修　士　論　文　の　和　文　要　旨

研究科・専攻	大学院　情報システム学研究科　情報ネットワークシステム学　専攻　博士前期課程		
氏　　　名	XU　YANSEN	学籍番号	1452011
論　文　題　目	A study of method for updating network configuration to avoid loops and packet loss in Software-Defined Networking SDN におけるパケット損失とループを防止するネットワーク設定更新方式の検討		

要　　旨

今のインターネットは世界を結ぶ情報社会の基盤となっている。様々なサービスやアプリケーションはインターネット上で増加しており、ネットワークはますます複雑になっている。このような状態を打破するため新たなネットワークアーキテクチャの設計が必要となる。ソフトウェア定義ネットワーク（SDN）は、コントロールプレーンとデータプレーンを分離するネットワークへの新しいアプローチである。SDN ネットワークにおいて、新しいトポロジーおよびサービスを適合させるため、ネットワークの設定を更新することは一般的である。この新しい技術を実現するため、OpenFlow という新しい技術を標準として導入される。OpenFlow プロトコルはコントロールプレーンとデータプレーンの間の通信が可能にする。しかし、全てのネットワーク機器において設定の更新が整合性を持たなければ、不一致性による問題が生じる。この問題により、ネットワークにおけるパケット損失やループなど不正確な動作が起こる可能性がある。

本稿では、SDN と OpenFlow に関する関連技術を紹介し、SDN における不一致性問題を定義し、先行研究を交えて議論する。そして、パケット損失とループを防止するネットワーク設定更新方式を提案する。具体的には SDN コントロールはネットワーク設定の更新前と更新後の転送経路を分析し、開放ループと閉合ループの有無により二つの経路間の関係を分類する。この関係を基づいて、スイッチ設定の更新の順番を計算し、コントロールはこの順番によってネットワーク設定を更新する。提案の正確性を検証するため、コントロール POX とネットワークエミュレータ Mininet 上で実装しシミュレーションを行い、TCP と UDP 二つのプロトコルでリンクのスループットやパケット損失を評価した。結果としては提案手法はリンクのスループットを保証し、パケット損失を抑制を実現した。また上記の提案手法を実装するためには、コントローラとスイッチ間の遅延が大きく作用するため、コントロールとスイッチの間に遅延の測定と設置の手法を提案し、評価を行った。最後に関連研究と比較して提案方式を議論した。

平成２７年度修士論文

A study of method for updating network configuration to avoid loops and packet loss in Software-Defined Networking

SDNにおけるパケット損失とループを防止するネットワーク設定更新方式の検討

大学院情報システム学研究科　情報ネットワークシステム学専攻

学 籍 番 号	: 1452011
氏　　　名	: Xu Yansen
主任指導教員	: Ved Kafle 客員准教授
指 導 教 員	: 吉永 努　教授
指 導 教 員	: 大坐畠 智　准教授
提 出 年 月 日	: 平成２８年１月２８日（木）

Abstract

The Internet has become the foundation of the modern information society. Kinds of services and applications are increasing on the Internet. The network has become more complex. The need of redesigning the network architecture has been felt. Software-Defined Networking (SDN) is a new network architecture, which separates the control plane from the data plane. OpenFlow is the standard protocol for communication interface between the control plane and data plane to realize the SDN architecture. Configuration changes are very common in the SDN networks to adapt a new topology and services. However, failing to perform update in the configurations consistently can introduce inconsistency problems. The inconsistency problems may cause the network to behave incorrectly such as creating of routing loops or misleading packets into a loop.

In this thesis, we introduced the concept of SDN and OpenFlow briefly. We defined and discussed the inconsistency problems and presented a scheme to avoid loops and packet loss during updating network configuration by analyzing and classifying relation between a new forwarding path and the old forwarding path. Based on this relation, the controller calculates the switches order and updates the configuration of switch based on the order. We conducted experiments to verify our scheme by using POX as controller and Mininet as network emulator. The experiments evaluated the throughput and packet loss using TCP and UDP based communications. The results show that our scheme can ensure the throughput and avoids packet loss during the network configuration update time. In order to realize the experiments, we also proposed an approach to measure and set the delay between controller and switches. Finally, we compared our scheme performance with related work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer network is one of the greatest innovations in the 20th century. Computer network has been developing for more than 40 years since Advanced Research Projects Agency Network (ARPANET) funded by the Defense Advanced Research Projects Agency (DARPA) of the United States Department of Defense.

After decades of development, the current Internet constituents many kinds of devices such as NICs, hubs, switches, routes, firewalls and other middle boxes which are the physical foundation of the networks. And at the same time, various kinds of contents, services including VoIP, IP-TV, online banking, sensor networking, content delivery developing and more protocols are added into the stack of network devices [1]. Through these devices, services, and protocols, billions of computers, mobile devices and sensors connect to the network and communicate with each other.

However, this increasing number of hardware and software makes networks more complicated and harder to control and manage. On the other side, current networks are also very difficult for researchers, network operators, and vendors to innovate, since it takes a long time to test and standardize new architectures, protocols and services [1].

To address these problems, Open Networking Foundation (ONF) has been formed and dedicated to the promotion and adoption of Software-Defined Networking (SDN) through open standards development [2].

SDN is an architecture based on decoupling of the control and data planes. This new architecture makes the network more intelligence and enables the logically centralized management. The underlying network infrastructure of SDN is abstracted from the applications. As a result, enterprises and carriers gain unprecedented programmability, automation, and network control, enabling them to build highly scalable, flexible networks that readily adapt to changing business needs [3]. And since the control planes and data planes in SDN are decoupled, these two planes are able to evolve separately, thus faster innovation of networking technologies.

OpenFlow is the first standard communications interface defined for exchanging information between the controller and forwarding layers of the SDN architecture [4]. It provides an open and standard way for a controller to communicate with the switches [5]. OpenFlow has caught attention of the researchers and the router vendors. It is developed under a clean-slate future Internet program by Stanford University.

Software Defined Network defines a centralized architecture, and this centralized approach introduces various challenges in terms of consistency.

The consistency update [6] from one configuration to another configuration in SDN guarantees that every packet flowing in the network is forwarded according to either the configuration prior to the update or the configuration after the update. If inconsistency update occurs during changes of network topology such as due to host migration, link state change, switch failure, or policy changing on load-balancing, resource sharing and security, the network may behave incorrectly and cause loops, packet loss and violations of policies. This kind of packet loss or loop cannot be tolerated. Packet loss may cause a session break or stopping of service such as interruption of HTTP applications and packet loop may cause broadcast storm and network congestion. Therefore, when the controller updates the data plane's forwarding rules, the controller must be capable to handle the inconsistency problem.

Contributions

In this thesis, we proposed a scheme to update network configuration to avoid packet loss and loop in SDN. We conducted experiments to verify the correctness and performance of our scheme. The results show that our scheme can ensure the throughput and avoid packet loss during the network configuration update time.

Thesis overview

The rest of this thesis is organized as follows:

In Chapter 2, we introduce the background of software-defined networking and OpenFlow protocol.

In Chapter 3, we introduce and define the inconsistency problem

Chapter 4 proposes the update scheme for network configuration without packet loop and loss.

In Chapter 5, we conduct experiments to verify the correctness and performance of our scheme.

In Chapter 6, we discuss our scheme and compare our scheme with related works.

In Chapter 7 presents the conclusion of this thesis.

Chapter 2

Background

Current networks have many limitations because of increasing of number of devices, services, contents and protocols. SDN and OpenFlow change the way to address the network problems. This chapter first introduces the current network limitations briefly and then provides the concept of SDN and OpenFlow.

2.1 Limitations of current networks

The current networks have been in use for many years and gained great success both in the academic and industrial field based on the hierarchical network architecture [15]. However, since the TCP/IP computer network was designed 40 years ago and with the development of technology and service, the complexity of network configuration and Internet traffic is increasing time by time, and the past design of network cannot meet the current and future requirements. Due to the integrated control plane and data plane, the innovation of network is slow and it makes the network equipment expensive. Many limitations of current networks hinder the development of network.

Distributed Control:

The networks are composed of many kinds of devices such as switches, routers and firewalls. The control software running in these devices are complex and distributed. Each devices is controlled by itself and cannot obtain a whole view of the network.

Hard to Management:

Due to distributed control, network management has become very complex. Network operators have to control and configure each device using different low-level configuration interfaces that vary across venders [9].

Difficulty to Extend:

Since the current network is a distributed control system and consists of numerous devices, any part of network change will lead to reconfiguration of all of these devices. It takes a long time to compute and install new configurations and to assure that the new network works well.

Expensive Equipment:

Current network devices are composed of both software and hardware. The most expensive part, control software contains millions of lines of source codes and the control software cannot be customized by customers even though there may be many functions that the customer may not need.

Slow Innovation [11]:

Current standardizing process is slow. It takes several years to improve the prototype perfectly and over 10 years to standardize. Even many good research ideas proposed by researchers cannot be tested at scale and on real networks with real user traffic.

Therefore, to manage networks easily, reduce network expenses, make network innovation fast and meet future requirements, we need a new technology with a clean-slate approach. Following subsections introduce the technology of software-defined networking and OpenFlow.

2.2 Software Defined Networking

Software Defined Networking (SDN) has originated from the Clean Slate project in Stanford University since 2006 [14] and first proposed by Professor Mckeown in 2009 [11]. SDN is a new approach to networking in which the network control is decoupled from the data forwarding function and is directly programmable [7]. It logically centralizes the network intelligence and state, and reduces network complexity through automation by writing program. And with the separation of the control and data planes, network switches become simple forwarding devices and the control logic is implemented in a logically centralized controller, simplifying policy enforcement and network configuration and evolution [12][13].

2.2.1 Overview of SDN Architecture

Figure 2- 1shows an overview of SDN architecture proposed by Open Network Foundation (ONF) [8]. The application layer, control layer and infrastructure layer are the main three parts in SDN architecture.

Application Layer (SDN Application) – The application layer defines all the features, services and policies in the network. This layer uses an abstract network view provided by controller to make decision and realize services and policies. Application layer consists of an SDN Application (APP) and a Northbound Interface (NBI) driver. The SDN APP is the main body of logic control to the network and the NBI driver is an interface to communicate with controller layer via NBI.

Control Layer (SDN Controller) – Located in the central position of SDN architecture. This layer communicates with network devices and obtains abstract view of network. It controls the network devices' behavior according to the application layer, and provides the abstract view of network to the application layer such as statistics, events and topology. This layer consists of an SDN control logic, an NBI agent via which control layer communicates with application layer, and an SDN Control Data Plane Interface (CDPI) via which the control layer communicates with the data plane.

Infrastructure Layer (Data Plane) – This layer is a collection of network devices. Devices in this layer are connected with each other physically. It forwards and processes packets as the controller commands and knows nothing about the information of network. This layer consists of a CDPI agent via which the data plane communicates with the control plane and a forwarding engine or processing function to forward and process packets.

SDN Northbound Interfaces (NBI) – NBI is an interface between the application layer and the control layer. The control layer provides application programming interfaces and abstract network views to the application layer.

SDN Control to Data Plane Interfaces (CDPI or Southbound Interfaces) – CDPI is an interface between the control layer and the infrastructure layer. This interface provides least programmatic control of all forwarding operations, capabilities advertisement, statistics reporting, and event notification.

‘

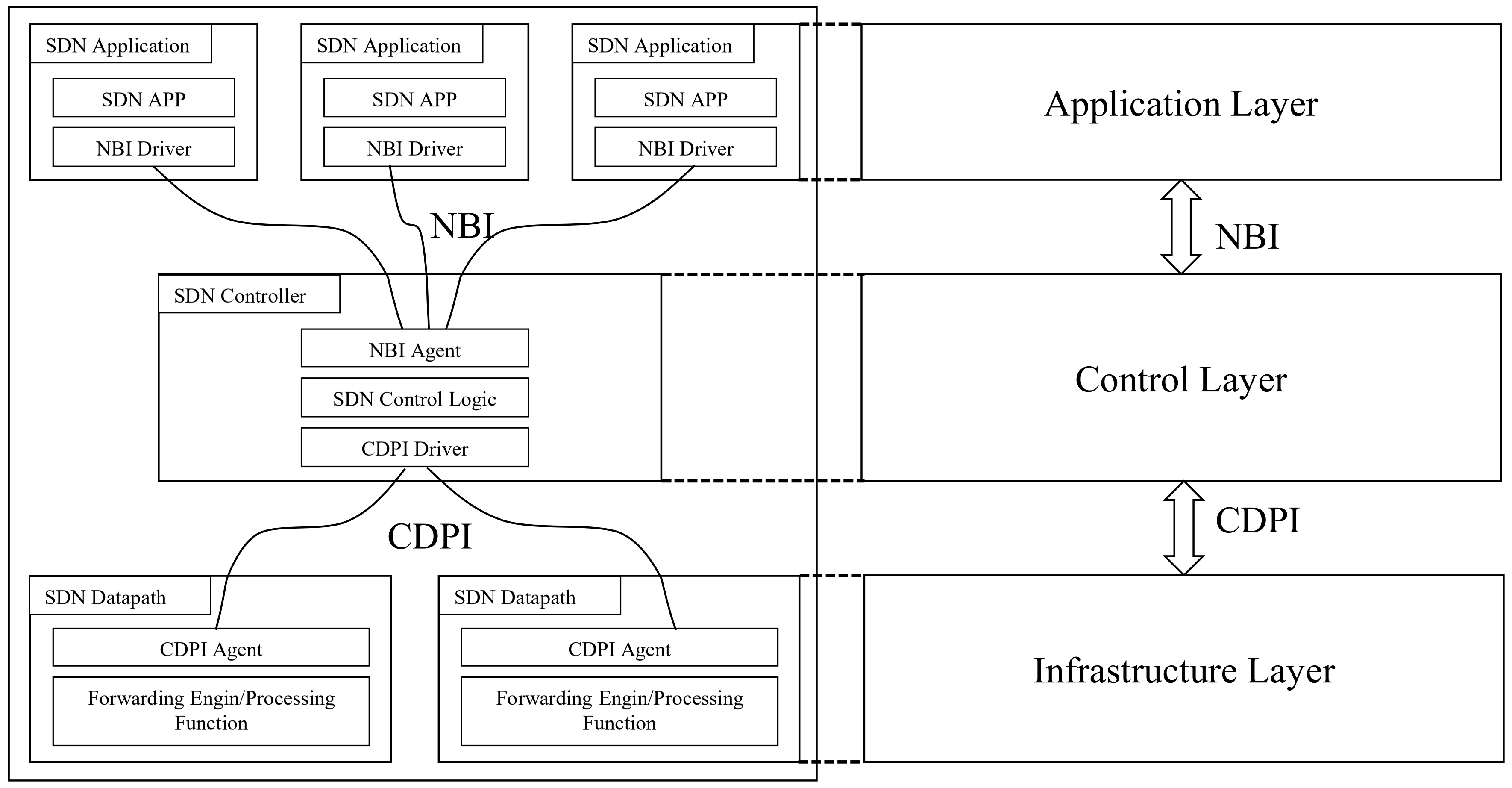

Figure 2- 1 Overview of SDN architecture

2.2.2 Benefits of SDN

The separation of control plane and data plane brings many benefits compared with the current network architecture. Some benefits are as following.

Programmable – Since the control logic is decoupled from the data plane and centralized to one location, and this control logic knows the abstract of network view, it is easy to use this feature to program the network directly.

Easy to Extend – Based on the programmable feature, the complexity to extend a network is reduced. Network administrators do not have to configure devices directly. The program on the controller reconfigures the network automatically.

Fast Innovation – Since the control plane and data plane are decoupled, these planes can evolve independently. Like the OSI reference model, each layer can develop its own new technology without affecting the other.

Cheap Equipment – Everyone could produce their own SDN equipment as long as their products follow the SDN standards.

2.2.3 SDN Controller

SDN controller is the most important part in the SDN architecture. It is the “brain” of the network. SDN controller abstracts the data plane and provides a whole view of network and installs applications’ deployment to the data plane. SDN controller can also be regarded as a network operating system (NOS).

The most well-known NOS in current networking is the Cisco IOS [21]. While in SDN, numbers of NOS have been created developed. Figure 2- 2 shows a SDN controller platform. This platform is based on the analysis of different SDN controllers and extracted common elements to provide a best attempt to use and research controller [13].

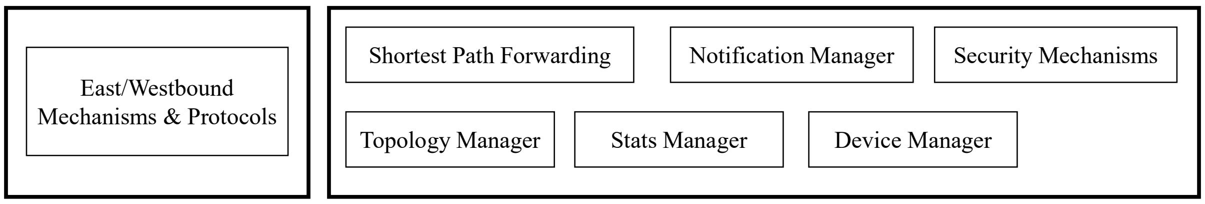

Figure 2- 2 SDN controller platform

Following are some well-known controllers.

Floodlight [22]

Floodlight Open SDN Controller is an enterprise-class controller licensed by Apache. It fully supports OpenFlow 1.0 and 1.3 and experimentally supports OpenFlow 1.1, 1.2 and 1.4 with easy-to-use, version-agnostic APIs. Floodlight uses Java programming language and has the latest version v1.1 released in April 2015.

OpenDaylight [23]

OpenDaylight is a highly available, modular, extensible, scalable and multi-protocol controller infrastructure built for SDN deployments on modern heterogeneous multi-vendor networks. OpenDaylight is licensed by Eclipse Public License (EPL-1.0) and uses Java as programming language. The latest release is third release, called LITHIUM, released in June 2015.

Trema [24]

Trema provides a high-level OpenFlow library and a network emulator that can create OpenFlow-based networks for testing on PC. It uses C and Ruby as programming language and is licensed by GNU General Public License version 2.0 (GPL-2.0). The latest version is v0.9.0 (unreleased).

Ryu [25]

Ryu is a component-based software defined networking framework. Ryu supports OpenFlow, Netconf, OF-config, etc. Ryu fully supports OpenFlow version 1.0, 1.2, 1.3, 1.4, 1.5 and Nicira Extensions. All the code is freely available under the Apache 2.0 license. Ryu uses Python as programming language. The latest release is v3.28 (January 2016).

