src

msg.match.dl_type = ethernet.IP_TYPE

msg.match.nw_src = serverip

msg.match.nw_dst = packet.next.srcip


# Append actions

# Set the src IP and MAC to load balancer's

# Forward the packet to client's port

msg.actions.append(of.ofp_action_nw_addr.set_src(self.service_ip))

msg.actions.append(of.ofp_action_dl_addr.set_src(self.mac))

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


self.connection.send(msg)


"Second install the forward rule from client to server"

msg = of.ofp_flow_mod()

msg.idle_timeout = IDLE_TIMEOUT

msg.hard_timeout = HARD_TIMEOUT

msg.buffer_id = None

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


# Set packet matching

# Match (in_port, MAC src, MAC dst, IP src, IP dst)

msg.match.in_port = event.port

msg.match.dl_src = packet.src

msg.match.dl_dst = self.mac

msg.match.dl_type = ethernet.IP_TYPE

msg.match.nw_src = packet.next.srcip
```

```
  msg.match.nw_dst = self.service_ip

  # Append actions
  # Set the dst IP and MAC to load balancer's
  # Forward the packet to server's port
  msg.actions.append(of.ofp_action_nw_addr.set_dst(serverip))
  msg.actions.append(of.ofp_action_dl_addr.set_dst(mac))
  msg.actions.append(of.ofp_action_output(port = sport))

  self.connection.send(msg)
  #log.warning("Installing %s <-> %s : %i" % (packet.next.srcip, serverip,self.counter))

def _handle_PacketIn (self, event):    # function No.4 内部函数
  packet = event.parse()
  inport = event.port

  if packet.type == packet.LLDP_TYPE or packet.type == packet.IPV6_TYPE:
    # Drop LLDP packets
    # Drop IPv6 packets
    # send of command without actions

    msg = of.ofp_packet_out()
    msg.buffer_id = event.ofp.buffer_id
    msg.in_port = event.port
    self.connection.send(msg)

  elif packet.type == packet.ARP_TYPE:
    # Handle ARP request for load balancer
    arpp = packet.find('arp')
    #print("%s is asking whos is %s from port %s" %(arpp.protosrc,self.service_ip,inport))
  if packet.next.protodst == self.service_ip:


    if arpp.opcode != arpp.REPLY and self.counter < 1000000000:
     self.handle_arp(packet, event.port)


  if arpp.protosrc in self.serversip:
    # Handle replies to our server-liveness probes
    if arpp.opcode == arpp.REPLY:
        if arpp.protosrc in self.outstanding_probes :
         # A server is (still?) up; cool.
         #print("reply from " + str(arpp.protosrc))
          del self.outstanding_probes[arpp.protosrc]
        if arpp.protosrc in self.live_servers:
          # Ah, nothing new here.
          pass
        else:
          # Ooh, new server.
          if arpp.protosrc==self.serversip[0] :
           if self.counter>=1000000000:
            self.counter=0
            self.reset=0
          print("Server %s %s" % (arpp.protosrc,inport))


          self.live_servers[arpp.protosrc] = arpp.hwsrc,inport


        return


  # Only accept ARP request for load balancer


elif packet.type == packet.IP_TYPE:
    # Handle client's request

    # Only accept ARP request for load balancer
    if packet.next.dstip != self.service_ip:
      return

    #log.warning("flow %i" % (self.counter))

    #log.warning("Receive an IPv4 packet from %s" % packet.next.srcip)
    self.handle_request(packet, event)

class load_balancer (EventMixin):    # my component

  global ThisController
  global OtherController

  def __init__ (self, service_ip,operation_mode, servers = []):
    self.listenTo(core.openflow)
    self.service_ip = IPAddr(service_ip)
    self.servers = [IPAddr(a) for a in servers]
    self.opmode=operation_mode

  def _handle_ConnectionUp (self, event):

    rconnection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))

    channel = rconnection.channel()

    if self.service_ip==IPAddr('10.1.1.3'):
      OtherController='TX4'
    ThisController='TX3'
    channel.queue_delete(queue='TX3')

  else:
    OtherController='TX3'
    ThisController='TX4'
    channel.queue_delete(queue='TX4')
  log.warning("  %s " %ThisController)

  state='notready'

  channel.queue_declare(queue='TX30')
  channel.queue_declare(queue='TX40')
  print("Connection %s" % event.connection)
  def callback(ch, method, properties, body):
    print(" [x] Received %s" % body)
    if body=='Ready':
      state='ready'
      log.warning("%s is ready" % OtherController)
          rconnection.close()
      LoadBalancer(event.connection,
self.service_ip,OtherController,ThisController,self.opmode,self.servers)

  channel.basic_consume(callback,
                queue=OtherController+'0',
                 no_ack=True)

  print(' [*] Waiting for messages. To exit press CTRL+C')
  channel.basic_publish(exchange='',
                 routing_key=ThisController +'0',
                 body='Ready')
    channel.start_consuming()



def launch (ip,operation_mode, servers):    # registering component
  # Start load balancer
  servers = servers.replace(","," ").split()
  servers = [IPAddr(x) for x in servers]
  ip = IPAddr(ip)
  from proto.arp_responder import launch as arp_launch
  arp_launch(eat_packets=False,**{str(ip):True})
  import logging
  logging.getLogger("proto.arp_responder").setLevel(logging.WARN)

  core.registerNew(load_balancer,ip, operation_mode,servers)    #
```