Beacon [26]

Beacon is a Java-based open source OpenFlow controller. It supports to start and stop existing applications or new applications at runtime. Beacon is now licensed by BSD. The current v1.0.4 is released in September 2013.

POX [27]

POX is an open source development platform written in Python. It is a sibling of NOX [28]. It currently supports OpenFlow 1.0 and a number of the Nicira extensions. The current version of POX is 0.3.0 (dart).

The following table is a comparison between SDN controllers.

Table 2- 1 SDN controllers

Name	Northbound API	License	Programming Language	Version
Floodlight	RESTful API	Apache 2.0	Java	V1.1
OpenDaylight	REST,RESTCONF	EPL-1.0	Java	LITHIUM
Trema	Ad-hoc API	GPL-2.0	Ruby	V0.9.0
Ryu	Ad-hoc API	Apache 2.0	Python	V3.28
Beacon	Ad-hoc API	BSD	Java	V1.0.4
POX	Ad-hoc API	Apache 2.0	Python	V0.3.0

2.2.4 ONF Standardization Activities

Technical communities in the Open Networking Foundation are organized to promote the adoption of SDN. Only ONF member companies and their representatives can participate in a variety of ways to fulfill ONF's missions [16]. The technical communities in ONF handle specific issues related to SDN and collaborate with the world's leading experts regarding SDN concepts, frameworks, architecture, software, standards and certifications. Four main areas is focused: Operator, Services, Specification and Market.

Operator Area

The Operator Area works to gather and validate network operator requirements, priorities, tradeoffs, and vision [17].

Table 2- 2 ONF Operator Area activities

Fields	Focus
Carrier Grade SDN	Focus on the unique needs of carrier operators of SDN environments.
Data Center	Content to come
Enterprise	Content to come
Migration	Produce methods and recommendations for migrating network services from a traditional network to an OpenFlow-based software defined network

Services Area

The Services Area works on technical projects to enable applications and network operator services with SDN technologies [18].

Table 2- 3 ONF Services Area Activities

Fields	Focus
Architecture & Framework	Help standardize SDN by defining the broad set of problems that the SDN architecture needs to address
Information Modeling	Responsible for a Core Information Model and forwarding technology-specific information models
L4-7 Services	Focuses on end-to-end services that require various L4-L7 functions and chaining
Northbound Interfaces	Develops concrete requirements, architecture, and working code for northbound interfaces
Security	Carry out the analysis of security issues with SDN and promote discussion of security considerations and recommendations

Specification Area

The specifications area is responsible for publishing all ONF technical specifications [19].

Table 2- 4 ONF Specification Area activities

Fields	Focus
Open Datapath	Maintain and evolve the OpenFlow protocol and associated datapath modeling technologies
OF-Config	Address core Operations, Administration, and Management issues
Open Transport	Address SDN and OpenFlow Standard-based control capabilities for transport technologies of different types
Protocol Independent Forwarding	Identify and employ SDN and OpenFlow standard based technology in mobile networks
Mobile Networks	Develop an interpreter for an Intermediate Representation
Testing & Interoperability	Accelerate the development and adoption of the OpenFlow Standard

Market Area

The Market Area's primary goals are to educate the SDN community on the value proposition of software-defined networks based on ONF Standards and promoting adoption for open SDN [20].

Table 2- 5 ONF Market Area activities

Fields	Focus
Liaisons	Establish relationships with a variety of organizations to partner and address issues of common concerns and collaborate on shared interests
Proofs of Concept	
Publications	
SDN Solutions Showcase	Highlight the adoption of SDN technologies
Skills Certification	
Workshops	

2.2.5 Challenges of SDN

Since SDN is a very new architecture compared with other concepts, a lot of ongoing research on SDN has been recently pursued in the world. The topic of SDN is the hottest field in the most top conferences in recent years. There are some challenges in SDN as following:

The design of forwarding plane

The forwarding plane can be designed as hardware or software. The hardware forwarding plane has a high packet forwarding rate whereas the software forwarding plane can be reconfigured easily when needed. As the evolution of networking, it is important to increase packet forwarding rate and support the network programmability.

Scalability and distribution of control plane

In many medium-size networks, the latency from every node to a single controller can meet the response-time goals of existing technologies. However, as the size of networks gets larger and larger, we need to consider where and how many controllers to deploy in the networks and how these controllers can get a consistency view of the network and the control plane [29].

Speed and availability of controller processing

The controller has to obtain the abstraction of data plane and control the behavior of data plane. As the services and contents increasing, how to improve the speed and availability of controller processing is important [29].

Testing techniques

In large-scale networks, one challenge is to detect and resolve security policy violation and to test and verify forwarding tables to find routing errors due to the frequent changes in routing state [13]. How to design and develop testing tools is also a very important issue in SDN.

Language for controller API

Like computer programming languages switched from low-level languages (assemble languages) to high-level languages (C, Python) [13], the programming languages for networking is also needed to switch from a low-level languages to a high-level languages which can abstract the data plane and make programming tasks easier [30].

Besides the above points, there are many other issues that need to be solved such as security issues, design of north/south bound interfaces, and application, and virtualization.

2.3 OpenFlow

In order to realize SDN architecture, there needs some methods for the control plane to communicate with the data plane. The Open Networking Foundation (ONF) introduced OpenFlow which enables remote programming of the forwarding plane. OpenFlow is the first standard communications interface between the data plane and control plane and a vital element in SDN architecture [4]. Following is a brief introduction about OpenFlow based on OpenFlow Switch Specification Version 1.0.0 released in December 2009 [31].

2.3.1 Requirements in OpenFlow based switches

An OpenFlow based switch has two parts as shown in Figure 2- 3: secure channel and flow table. OpenFlow switch communicates with a controller over the secure channel by using OpenFlow protocol. The controller configures and manages devices and receives events form data plane through the secure channel interface. Transport Layer Security (TLS) is one choice to establish a secure connection over OpenFlow protocol.

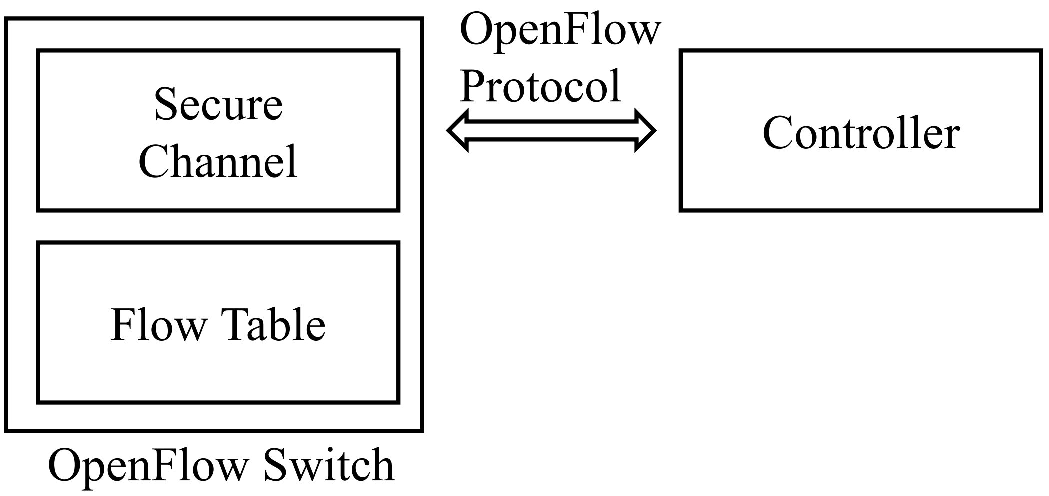

Figure 2- 3 An OpenFlow switch communicates with a controller

The flow table contains a set of flow entries, activity counters and actions. Packets entering into switches are processed according to the flow entries. If a matching entry is found, the action in this entry is applied to the packet and if no matching entry is found, the packet is forwarded to the controller over the secure channel.

2.3.2 Flow table

A flow table contains a set of flow entries, consisting of header fields, counters, and actions.

The header field is matched against packets and contains a specific value, or ANY. Table 2- 6 lists the layers and fields a header field can match. OpenFlow v1.0 supports 12-tuple match fields and if the switch supports subnet masks on IP fields, these can more precisely specify matches.

The counter field is to record the static network information. This field is updated as a flow entry is matched. Counters can obtain information of network on per-table, per-flow, per-port and per-queue. Table 2- 7, Table 2- 8, Table 2- 9, and Table 2- 10list the counters used in statistics messages.

Table 2- 6 Match fields and lengths

Layer	Field	Length (number of bits)
L1	Ingress port	Implementation dependent
L2	Ethernet source address	48
	Ethernet destination address	48
	Ethernet type	16
	VLAN id	12
	VLAN priority	3
L3	IP source address	32
	IP destination address	32
	IP protocol	8
	ToS	6
L4	Transport source port / ICMP type	16
	Transport destination port / ICMP code	16

Table 2- 7 Per Table Counter

Counter	Length (number of bits)
Active Entries	32
Packet Lookups	64
Packet Matches	64

Table 2- 8 Per Flow Counter

Counter	Length (number of bits)
Received Packets	64
Received Bytes	64
Duration (seconds)	32
Duration (nanoseconds)	32

Table 2- 9 Per Port Counter

Counter	Length (number of bits)
Received Packets	64
Transmitted Packets	64
Received Bytes	64
Transmitted Bytes	64
Receive Drops	64
Transmit Drops	64
Receive Errors	64
Transmit Errors	64
Receive Frame Alignment Errors	64
Receive Overrun Errors	64
Receive CRC Errors	64
Collisions	64

Table 2- 10 Per Queue Counter

Counter	Length (number of bits)
Transmit Packets	32
Transmit Bytes	64
Transmit Overrun Errors	64

The action field dictates how the switch processes packets matched with the flow entry. If no forward actions in the flow entry, the switch drops the packets. The action field supports four kinds of actions: Forward, Enqueue, Drop, and Modify-Field.

Forward action is either required action or optional action. Table 2- 11 lists the Forward action's classification. OpenFlow-compliant switches are of two types: OpenFlow-only and OpenFlow-enable. The required action is supported only by OpenFlow-only switches, whereas the optional action is also supported by the NORMAL action. Either type of switch supports FLOOD action.

Enqueue action is an optional action. It forwards a packet through a queue attached to a port.

Drop action is a required action. If there is no action specified in a flow entry, the packet matching this flow entry is dropped.

Modify-Field action is an optional action. This action can modify the contents in a packet. Generally, VLAN ID/priority can be set, VLAN header can be stripped, and Ethernet source/destination MAC address, IPv4 source/destination address, IPv4 ToS bits, and transport source/destination port can be modified. Switches do not necessarily support this action but the specification suggests that VLAN modification be supported.

Table 2- 11 Forward action's classification

Required/Optional Action	Action name	Description
Required Action	-	Forward out from assigned port
	ALL	Send the packet out all interfaces, not including the incoming interface
	CONTROLLER	Send the packet to controller
	LOCAL	Send the packet to the switches local networking stack
	TABLE	Perform actions in flow table
	IN_PORT	Send the packet out the input port
Optional Action	NORMAL	Performance as a traditional switch
	FLOOD	Flood the packet along the minimum spanning tree, not including the incoming interface

2.3.3 Matching

Matching occurs every time a packet enters a switch. The switch looks up the flow entry according to the header of the packet and if the flow entry is matched with the header, the packet is processed as the action field indicates.

Figure 2- 4 shows how a packet is processed inside the switches and Figure 2- 5 shows how header fields are parsed.

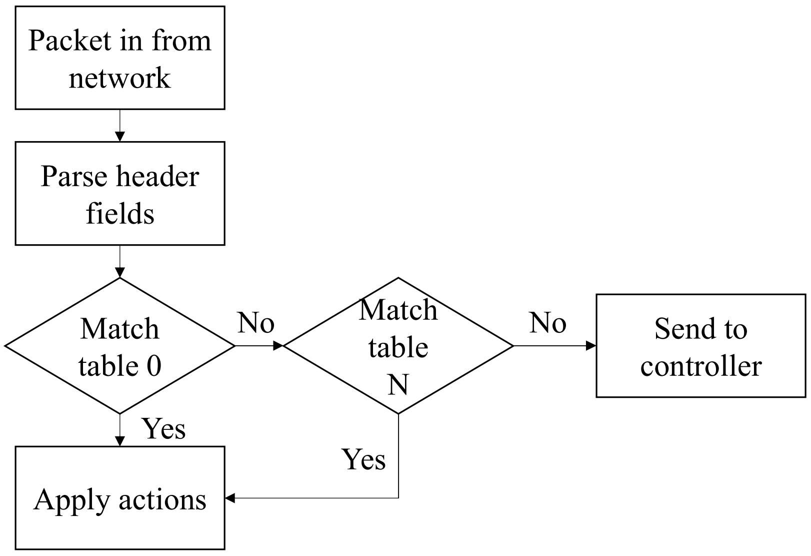

Figure 2- 4 Packet processing in switch

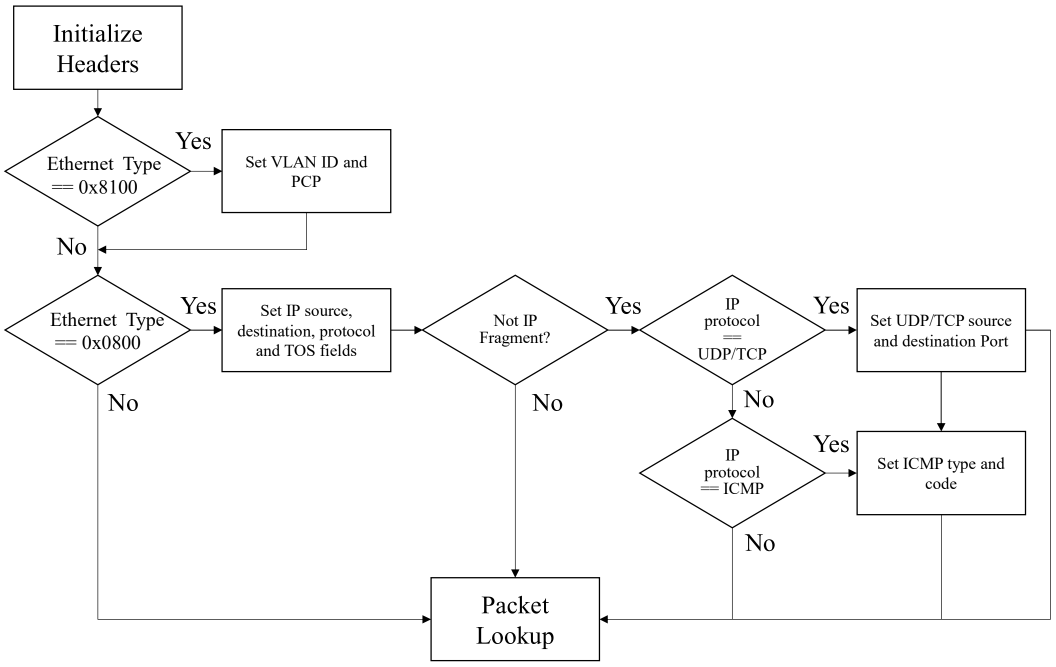

Figure 2- 5 Flowchart of header fields parsing process

2.3.4 OpenFlow Protocol

There are three types of messages in OpenFlow protocol: controller-to-switch message, asynchronous message, and symmetric message.

The controller-to-switch message is initialed by the controller and the switch may or may not respond the message according to the sub message type. Table 2- 12 lists the sub-type messages of controller-to-switch messages.

The asynchronous message is a message sent by switches to the controller. This message is used to send packets with no matched flow entry, notify switch's state change, or inform error in switch. Table 2- 13 lists he sub-type messages of asynchronous message.

The symmetric message is a message initialed either by the controller or by a switch. Table 2-14 lists the sub-type messages of symmetric message.

Table 2- 12 Sub-type Messages of Controller-to-switch

Message	Description
Features	Request the capabilities supported by switch.
Configuration	Set and query configuration in the switch.
Modify-State	Add, delete and modify flows in the flow table and set switch port properties.
Read-State	Collect information from switches
Packet-Out	Direct switch to send packets out of a specified port
Barrier	Ensure the message switch received have been completely implemented

Table 2- 13 Sub-type Messages of Asynchronous

Message	Description
Packet-In	Forward packets, which have no matching flow entry, to controller.
Flow-Removed	Notify controller that a flow entry is removed.
Port-Status	Notify controller about the status change of port
Error	Notify controller about problems or errors in switch.

Table 2- 14 Sub-type Messages of Symmetric

Message	Description
Hello	Exchanged between switch and controller upon connection startup
Echo	Echo request/reply messages sent by either controller or switch and must be replied
Vendor	For future vendor's additional functions

2.3.5 OpenFlow Message construction

This section introduces formats of some OpenFlow messages.

OpenFlow Header

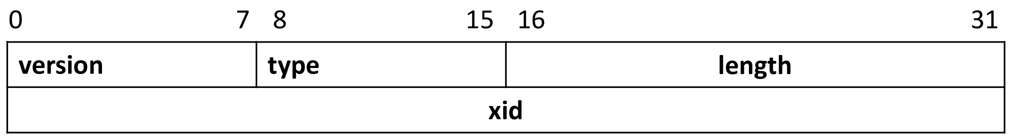

Figure 2- 6 OpenFlow Header

OpenFlow message is started from an OpenFlow header. Each header contains a version field specifying the OpenFlow protocol version, a type field indicating the message type, length field indicating the total length of message and *xid* field as identifier.

Packet-In Message

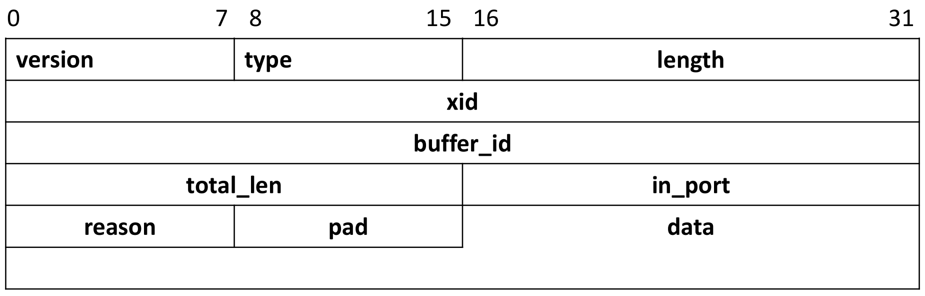

Figure 2- 7 Packet-In Message

The reason field in Packet-In message indicates the reason switch forwards the packet to controller. Two reasons are specified: no match flow and action explicitly output to controller.

Packet-Out message

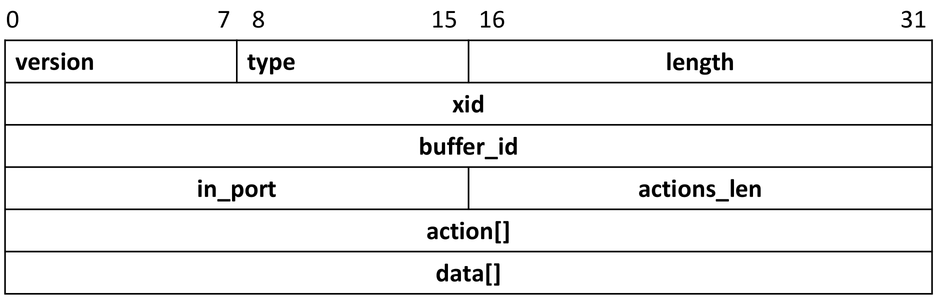

Figure 2- 8 Packet-Out Message

Flow-Modify Message

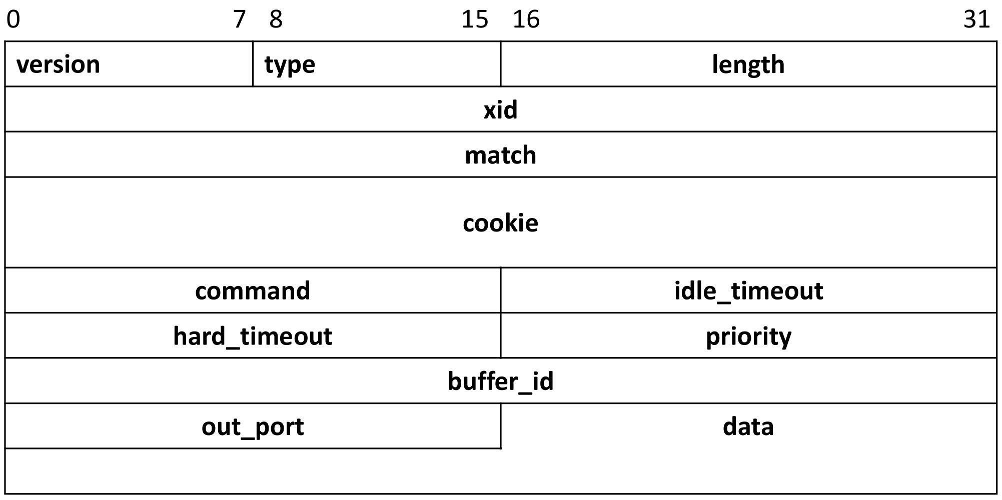

Figure 2- 9 Flow-Modify Message

The command field indicates that how controller directs the switch to process the flow entry. This field can contain one of the following commands:

OFPFC_ADD: Install a new flow entry.

OFPFC_MODIFY: Modify all matching flows entry.

OFPFC_MODIFY_STRICT: Modify flow entry strictly matching wildcards

OFPFC_DELETE: Delete all matching flows entry.

OFPFC_DELETE_STRICT: Strictly match wildcards and priority.

Chapter 3

Inconsistency Problem

SDN decouples the control plane and data plane and enables centralized management on control plane. But this method is also facing many problems in terms of consistency and performance.

Similar to traditional networks, the network topology and configuration in SDN change time to time for routing, load balancing, etc. The controller in SDN has to update the configurations of switches in the data plane according to the topology change. However, the data plane is actually a distributed system and, as most of other distributed systems, it encounters CAP theorem (also known as Brewer's theorem) [32].

The CAP theorem shows that it is impossible for a distributed computer system to provide the following three guarantees: consistency, availability, and partition tolerance. The systems have to be trade-off on these three aspects. The same problem also exists in SDN [33]. In the following subsections, we discuss SDN from the aspect of inconsistency in the network state.

3.1 Problem Definition

In the traditional network, the inconsistency problem has been researched for many years. Many protocols and methods have been proposed to solve the inconsistent behavior during reconfiguration of the network. These protocols and methods solve some aspects of network such as in the field of BGP [34], IGP [35], OSPF [36], and some other routing protocols.

Many researchers have noticed the same inconsistency problem in SDN and done much research on this problem. To address the inconsistency problem, the author in [6] defined the inconsistency problem into two levels: per packet consistency and per flow consistency.

Per packet consistency: the per packet consistency update guarantees that when a configuration update occurs, every packet in the network is processed either using the configuration rules existing prior to the update, or the configuration rules existing after the update.

Per flow consistency: the per flow consistency update guarantees that all packets of a flow are processed by the same version of the configuration rules.

3.2 Related Work

In [6] [42], the authors provided a two-phase commit for per-packet consistency. The two-phase commit installs new configuration rules assigned with a new version number in the internal network and then installs the new configuration rules assigned with the new version number in the ingress switch. The ingress switch tags new packets with the new version number and when all packets with the old version number go out of network, the old configuration rules are deleted. But since the switches have to save two versions of configuration rules for some time, this method wastes the memory resource of switches, such as TCAM. It thus incurs high hardware cost, high power consumption and heat generation, and may result in lowering the switch capacity [37]. For per-flow consistency update mechanisms, the authors purposed three implementations: switch rules with timeouts, wildcard cloning, and end-host feedback. However, the latter two of these three implementations are not dependent on OpenFlow technology.

In [38], the authors proposed an incremental consistent update algorithm. This algorithm breaks an update into K rounds and in each round the algorithm moves a part of traffic to the new configuration. This algorithm reduces rule spaces by increasing the update time. However, it can only guarantee the packet-level consistency, not the flow-level consistency.

In TIMECONF [39] method, the controller enforces coordinated updates by incorporating a scheduled execution time, T, in every configuration message. Thus, every update procedure starts with an offline preprocessing stage, where the controller computes the update time T and distributes the configuration messages. Consequently, every switch executes the configuration update at the scheduled time, T, and thus all updates are performed during the period (T-d; T+d), where d denotes the clock synchronization accuracy in the system.

In [40], authors proposed a method in which switches send all packets affected by the update procedure to the controller. The controller caches the packets and update switches' configurations. When all switches have been updated, controller sends the packets back to switches. This method consumes bandwidth between the controller and switches.

In [41], the authors proposed a K-prefix covering scheme to guarantee flow-level consistency. This scheme computes K optimal prefixes, which can cover existing flows by collecting the header information. Then it installs the new rule and K old sub-rules with lower and higher priorities, respectively. When the old rules reach time out, the new rules come into effect.

In [43], the authors proposed an updating method based on classification of rules. In this method, the controller first divides the network devices into two parts, entry network devices and other network devices, and then divides flow entries into four parts: new flow entry, shared flow entry, deleted flow entry, and modified flow entry. Based on these classifications, the controller installs and deletes different kinds of flows in different types of switches.

In [44], the authors discussed where to place the controller and how many controllers are needed to satisfy the requirement of network on the basis of the analysis of the average latency and the worst-case latency.

3.3 Example of Inconsistency Problem

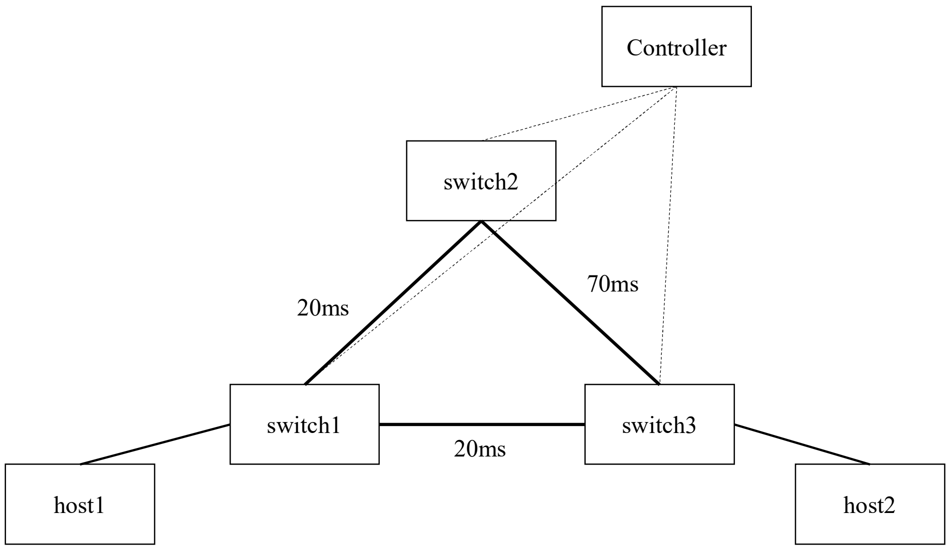

Figure 3- 1 Example of inconsistency problem – per packet consistency

Example 1. Per packet consistency

Figure 3- 1 shows an example of inconsistency problem [45]. Three switches and two hosts are in the network topology.

Loop

The delay between switch1 and switch2 and between switch1 and switch3 are 20ms, the delay between switch2 and switch3 is 70ms. The controller runs an application that forwards packets along the link with a minimum delay.

Initially, the link between switch1 and switch3 is down, the packets from host1 to host2 are forwarded via switch1 and switch2 and the link delay is 90ms. At some time, the link between switch1 and switch3 is up. Then the minimum delay between host1 and host2 is 20ms via switch1 and switch2. When controller detects this change, controller updates the configurations in every switch to forward the packets. However, if the switch1's configuration is updated prior to switch2's, then the packets from switch1 towards switch2, will be sent back to switch1 by switch2 because at this time the minimum delay from switch2 to host2 is via switch1. And loop happens between switch1 and switch2.

Packet loss:

Another situation is that at first packets are forwarded along the path with switch1 and switch3, and then forwarded along the path with switch1, switch2, and switch3. So the controller has to update switch1 and switch2. However, if swich1 is updated before switch2, the packet is forward towards switch2 where there is no rule for these packets in switch2, thus the packets loop between switch1 and switch2.

The switch1's configuration is updated prior to switch2's due to the differences in the delay between switches and controller or differences in the process speed of different switches.

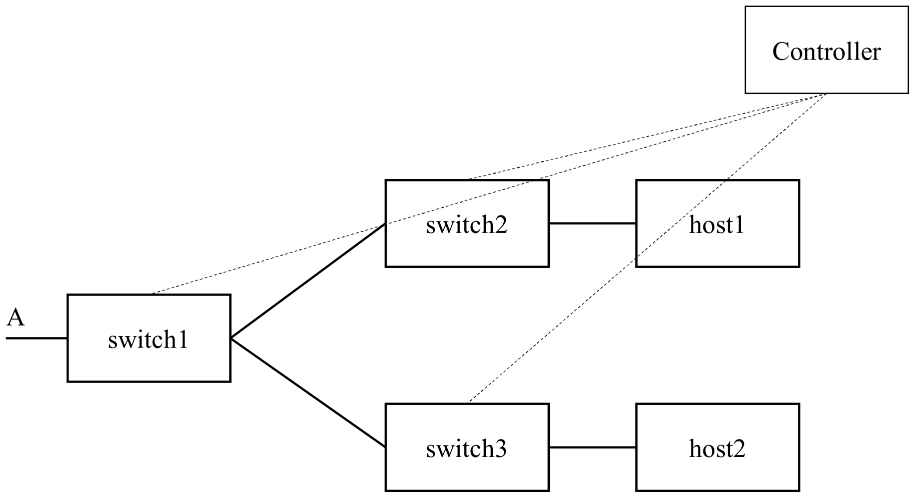

Figure 3- 2 Example of inconsistency problem – per flow consistency

Example 2. Per flow consistency

Figure 3- 2 shows another inconsistency problem. Three switches and two hosts are in the network topology. Host2 is a replica of host1. Switch1 acts like a load balancer.

Initially, flows from A (say A as a network) are forwarded to host1 via switch1 and switch2. At some time, controller detects that the load of host1 or the bandwidth between switch1 and switch2 is getting heavily used. The controller switches the flow from the upper link to the lower link in Figure 3- 2. But the controller has to process carefully so that the flow is not divided into two parts, of which one part gets forwarded to host1 and another to host2. This will cause the service broken.

Another problem in this network topology is that if the switch3's configuration is updated later than switch1's, switch1 will forward packets and flows to switch3, and since switch3 has not been updated yet, the packets will be lost between switch1 and switch3. Although switches may send the packets to controller because no rules match to these packets, but since at this time, the controller has sent the new rules to switches and due to delay between switches and controller, the new rules have not been installed yet, but controller's status has changed. If the application in controller doesn't handle it very well, the controller may not know how to process these packets but dropped.

The inconsistency problem could cause packets loss, loop, and flow break if the controller does not handle the update carefully. It is very important that the controller is capable to handle the inconsistency problem.

Chapter 4

Network configuration update scheme

In this part, we will introduce our proposed scheme to update network configuration. This scheme can update network configuration without loop and packet loss. It analyzes the old and new configurations and computes a correct updating order for the switches.

4.1 Assumptions

a) Data plane is simply forwarding packets based on destination IP addresses;

b) Data plane does not modify any packet header by neither old nor new rules (except modification of destination IP address in type II, see 4.4.1 type II relation);

c) One rule is exactly matched by one flow. That is, no two rules are matched by the same flow.

Based on the assumptions, the controller analyzes old and new rules and classifies the relation between old and new paths into two types according to whether an open loop is included in the graph surrounded by the new and old paths.

4.2 Basic concept

Before introducing the scheme, we first clarify some concept concerning with the scheme. This section introduces the concept of path, new path, old path, loop, open loop, etc. used in the description of the network configuration updating procedure.

Path:

A path is a finite or infinite sequence of edges, which connect a sequence of distinct vertices. A forwarding path is a path with finite sequence of links (edges), which connect a sequence of switches and hosts (vertices). Figure 4- 1 shows a path composed of two hosts and two switches.

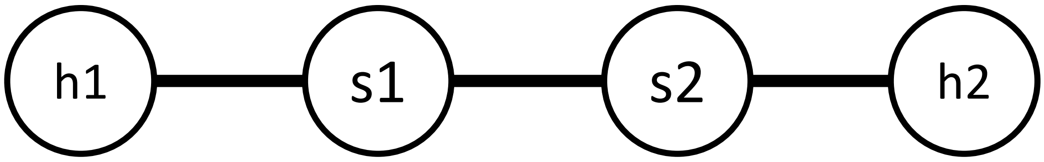

Figure 4- 1 Path

New path and old path:

A new path is a path after the network configuration is updated. An old path is a path before the network configuration update. Figure 4- 2 shows new path and old path.

If packets first flow along with path [h1, s1, s2, s3, h2] and then flow along with path [h1, s1, s4, s5, s3, h2] or along with path [h1, s1, s4, s5, h3], then path [h1, s1, s2, s3, h2] is old path and path [h1, s1, s4, s5, s3, h2] or path [h1, s1, s4, s5, h3] is the new path.

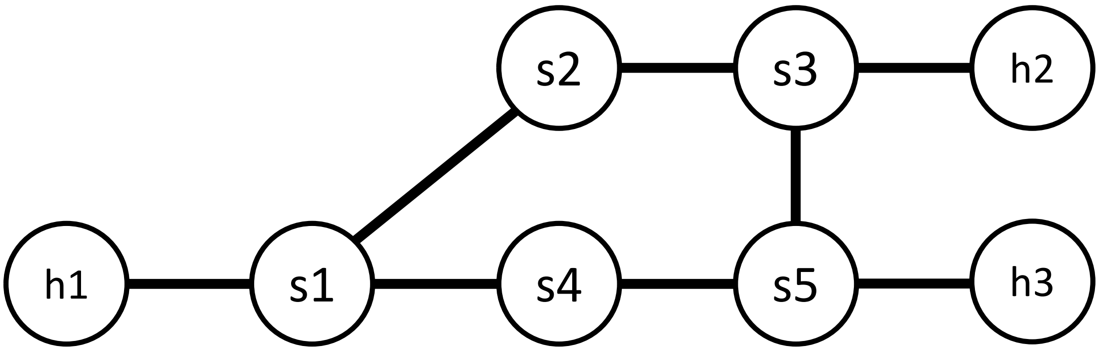

Figure 4- 2 New path and old path

Loop:

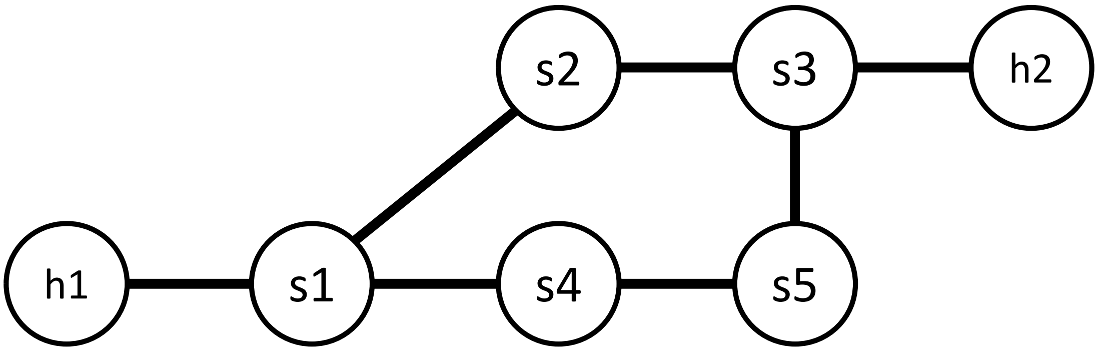

Figure 4- 3 Loop

A loop is a closed path that starts and ends at the same vertex and does not travel to any vertex twice. A loop is also a closed loop with an ingress and an egress. Figure 4- 3 shows a loop composed of four switches [s1, s2, s3, s5, s4]. s1 is the ingress vertex and s3 is the egress vertex.

Open loop:

An open loop is a loop with only ingress and no egress. An open loop is surrounded by old path and new path. Figure 4- 4 shows [h2, s3, s2, s1, s4, s5, h3] as the open loop and s1 is ingress.

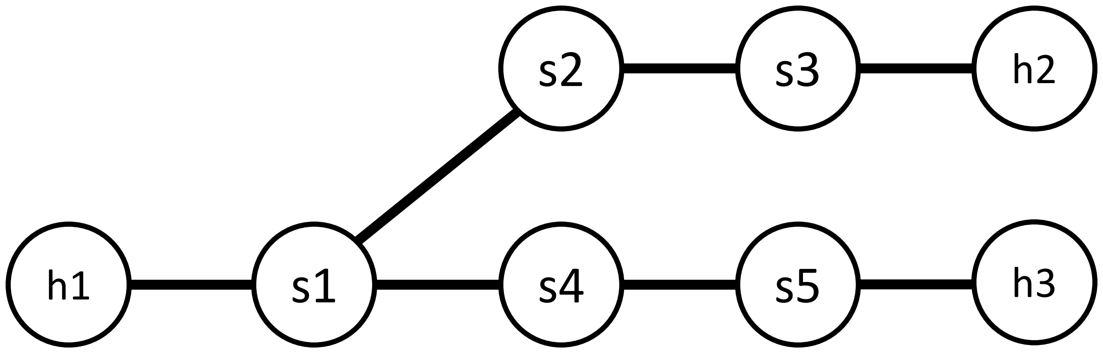

Figure 4- 4 Open loop

4.3 New and old forwarding path relation analysis

A current forwarding path will change to a new forwarding path due to a network status change. Basically, there are two types of path change scenarios:

(1) Path changes but the source host and destination host are the same, and

(2) Path changes and the destination host also changes.

The first scenario may occur in the situation of link or switch up (or down), link congestion or long latency, in which the network has to find a new path to forward packets to the destination. The second scenario may occur in the situation of load balancing, in which the network has to forward traffic to a new replica of a server.

Intuitively, based on the above two kinds of change scenarios, the relation between new and old paths can be classified into two types according to whether an open loop is included in the graph surrounded by the new and old paths.

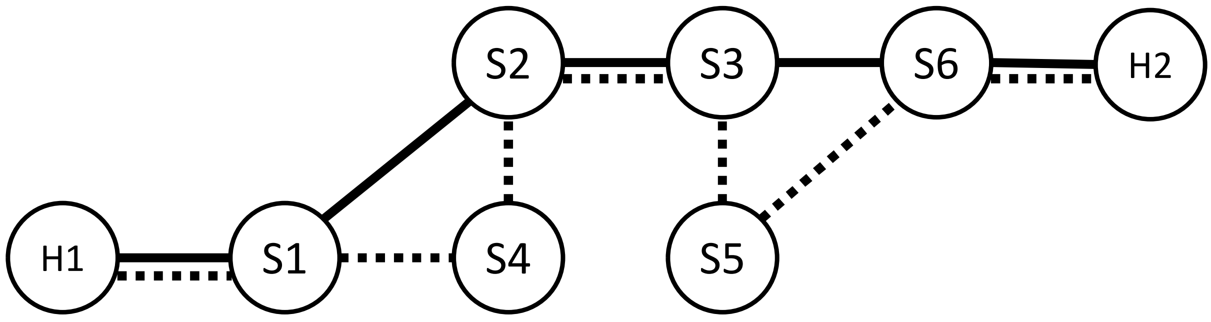

Figure 4- 5 Type I relation: No open loop

Type I relation

Figure 4- 5 shows type I relation. Path[H1,S1,S2,S3,S6,H2] is old path and Path[H1,S1,S4,S2,S3,S5,S6,H2] is new path. In this type, all packets forwarded along with the path finally go to the same destination (H2) and there is no open loop, which means for any ingress (S1, S3), there is an egress (S2, S6).

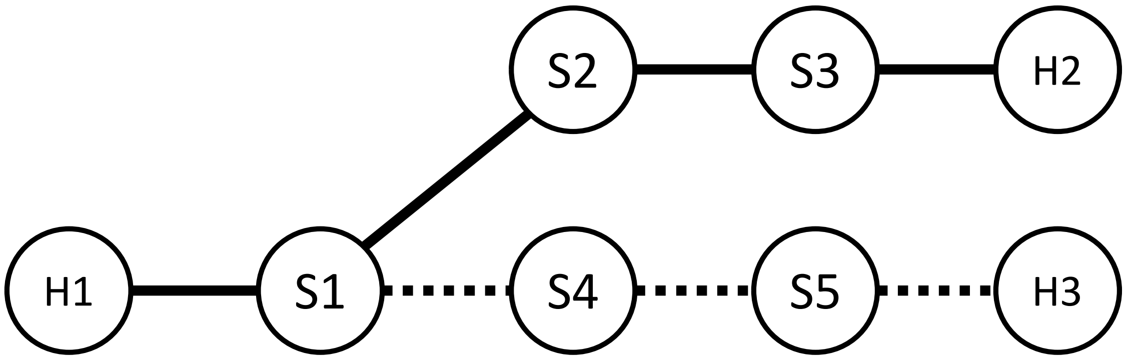

Figure 4- 6 Type II relation: one and only one open loop

Type II relation

Figure 4- 6 show type II relation. Path [H1,S1,S2,S3,H2] is the old path and Path [H1,S1,S4,S5,H3] is the new path. In this type, packets are forwarded to another destination (H1) and there is one and only one open loop. That is, for ingress (S1) there is no egress vertex.

We can prove that there is no open loop in the first type because if there is an open loop, there would be an ingress with no egress, and packets would be forwarded to another host and cannot be forwarded back to the original host since a path cannot traverse to any vertex twice. And also in the second type, there is only one open loop because if there is no open loop, packets will be forwarded to the same host and if there are more than one open loop, a path will traverse some vertices twice.

4.4 Loop-free and no packet loss update scheme

In this section, we introduce the update scheme. We first summarize the update procedure by showing how the scheme guarantees no packet loss and then describe how the scheme guarantees loop free updates.

4.4.1 No packet loss update

No packet loss update is classified according to the type I relation or type II relation. Since Type II relation may contain type II relation, we first describe type I relation update scheme, and by using type I relation update scheme, we describe type II relation update scheme.

Type I relation

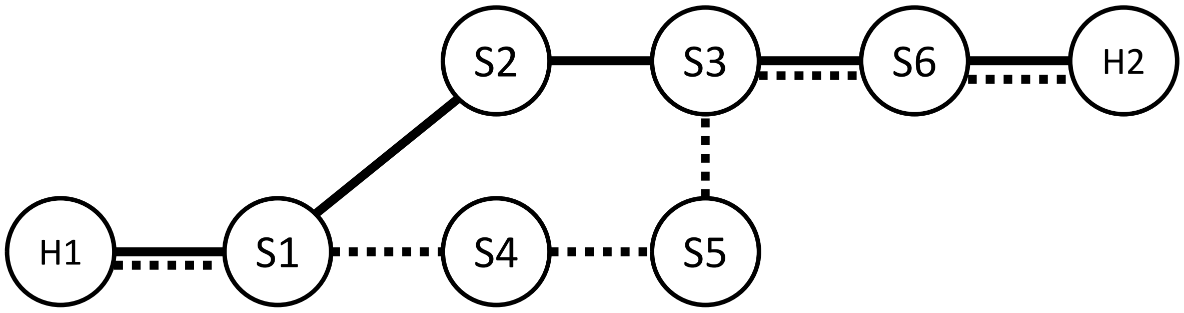

Figure 4- 7 Type I relation with one loop

Figure 4- 7 shows the simplest type I relation. Old path [H1, S1, S2, S3, S6, H2], and new path [H1,S1,S4,S5,S3,S6,H2] compose only one closed loop [S1,S4,S5,S3,S2,S1]. S1 is ingress and S3 is egress vertix. In order to update from the old path to new path with loop free and no packet loss, controller first updates all the inner switches on new path and egress switch (S4, S5, S3) remaining ingress S1 and old path switch S2. During this period, packets are forwarded via S1, S2 and S3 that incurs no packet loss. Once the controller completes updating inner and egress switches, it updates ingress S1 and sets a timeout on the old rule in S2, then packets are forwarded to the new path and packets in-flight in the old path will still be forwarded to the correct destination without loss. Old rules in S2 will be timeout and removed automatically once in-flight packets in the old path are all out of the network.

We summarize the update procedure for type I relation as following:

1. Find out closed loops in type I relation;
2. Determine ingress and egress of each closed loops;
3. Update inner switches on the new path of closed loop first;
4. Wait for all switches update completed in step 3;
5. Update ingress and set a timeout on inner switches on old path of closed loop;
6. Repeat step 3 to step 5 for each closed loop.

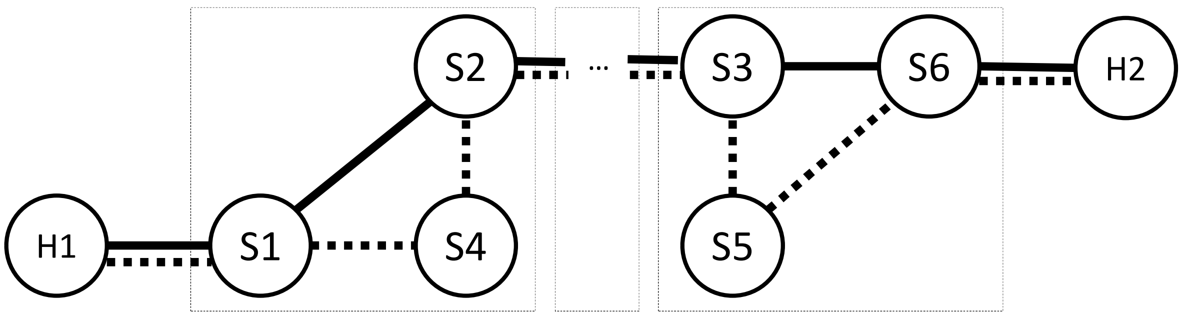

Figure 4- 8 Type I relation with more than one loop

Figure 4- 8 shows type I relation with more than one closed loop. We can see these closed loops as some different independent closed loops and update one by one following the procedure mentioned above repeatedly. Because for each closed loop, it can guarantee loop-free and no packet loss, the controller can update the whole network with loop-free and

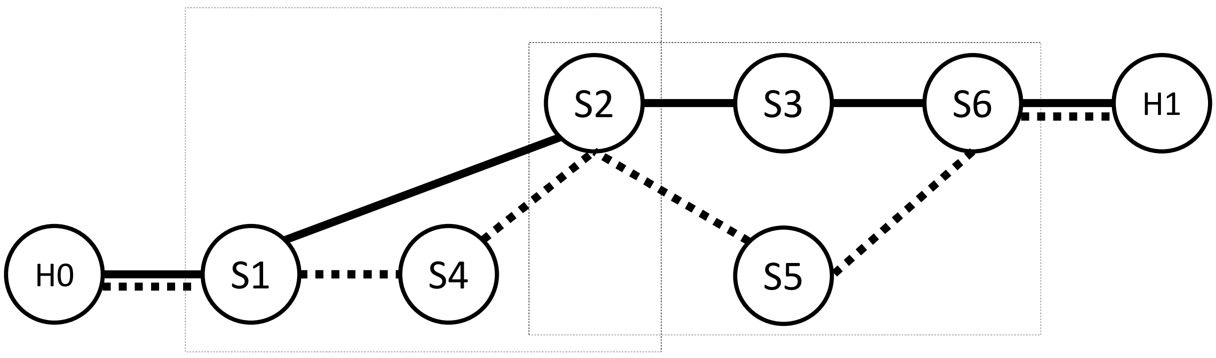

Figure 4- 9 Type I relation with two loops connected by on switch no packet loss.

Figure 4- 9 shows a special situation where two closed loops are connected by one switch. In this situation, the first closed loop's egress is the second closed loop's ingress vertex. Under the assumption the "data plane is simply forwarding packets based on destination IP addresses", new rules in the switch are separated into two parts: one part belonging to the previous closed loop and the other part belonging to the latter closed loop. This kind of topology can be updated following the procedure of type I as well.

Type II relation

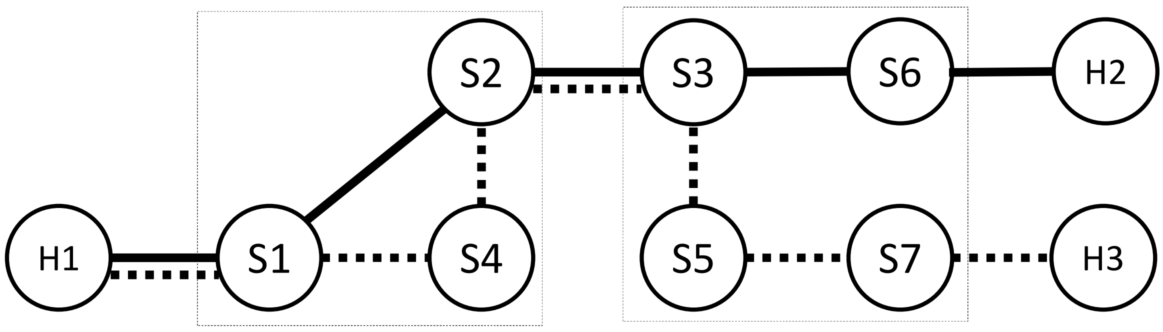

Figure 4- 10 Type II relation with a closed loop and an open loop

Figure 4- 10 shows the type II relation. Path [H1,S1,S2,S3,S6,H2] is the old path and path[H1,S1,S4,S2,S3,S5,S7,H3] is the new path. [S1, S4, S2] compose a closed loop where S1 is ingress and S2 is egress. [H2, S6, S3, S5, S7, H3] composes an open loop where S3 is ingress and there is no egress for it. Host H3 could be a replica of host H2. S3 modifies the header of packets destined for H3 for load balancing. In order to update from the old path to a new path with loop free and no packet loss, the controller first updates the closed loop by following the procedure mentioned in type I relation. Then the controller updates switches on the new path of open loop S5 and S7. During this period, packets are still forwarded to H2 with no packet loss. After this completion, the controller updates the ingress S3 and sets a timeout on S6's rule. Then packets from H1 will be forwarded to H3. Packets in-flight in the old path will still be forwarded to H1 without any loss. Old rules in S6 will be timeout and removed once all in-flight packets are out of the network.

For type II relation, we summarize the update procedure as follows:

1. Find out closed and open loops;

2. Update closed loops;

3. Determine ingress of open loop;

4. Update switches on new path of open loop;

5. Wait for all switches update completed in step 4;

6. Updated ingress and set a timeout on old path of open loop.

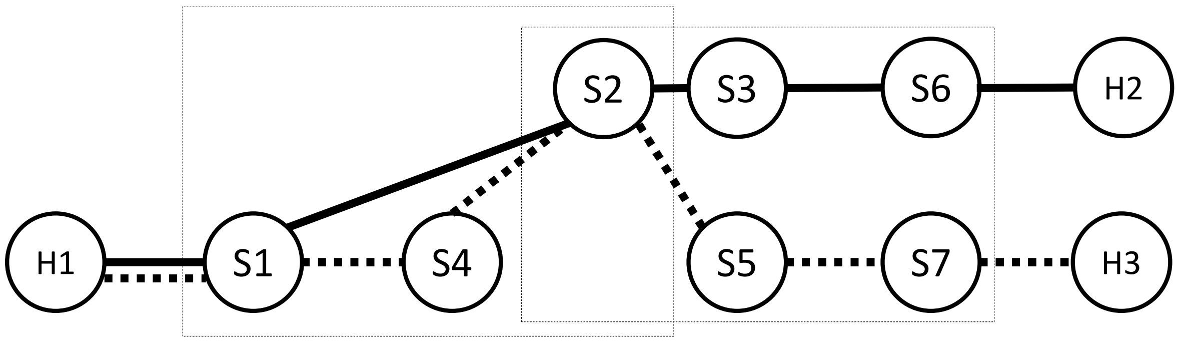

Figure 4- 11 Type II relation with closed loop and open loop connected by switch S2

Figure 4- 11 shows a special situation that the last closed loop's egress may be the open loop's ingress. Controller can also update the network configuration by using the procedure described above.

4.4.2 Loop free update

Consider the topology shown in Figure 4- 12, which is a loop-prone topology. Suppose that the current forwarding path is [H1,S1,S4,S2,S3,S5,S6,H2], shown by the solid line and new forwarding path is [H1,S1,S2,S3,S4,S5,S6,H2] shown by the dotted line. This forwarding path can be classified as Type I relation, thus we can use the Type I update scheme to update this network.

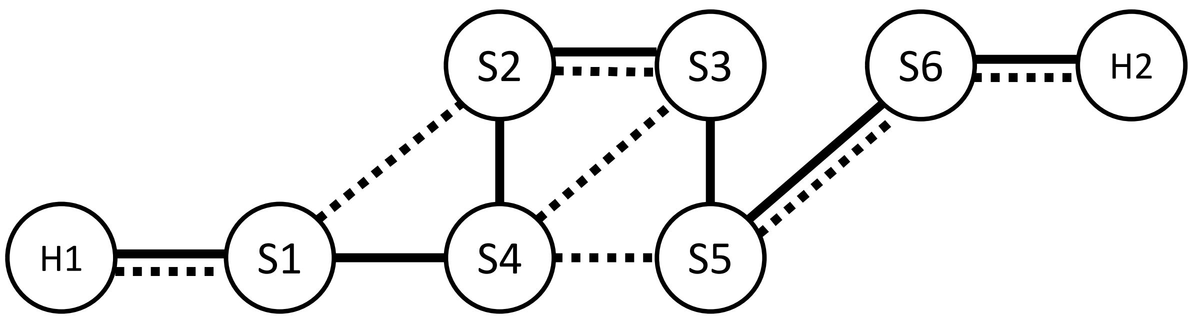

Figure 4- 12 Loop-prone topology

The controller finds out all the closed loops in the network: loop [S1,S4,S2] and loop [S3,S4,S5]. The controller first updates the loop [S1,S4,S2], as shown in Figure 4- 13. For now the packets are forwarded via [S1,S2,S3,S5,S6], and no loop occurs. After confirming the update completion of the first loop, we update the loop [S3,S4,S5], as shown in Figure 4- 14. After the second loop update finishes, the forwarding path is switched to the new path and during update period, packet loop is diminished.

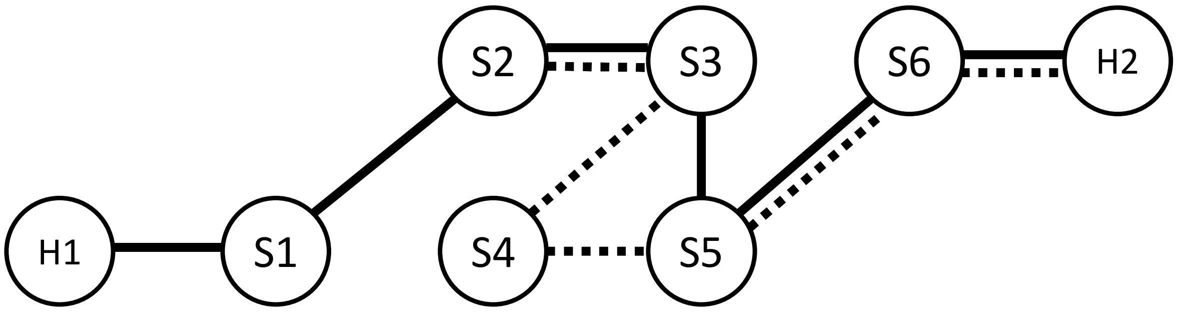

Figure 4- 13 Network topology after updating loop [S1, S4, S2]

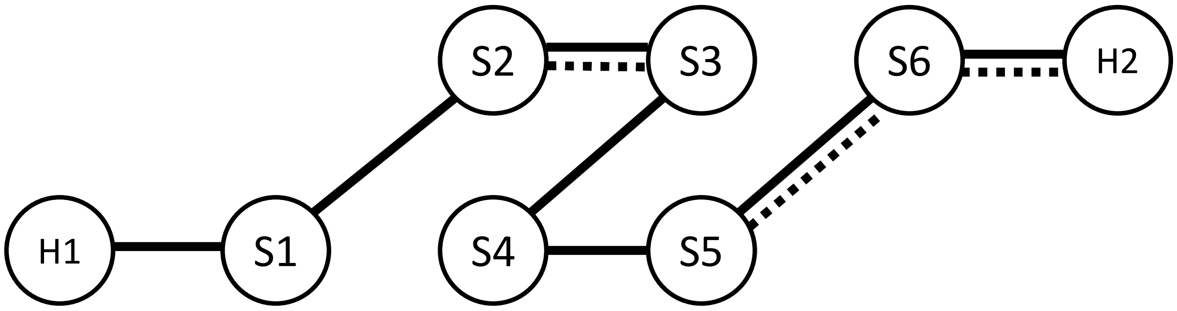

Figure 4- 14Forwarding path after updating loop [S3, S4, S5]

4.5 Algorithm to find a loop and determine ingress and egress

As mentioned in the previous section, finding out closed and open loops surrounded by old and new paths and determining ingress and egress switches are the key to realize loop-free and no packet loss update. In this section, we describe how to find the loops and how to determine ingress and egress vertices.

4.5.1 Forwarding graph

A serial of switches and links in order can represent a forwarding path. We use $G(V,E)$ to present a forwarding path graph. V represents a set of switches and E represents a set of links.

New forwarding path can be expressed as $\mathrm{G}_n(V_n, E_n)$, V_n is a set of switches and E_n is a set of links along with new path.

Old forwarding path can be expressed as $\mathrm{G}_o(V_o, E_o)$, V_o is a set of switches and E_o is a set of links along with old path.

$\mathrm{G}_{on}(V_{on}, E_{on}) = \mathrm{G}_o \cup \mathrm{G}_n$ represents the whole graph composed by old and new paths where $V_{on} = \mathrm{V}_o \cup \mathrm{V}_n$ and $E_{on} = \mathrm{E}_o \cup \mathrm{E}_n$.

4.5.2 Algorithm to find a loop

We use the graph abstraction described above to find loops in the forwarding path graph.

In [46] the author describes an algorithm to find a fundamental set of loops of a graph. We borrow this algorithm to find loops. Controller first analyzes the old path $\mathrm{G}_o(V_o, E_o)$ and the new path $\mathrm{G}_n(V_n, E_n)$, and merges them into a whole graph $\mathrm{G}_{on}(V_{on}, E_{on})$. If the graph belongs to type II relation, controller adds an edge between the final two end hosts into the graph (not to add a link into data plane) in order to make an easy computation. Then, controller computes a spanning tree and finds out all loops. The algorithm is as follows:

T is the set of vertices in spanning tree and X is the set of vertices not yet examined. At the beginning, let $T = \emptyset$, $X = V$, then take any vertex v from X as the root of the spanning tree, and add v into T, then $T = \{v\}$, $X = V$. Next take any vertex z from $T \cap X$, if z is not none, examine z: if z's neighbor w is not in T, add w into T, if w is in T, a loop is found. This loop consists of edge(z, w) and the unique path in tree T from w to z. In each case, delete edge (z, w). After all z's edges have been visited, remove z from X and examine a new z.

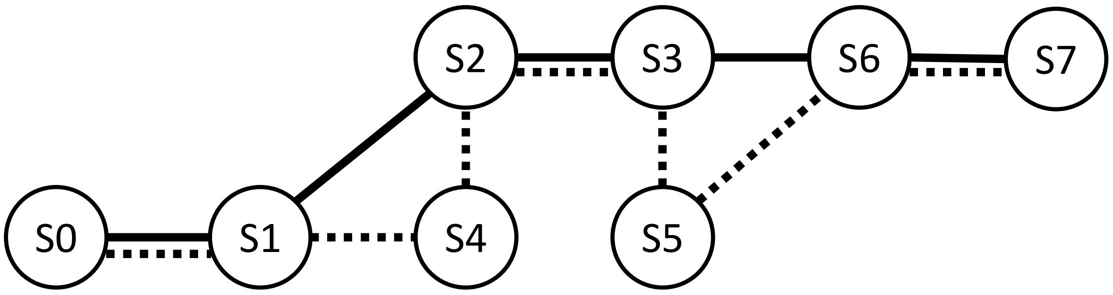

Figure 4- 15 Example of finding loops by creating spanning tree (a)

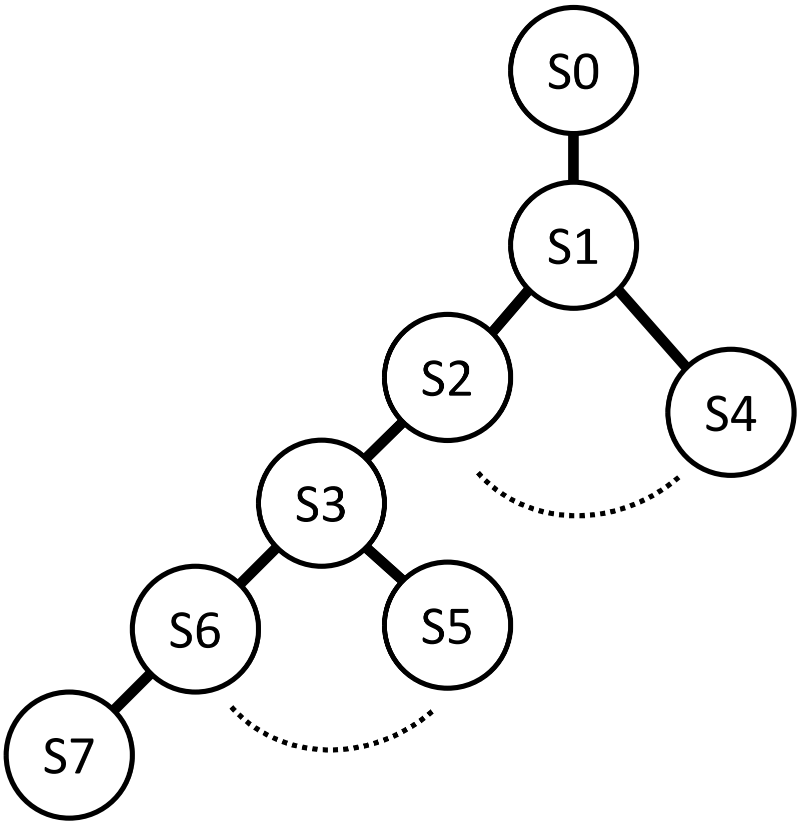

Figure 4- 16 Example of finding loops by creating spanning tree (b)

Figure 4- 15 and Figure 4- 16 shows an example of finding loops by creating a spanning tree. Figure 4- 15 is the graph to be examined and Figure 4- 16 is the spanning tree created. First take S0 as tree's root and examine S0's neighbor S1. Since S1 is not in tree, add S1 into tree, and then examine S1 and its neighbors. Since the neighbors S2, S4, S2 and S4 are not in tree, add them into tree. Next examine S2 and its neighbor S3 and S4. Since S3 is not in tree, add S3 into the tree whereas S4 has been in tree. Then, we found a loop constructed by S2, S4 and S1. Using the same algorithm we can find another loop [S3, S6, S5].

4.5.3 Determine ingress and egress

Using the algorithm in [46], we can find loops in graph. But, it simply gives out a set of vertices, and the order of vertices in set is uncertain because of various roots of spanning trees and various searching ways via the tree. In order to update data plane in a correct order, the controller must know the ingress, egress and switches' order between ingress and egress. We compare the set of loops with V_{op} and V_{np} respectively and based on the order of V_{op} and V_{np}, we can get a correct order of switches.

For example, in Figure 4- 7, the loop is a set of vertices [S1,S2,S3,S5,S4], and the new path is a set of vertices [H1,S1,S4,S5,S3,S6,H2]. We can first compute Loop∩Path = [S1,S3,S5,S4], and compare this with Path to get a correct order [S1,S4,S5,S3], where S1 is ingress and S3 is egress. For type II relation, we can find ingress and egress use in the same way, whereas the last loop's egress is a host not a switch, then we can know the first switch in the vertices set is the ingress of the last open loop.

4.6 A framework for updating network configuration

In this section, we describe a framework for the loop-free and no packet loss update scheme.

Figure 4- 17 shows the framework for update. There are several modules in this framework as described below.

Topology Discovery module

It discovers the network topology for the other modules.

Network Monitor module

It interacts with data plane and obtains network statics information such as flow statics, table statics, packets statics, load statics, link status and so on. It also sends information obtained to Update Analysis module.

Update Analysis module

It analyzes received information and decides to update data plane if one or more statics parameter is beyond a threshold. It calculates a new path based on the topology given out by Topo Discovery module and sends the old path and new path to Type Identification module.

Type Identification module

It identifies relation type according to old and new paths, forms a forwarding path graph and sends the graph to Graph Analysis module.

Graph Analysis module

It analyzes the graph and gives out cycles, ingress, egress and switch update order to Update module.

Update module

It updates data plane by using the switch update order received from Graph Analysis.

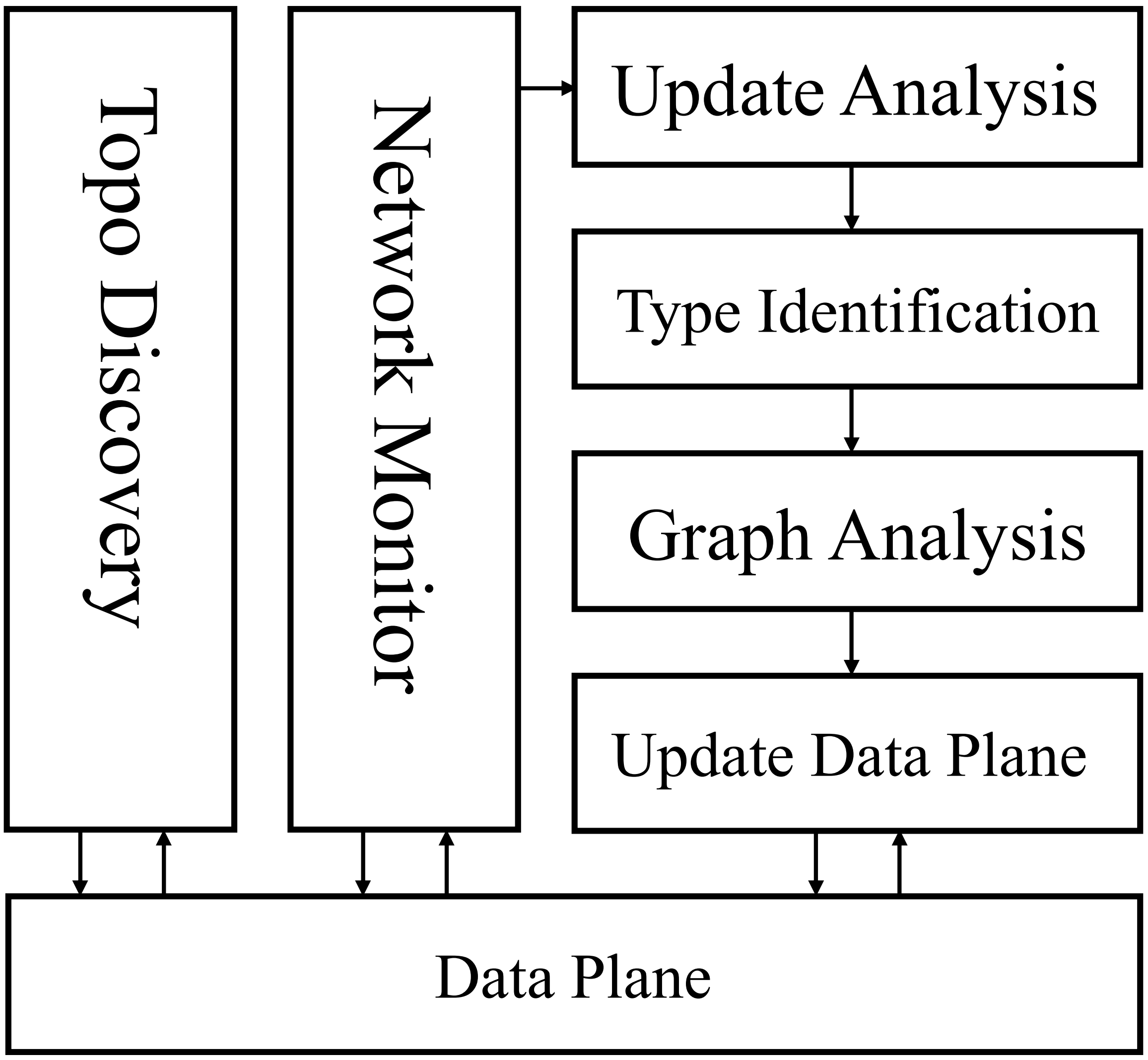

Figure 4- 17 Framework for loop-free and no packet loss update scheme

Chapter 5

Experiments

In this chapter, we conducted experiments to verify the proposed scheme. We first introduce tools used in the experiments. Then we proposed a method to measure the delay between the controller and data plane and compared with a related method. We describe how to set a delay value to each link between switches in data plane and controller. Finally, we test our scheme by using TCP and UDP applications.

5.1 Tools for experiment

Mininet

Mininet [47] is a network emulator for rapidly prototyping large networks on a single computer. Since code developed in Mininet can deployed in a real network, it is more realistic compared with other simulators such as ns-2 [48] or Opnet [49]. Mininet can also create lightweight virtual machines for network performance tests. Figure 5- 1 shows the components and connections in a two-host network created with Mininet.

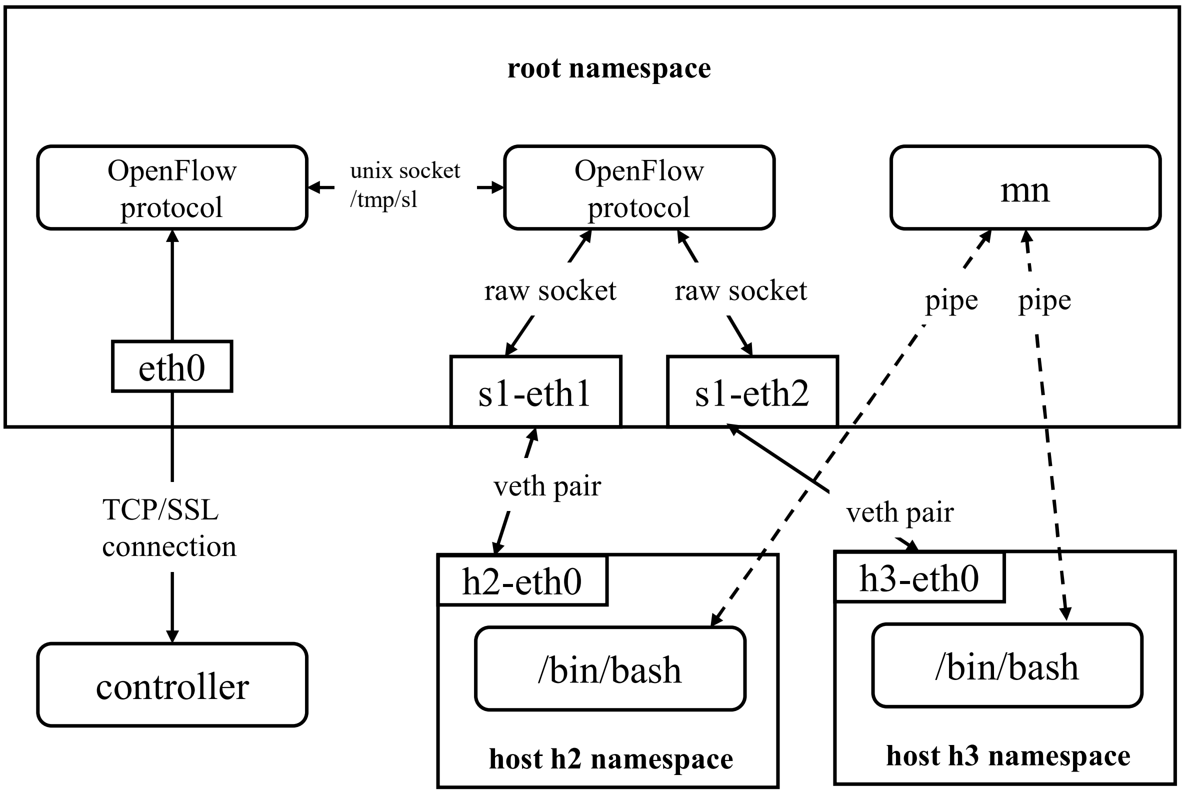

Figure 5- 1 Components and connections in a two-host network created with Mininet

POX

As mentioned in Chapter 2, POX is an open source OpenFlow controller platform. POX uses python as programming language. POX is a publish/subscribe paradigm controller. In a publish/subscribe pattern [50], there are 3 parts -- publisher, subscriber and event. Publisher can publish raise an event without any knowledge of subscriber. A subscriber can subscribe a publisher. Multiple subscribers can subscribe a publisher, and a subscriber can subscribe multiple publishers as well. POX works in publish/subscribe pattern and is an event-driven system. Figure 5- 2 shows a publish-subscribe paradigm.

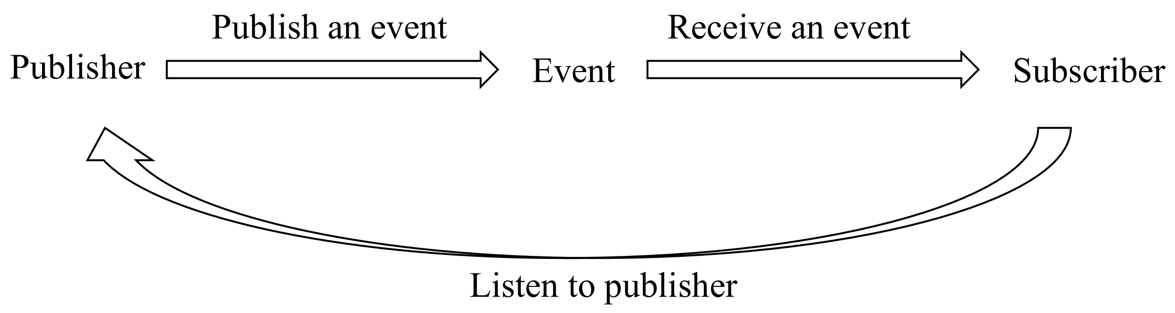

Figure 5- 2 Publish-subscribe paradigm

Wireshark

Wireshark is a network protocol analyzer [51]. It supports hundreds of protocols including OpenFlow. Wireshark captures live packets and analyzes packets real-time and offline. It can run on multi-platform such as Windows, Linux, OS X, FreeBSD, NetBSD.

Iperf

Iperf [52] is a commonly used network testing tool that can create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams and measure the throughput of a network that is carrying them.

***netem* and Traffic Control (*tc*)**

netem [10] provides Network Emulation functionality for testing protocols by emulating the properties of wide area networks. *tc* is a Linux command to configure the Linux kernel's network scheduler to show and manipulate Network traffic control settings.

5.2 Delay measurement between controller and data plane

It is important to measure the delay between a controller and data plane. Results of measurement can be used to evaluate the update scheme and monitor the actual delay when the delay is set between the controller and switches.

We first introduce a method to test the delay between controller and data plane. We utilize echo request/reply messages to measure the delay between controller and switches. The Echo request/reply messages are usually used to keep alive the connection between controller and data plane.

Echo request/reply message is a symmetric message, which means that either controller or switch can send this message without prior solicitation to the other.

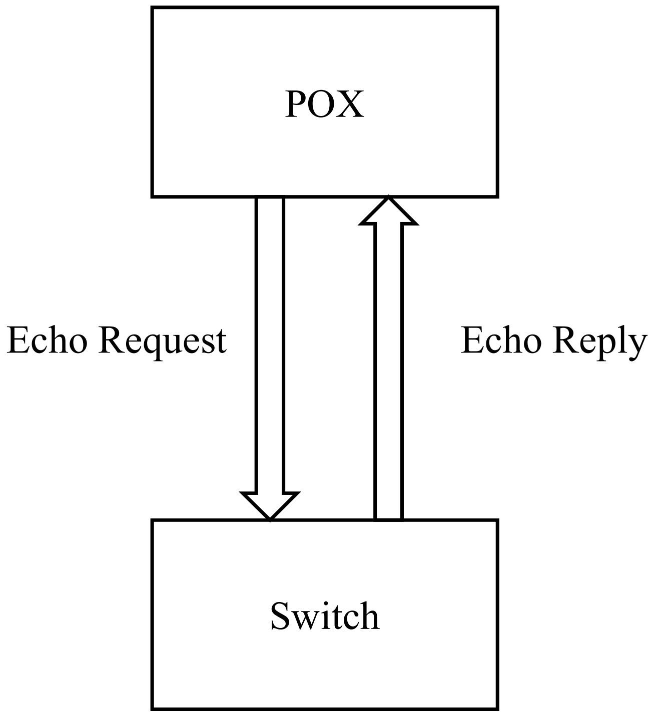

Figure 5- 3 Echo request and reply messages

Figure 5- 3 shows the Echo request and reply messages. In the system composed of POX and Mininet, the echo request message is usually initially sent by Mininet and POX returns an echo reply every 5 seconds. However, the echo request message initially sent by data plane cannot do any measurement on the controller side. We thus modified the source code of POX to send the echo request message initially.

Experiment setup:

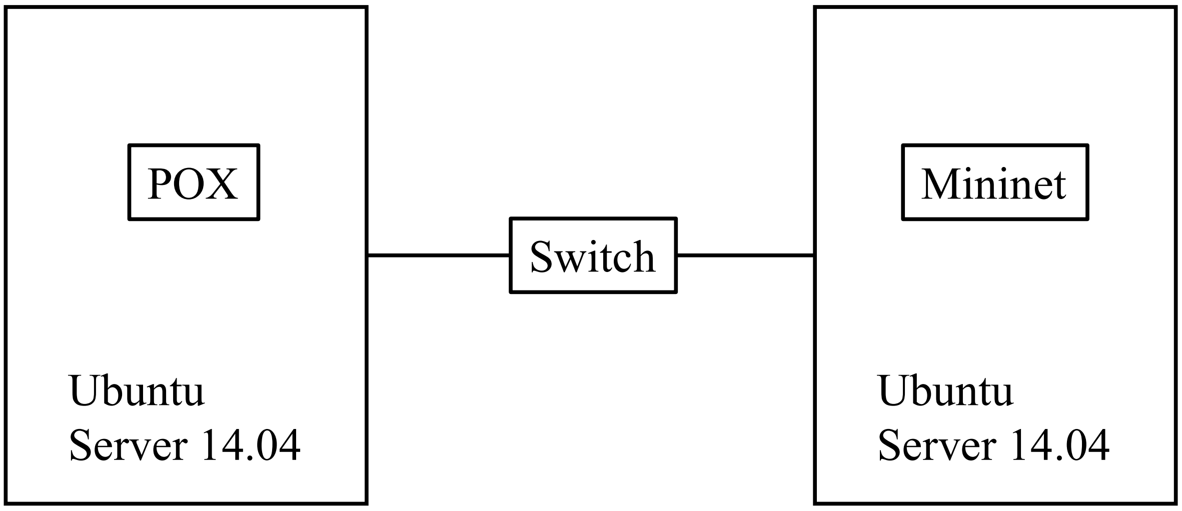

Figure 5- 4 Experiment setup

Figure 5- 4 shows the experiment setup. POX and Mininet are installed in two different Ubuntu Servers 14.04. We tested two types of round-trip delay setups. In the first setup, the controller and switch are next to each other and connected via a switch. Both POX and Mininet are in the same local area network. In the second setup, we assume a long distance round-trip measurement that the controller is located in America whereas Mininet is located in Japan.

Experiment results

We modified the POX source code to make Echo request/reply message supporting delay measurement between controller and switches. Controller sends a echo request message to a switch every second and waits for the switch's reply message. We didn't set any delay value for the communication path between the controller and switch. Controller records the Echo request sent time $T1$ and the corresponding Echo reply receiving time $T2$. It calculates the round-trip delay $T\ as\ follows$:

$$T_d = T2 - T1$$

Figure 5- 5 shows the results of round-trip delay measurement between a controller and a switch which are located in the same local area network.

Figure 5- 6 shows the results of round-trip delay between a controller and a switch, which are separated by a long distance that the controller was in a server located in America borrowed from University of Massachusetts Amherst and switch was located in Japan.

For each experiment, we measured the round-trip delay for 120 seconds and compared this method with the results of delay measured by “ping” command at the same time.

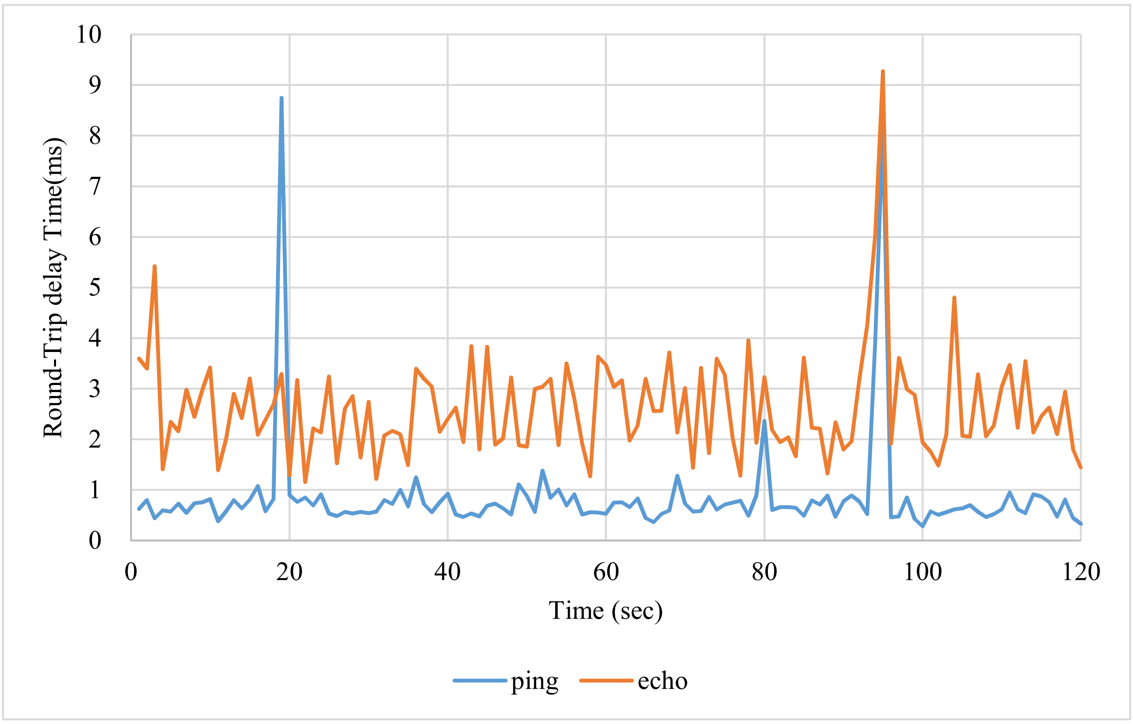

Figure 5- 5 Round-trip delay: Controller and switch are in the same LAN

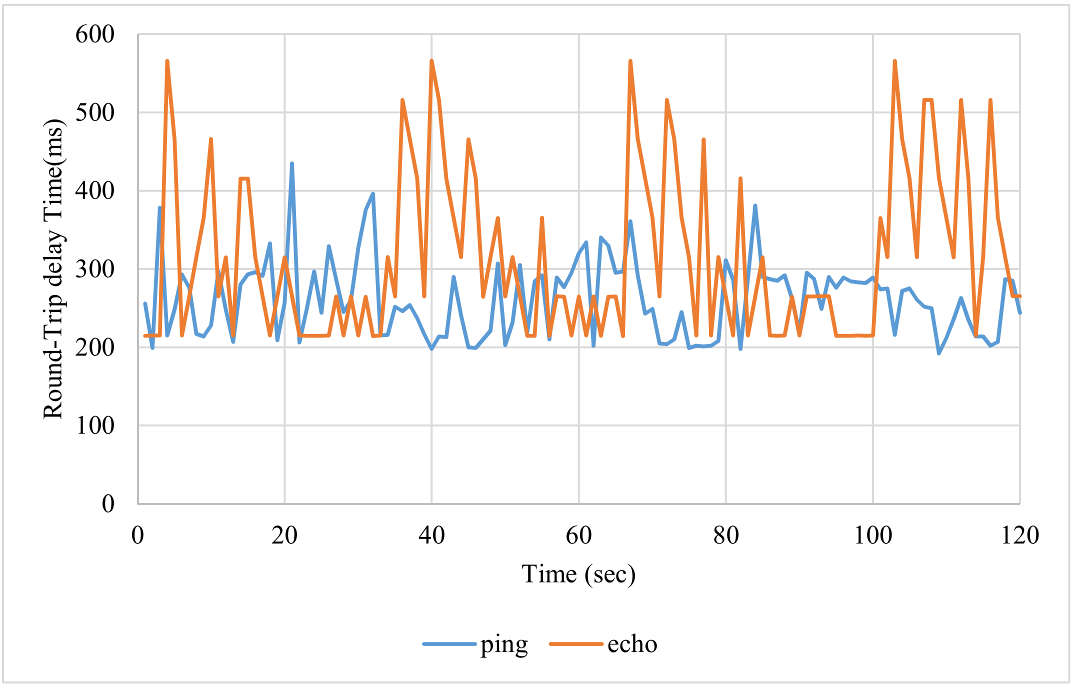

Figure 5- 6 Round-trip delay: Controller and switch are separated in a long distance

5.3 Setting delay between a controller and switches

In order to verify the update scheme, we need to set delay between the controller and switches to establish an inconsistent network configuration update environment. We used traffic control on interface in Linux to control the traffic flowed through the interface.

According to Figure 5- 1, Mininet communicates with controller via an interface eth0, and switches emulated by Mininet are in the root namespaces. If we use *tc* command to control traffic on this interface, the delay between controller and each switch will remain the same value.

By analyzing the packets between controller and switches, we found that although Mininet uses one interface and one IP address to communicate with POX, POX establishes multiple connections with each switch using different TCP port numbers. We thus could set different delay values for each connection between the controller and switches by filtering on these port numbers.

Usually, the link delay is a bi-direction feature. For example, if a link between A and B has a delay of 10ms, then a packet suffers 10ms delay no matter if this packet is from A to B or from B to A. And since *tc* can only control packets out of an interface, we used *tc* to set delay both on POX and Mininet sides.

```
$sudo tc qdisc add dev eth0 root handle 1: prio
$sudo tc qdisc add dev eth0 parent 1:1 handle 10: netem delay 100ms
$sudo tc filter add dev eth0 protocol ip parent 1:0 prio 3 u32 match ip src
172.21.66.182/32 match ip sport 44549 0xffff flowid 1:1
```

Above commands show an example to set 100ms delay for packets with source IP address of 172.21.66.182 and TCP source port of 44549 coming out of interface eth0.

Results

Figure 5- 7 show the round-trip delay results when 100ms delay was set for the link between the controller and switch. The round-trip delay was measured by exchanging the Echo messages. The test period was 120 seconds.

Figure 5- 8 show the round-trip delay results when setting 100ms, 200ms and 300ms delay to each link between the controller and switch. The round-trip delay was measured by exchanging the Echo messages. The test period was 120 seconds.

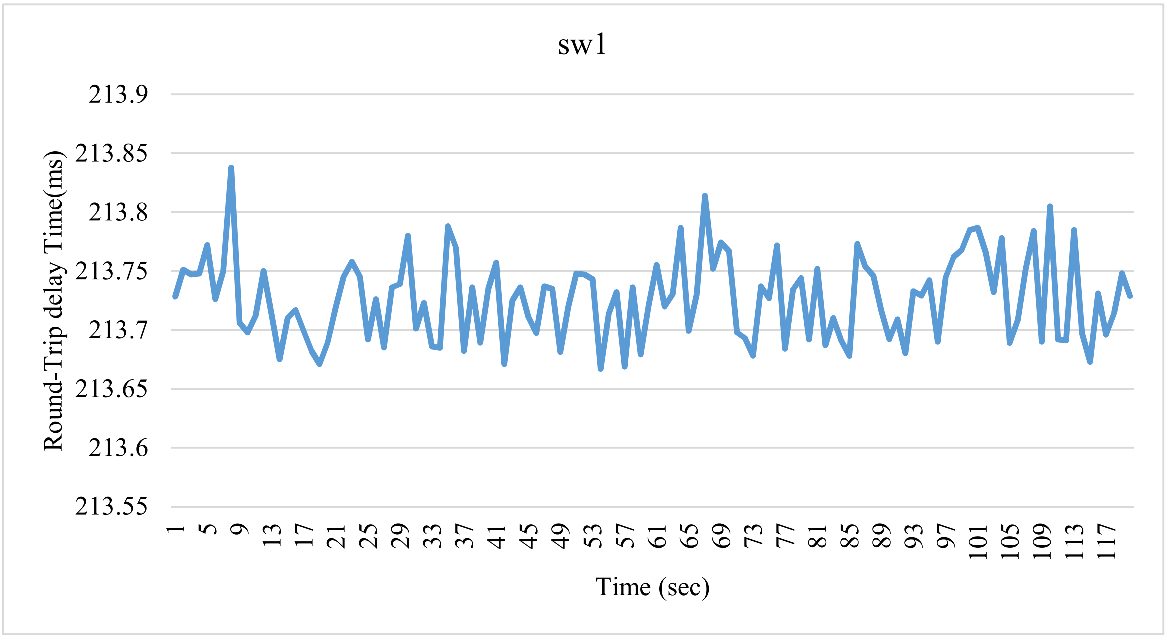

Figure 5- 7 Set 100ms delay between controller and switch

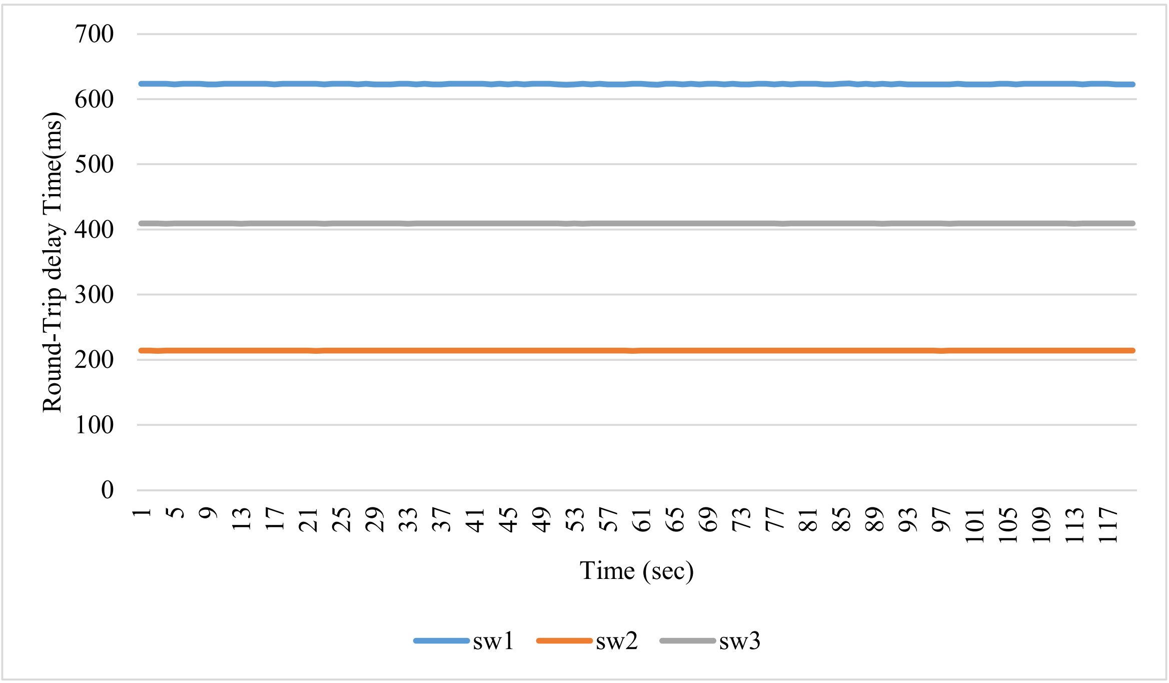

Figure 5- 8 Setting different delays to links between the controller and three switches

5.4 Validation of scheme for updating without loop and packets loss

The experiment setup is the same as shown in Figure 5- 4. POX and Mininet are located in different servers. The network topology is shown as Figure 5- 9. Packets are sent from H1 to H2. The old path is [H1, S1, S2, S3, S6, H2] and the new path is [H1, S1, S4, S5, S3, S6, H2]. Controller switches the forwarding path from the old to new path during the test period.

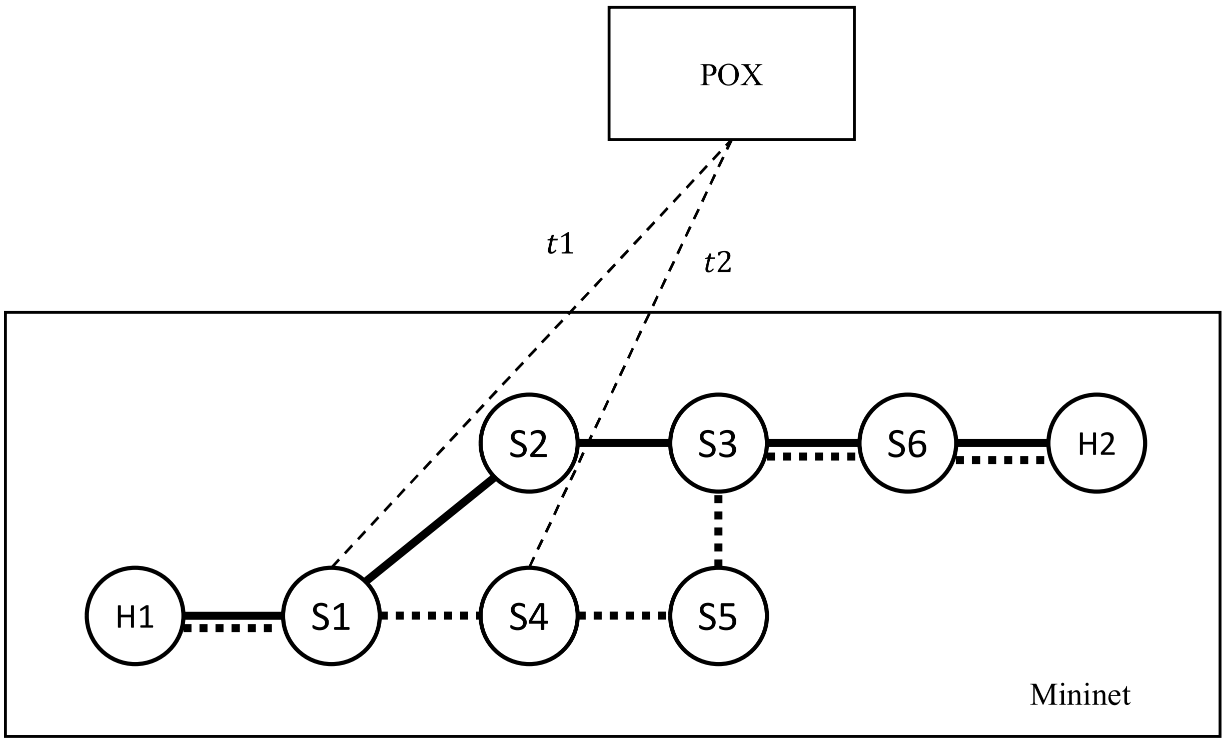

Figure 5- 9 Experiment topology

We defined Δt as the delay difference between the links form the controller to S1 and from the controller to S4. As shown in Figure 5- 9, if the delay between S1 and controller is $t1$ and the delay between S4 and controller is $t2$, then

$$\Delta \boldsymbol{t} = |t2 - t1|$$

Here we suppose that all other delays from controller to switches are smaller than the delay from controller and S4.

The reason to define this variable is because the inconsistency problem is caused mainly by the link delay differences as they lead to switches updating their configurations asynchronously.

5.4.1 TCP test

We first use TCP packets to test on the network with and without our proposed updating scheme.

When the system is started up, the controller first installed a set of rules along the old path to guarantee that *iperf* could establish a connection between H1 and H2. Once *iperf* established the link and sent TCP packets along old forwarding path, the controller updated the network to switch the forwarding path from the old to the new one, this can be triggered by events such as link congestion.

We tested with the network topology by setting Δt varied from 100ms to 1000ms both with and without our updating scheme to show how our scheme guarantees the network's correct behavior.

We recorded the statistics about the data transferred by *iperf* and calculated the throughput every 0.5 second. For each Δt value, the total testing period was 20 seconds.

5.4.2 General throughput characteristic of paths

We tested the new and old path throughputs without any configuration update to understand the general throughput characteristic of both paths.
Figure 5- 10 shows the results of the test. Both new (blue line) and old (orange line) paths' throughputs are in the range between 3.3 GBytes/sec and 4 GBytes/sec.

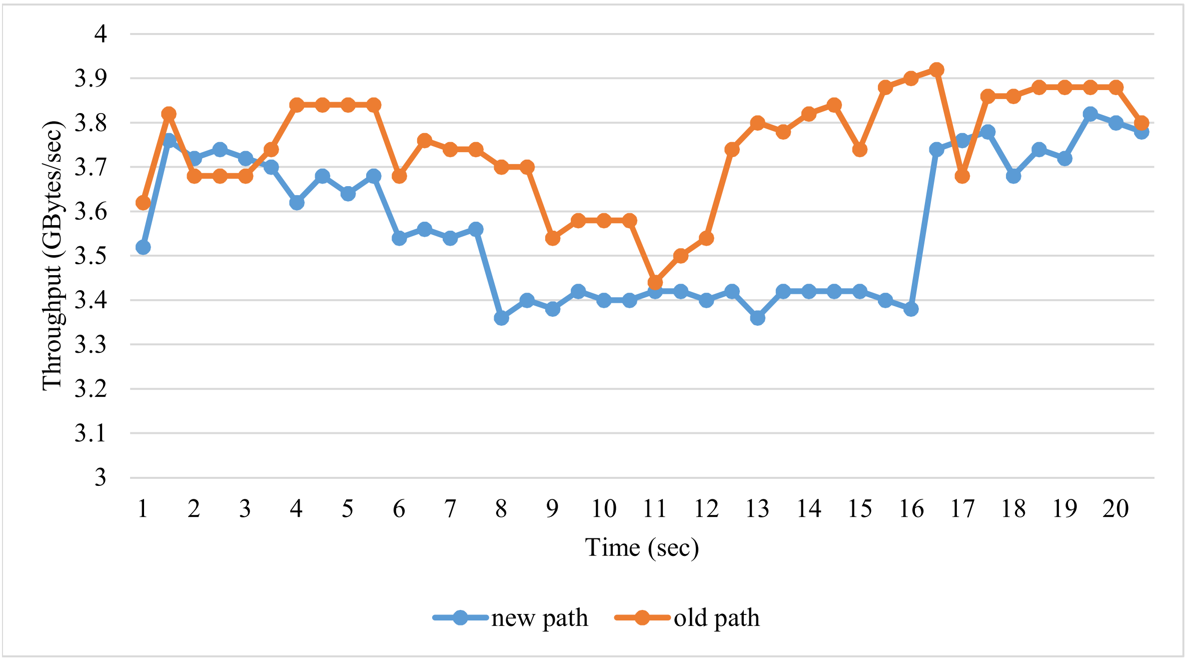

Figure 5- 10 General throughput characteristic of paths

5.4.3 Performance measurement of the proposed scheme

The controller switched the forwarding path from the old path to the new path, and we measured the throughput twice: once with our scheme and another without using our scheme for each Δt.

Figure 5- 11 shows the throughput with our scheme during update and Figure 5- 12 shows a comparison between the throughput with our scheme (blue line) and the one without scheme (orange line) during network configuration updating. We can see that during the update period,

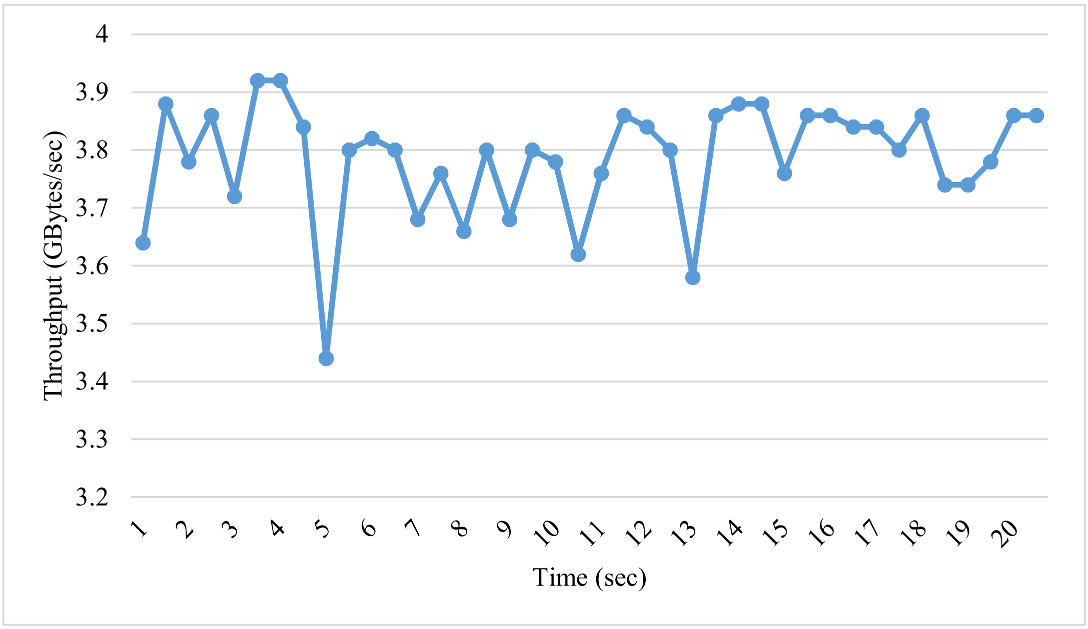

Figure 5- 11 Throughput with our scheme $\Delta t = 100$ ms

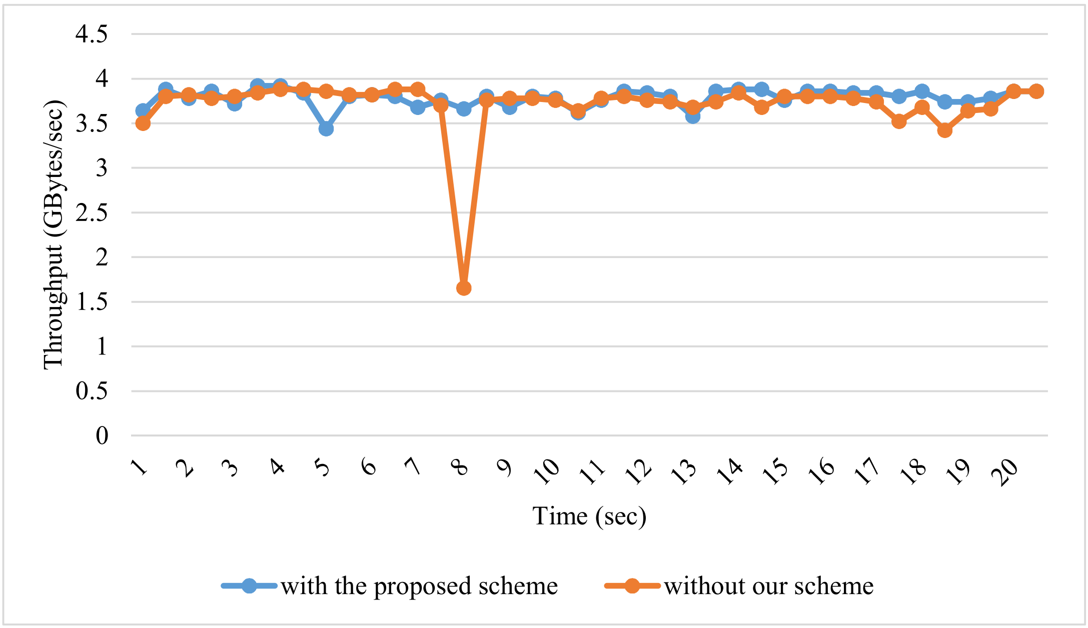

Figure 5- 12 Throughput with and without using our scheme $\Delta t = 100$ ms

the throughput without our scheme got significantly low (1.65 GBytes/sec), almost half of the general throughput whereas the throughput with proposed scheme did not got obviously low. Because of two times measurement, the throughput variations with our scheme are different from the one without our scheme. But compared with the general throughput characteristic shown in Figure 5- 10, the throughput with our scheme is within the lower bound value of 3.3 GBytes/sec.

Figure 5- 13 and Figure 5- 14 show the throughput with our scheme and comparison with the one without our scheme when $\Delta t = 500$ ms, we can see that the throughput without the proposed scheme got much lower (0.096GBytes/sec).

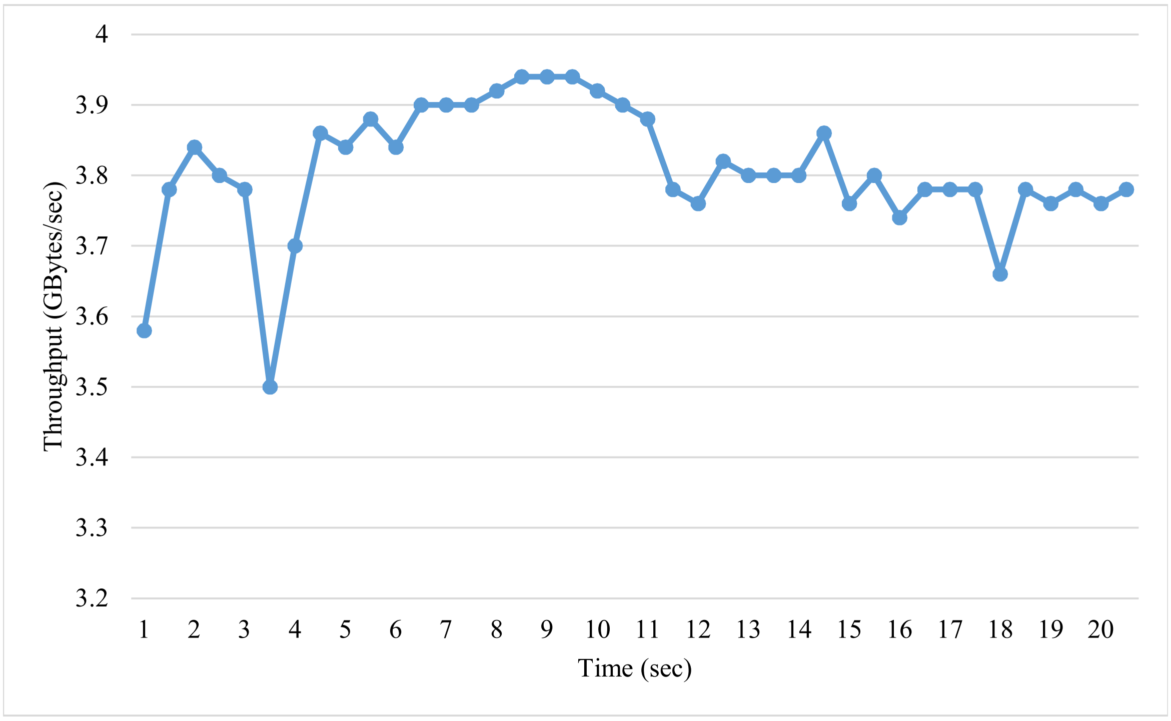

Figure 5- 13 Throughput with our scheme $\Delta t = 500$ ms

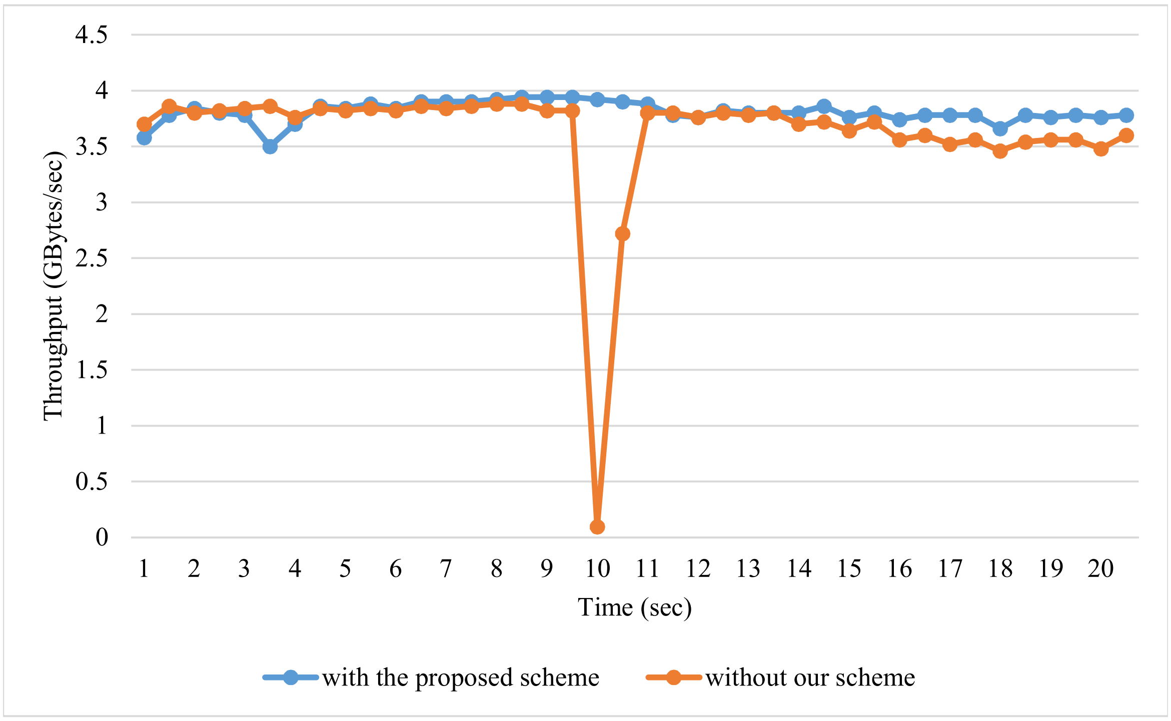

Figure 5- 14 Throughput with and without using our scheme $\Delta t = 500$ ms

Figure 5- 15 and Figure 5- 16 show the throughput with our scheme and comparison with the one without our scheme when $\Delta t = 900$ ms, we can see that the throughput without proposed scheme got zero for more than 2 seconds.

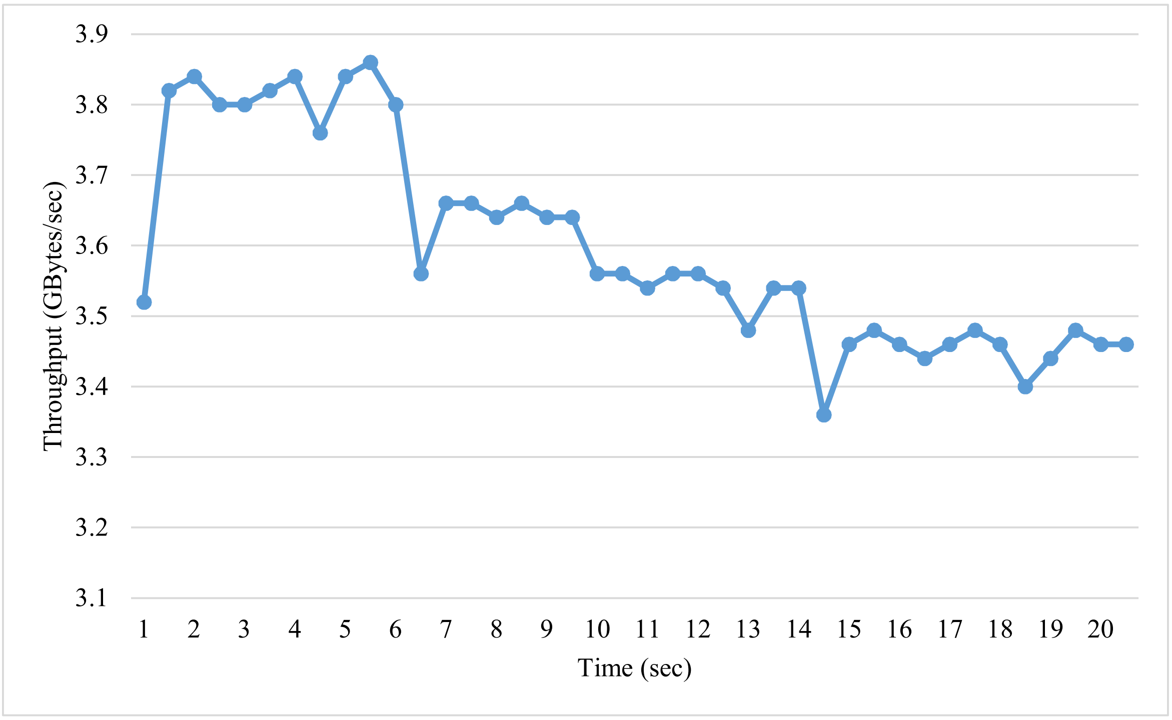

Figure 5- 15 Throughput with our scheme $\Delta t = 900$ ms

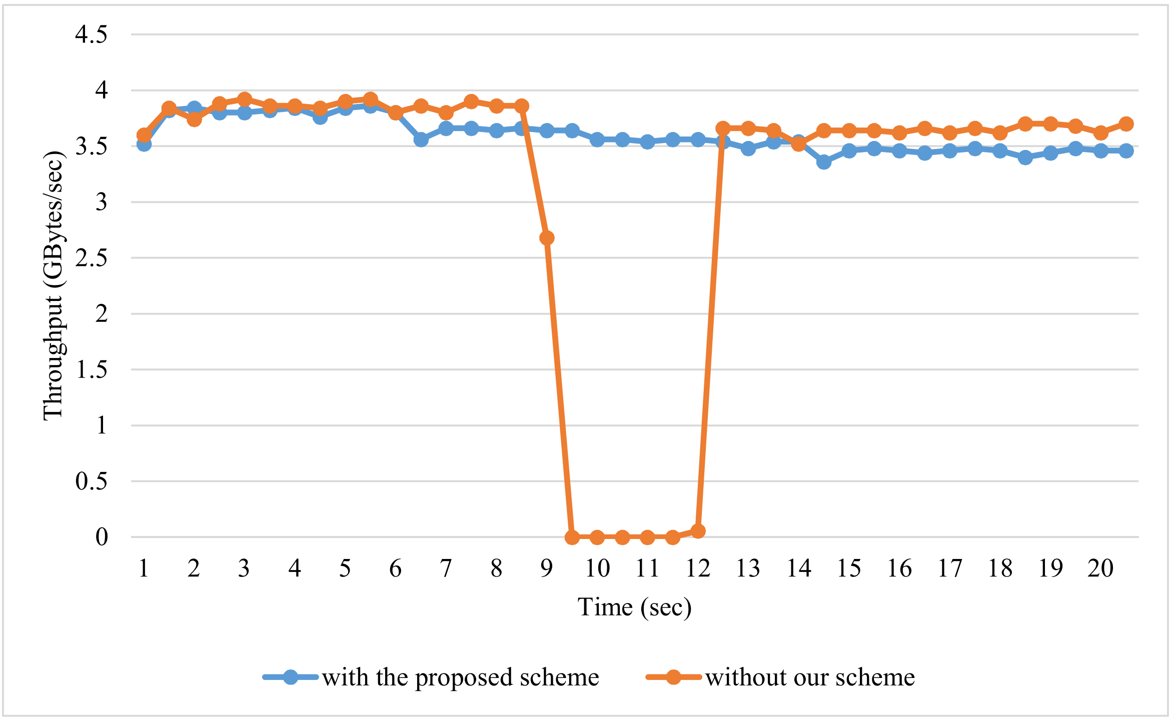

Figure 5- 16 Throughput with and without using our scheme $\Delta t = 900$ ms

The figures from Figure 5- 11 to Figure 5- 16 show the results of throughput measured during network configuration updating. As the results shown, as the value of Δt increased, the throughput without our proposed scheme got lower. If the controller did not handle the consistency problem, the network would behave terribly worse as shown Figure 5- 14 and Figure 5- 16 during the updating period. The throughput was low (as in Figure 5- 12) or even zero (as in Figure 5- 16) when Δt had a large value.

The results also showed that compared with the general throughput characteristic of paths, the throughput with our scheme is within the lower bound value 3.3 GBytes/sec which mean that our scheme could guarantee the throughput when network configuration update even when Δt had a large value.

5.4.4 UDP test

We tested on the network topology based on the same scenario but *iperf* sends UDP packets. We repeated the experiment with different value of Δt with and without using our scheme to illustrate how this delay difference between the controller and switches created inconsistency problems when our scheme was not applied.

Figure 5- 17 shows the number of lost packets and the loss rate during network updates without using the proposed scheme. Upper (blue) line represents the number of lost packets out of total 1785 packets and lower (orange) line represents packet loss rate during 20 seconds.

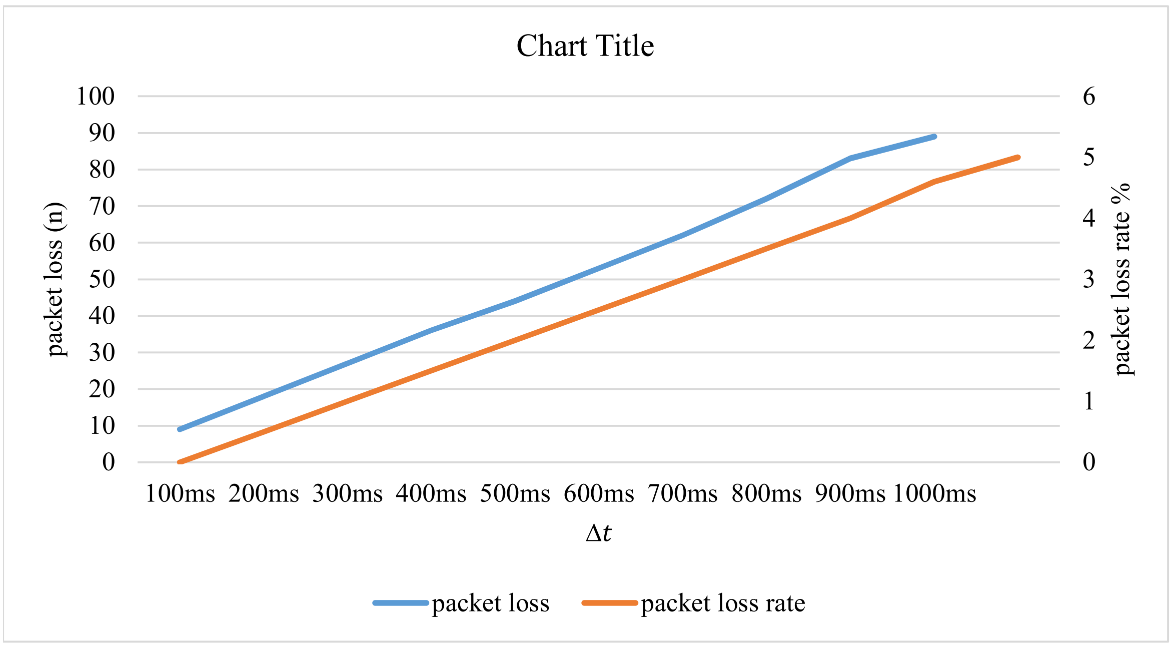

Figure 5- 17 Packet loss and packet loss rate (out of total 1785 packets sent) when our scheme was not used.

We repeated the experiment by using our scheme to update network on the same system and we confirm no packet loss detected in the *iperf* output.

Chapter 6

Discussion

In Chapter 5, we described an approach to test the delay between controller and data plane, showed how the inconsistency makes the network behave incorrectly, and verified the correctness of our proposed scheme.

Chapter 3 introduced related work in the field of the consistency problem, and compared our scheme with previous work.

The proposed scheme analyzes old path and new paths, computes update order, and updates the data plane in such a way that there are no loops and no packet losses. Our scheme guarantees the bandwidth between controller and data plane since our scheme does not have to forward any data packets to the controller.

Both Type I and Type II relations in our scheme can guarantee per-packet consistency. Type I relation also guarantees per-flow consistency since every packet in a flow finally arrives at the same and correct destination but some flows may suffer out-of-order packet delivery depending on the network topology.

There are some limitations of our scheme. It is based on the assumption that only one rule is matched by a flow and it cannot support any modification of packet header through the forwarding path, which may be required for host mobility

In future, we will improve the proposed scheme to support header modification, multi-flow match and per flow consistency.

Chapter 7

Conclusion

This thesis introduced software-defined networking (SDN), gave an overview of the architecture, and listed its benefits and limitations. It also described OpenFlow as the most popular protocol used to communication between the control plane and the data plane of SDN.

It defined and analyzed the consistency problem, which lead to an incorrect network behavior in SDN. In order to solve this problem, it proposed a scheme to update SDN networks in such a way that there exist no packet loops or losses.

The controller analyzes the rules and network topology in the data plane, and classifies the relation between old and new paths into two types, based on whether an open loop is included in the topology or not.

It described the procedure to find loops included by old and new paths by using the graph theory concept and presented the update procedure. It also explained about the experiment conducted to verify the proposed scheme. From the results of the experiments, it is verified that the proposed scheme can guarantee the throughput and with no packet loss during the update period.

Acknowledgments

This Master thesis has been accomplished at the laboratory of Network Architecture in the Department of Information Network Systems in the University of Electro-Communications and at National Institute of Information and Communications Technology (NICT).

I would like to express my appreciation towards my advisor Professor Ved P. Kafle, for his constructive advices. He always teach me how to find problems in research and instruct me to do real research in a correct way.

Thanks to YUASA International Student Scholarship and JEES who gave me financial support in the second year. I could focus more on the research.

I would like to specially thank Zhang Zhiqi, and Xu Wang who are always supporting behind me.

Thanks to my friends and lab members who are friendly to me.

Thanks to my families who are supporting me and encouraging me all the time.

References

[1] K. Suzuki et al., “A Survey on OpenFlow technologies,” IEICE Transactions on Communications, 97(2):375-386, 2014.

[2] https://www.opennetworking.org/about/onf-overview

[3] “Software-defined networking: The new norm for networks,” ONF White Paper, 2012.

[4] https://www.opennetworking.org/sdn-resources/openflow

[5] N. McKeown et al., “OpenFlow: Enabling Innovation in Campus Networks,” ACM SIGCOMM Computer Communication Review, 38(2):69-74, 2008.

[6] M. Reitblatt et al., “Abstractions for Network Update,” ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 2012.

[7] https://www.opennetworking.org/sdn-resources/openflow

[8] Open Networking Foundation, “SDN Architecture Overview Version 1.0,” December 2013

[9] N. Feamster, J. Rexford, and E. Zegura, “The road to SDN: an intellectual history of programmable networks,” ACM SIGCOMM Computer Communication Review 44(2): 87-98, 2014.

[10] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[11] Nick McKeown, “Software-defined Networking,” Proc. of the INFOCOM, April 2009.

[12] H. Kim and N. Feamster, ‘‘Improving Network Management with Software Defined Networking,’’ IEEE Communications Magazine, 51(2):114-119, 2013.

[13] D. Kreutz et al., “Software-defined Networking: A Comprehensive Survey,” Proceedings of the IEEE, 103(1): 14-76, 2015.

[14] Stanford University, Clean Slate Program. 2006. http://cleanslate.stanford.edu/

[15] C.K. Zhang et al., “State-of-the-Art Survey on Software-defined Networking (SDN),” Journal of Software, 26(1):62−81, 2015.

[16] Technical Communities Overview, https://www.opennetworking.org/technical-communities

[17] Operator Area, https://www.opennetworking.org/technical-communities/areas/operator

[18] Services Area, https://www.opennetworking.org/technical-communities/areas/services

[19] Specification Area, https://www.opennetworking.org/technical-communities/areas/specification

[20] Market Area, https://www.opennetworking.org/technical-communities/areas/market

[21] V. Bollapragada, C. Murphy, and R. White, “Inside Cisco IOS Software Architecture, 1st ed,” Indianapolis, IN, USA: Cisco Press, July 2000.

[22] Project Floodlight, http://www.projectfloodlight.org/

[23] OpenDayLight, https://www.opendaylight.org/

[24] Trema https://github.com/trema/trema

[25] https://osrg.github.io/ryu/

[26] D. Erickson, ‘‘The Beacon OpenFlow Controller,’’ in Proc. 2nd ACM SIGCOMM Workshop Hot Topics Software Defined Networking, 2013.

[27] POX, https://github.com/noxrepo/pox

[28] N. Gude et al., “NOX: Towards an Operating System for Networks,” ACM SIGCOMM Computer Communication Review 38(3): 105-110, 2008.

[29] A. Tootoonchian et al., “On Controller Performance in Software-defined Networks,” In: Proc. of the USENIX Hot-ICE, 2012.

[30] N. Foster et al., ‘‘Frenetic: A Network Programming Language,’’ SIGPLAN Notes, 46(9):279–291, 2011.

[31] OpenFlow Switch Specification Version 1.0.0 (Wire Protocol 0x01) December 2009

[32] E. A. Brewer, “Towards Robust Distributed Systems,” Principles of Distributed Computing, Portland, Oregon, July 2000

[33] A. Panda et al., “CAP for Networks,” Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, 2013.

[34] P. Francois et al., “Avoiding Disruptions During Maintenance Operations on BGP Sessions,” IEEE Trans. on Network and Service Management, 4(3):1-11, 2007.

[35] L. Vanbever et al., “Seamless Network-wide IGP Migration,” ACM SIGCOMM Computer Communication Review 41(4):314-325, 2011.

[36] P. Francois, M. Shand, and O. Bonaventure, “Disruption-free Topology Reconfiguration in OSPF networks,” in IEEE INFOCOM, May 2007.

[37] A. X. Liu, C. R. Meiners, and E. Torng, “TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs,” IEEE/ACM Trans. Networking, 18(2):490–500, 2010.

[38] N.P. Katta, J. Rexford, and D. Walker. “Incremental Consistent Updates,” ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, 2013.

[39] T. Mizrahi and Y. Moses, “Time-based Updates in Software Defined Networks,” HotSDN’ 13, August 2013.

[40] R. McGeer, “A Safe, Efficient Update Protocol for OpenFlow Networks,” Proceedings of First ACM Workshop on Hot Topics in Software Defined Networks, New York, 2012.

[41] K. Zhao, Q. Li, and Y. Jiang, “Flow-level Consistent Update in SDN Based on K-prefix Covering,” IEEE Global Communications Conference, 2014.

[42] M. Reitblatt et al., “Software Updates in OpenFlow Networks: Change You Can Believe in,” In Proceedings of HotNets, 2011.

[43] Z. Ye et al., “Classification Based Consistent Flow Update Scheme in Software Defined Network,” Journal of Electronics & Information Technology, 35(7): 1746-1752, 2013.

[44] B. Heller, R. Sherwood, and N. McKeown, “The Controller Placement Problem,” Proceedings of the first ACM Workshop on Hot topics in software defined networks, 2012.

[45] P. Peresini et al., “OF. CPP: Consistent Packet Processing for OpenFlow,” Proceedings of the second ACM SIGCOMM Workshop on Hot topics in software defined networking, 2013.

[46] K. Paton, “An Algorithm for Finding a Fundamental Set of Cycles of a Graph,” Comm. ACM, 12(9): 514-518, 1969.

[47] B. Lantz, B. Heller, and Nick McKeown, “A Network in A Laptop: Rapid Prototyping for Software-Defined Networks,” Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010.

[48] The Network Simulator ns-2, http://www.isi.edu/nsnam/ns/

[49] Opnet, http://www.opnet.com/solutions/network_rd/modeler.html

[50] Y. Liu and B. Plale, “Survey of Publish Subscribe Event Systems,” Computer Science Dept., Indian University 16 (2003).

[51] Wireshark, https://www.wireshark.org/

[52] Iperf, https://iperf.fr/